Учебник Jakarta® EE

Команда Jakarta Platform, https://projects.eclipse.org/projects/ee4j.jakartaee-platform
https://accounts.eclipse.org/mailing-list/jakartaee-platform-dev
Release 9.1, декабрь 2021
Final

Содержание

- Предисловие
  - Целевая аудитория
  - Прежде чем читать эту книгу
  - Сопутствующая документация
  - Условные обозначения
  - Пути по умолчанию и имена файлов
- Часть I: Введение
  - Глава 1. Обзор
    - Введение в Jakarta EE
    - Основные моменты платформы Jakarta EE 9
    - Модель приложения Jakarta EE
    - Распределённые многослойные приложения
    - Контейнеры Jakarta EE
    - Поддержка веб-сервисов
    - Сборка и развёртывание приложений Jakarta EE
    - API Jakarta EE
    - API Jakarta EE 9 в Java Platform, Standard Edition 8
    - Утилиты Eclipse GlassFish Server
  - Глава 2. Использование примеров учебника
    - Требуемое программное обеспечение
    - Запуск и остановка GlassFish Server
    - Запуск Консоли администрирования
    - Запуск и остановка Apache Derby
    - Сборка примеров
    - Структура каталогов примеров из учебника
    - Maven-архетипы Jakarta EE в учебнике
    - Отладка приложений Jakarta EE
- Часть II: Основы платформы
  - Глава 3. Создание ресурса
    - Ресурсы и именование JNDI
    - Объекты источника данных и пулы соединений
    - Административное создание ресурсов
  - Глава 4. Инъецирование
    - Инъецирование ресурса
    - Инъецирование зависимостей
    - Основные различия между инъецированием ресурса и инъецированием зависимости
  - Глава 5. Упаковка
    - Упаковка приложений
    - Упаковка Enterprise-бинов
    - Упаковка веб-архивов
    - Упаковка архивов адаптера ресурсов
- Часть III: Веб-слой
  - Глава 6. Начало работы с веб-приложениями
    - Веб-приложения
    - Жизненный цикл веб-приложения
    - Веб-модуль с использованием Jakarta Faces: пример hello1

Веб-модуль с использованием Jakarta Servlet: пример hello2
Настройка веб-приложений
Дополнительная информация о веб-приложениях
Глава 7. Jakarta Faces
Введение в Jakarta Faces
Что такое приложение Jakarta Faces?
Преимущества Jakarta Faces
Простое приложение Jakarta Faces
Модель компонентов пользовательского интерфейса
Модель навигации
Жизненный цикл приложения Jakarta Faces
Частичная обработка и частичное отображение
Дополнительная информация о Jakarta Faces
Глава 8. Введение в Facelets
Что такое Facelets?
Жизненный цикл приложения Facelets
Разработка простого приложения Facelets: пример guessnumber-jsf
Использование шаблонов Facelets
Составные компоненты
Веб-ресурсы
Перемещаемые ресурсы
Контракты библиотеки ресурсов
HTML5-совместимая разметка
Глава 9. Язык выражений (EL)
Обзор EL
Немедленное и отложенное выполнение
Выражения значений и методов
Операции над коллекциями
Операторы
Зарезервированные слова
Примеры выражений EL
Дополнительная информация о EL
Глава 10. Использование Jakarta Faces на веб-страницах
Настройка страницы
Добавление компонентов на страницу с помощью библиотеки тегов HTML
Использование основных тегов
Глава 11. Использование конвертеров, слушателей и валидаторов
Использование стандартных конвертеров
Регистрация слушателей в компонентах
Использование стандартных валидаторов
Ссылка на метод Managed-бина
Глава 12. Разработка с использованием Jakarta Faces
Managed-бины в Jakarta Faces
Запись свойств бина
Пишем методы Managed-бинов
Глава 13. Использование Ajax с Jakarta Faces
Обзор Ajax
Использование Ajax в Jakarta Faces

Использование Ajax с Facelets
Отправка запроса Ajax
Мониторинг событий на клиенте
Обработка ошибок
Получение ответа Ajax
Жизненный цикл запроса Ajax
Группировка компонентов
Загрузка JavaScript как ресурса
Приложение ajaxguessnumber
Дополнительная информация об Ajax в Jakarta Faces
Глава 14. Составные компоненты: дополнительные темы и примеры
Атрибуты составного компонента
Вызов Managed-бина
Валидация значений составного компонента
Пример compositecomponentexample
Глава 15. Создание кастомных компонентов интерфейса пользователя и других кастомных объектов
Введение в создание кастомных компонентов
Определение того, требуется ли кастомный компонент или отрисовщик
Объяснение на примере карты изображения
Шаги для создания кастомного компонента
Создание классов кастомного компонента
Делегирование отрисовки отрисовщику
Реализация слушателя событий
Обработка событий для кастомных компонентов
Определение тега кастомного компонента в дескрипторе библиотеки тегов
Использование кастомного компонента
Создание и использование кастомного конвертера
Создание и использование кастомного валидатора
Связывание значений компонентов и объектов со свойствами Managed-бина
Связывание конвертеров, слушателей и валидаторов со свойствами Managed-бинов
Глава 16. Настройка приложений Jakarta Faces
Введение в настройку приложений Jakarta Faces
Использование аннотаций для настройки Managed-бинов
Файл конфигурации приложения
Использование Faces Flows
Настройка Managed-бинов
Регистрация сообщений приложения
Использование валидаторов по умолчанию
Регистрация кастомного валидатора
Регистрация кастомного конвертера
Настройка правил навигации
Регистрация кастомного отрисовщика с помощью инструментария отрисовки (Render Kit)
Регистрация кастомного компонента
Основные требования к приложениям Jakarta Faces
Глава 17. Использование веб-сокетов с Jakarta Faces
О веб-сокетах в Jakarta Faces
Конфигурирование веб-сокетов
Использование тега f:websocket

Области видимости и пользователи веб-сокетов
Условное подключение веб-сокетов
Вопросы безопасности в веб-сокетах
Использование Ajax с веб-сокетами
Глава 18. Технология Jakarta Servlet
Что такое сервлет?
Жизненный цикл сервлета
Обмен информацией
Создание и инициализация сервлета
Написание сервисных методов
Фильтрация запросов и ответов
Вызов других веб-ресурсов
Доступ к веб-контексту
Поддержка состояния клиента
Финализация сервлета
Загрузка файлов с помощью сервлетов Jakarta
Асинхронная обработка
Неблокирующий ввод/вывод
Серверный Push
Обработка обновления протокола
HTTP Trailer
Приложение mood
Приложение fileupload
Приложение dukeetf
Дополнительная информация о сервлета Jakarta
Глава 19. Jakarta WebSocket
Введение в веб-сокеты
Создание приложений веб-сокетов в платформе Jakarta EE
Программные конечные точки
Аннотированные конечные точки
Отправка и получение сообщений
Поддержка состояния клиента
Использование кодировщиков и декодировщиков
Параметры пути
Обработка ошибок
Указание класса конфигуратора конечной точки
Приложение dukeetf2
Приложение websocketbot
Дополнительная информация о веб-сокетах
Глава 20. Обработка JSON
Введение в JSON
Обработка JSON в платформе Jakarta EE
Использование API объектной модели
Использование потокового API
JSON в Jakarta EE RESTful веб-сервисах
Приложение jsonpmodel
Приложение jsonpstreaming
Дополнительная информация о Jakarta JSON Processing

Глава 21. Связывание с JSON
Связывание с JSON в платформе Jakarta EE
Обзор JSON Binding API
Запуск приложения jsonbbasics
Дополнительная информация о Jakarta JSON Binding
Глава 22. Интернационализация и локализация веб-приложений
Классы локализации платформы Java
Предоставление локализованных сообщений и меток
Форматирование дат и чисел
Наборы и кодировки символов
Часть IV: Bean Validation
Глава 23. Введение в Jakarta Bean Validation
Обзор Jakarta Bean Validation
Использование ограничений Jakarta Bean Validation
Повторяющиеся аннотации
Валидация строк на null и пустоту
Валидация конструкторов и методов
Дополнительная информация о Jakarta Bean Validation
Глава 24. Валидация бинов: дополнительные темы
Создание пользовательских ограничений
Реализация временны́х ограничений с помощью ClockProvider
Пользовательские ограничения
Использование предустановленных экстракторов значений в кастомных контейнерах
Настройка сообщений валидатора
Группировка ограничений
Использование ограничений методов в иерархиях типов
Часть V: Контексты и инъецирование зависимостей Jakarta EE
Глава 25. Введение в Jakarta CDI
Начало работы
Обзор CDI
О бинах
О Managed-бинах CDI
Бины как инъецируемые объекты
Использование квалификаторов
Инъецирование бинов
Использование областей видимости
Присвоение бинам имён EL
Добавление set- и get- методов
Использование Managed-бина на странице Facelets
Инъецирование объектов с использованием методов-производителей
Настройка приложения CDI
Использование аннотаций @PostConstruct и @PreDestroy с классами Managed-бинов CDI
Дополнительная информация о CDI
Глава 26. Базовые примеры инъецирования контекстов и зависимостей
Сборка и запуск примеров CDI
Пример simplegreeting на CDI
Пример CDI gunessnumber-cdi
Глава 27. Jakarta CDI: дополнительные темы

Упаковка CDI-приложений
Использование альтернатив в приложениях CDI
Использование методов-производителей, полей-производителей и методов закрытия в приложениях CDI
Использование предопределённых бинов в приложениях CDI
Использование событий в приложениях CDI
Использование Interceptor-ов в приложениях CDI
Использование декораторов в приложениях CDI
Использование стереотипов в приложениях CDI
Использование встроенных литералов аннотации
Использование интерфейсов конфигураторов
Глава 28. Начальная загрузка контейнера CDI в Java SE
Bootstrap API
Конфигурирование контейнера CDI
Глава 29. Дополнительные примеры инъецирования контекстов и зависимостей
Сборка и запуск дополнительных примеров CDI
Пример encoder: использование альтернатив
Пример producermethods: использование метода-производителя для выбора реализации компонента
Пример producerfields: использование полей-производителей для создания ресурсов
Пример billpayment: использование событий и Interceptor-ов
Пример decorators: декорирование бинов
Часть VI: Веб-сервисы
Глава 30. Введение в веб-сервисы
Что такое веб-сервисы?
Типы веб-сервисов
Выбор типа веб-сервиса для использования
Глава 31. Создание веб-сервисов с Jakarta XML Web Services
Обзор Jakarta XML Web Services
Создание простого веб-сервиса и клиентов с Jakarta XML Web Services
Типы, поддерживаемые Jakarta XML Web Services
Совместимость веб-сервисов и Jakarta XML Web Services
Дополнительная информация о Jakarta XML Web Services
Глава 32. Создание RESTful веб-сервисов с Jakarta REST
Что такое RESTful веб-сервисы?
Создание класса корневого ресурса RESTful
Примеры приложений Jakarta REST
Дополнительная информация о Jakarta REST
Глава 33. Доступ к ресурсам REST с помощью клиентского API Jakarta REST
Обзор клиентского API
Использование клиентского API в примере приложения Jakarta REST
Дополнительные возможности клиентского API
Глава 34. Jakarta REST: дополнительные темы и пример
Аннотации для полей и свойств компонентов классов ресурсов
Валидация данных ресурса с Bean Validation
Подресурсы и разрешение ресурсов времени выполнения
Интеграция Jakarta REST с Jakarta Enterprise Beans и CDI
Условные HTTP-запросы
Согласование содержимого во время выполнения
Использование Jakarta REST с Jakarta XML Binding

Приложение customer
Часть VII: Enterprise-бины
Глава 35. Enterprise-бины
Что такое Enterprise-бин?
Что такое сессионный бин?
Что такое бин, управляемый сообщениями?
Доступ к Enterprise-бинам
Содержимое Enterprise-бина
Соглашения об именовании Enterprise-бинов
Жизненные циклы Enterprise-бинов
Дополнительная информация об Enterprise-бинах
Глава 36. Начало работы с Enterprise-бинами
Начало работы с Enterprise-бинами
Создание Enterprise-бина
Изменение приложения Jakarta EE
Глава 37. Запуск примеров Enterprise-бина
Обзор примеров Jakarta Enterprise Beans
Пример cart
Сессионный компонент-синглтон counter
Пример веб-сервиса: helloservice
Использование сервиса таймера
Обработка исключений
Глава 38. Использование встроенного EJB-контейнера
Обзор встроенного EJB-контейнера
Разработка встраиваемых приложений Enterprise-бинов
Приложение standalone
Глава 39. Использование асинхронного вызова методов в сессионных компонентах
Вызов асинхронных методов
Приложение async
Часть VIII: Персистентность
Глава 40. Введение в Jakarta Persistence
Обзор Jakarta Persistence
Сущности
Наследование сущностей
Управление сущностями
Выборка объектов сущностей
Создание схемы базы данных
Дополнительная информация о персистентности
Глава 41. Запуск примеров персистентности
Обзор примеров персистентности
Приложение order
Приложение roster
Приложение address-book
Глава 42. Язык запросов Jakarta Persistence
Обзор языка запросов Jakarta Persistence
Терминология языка запросов
Создание запросов с использованием языка запросов Jakarta Persistence
Упрощённый синтаксис языка запросов

Примеры запросов
Полный синтаксис языка запросов
Глава 43. Использование Criteria API для создания запросов
Обзор Criteria и Metamodel API
Использование Metamodel API для моделирования классов объектов
Использование Criteria и Metamodel API для создания типобезопасных запросов
Глава 44. Создание и использование строковых запросов Criteria
Обзор строковых запросов Criteria
Создание строковых запросов
Выполнение строковых запросов
Глава 45. Управление параллельным доступом к данным объекта с помощью блокировки
Обзор блокировки сущностей и параллелизма
Режимы блокировки
Глава 46. Создание планов выполнения с помощью графов сущностей
Обзор использования планов выполнения и графов сущностей
Основы графа сущностей
Использование именованных графов сущностей
Использование графов сущностей в запросах
Глава 47. Использование кэша второго уровня в приложениях Jakarta Persistence
Обзор кэша второго уровня
Задание настроек режима кэширования для повышения производительности
Часть IX: Обмен сообщениями
Глава 48. Концепции обмена сообщениями в Jakarta
Обзор Jakarta Messaging
Основные концепции Jakarta Messaging
Модель программирования Jakarta Messaging
Дополнительные функции JMS
Использование Jakarta Messaging в приложениях Jakarta EE
Дополнительная информация о Jakarta Messaging
Глава 49. Примеры обмена сообщениями Jakarta
Примеры сборки и запуска Jakarta Messaging
Обзор примеров JMS
Простые приложения JMS
Более сложные приложения JMS
Высокопроизводительные и масштабируемые приложений JMS
Отправка и получение сообщений в простом веб-приложении
Асинхронный приём сообщений с использованием бина, управляемого сообщениями
Отправка сообщений из сессионного компонента в MDB
Использование сущности для объединения сообщений из двух MDB
Создание ресурсов Jakarta Messaging в IDE NetBeans
Часть X: Безопасность
Глава 50. Введение в безопасность Jakarta EE Platform
Обзор Jakarta Security
Механизмы безопасности
Использование подключаемых провайдеров
Защита контейнеров
Обеспечение безопасности GlassFish Server
Работа с хранилищами идентификаторов

Работа с областями безопасности, пользователями, группами и ролями
Установление безопасного соединения с использованием SSL
Дополнительная информация о безопасности
Глава 51. Начало работы по защите веб-приложений
Обзор безопасности веб-приложений
Защита веб-приложений
Использование программной безопасности в веб-приложениях
Примеры: защита веб-приложений
Глава 52. Начало работы по защите EJB-приложений
Основные задачи безопасности для EJB-приложений
Защита Enterprise-бинов
Примеры: защита Enterprise-бинов
Глава 53. Использование Jakarta Security
О Jakarta Security
Обзор интерфейса механизма аутентификации HTTP
Обзор интерфейсов хранилища идентификаторов
Выполнение примера встроенного хранилища идентификаторов базы данных
Запуск примера кастомного хранилища идентификаторов
Глава 54. Безопасность Jakarta EE: дополнительные темы
Работа с цифровыми сертификатами
Механизмы аутентификации
Использование области безопасности JDBC для аутентификации пользователя
Защита ресурсов HTTP
Защита клиентских приложений
Защита информационных систем (ИС) предприятия
Настройка безопасности с использованием дескрипторов развёртывания
Дополнительная информация о разделах повышенной безопасности
Часть XI: Технологии, поддерживаемые Jakarta EE
Глава 55. Транзакции
Обзор транзакций
Транзакции в приложениях Jakarta EE
Что такое транзакция?
Управляемые контейнером транзакции
Управляемые бином транзакции
Тайм-ауты транзакций
Обновление нескольких баз данных
Транзакции в веб-компонентах
Дополнительная информация о транзакциях
Глава 56. Адаптеры и контракты ресурсов
Что такое адаптер ресурсов?
Аннотации
Общий клиентский интерфейс
Использование адаптеров ресурсов с инъецированим контекстов и зависимостей Jakarta (CDI)
Дополнительная информация об адаптерах ресурсов
Глава 57. Примеры адаптеров ресурсов
Обзор примеров адаптеров ресурсов
Пример trading
Пример traffic

Глава 58. Использование Interceptor-ов Jakarta EE
Обзор Interceptor-ов
Использование Interceptor-ов
Пример interceptor
Глава 59. Пакетная обработка
Введение в пакетную обработку
Пакетная обработка в Jakarta EE
Простой вариант использования
Использование языка спецификации задания
Создание пакетных артефактов
Отправка заданий в пакетную среду выполнения
Пакетные приложения
Пример webserverlog
Пример phonebilling
Дополнительная информация о пакетной обработке
Глава 60. Параллелизм Jakarta
Основы параллелизма
Основные компоненты утилит параллелизма
Параллелизм и транзакции
Параллелизм и безопасность
Пример jobs
Пример taskcreator
Дополнительная информация о Jakarta Concurrency
Часть XII: Учебные примеры
Глава 61. Пример Duke's Bookstore
Дизайн и архитектура Duke's Bookstore
Интерфейс Duke's Bookstore
Запуск Duke's Bookstore
Глава 62. Пример Duke’s Tutoring
Дизайн и архитектура Duke’s Tutoring
Основной интерфейс
Интерфейс администрирования
Запуск Duke’s Tutoring
Глава 63. Пример Duke's Forest
Обзор примера Duke’s Forest
Дизайн и архитектура Duke's Forest
Сборка и развёртывание Duke’s Forest
Запуск Duke's Forest

```
Учебник Jakarta® EE

Версия: Release 9.1

Статус: Завершён

Релиз: Декабрь 2021
```


Оригинальный перевод: Jakarta EE Tutorial — русскоязычная версия
(https://www.bychkov.name/Учебник%20Jakarta%20EE.pdf).

Автор перевода Владимир Бычков: github@bychkov.name

Предисловие

Этот учебник представляет собой руководство по разработке корпоративных приложений для платформы Jakarta EE 9 с использованием Eclipse GlassFish Server.

Eclipse GlassFish Server — ведущая платформа с открытым исходным кодом для создания и развёртывания приложений и сервисов следующего поколения. Eclipse GlassFish Server, разработанный сообществом открытого исходного кода Eclipse GlassFish, доступен по ссылке https://projects.eclipse.org/projects/ee4j.glassfish и является совместимой реализацией спецификации платформы Jakarta EE 9. Этот легковесный, гибкий и открытый сервер приложений позволяет организациям не только использовать новые возможности, представленные в спецификации Jakarta EE 9, но и добавлять свои функции за счёт более быстрого и оптимизированного цикла разработки и развёртывания. Eclipse GlassFish Server далее именуется GlassFish Server.

Целевая аудитория

Данное учебное пособие предназначено для программистов, интересующихся разработкой и развёртыванием приложений Jakarta EE 9. Оно охватывает технологии, входящие в платформу Jakarta EE, и описывает, как разрабатывать компоненты Jakarta EE и развёртывать их в Eclipse GlassFish.

Прежде чем читать эту книгу

Прежде чем приступить к изучению этого учебника, вы должны обладать хорошими знаниями Java. Хороший способ достичь этого — проработать учебники Java™ https://docs.oracle.com/javase/tutorial/index.html.

Сопутствующая документация

Документация по GlassFish Server описывает планирование развёртывания и установку системы. Чтобы получить документацию по GlassFish Server перейдите по ссылке https://glassfish.org/docs.

Спецификацию Jakarta EE 9 API можно просмотреть по ссылке https://jakarta.ee/specifications/platform/9/.

Кроме того, могут быть полезны спецификации Jakarta EE по ссылке https://jakarta.ee/specifications.

Дополнительная информация о создании приложений J2EE в IDE NetBeans приведена на веб-странице https://netbeans.apache.org/kb/.

Для получения информации об использовании Apache Derby с GlassFish Server см. https://db.apache.org/derby/docs/10.14/adminguide/.

Проект GlassFish Samples представляет собой набор примеров приложений, демонстрирующих широкий спектр технологий Jakarta EE. Примеры GlassFish доступны на странице проекта примеров GlassFish по ссылке https://github.com/eclipse-ee4j/glassfish-samples/.

Условные обозначения

В следующей таблице описаны типографские соглашения, используемые в этой книге.

Условное обозначение	**Значение**	**Пример**

Условное обозначение	Значение	Пример
Жирный	Жирный шрифт обозначает элементы графического интерфейса пользователя, связанные с действием или терминами, определёнными в тексте.	В меню **Файл** выберите **Открыть проект**. **cache** — это копия, которая хранится локально.
`Monospace`	Моноширинный тип указывает имена файлов и каталогов, команды внутри абзаца, URL, код в примерах, текст, который появляется на экране, или текст, который вы вводите.	Отредактируйте файл `.login`. Используйте `ls -a` для просмотра списка всех файлов. *`machine_name`*`% у вас есть почта.`
курсивный	Курсив указывает на названия книг, выделение или переменные-заполнители, для которых вы указываете конкретные значения.	Прочтите главу 6 в *Руководстве пользователя*. *Не* сохраняйте файл. Команда для удаления файла: `rm` *`filename`*.

Пути по умолчанию и имена файлов

В следующей таблице описаны пути и имена файлов по умолчанию, которые используются в этой книге.

Обозначение	Описание	Значение по умолчанию
`as-install`	Представляет каталог установки GlassFish Server.	Установки в операционной системе Solaris, Linux и Mac: *`user's-home-directory`*`/glassfish6/glassfish` Windows, все системы: *`SystemDrive`*`:\glassfish6\glassfish`
`as-install-parent`	Родительский элемент базового каталога установки для GlassFish Server.	Установки в операционной системе Solaris, Linux и Mac: *`user's-home-directory`*`/glassfish6` Windows, все системы: *`SystemDrive`*`:\glassfish6`
`tut-install`	Представляет базовый каталог установки для материалов Учебника Jakarta EE после его загрузки и распаковки.	*`user's-home-directory`*`/jakartaee-tutorial`

Обозначение	Описание	Значение по умолчанию
domain-dir	Каталог, в котором хранится конфигурация домена.	*as-install*/domains/domain1

Часть I: Введение

Часть I представляет платформу, учебное пособие и примеры.

Глава 1. Обзор

Эта глава знакомит вас с разработкой корпоративных приложений на Jakarta EE. Здесь вы ознакомитесь с основами разработки, узнаете об архитектуре и API Jakarta EE, познакомитесь с важными терминами и концепциями и узнаете, как подходить к программированию, сборке и развёртыванию приложений Jakarta EE.

Введение в Jakarta EE

Сегодня разработчики всё больше осознают необходимость в распределённых, транзакционных и переносимых приложениях, которые используют скорость, безопасность и надёжность серверных технологий. Корпоративные приложения обеспечивают бизнес-логику для предприятия. Они управляются централизованно и часто взаимодействуют с другим корпоративным программным обеспечением. В мире информационных технологий проектирование, конструирование и поставка приложений должно осуществляться с меньшей стоимостью, большей скоростью и меньшими ресурсами.

С Jakarta EE разработка корпоративных приложений Java становится простой и быстрой как никогда раньше. Цель платформы Jakarta EE — предоставить разработчикам мощный набор API, сократив время разработки, снизив сложность и повысив производительность приложений.

Платформа Jakarta EE разрабатывается в рамках Jakarta EE Specification Process. Экспертные группы состоят из заинтересованных сторон и создают спецификации Jakarta, чтобы определить различные технологии Jakarta EE. Работа сообщества Jakarta в рамках программы Jakarta EE Specification Process помогает обеспечить стабильность и кроссплатформенность стандартов технологии Java.

Платформа Jakarta EE использует упрощённую программную модель. Дескрипторы развёртывания XML являются необязательными. Вместо этого разработчик может указать информацию в аннотации непосредственно в исходном файле Java, а сервер Jakarta EE настроит компонент в процессе развёртывания и выполнения. Эти аннотации обычно используются для встраивания данных в программу, которые иначе были бы представлены в дескрипторе развёртывания. С аннотациями информация помещается непосредственно в код конфигурируемого элемента программы.

В платформе Jakarta EE инъецирование зависимостей может применяться ко всем ресурсам, в которых нуждается компонент, эффективно скрывая создание и поиск ресурсов из кода приложения. Инъецирование зависимостей можно использовать в контейнерах корпоративных компонентов, веб-контейнерах и клиентских приложениях. Инъецирование зависимостей позволяет контейнеру Jakarta EE автоматически вставлять ссылки на другие необходимые компоненты или ресурсы, используя аннотации.

В этом руководстве используются примеры для описания функций, доступных в платформе Jakarta EE для разработки корпоративных приложений. Независимо от того, являетесь ли вы новичком или опытным разработчиком, примеры и сопровождающие их пояснения станут ценной и доступной базой знаний для создания собственных решений.

Основные моменты платформы Jakarta EE 9

Цель выпуска Jakarta EE 9 состоит в том, чтобы предоставить набор спецификаций, функционально схожих с Jakarta EE 8, но в новом пространстве имён Jakarta EE 9 jakarta.*.

Кроме того, с выпуском Jakarta EE 9 исчезает небольшой набор спецификаций из Jakarta EE 8 — устаревших или необязательных — для уменьшения общего количества API с целью облегчить новым поставщикам вход в экосистему, а также уменьшить затраты на внедрение, миграцию и обслуживание этих API.

Следующие технологии Jakarta EE были удалены в новой версии платформы Jakarta EE:

- XML Registries 1.0
- XML RPC 1.1
- Deployment 1.7
- Management 1.1
- Распределённая совместимость (EJB 3.2 Core Specification, Chapter 10)

Помимо удалённых технологий, некоторые технологии в выпуске Jakarta EE 9 помечены как опциональные. Причина в том, что некоторые технологии, изначально включённые в Jakarta EE, уже не так актуальны, как во время добавления в платформу.

Проект спецификации платформы может принять решение официально «удалить» «необязательную» функцию из платформы в одном из последующих выпусков.

Необязательными являются следующие технологии:

- Jakarta Enterprise Beans 3.2 и более ранние бины сущностей и ассоциированный с ними Jakarta Enterprise Beans QL
- Группа API Jakarta Enterprise Beans 2.x
- Jakarta Enterprise Web Services 2.0
- Jakarta SOAP with Attachments 2.0
- Jakarta Web Services Metadata 3.0
- Jakarta XML Web Services 3.0
- Jakarta XML Binding 3.0

Модель приложения Jakarta EE

Модель приложения Jakarta EE начинается с языка программирования Java и виртуальной машины Java. Проверенная переносимость, безопасность и производительность разработки, которые они обеспечивают, составляют основу модели приложения. Jakarta EE предназначена для поддержки приложений, которые реализуют корпоративные сервисы для клиентов, сотрудников, поставщиков, партнёров и всех, кто предъявляет требования к предприятию или вносит вклад в его работу. Такие приложения по своей природе являются сложными, потенциально могут предоставлять доступ к данным множества источников и распределённых приложений различным клиентам.

Для лучшего контроля и управления этими приложениями бизнес-функции для поддержки этих различных пользователей сосредоточены в среднем слое. Средний слой представляет собой среду, которая строго контролируется отделом информационных технологий предприятия. Средний слой обычно работает на выделенном серверном оборудовании и имеет доступ ко всем сервисам предприятия.

Модель приложения Jakarta EE определяет архитектуру для реализации сервисов как многослойных приложений, которые обеспечивают масштабируемость, доступность и управляемость, необходимые приложениям уровня предприятия. Эта модель разделяет работу, необходимую для реализации многослойного сервиса, на следующие части:

- Бизнес-логика и логика представления данных, которые реализуются разработчиком
- Стандартные системные сервисы, предоставляемые платформой Jakarta EE

Разработчик может положиться на платформу, чтобы обеспечить решение сложных проблем системного уровня при разработке многослойного сервиса.

Распределённые многослойные приложения

Платформа Jakarta EE использует модель распределённых многослойных приложений для корпоративных приложений. Логика приложения делится на компоненты в зависимости от функции, а компоненты приложения Jakarta EE устанавливаются на различных компьютерах в зависимости от слоя, к которому относится компонент в многослойной среде Jakarta EE.

Рисунок 1-1 показывает два многослойных приложения Jakarta EE, разделённых на слои, описанные в следующем списке. Части приложения Jakarta EE, показанные на рисунке 1-1, представлены в Компоненты Jakarta EE.

- Компоненты клиентского слоя работают на клиентском компьютере.
- Компоненты веб-слоя работают на сервере Jakarta EE.
- Компоненты слоя бизнес-логики работают на сервере Jakarta EE.
- Иные информационные системы (ИС) предприятия работает на своих серверах.

Хотя приложение Jakarta EE может состоять из всех слоёв, показанных на рисунке 1-1, многослойные приложения Jakarta EE обычно рассматриваются как трёхслойные приложения, поскольку они распределены по трём местоположениям: клиентские машины, компьютер Jakarta EE-сервера и база данных или иные информационные системы (ИС) предприятия. Трёхслойные приложения, которые работают таким образом, расширяют стандартную двухслойную модель клиент-сервер, размещая многопоточный сервер приложений между клиентским приложением и серверным хранилищем.

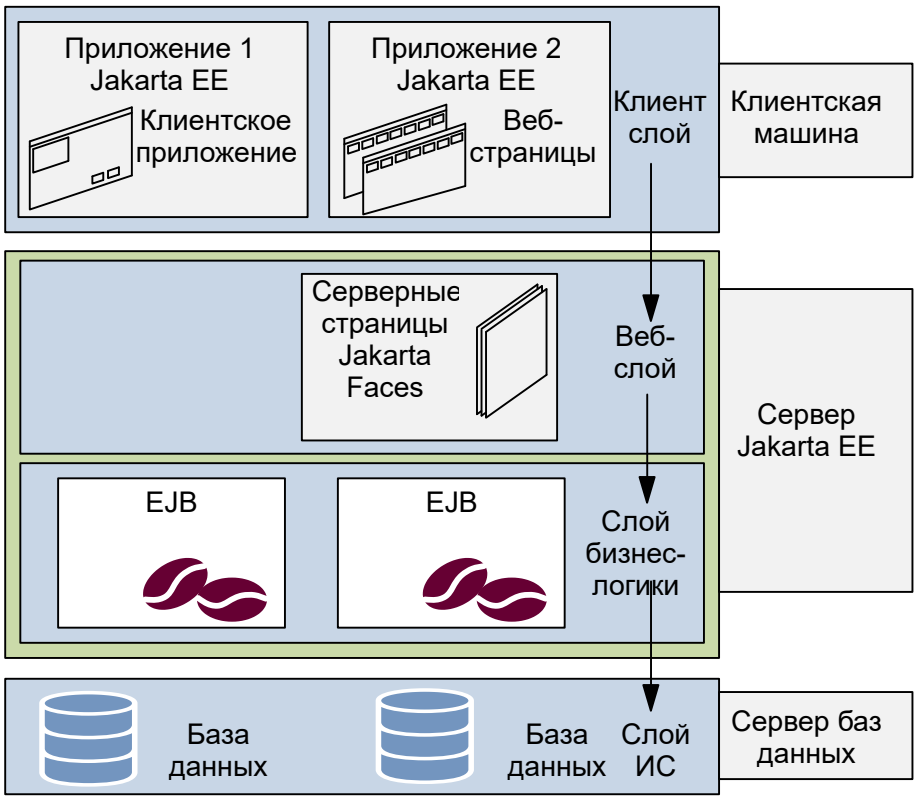

Рисунок 1-1 Многослойные приложения

Безопасность

Хотя для других моделей корпоративных приложений требуются специфичные для каждой платформы меры безопасности в каждом приложении, среда безопасности Jakarta EE позволяет задать ограничения безопасности во время развёртывания. Платформа Jakarta EE делает приложения переносимыми для широкого спектра реализаций безопасности, ограждая разработчиков приложений от сложностей реализации функций безопасности.

Платформа Jakarta EE предоставляет стандартные декларативные правила контроля доступа, которые определяются разработчиком и интерпретируются при развёртывании приложения на сервере. Jakarta EE также предоставляет стандартные механизмы входа в систему, поэтому разработчикам приложений не нужно инъецировать эти механизмы в свои приложения. Одно и то же приложение работает в различных средах безопасности без изменения исходного кода.

Компоненты Jakarta EE

Приложения Jakarta EE состоят из компонентов. Компонент Jakarta EE представляет собой автономный функциональный программный модуль, который компонуется в приложение Jakarta EE со связанными с ним классами и файлами и взаимодействует с другими компонентами.

Спецификация Jakarta EE определяет следующие компоненты Jakarta EE:

- Клиентские приложения и апплеты — это компоненты, которые работают на клиенте.
- Технологии Jakarta Servlet, Jakarta Faces и Jakarta Server Pages — это веб-компоненты, выполняющиеся на сервере
- Компоненты EJB (Enterprise-бины) являются серверными компонентами бизнес-логики.

Компоненты Jakarta EE написаны на Java и компилируются так же, как и любая программа на этом языке. Различия между компонентами Jakarta EE и «стандартными» классами Java заключаются в том, что компоненты Jakarta EE собираются в приложение Jakarta EE, верифицируются на соответствие спецификации Jakarta EE, развёртываются и выполняются под управлением серверов Jakarta EE.

Клиенты Jakarta EE

Клиент Jakarta EE обычно является либо веб-клиентом, либо клиентским приложением.

Веб-клиенты

Веб-клиент состоит из двух частей:

- Динамические веб-страницы, содержащие различные типы языков разметки (HTML, XML и т. д.), которые генерируются веб-компонентами, работающими в веб-слое
- Веб-браузер, который отображает страницы, полученные с сервера

Веб-клиент иногда называют тонким клиентом. Тонкие клиенты обычно не выполняют запросов к базам данных и сложных бизнес-правил. При использовании тонкого клиента эти тяжеловесные операции выполняются Enterprise-бинами, выполняемыми на сервере Jakarta EE, где они могут использовать все возможности безопасности, скорости, все сервисы и надёжность технологий на стороне сервера Jakarta EE.

Клиентские приложения

Клиентское приложение работает на клиентском компьютере и предоставляет пользователям возможность выполнять задачи, для которых требуется более богатый пользовательский интерфейс, чем тот, который может обеспечить язык разметки. Клиентское приложения обычно имеет графический интерфейс пользователя (GUI), созданный из API Swing или API Abstract Window Toolkit (AWT), но безусловно возможен и интерфейс командной строки.

Клиентские приложения получают прямой доступ к Enterprise-бинам, работающим в слое бизнес-логики. Однако, если этого приложению это необходимо, клиентские приложения могут открыть HTTP-соединение, чтобы установить связь с сервлетом, работающим в веб-слое. Приложения, написанные на языках, отличных от Java, могут взаимодействовать с серверами Jakarta EE, что позволяет платформе Jakarta EE взаимодействовать с другими информационными системами и клиентами.

Апплеты

Веб-страница, полученная от веб-слоя, может содержать встроенный апплет. Написанный на Java апплет представляет собой небольшое клиентское приложение, которое выполняется на виртуальной машине Java, установленной в веб-браузере. Однако клиентским системам, вероятно, понадобится Java Plug-in и, возможно, файл политики безопасности для успешного выполнения апплета в веб-браузере.

Веб-компоненты являются предпочтительным API для создания программы веб-клиента, поскольку в клиентских системах не требуются Plug-in-ы или файлы политики безопасности. Кроме того, веб-компоненты обеспечивают более чёткое и модульное проектирование приложений, поскольку они позволяют отделить программирование логики приложения от веб-дизайна. Таким образом, персонал, участвующий в разработке веб-страниц, может не понимать синтаксис Java, чтобы выполнять свою работу.

Архитектура компонентов JavaBeans

Слои сервера и клиента могут также включать компоненты на основе архитектуры компонентов JavaBeans (Java-бинов) для управления потоком данных между следующими:

- Клиентские приложения или апплет и компоненты, работающие на сервере Jakarta EE
- Серверные компоненты и база данных

Компоненты JavaBeans не являются компонентами Jakarta EE в соответствии со спецификацией Jakarta EE.

Компоненты JavaBeans имеют свойства и методы `get` и `set` для доступа к этим свойствам. Компоненты JavaBeans, используемые таким образом, обычно просты в разработке и реализации, но должны соответствовать соглашениям об именах и проектировании, изложенным в архитектуре компонентов JavaBeans.

Связь с сервером Jakarta EE

Рисунок 1-2 показывает различные элементы, которые могут составлять клиентский слой. Клиент связывается с уровнем бизнес-логики, работающим на сервере Jakarta EE, напрямую или, как в случае браузерного клиента, через веб-страницы или сервлеты, работающие на веб-уровне.

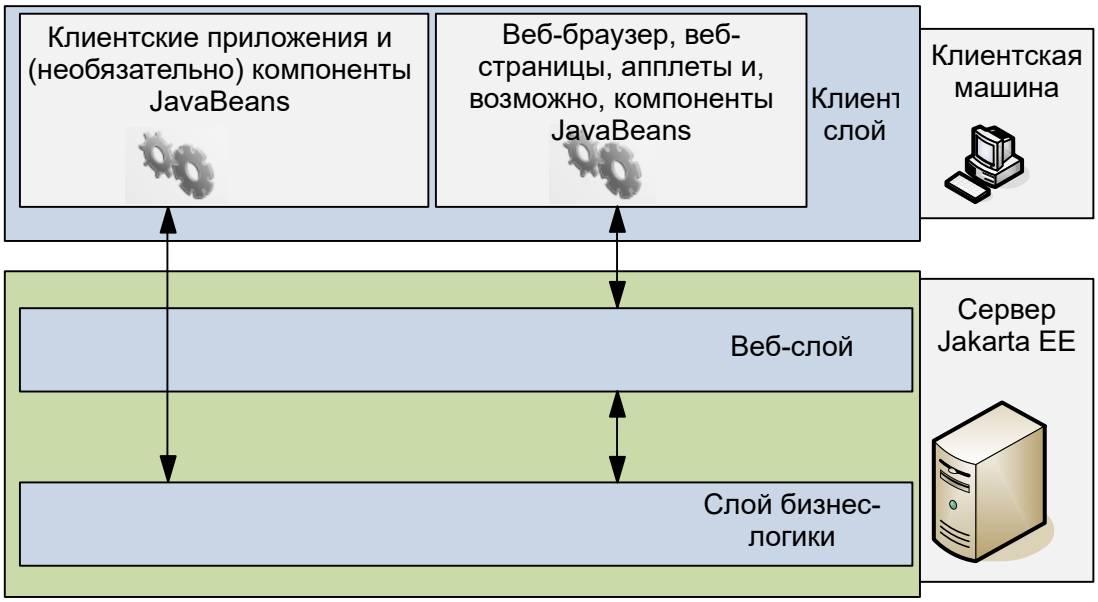

Рисунок 1-2 Связь с сервером

Веб-компоненты

Веб-компоненты Jakarta EE представляют собой сервлеты или веб-страницы, созданные средствами Jakarta Faces и/или Jakarta Server Pages. Сервлеты — это программные классы Java, которые динамически обрабатывают запросы и выдают ответы. Jakarta Server Pages — это текстовые документы, которые выполняются как сервлеты, но допускают использовать более естественный подход к созданию статического содержимого. Технология Jakarta Faces основана на сервлетах и Jakarta Server Pages и предоставляет фреймворк компонентов пользовательского интерфейса для веб-приложений.

Статические HTML-страницы и апплеты связаны с веб-компонентами во время сборки приложения, но не рассматриваются как веб-компоненты в спецификации Jakarta EE. Серверные служебные классы также могут быть связаны с веб-компонентами и, как и HTML-страницы, не считаются веб-компонентами.

Как показано на рис. 1-3, веб-слой, как и клиентский слою, может включать Java-бин для управления пользовательским вводом и отправки этого ввода Enterprise-бинам, работающим в слое бизнес-логики, для обработки.

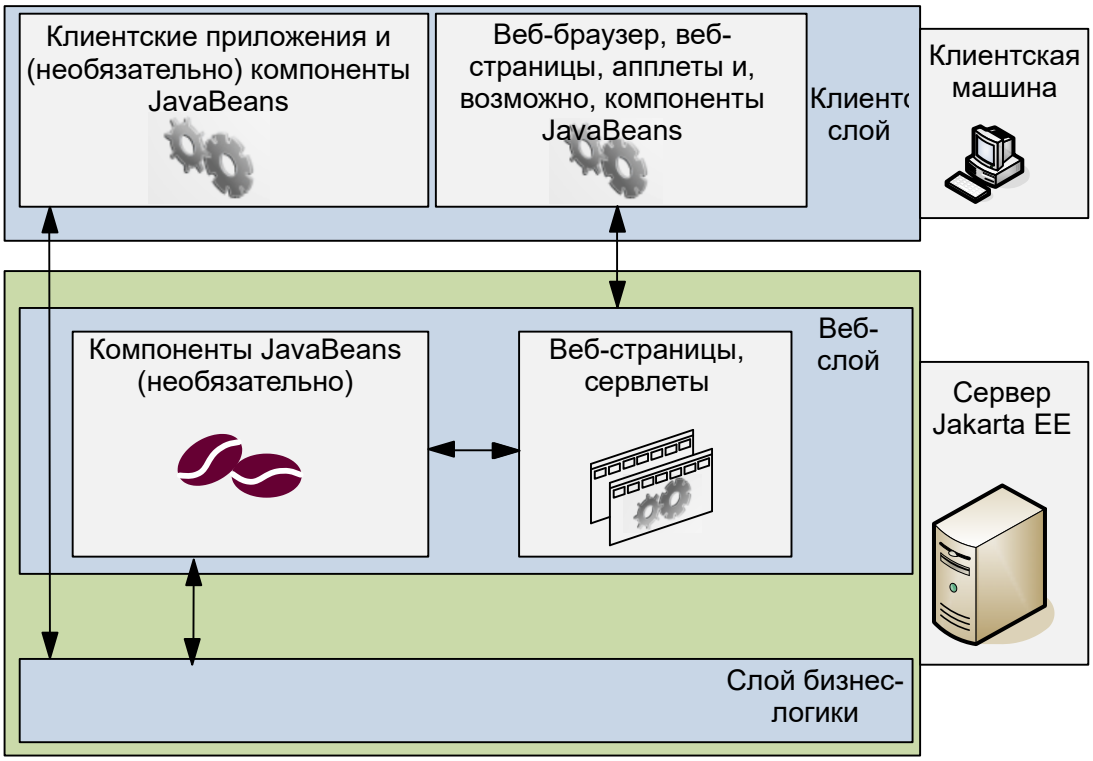

Рисунок 1-3 Веб-уровень и приложения Jakarta EE

Бизнес-компоненты

Бизнес-код, который представляет собой логику, которая решает или удовлетворяет потребности конкретной бизнес-области, такого как банковское дело, розничная торговля или финансы, обрабатывается Enterprise-бинами, работающими либо в слое бизнес-логики, либо в веб-слое. Рисунок 1-4 показывает, как Enterprise-бин получает данные от клиентских программ, обрабатывает их (при необходимости) и отправляет их в базу данных или другую информационную систему предприятия для хранения. Enterprise-бин также извлекает данные из хранилища, обрабатывает их (при необходимости) и отправляет обратно клиентской программе.

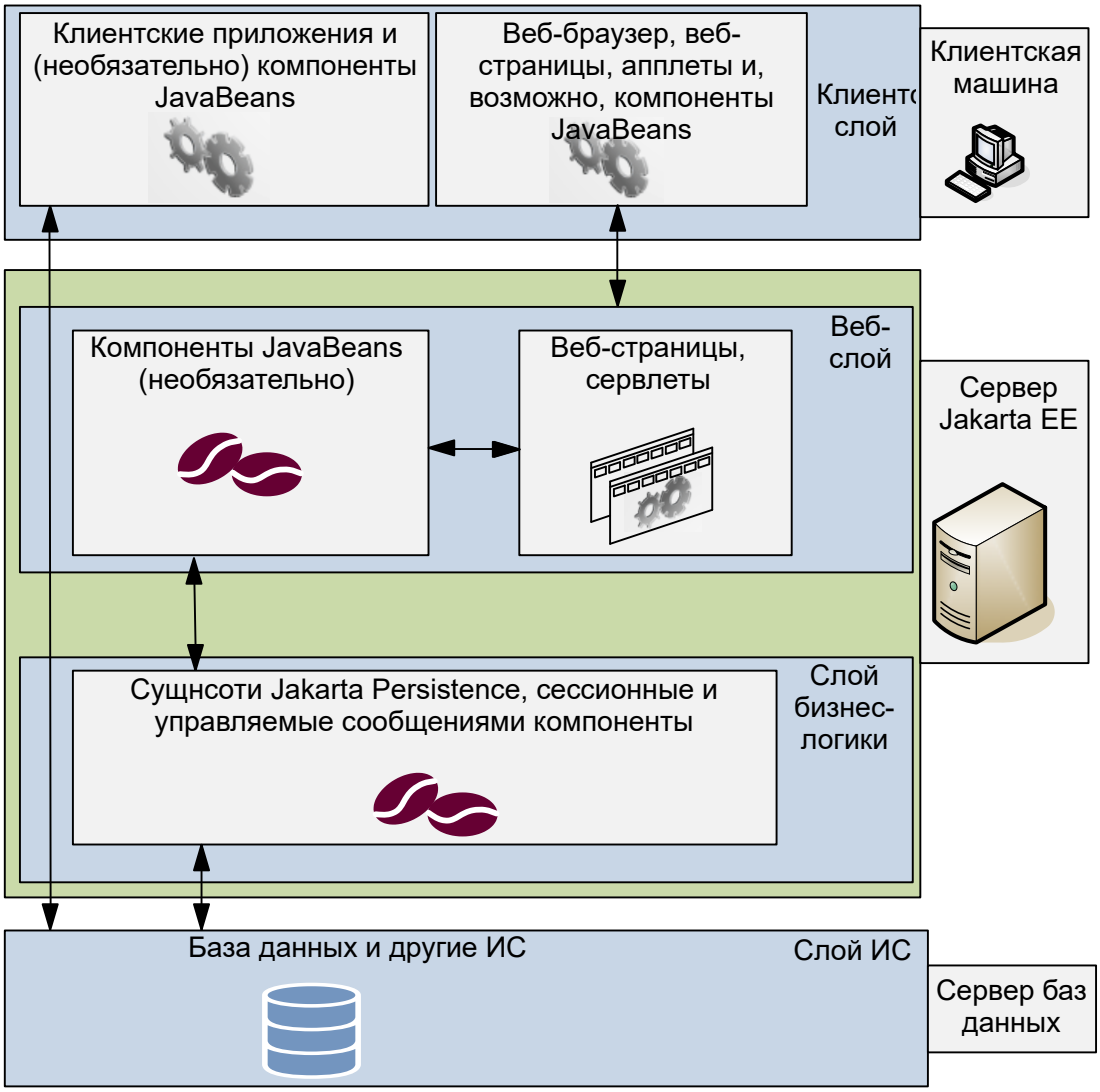

Рисунок 1-4 Слои бизнес-логики и информационных систем (ИС) предприятия

Слой информационных систем предприятия

Слой информационных систем предприятия включает программное обеспечение ИС и системы инфраструктуры предприятия, такие как планирование ресурсов предприятия (ERP), обработка транзакций мэйнфреймов, системы управления базами данных и другие информационные системы. Например, компонентам приложения Jakarta EE может потребоваться доступ к корпоративным информационным системам для подключения к базе данных.

Контейнеры Jakarta EE

Обычно многослойные приложения для тонких клиентов сложно писать, поскольку они содержат много строк сложного кода для управления транзакциями и состояниями, многопоточностью, пулами ресурсов и другими сложными низкоуровневыми деталями. Компонентная и платформо-независимая архитектура Jakarta EE облегчает написание приложений, поскольку бизнес-логика организована в повторно используемые компоненты. Кроме того, сервер Jakarta EE предоставляет базовые сервисы в форме контейнера для каждого типа компонента. Поскольку вам не нужно разрабатывать эти сервисы самостоятельно, вы можете сосредоточиться на решении имеющейся бизнес-задачи.

Сервисы контейнера

Контейнеры — это интерфейс между компонентом и низкоуровневой платформо-зависимой функциональностью, которая поддерживает компонент. Перед выполнением веб-компонент, Enterprise-бин или клиентский компонент приложения должны быть упакованы в модуль Jakarta EE и развёрнуты в его контейнере.

Процесс сборки включает в себя указание настроек контейнера для каждого компонента в приложении Jakarta EE и для самого приложения Jakarta EE. Параметры контейнера настраивают базовую среду, предоставляемую сервером Jakarta EE, включая такие сервисы, как безопасность, управление транзакциями, использованием Java Naming and Directory Interface (JNDI) и удалённое подключение. Вот некоторые из основных моментов.

- Модель безопасности Jakarta EE позволяет настроить веб-компонент или Enterprise-бин так, чтобы к системным ресурсам обращались только авторизованные пользователи.
- Модель транзакций Jakarta EE позволяет определять отношения между методами, составляющими транзакцию, так что все методы в этой транзакции рассматриваются как единое целое.
- Службы поиска JNDI предоставляют унифицированный интерфейс для нескольких сервисов имён и каталогов предприятия, чтобы компоненты приложения могли получить доступ к этим службам.
- Модель удалённого подключения Jakarta EE управляет связью между клиентами и Enterprise-бинами на низком уровне. После создания Enterprise-бина клиент вызывает его методы, как если бы они находились на одной виртуальной машине.

Поскольку архитектура Jakarta EE предоставляет настраиваемые сервисы, компоненты в одном и том же приложении могут вести себя по-разному в зависимости от того, где они развёрнуты. Например, Enterprise-бин может иметь параметры безопасности, которые позволяют ему определённый уровень доступа к данным базы данных в одной производственной среде и другой уровень доступа к базе данных в другой производственной среде.

Контейнер также управляет неконфигурируемыми сервисами, такими как жизненные циклы Enterprise-бинов и сервлетов, пул ресурсов соединени с базой данных, персистентность данных и доступ к API-интерфейсам платформы Jakarta EE (см. API-интерфейсы Jakarta EE.

Типы контейнеров

Процесс развёртывания устанавливает компоненты приложения Jakarta EE в контейнеры Jakarta EE, как показано на рисунке 1-5.

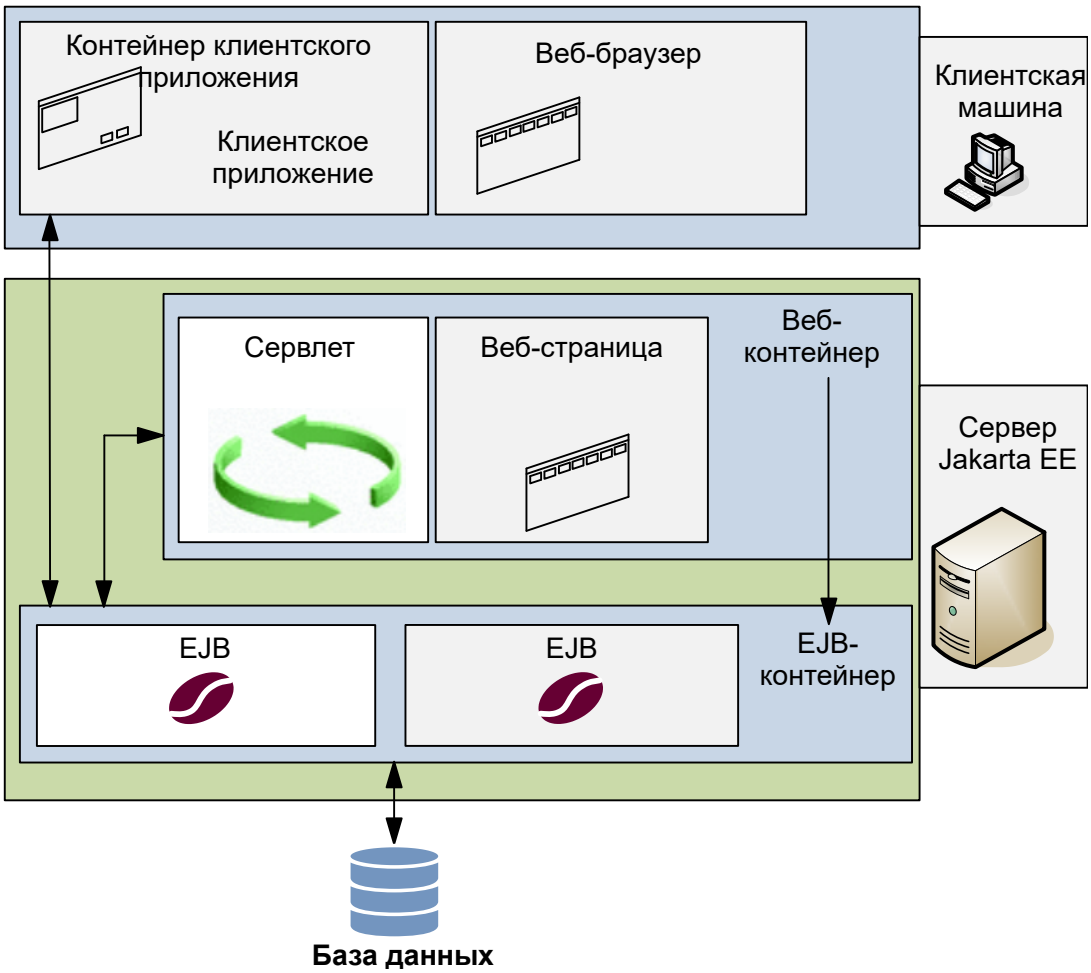

Рисунок 1-5. Сервер и контейнеры Jakarta EE

Сервер и контейнеры:

- Сервер Jakarta EE: часть времени выполнения продукта Jakarta EE. Сервер Jakarta EE предоставляет корпоративные и веб-контейнеры.
- EJB-контейнер Jakarta управляет объектами EJB для приложений Jakarta EE. EJB-компоненты и EJB-контейнер работают на сервере Jakarta EE.
- Веб-контейнер управляет выполнением веб-страниц, сервлетов и некоторых EJB-компонентов для приложений Jakarta EE. Веб-компоненты и их контейнер работают на сервере Jakarta EE.
- Клиентский контейнер приложения: управляет выполнением клиентских компонентов приложения. Клиентские приложения и их контейнер запускаются на клиенте.
- Контейнер апплетов: управляет выполнением апплетов. Состоит из веб-браузера и подключаемого Java Plug-in-a, работающих совместно на клиенте.

Поддержка веб-сервисов

Веб-сервисы — это корпоративные веб-приложения, которые используют открытые стандарты на основе XML и транспортные протоколы для обмена данными с вызывающими их клиентами. Платформа Jakarta EE предоставляет API-интерфейсы XML и инструменты, необходимые для быстрого проектирования, разработки, тестирования и развёртывания веб-сервисов и клиентов, которые полностью совместимы с другими веб-сервисами и клиентами, работающими на платформах, как основанных на Java, так и основанных на других языках программированиям.

Для написания веб-сервисов и клиентов с API Jakarta EE XML всё, что нужно сделать, это передать данные в параметрах вызова методов и обработать данные, полученные в ответе. Для документоориентированных веб-сервисов документы с данными отправляются сервису принимается документ с данными от сервиса. Нет необходимости в низкоуровневом программировании, потому что реализации XML API выполняют работу по преобразованию данных приложения в поток данных на основе XML, который отправляется через стандартизированные транспортные протоколы на основе XML. Эти основанные на XML стандарты и протоколы представлены в следующих разделах.

Перевод данных в стандартизированный поток на основе XML — именно это делает веб-сервисы и клиенты, написанные с API Jakarta EE XML, полностью совместимыми. Это не обязательно означает, что транспортируемые данные включают в себя теги XML, поскольку транспортируемые данные сами по себе могут быть простым текстом, данными XML или любыми бинарными данными, такими как аудио, видео, карты, файлы программ, файлы компьютерного проектирования (CAD), документы и тому подобное. В следующем разделе представлен XML и объясняется, как бизнес-партнёры могут использовать теги и схемы XML для эффективного обмена данными.

XML

Extensible Markup Language (XML) — это кроссплатформенный, расширяемый текстовый стандарт для представления данных. Стороны, обменивающиеся данными XML, могут создавать свои собственные теги для описания данных, настраивать схемы для указания, какие теги могут использоваться в конкретном виде XML-документа, и использовать таблицы стилей XML для управления отображением и обработкой данных.

Например, веб-сервис может использовать XML и схему для создания прайс-листов, а компании, получающие прайс-листы и схемы, могут иметь свои собственные таблицы стилей для обработки данных таким образом, который больше соответствует их потребностям. Вот примеры.

- Одна компания может разместить информацию о ценах в XML и через программу перевода XML в HTML публиковать прайс-листы в своей внутренней сети.
- Партнёрская компания может передавать информацию о ценах в XML и через утилиту создавать маркетинговые презентации.
- Другая компания может считывать информацию о ценах в XML в приложение для обработки.

Транспортный протокол SOAP

Клиентские запросы и ответы веб-сервисов передаются в виде сообщений SOAP (Simple Object Access Protocol) по HTTP для обеспечения полностью совместимого обмена между клиентами и веб-слервисами, работающими на разных платформах и в разных местах в Интернете. HTTP является общепринятым стандартом запросов и ответов при обмене текстовыми сообщениями через Интернет, а SOAP — это протокол на основе XML, соответствующий модели запросов и ответов HTTP.

SOAP-часть транспортируемого сообщения выполняет следующие действия:

- Определяет конверт (на основе XML) для описания содержимого сообщения и инструкций к его обработке
- Включает правила кодирования (на основе XML) типов данных сообщения, специфичных для приложения
- Определяет соглашение (на основе XML) об отправке запроса удалённому сервису и получении от него ответа

Стандарт формата WSDL

Язык описания веб-сервисов (WSDL) — это стандартизированный формат XML для описания сетевых сервисов. Описание включает в себя название сервиса, местоположение сервиса и способы связи с сервисом. Описания сервисов в виде WSDL можно публиковать в Интернете. Eclipse GlassFish Server предоставляет инструмент для создания спецификации WSDL веб-сервиса, которая использует удалённые вызовы процедур для связи с клиентами.

Сборка и развёртывание приложений Jakarta EE

Приложение Jakarta EE упаковывается в один или несколько стандартных модулей для развёртывания в любой системе, совместимой с платформой Jakarta EE. Каждый модуль содержит

- Функциональный компонент или компоненты, такие как Enterprise-бин, веб-страница, сервлет или апплет
- Необязательный дескриптор развёртывания, который описывает его содержимое

Модуль Jakarta EE готов к развёртыванию сразу после сборки. Развёртывание обычно включает использование утилиты развёртывания на целевой платформе с указанием специфичной информации, такой как список конкретных пользователей, имеющих к ней доступ, и имя конкретной базы данных. Сразу после развёртывания на целевой платформе приложение готово к запуску.

API Jakarta EE

На рис. 1-6 показаны взаимосвязи между контейнерами Jakarta EE.

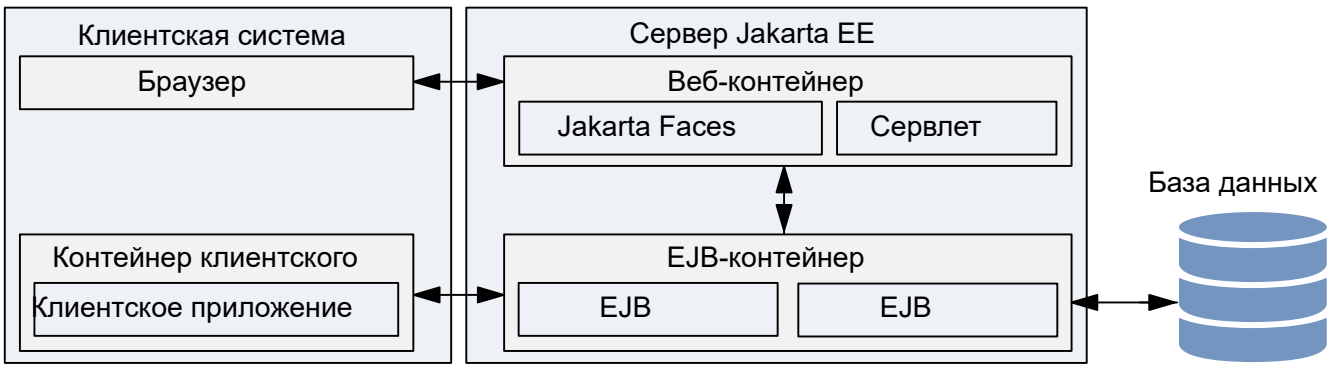

Рис. 1-6 Контейнеры Jakarta EE

На рис. 1-7 показана доступность API Jakarta EE в веб-контейнере.

Рисунок 1-7 API Jakarta EE в веб-контейнере

На рис. 1-8 показана доступность API Jakarta EE в EJB-контейнере.

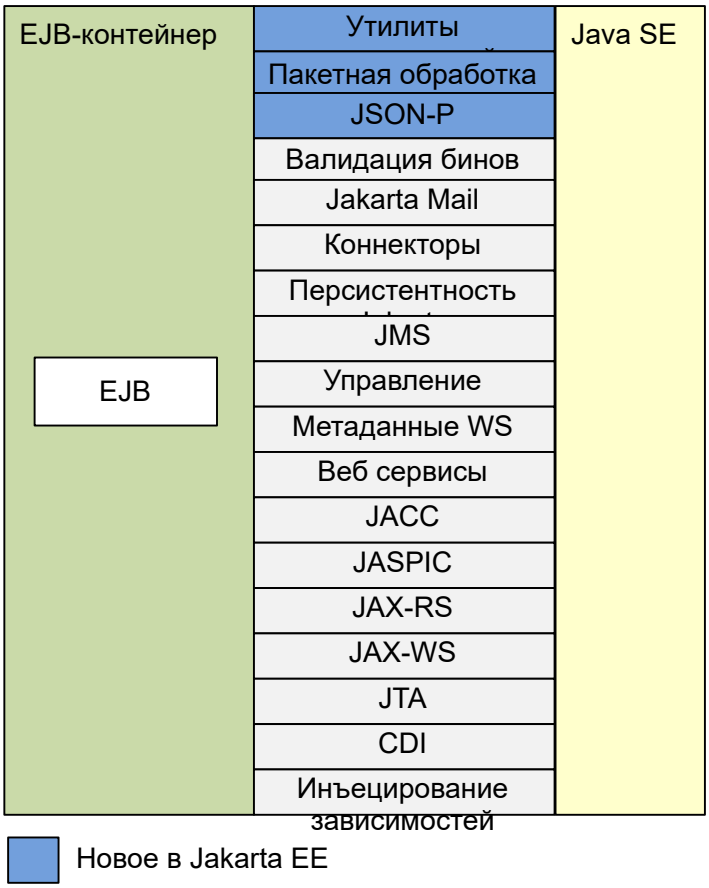

Рисунок 1-8 API Jakarta EE в EJB-контейнере

На рис. 1-9 показана доступность API Jakarta EE в контейнере клиентских приложений.

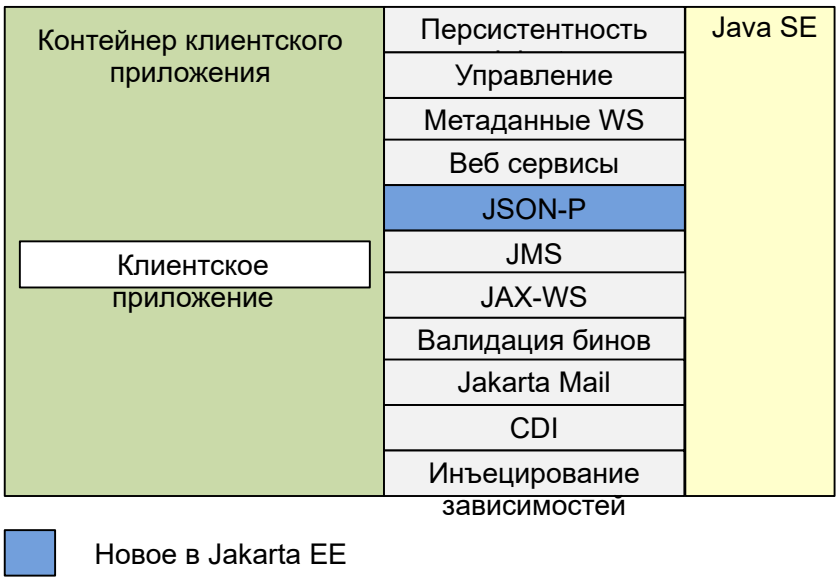

Рисунок 1-9 API Jakarta EE в контейнере клиентских приложений

В следующих разделах даётся краткое описание технологий, требуемых платформой Jakarta EE, и API, используемых в приложениях Jakarta EE.

Технология Jakarta Enterprise Beans

Компонент EJB — это совокупность кода, имеющего поля и методы для реализации модулей бизнес-логики. Вы можете думать об Enterprise-бине как о строительном блоке, который можно использовать отдельно или совместно с другими Enterprise-бинами для выполнения бизнес-логики на сервере Jakarta EE.

Понятие Enterprise-бинов включает в себя сессионные компоненты и компоненты, управляемые сообщениями.

- Сессионный компонент обслуживает вре́менное взаимодействие с клиентом. Когда клиент завершает выполнение, сессионный компонент завершает работу и его данные исчезают.
- Компонент, управляемый сообщениями, сочетает в себе функции сессионного компонента и приёмника сообщений, что позволяет бизнес-компоненту получать сообщения асинхронно. Обычно это сообщения Jakarta Messaging.

Платформа Jakarta EE 9 требует Jakarta Enterprise Beans 4.0 и Jakarta Interceptors 2.0.

Технология Jakarta Servlet

Технология Jakarta Servlet позволяет создавать специфичные для HTTP классы сервлетов. Класс сервлета расширяет возможности серверов с размещёнными на них приложениями, доступ к которым описывается моделью запрос-ответ. Хотя сервлеты могут отвечать на любые запросы, они обычно используются для расширения приложений, размещаемых на веб-серверах.

Новые функции Jakarta Servlet в платформе Jakarta EE 8:

- Серверный Push
- HTTP Trailer

Для платформы Jakarta EE 9 требуется Servlet 5.0.

Jakarta Faces

Jakarta Faces — это набор компонентов пользовательского интерфейса для создания веб-приложений. Основными компонентами Jakarta Faces являются:

- Набор компонентов GUI.
- Гибкая модель для отрисовки компонентов в различных видах HTML, на различных языках и технологиях разметки. Объект `Renderer` генерирует разметку для отрисовки компонента и конвертирует данные из объектной модели к виду, удобному для отображения.
- Стандартный `RenderKit` для генерации разметки HTML 4.01.

Следующие функции поддерживают компоненты GUI:

- Валидация входных данных
- Обработка событий
- Преобразование данных между объектной моделью и компонентами
- Создание управляемой объектной модели
- Конфигурация навигации по страницам
- Язык выражений Jakarta

Все эти функции доступны в стандартных API Java и XML-файлах конфигурации.

Новые функции Jakarta Faces в платформе Jakarta EE 8:

- Прямая поддержка веб-сокетов с новым тегом `<f:websocket>`
- Валидация бина на уровне класса с новым тегом `<f:validateWholeBean>`
- Аннотации `@ManagedProperty`, совместимые с инъецированием контекстов и зависимостей Jakarta
- Усовершенствованный набор выражений (EL)

Платформа Jakarta EE 9 требует Jakarta Faces 3.0 и Jakarta Expression Language 4.0.

Технология Jakarta Server Pages

Технология Jakarta Server Pages позволяет вставлять фрагменты кода сервлета непосредственно в текстовый документ. Страница Jakarta Server Pages — это текстовый документ, содержащий текст двух типов:

- Статические данные, которые могут быть выражены в любом текстовом формате, например HTML или XML
- Элементы JSP, определяющие, как страница генерирует динамический контент

Технология Jakarta Server Pages основана на технологии JavaServer Pages (JSP) и совместима с ней.

Для получения информации о JSP см. *Учебник Java EE 5* по ссылке https://docs.oracle.com/javaee/5/tutorial/doc/.

Платформа Jakarta EE 9 требует Jakarta Server Pages 3.0 для совместимости с более ранними версиями, но рекомендует использовать Facelets в качестве технологии отображения в новых приложениях.

Стандартная библиотека тегов Jakarta

Стандартная библиотека тегов Jakarta включает в себя основные функции, общие для большинства приложений Jakarta Server Pages. Вместо того чтобы смешивать теги множества поставщиков в приложениях Jakarta Server Pages, вы можете использовать единый стандартный набор тегов. Такая стандартизация позволяет развёртывать приложения в любом контейнере Jakarta Server Pages, поддерживающем стандартную библиотеку тегов Jakarta, и повышает вероятность того, что реализация тегов будет оптимальной.

В стандартной библиотеке тегов Jakarta есть теги итератора и условные теги для управления потоком, теги для управления документами XML, теги интернационализации, теги для доступа к базам данных с помощью SQL и теги для часто используемых функций.

Платформа Jakarta EE 9 требует Jakarta Standard Tag Library 2.0.

Персистентность Jakarta

Jakarta Persistence — это основанное на стандартах Java решение для обеспечения персистентности. Персистентность использует объектно-реляционный подход отображения для преодоления разрыва между объектно-ориентированной моделью и реляционной базой данных. Персистентность Jakarta также может использоваться в приложениях Java SE за пределами среды Jakarta EE. Персистентность в Jakarta состоит из следующих областей:

- Персистентность Jakarta
- Язык запросов
- Метаданные объектно-реляционного отображения

Платформа Jakarta EE 9 требует Jakarta Persistence 3.0.

Транзакции Jakarta

Транзакции Jakarta предоставляют стандартный интерфейс для разграничения транзакций. Архитектура Jakarta EE по умолчанию обеспечивает автоматическую фиксацию и откат транзакций. Автоматическая фиксация предполагает, что любые другие приложения, просматривающие данные, будут видеть обновлённые данные после каждой операции чтения или записи в базу данных. Однако, если ваше приложение выполняет две отдельные операции доступа к базе данных, которые зависят друг от друга, вы можете захотеть использовать транзакции Jakarta, чтобы указать, что обе операции входят в состав одной транзакции, то есть выполняются атомарно.

Платформа Jakarta EE 9 требует Jakarta Transactions 2.0

Jakarta RESTful Web Service

Jakarta RESTful Web Service определяет API-интерфейсы для разработки веб-сервисов, построенных в соответствии со стилем REST. RESTful приложение Jakarta — это веб-приложение, которое состоит из классов, упакованных в виде сервлетов в файл WAR вместе с необходимыми библиотеками.

Новые функции RESTful веб-сервисов платформы Jakarta EE 8:

- Реактивный клиентский API
  Когда получены результаты вызова целевого ресурса, усовершенствования в соответствующих API в Java SE позволяют создавать очередь из этих результатов, приоритезировать и комбинировать их, а также обрабатывать возникающие исключительные ситуации.
- Улучшения в поддержке серверных событий
  Клиенты могут подписаться на уведомления о серверных событиях, используя долгоживущее соединение. Добавлена поддержка нового тип MIME — text/event-stream.
- Поддержка Jakarta JSON и улучшенная интеграция с технологиями инъецирования контекстов и зависимостей Jakarta, Jakarta Servlet и Jakarta Bean Validation.

Платформа Jakarta EE 9 требует Jakarta RESTful Web Services 3.0.

Managed-бины Jakarta

Managed-бины Jakarta — легковесные объекты, управляемые контейнером (POJO) с минимальными требованиями, поддерживают небольшой набор базовых служб, таких как инъецирование ресурсов, Callback-вызовы жизненного цикла и Interceptor-ы. Managed-бины представляют собой обобщение Managed-бинов, описанных в Jakarta Faces, и могут использоваться в любом месте приложения Jakarta EE, а не только в веб-модулях.

Спецификация Jakarta Managed Beans является частью спецификации платформы Jakarta EE 9. Платформа Jakarta EE 9 требует Jakarta Managed Beans 2.0.

Инъецирование контекстов и зависимостей Jakarta

Jakarta Contexts and Dependency Injection (CDI) определяет набор контекстных сервисов, предоставляемых контейнерами Jakarta EE, позволяющим разработчикам легко использовать Enterprise-бины вместе с Jakarta Faces в веб-приложениях. Спроектированный для использования с объектами, сохраняющими состояние, CDI имеет также и более широкое применение, что даёт разработчикам большую гибкость при интегрировании различных видов компонентов слабосвязанными, но типобезопасными (typesafe) способами.

Новые функции CDI в платформе Jakarta EE 8:

- API для начальной загрузки контейнера CDI в Java SE 8

- Поддержка порядка, в котором вызываются методы наблюдателя для определённого события, а также поддержка асинхронного запуска событий
- Интерфейсы конфигураторов, которые используются для динамического назначения и изменения CDI
- Встроенные литералы аннотаций, удобная функция для создания аннотаций и многое другое

Платформа Jakarta EE 9 требует Jakarta Contexts и Dependency Injection 3.0.

Инъекция зависимостей Jakarta

Инъецирование зависимостей Jakarta определяет стандартный набор аннотаций (и один интерфейс) для использования в инъецируемых классах.

В платформе Jakarta EE CDI обеспечивает поддержку инъецирования зависимостей. Вообще говоря, точки инъецирования могут использоваться только в приложении с поддержкой CDI.

Платформа Jakarta EE 9 требует Jakarta Dependency Injection 2.0.

Jakarta Bean Validation

Спецификация Jakarta Bean Validation определяет модель метаданных и API для валидации данных в компонентах JavaBeans. Вместо того, чтобы распределять валидацию данных по нескольким слоям, таким как браузер и серверная сторона, можно задать ограничения валидации в одном месте и использовать эти ограничения валидации несколькими слоями совместно.

Новые функции Jakarta Bean Validation в платформе Jakarta EE 8 включают следующее:

- Поддержка новых функций в Java SE 8, таких как API даты и времени
- Добавление новых встроенных ограничений Jakarta Bean Validation.

Платформа Jakarta EE 9 требует Jakarta Bean Validation 3.0.

Jakarta Messaging

Jakarta Messaging — это стандарт обмена сообщениями, который позволяет компонентам приложения Jakarta EE создавать, отправлять, получать и читать сообщения. Он обеспечивает распределённое, слабосвязное, надёжное и асинхронное взаимодействие.

Платформа Jakarta EE 9 требует Jakarta Messaging 3.0.

Коннекторы Jakarta

Коннекторы Jakarta используются провайдерами инструментальных средств и системными интеграторами для создания адаптеров ресурсов, поддерживающих доступ к корпоративным информационным системам, которые могут использоваться в любом приложении Jakarta EE. Адаптер ресурсов — это программный компонент, который позволяет компонентам приложения Jakarta EE получать доступ и взаимодействовать с базовым менеджером ресурсов ИС. Поскольку адаптер ресурса специфичен для его менеджера ресурсов, для каждого типа базы данных или информационной системы предприятия обычно существует отдельный адаптер ресурсов.

Коннекторы Jakarta также обеспечивают ориентированную на производительность, безопасную, масштабируемую и основанную на сообщениях транзакционную интеграцию веб-сервисов на платформе Jakarta EE с существующими ИС, которые могут быть синхронными или асинхронными. Существующие приложения и ИС, интегрированные через коннекторы Jakarta в платформу Jakarta EE, могут быть

представлены как веб-сервисы на основе XML с помощью моделей компонентов Jakarta XML Web Services и Jakarta EE. Таким образом, Jakarta XML Web Services и Jakarta Connectors являются дополнительными технологиями для интеграции корпоративных приложений (EAI) и сквозной бизнес-интеграции.

Платформа Jakarta EE 9 требует Jakarta Connectors 2.0.

Jakarta Mail

Приложения Jakarta EE используют Jakarta Mail для отправки уведомлений по электронной почте. Jakarta Mail состоит из двух частей:

- Интерфейс уровня приложения, используемый компонентами приложения для отправки почты
- Интерфейс поставщика услуг

Платформа Jakarta EE включает Jakarta Mail с поставщиком услуг, который позволяет компонентам приложений отправлять электронную почту в Интернете.

Платформа Jakarta EE 9 требует Jakarta Mail 2.0.

Jakarta Authorization

Спецификация Jakarta Authorization определяет контракт между сервером приложений Jakarta EE и поставщиком политики авторизации. Все контейнеры Jakarta EE поддерживают этот контракт.

Спецификация Jakarta Authorization определяет классы `java.security.Permission`, которые удовлетворяют модели авторизации Jakarta EE. Спецификация определяет привязку решений о предоставлении контейнером доступа к операциям с объектами классов этих разрешений. Он задаёт семантику провайдеров политик, которые используют новые классы разрешений для удовлетворения требований авторизации платформы Jakarta EE, включая определение и использование ролей.

Платформа Jakarta EE 9 требует Jakarta Authorization 2.0.

Jakarta Authentication

Спецификация Jakarta Authentication определяет интерфейс поставщика услуг (SPI), с помощью которого поставщики аутентификации, реализующие механизмы аутентификации сообщений, могут быть интегрированы в контейнеры или среду выполнения для обработки сообщений клиента или сервера. Поставщики аутентификации, интегрированные через этот интерфейс, работают с сетевыми сообщениями, которые им передают вызывающие их контейнеры. Поставщики аутентификации преобразуют исходящие сообщения, так что источник каждого сообщения может быть аутентифицирован получающим контейнером, а получатель сообщения может быть аутентифицирован отправителем сообщения. Поставщики аутентификации аутентифицируют каждое входящее сообщение и возвращают вызывающим их контейнерам результат идентификации, установленный в результате аутентификации сообщения.

Платформа Jakarta EE 9 требует Jakarta Authentication 2.0.

Jakarta Security

Спецификация Jakarta Security определяет переносимые подключаемые интерфейсы для HTTP-аутентификации и хранилищ идентификаторов, а также инъецируемый `SecurityContext`, который предоставляет API для программной безопасности.

Реализации интерфейса `HttpAuthenticationMechanism` могут использоваться для аутентификации вызывающих веб-приложение субъектов. Приложение может предоставить свой собственный `HttpAuthenticationMechanism` или использовать одну из реализаций по умолчанию, предоставляемых

контейнером.

Реализации интерфейса `IdentityStore` можно использовать для проверки учётных данных пользователя и получения информации о группе. Приложение может предоставить собственный `IdentityStore` или использовать встроенное хранилище LDAP или базы данных.

API `HttpAuthenticationMechanism` и `IdentityStore` предоставляют то преимущество перед реализациями, предоставленными контейнером, что они позволяют приложению контролировать процесс аутентификации и хранилища идентификаторов, используемые для аутентификации, стандартным переносимым способом.

API `SecurityContext` предназначен для использования кодом приложения для взаимодействия с текущим контекстом безопасности. В спецификации также предусмотрено назначение группы ролям по умолчанию и определён основной тип с именем `CallerPrincipal` , который может представлять результат идентификации вызывающего приложение субъекта.

Платформа Jakarta EE 9 требует Jakarta Security 2.0.

Jakarta WebSocket

Веб-сокеты — это прикладной протокол, который обеспечивает полнодуплексный канал связи между двумя узлами по TCP. Jakarta WebSocket позволяет приложениям Jakarta EE создавать конечные точки с помощью аннотаций, которые определяют настройки конечной точки и обозначают Callback-методы её жизненного цикла.

Платформа Jakarta EE 9 требует Jakarta WebSocket 2.0.

Обработка JSON

JavaScript Object Notation (JSON) — это текстовый формат обмена данными, пришедший из JavaScript и использующийся в веб-сервисах и других связанных приложениях. Jakarta JSON Processing позволяет приложениям Jakarta EE анализ, преобразование и выборку данных из JSON-документа, используя объектную или потоковую модель.

Новые функции Jakarta JSON Processing платформы Jakarta EE 8 включают следующее:

- Указатель JSON
  Задают строковый синтаксис для ссылки на определённое значение в документе JSON. JSON Pointer включает в себя API для извлечения значений из целевого документа и преобразования их при создании нового документа JSON.
- JSON Patch
  Определяет формат для выражения последовательности операций, которые будут применены к документу JSON.
- JSON Merge Patch
  Определяет формат и правила обработки для применения операций к документу JSON основываясь на специфике содержимого целевого документа.
- В базовый набор функций JSON по обработке документов добавлены новые функции редактирования и преобразования.
- Вспомогательные классы и методы, называемые JSON Collectors, которые используют возможности Stream API, представленный в Java SE 8.

Платформа Jakarta EE 9 требует Jakarta JSON Processing 2.0.

Jakarta JSON Binding

Jakarta JSON Binding обеспечивает связующий слой преобразования объектов Java в сообщения JSON и обратно. Jakarta JSON Binding также поддерживает возможность настройки процесса сопоставления по умолчанию, используемого в связующем слое, путём использования аннотаций Java для заданного поля, свойства JavaBean, типа или пакета или путём предоставления реализации стратегии именования свойств.

Платформа Jakarta EE 9 требует Jakarta JSON Binding 2.0.

Параллелизм Jakarta

Jakarta Concurrency — это стандартный API-интерфейс для предоставления асинхронных возможностей компонентам приложения Jakarta EE через следующие типы объектов: managed executor service, managed scheduled executor service, managed thread factory и сервис контекста.

Платформа Jakarta EE 9 требует Jakarta Concurrency 2.0.

Jakarta Batch

Пакетные задачи — это задачи, которые могут быть выполнены без взаимодействия с пользователем. Спецификация Batch Applications for the Java Platform — это фреймворк, который обеспечивает поддержку для создания и запуска пакетных заданий в приложениях Java. Фреймворк пакетной обработки состоит из пакетной среды выполнения, языка спецификации задания на основе XML, API Java для взаимодействия с пакетной средой выполнения и API Java для реализации пакетных артефактов.

Платформа Jakarta EE 9 требует Jakarta Batch 2.0.

Jakarta Activation

Jakarta Activation используется Jakarta Mail. Jakarta Activation предоставляет стандартные службы для определения типа произвольного фрагмента данных, инкапсуляции доступа к нему, обнаружения доступных операций и создания соответствующего компонента JavaBeans для выполнения этих операций.

Платформа Jakarta EE 9 требует Jakarta Activation 2.0.

Jakarta XML Binding

Jakarta XML Binding обеспечивает удобный способ связывания XML-схемы с объектными данными Java. XML Binding может использоваться как отдельно, так и совместно с Jakarta XML Web Services, и в этом случае она обеспечивает стандартное связывание данных для сообщений веб-сервисов. Все клиентские контейнеры Jakarta EE, веб-контейнеры и EJB-контейнеры Jakarta поддерживают XML Binding API.

Платформа Jakarta EE 9 требует Jakarta XML Binding 3.0.

Jakarta XML Web Services

Спецификация Jakarta XML Web Services обеспечивает поддержку веб-сервисов, использующих API Jakarta XML Binding для связывания данных XML с объектами Java. Спецификация Jakarta XML Web Services определяет клиентские API для доступа к веб-сервисам, а также методы для реализации конечных точек веб-сервисов. Enterprise Web Services описывает развертывание служб и клиентов на основе веб-служб Jakarta XML. Спецификации Jakarta Enterprise Beans и Jakarta Servlet также описывают аспекты такого развёртывания. Приложения на основе XML веб-сервисов Jakarta могут быть развёрнуты с использованием любой из этих моделей развёртывания.

Спецификация Jakarta XML Web Services описывает поддержку обработчиков сообщений, которые могут обрабатывать запросы и ответы сообщений. Как правило, эти обработчики сообщений выполняются в том же контейнере и с теми же привилегиями и контекстом выполнения, что и клиент или компонент конечной

точки XML веб-сервиса Jakarta, с которым они связаны. Эти обработчики сообщений имеют доступ к тому же пространству имён JNDI, что и связанный с ними компонент. Пользовательские сериализаторы и десериализаторы, если они поддерживаются, обрабатываются так же, как и обработчики сообщений.

Платформа Jakarta EE 9 требует Jakarta XML Web Services 3.0.

Jakarta SOAP with Attachments

Jakarta SOAP with Attachments — это низкоуровневый API, от которого зависит Jakarta XML Web Services. Jakarta SOAP with Attachments позволяет создавать и использовать сообщения, соответствующие спецификациям SOAP 1.1 и 1.2 и примечанию Jakarta SOAP with Attachments. Большинство разработчиков используют не Jakarta SOAP with Attachments, а API-интерфейсы XML веб-сервисов Jakarta более высокого уровня.

Аннотации Jakarta

Аннотации обеспечивают декларативный стиль программирования в платформе Java.

Платформа Jakarta EE 9 требует Jakarta Annotations 2.0.

API Jakarta EE 9 в Java Platform, Standard Edition 8

Несколько API, которых требует платформа Jakarta EE 9, уже включены в Java Platform, Standard Edition 8 (Java SE 8) и доступны для приложений Jakarta EE.

API Java подключения к базам данных

API Java Database Connectivity (JDBC) позволяет вызывать команды SQL из методов Java. Если сессионному компоненту требуется доступ к базе данных, он может использовать API JDBC для работы с ней. Также есть возможность использовать JDBC API из сервлета или страницы JSP для прямого доступа к базе данных, не обращаясь к Enterprise-бину.

JDBC API состоит из двух частей:

- Интерфейс уровня приложения, используемый компонентами приложения для доступа к базе данных
- Интерфейс поставщика услуг для подключения драйвера JDBC к платформе Jakarta EE

Платформа Jakarta EE 9 требует JDBC 4.1.

API имён и каталогов Java

API Java Naming and Directory Interface (JNDI) предоставляет функции имён и каталогов, позволяя приложениям обращаться к нескольким сервисам имён и каталогов, таким как LDAP, DNS и NIS. JNDI API предоставляет приложениям методы для выполнения стандартных операций с каталогами, таких как связывание атрибутов с объектами и поиск объектов с использованием их атрибутов. Используя JNDI, приложение Jakarta EE может хранить и извлекать любой тип именованного объекта Java, позволяя приложениям Jakarta EE сосуществовать со другими приложениями и информационными системами.

Сервисы имён Jakarta EE предоставляют клиентским приложениям, Enterprise-бинам и веб-компонентам доступ к среде именования JNDI. Среда именования позволяет настраивать компонент без необходимости доступа или изменения исходного кода компонента. Контейнер реализует среду именования JNDI и предоставляет её компоненту.

Среда именования предоставляет четыре логических пространства имён: `java:comp`, `java:module`, `java:app` и `java:global` для объектов, доступных компонентам, модулям или приложениям и совместно используется всеми развёрнутыми приложениями. Компонент Jakarta EE может обращаться к именованным системным и пользовательским объектам. Имена некоторых предоставляемых системой объектов, таких как объект JDBC

`DataSource` по умолчанию, фабрика соединений по умолчанию для обмена сообщениями и объект `UserTransaction`, хранятся в пространстве имён `java:comp`. Платформа Jakarta EE позволяет компоненту именовать определённые пользователем объекты, такие как Enterprise-бины, записи среды, объекты JDBC `DataSource` и пункты назначения сообщений.

Компонент Jakarta EE также может найти свой контекст именования среды с помощью интерфейсов JNDI. Компонент может создать объект `javax.naming.InitialContext` и найти контекст именования среды в `InitialContext` под именем `java:comp/env`. Среда именования компонента хранится непосредственно в контексте именования среды или в любом из его прямых или косвенных подконтекстов.

API Java для обработки XML

Java API для обработки XML (JAXP), часть платформы Java SE, поддерживает обработку документов XML с использованием объектной модели документов (DOM), простого API для XML (SAX) и преобразований языка расширяемой таблицы стилей (XSLT). JAXP позволяет приложениям анализировать и преобразовывать XML-документы независимо от конкретной реализации XML-обработки.

JAXP также предоставляет поддержку пространства имён, которая позволяет работать со схемами, отсутствие которых могло бы привести к конфликту имён. Гибкость JAXP позволяет использовать любой XML-совместимый синтаксический анализатор или XSL-процессор из приложения и поддерживает схему Worldwide Web Consortium (W3C). Вы можете найти информацию о схеме W3C на https://www.w3.org/XML/Schema.

Сервис аутентификации и авторизации Java

Служба аутентификации и авторизации Java (JAAS) предоставляет приложению Jakarta EE способ аутентификации и авторизации конкретного пользователя или группы пользователей для его запуска.

JAAS является инструментом подключаемого модуля аутентификации (Pluggable Authentication Module — PAM) Java, который расширяет архитектуру безопасности платформы Java для поддержки авторизации пользователя.

Утилиты Eclipse GlassFish Server

Eclipse GlassFish Server — это реализация платформы Jakarta EE. Помимо поддержки всех API-интерфейсов, описанных в предыдущих разделах, Eclipse GlassFish Server включает в себя ряд утилит Jakarta EE, которые не являются частью платформы Jakarta EE, но могут быть полезны разработчику.

В этом разделе кратко описаны утилиты, входящие в состав Eclipse GlassFish Server. Инструкции по запуску и остановке Eclipse GlassFish Server, запуску Консоли администрирования, а также запуску и остановке Apache Derby приведены в главе 2 *Использование учебных примеров*.

В таблице 1-1 перечислены утилиты, входящие в состав Eclipse GlassFish Server. Основная информация об использовании многих инструментов представлена в учебнике. Для получения подробной информации см. онлайн справку по инструментам графического интерфейса.

Таблица 1-1 Утилиты Eclipse GlassFish Server

Инструмент	**Описание**
Консоль администрирования	Утилита администрирования Eclipse GlassFish Server с графическим веб-интерфейсом. Используется для остановки Eclipse GlassFish Server и управления пользователями, ресурсами и приложениями.

Инструмент	Описание
`asadmin`	Утилита администрирования Eclipse GlassFish Server из командной строки. Используется для запуска и остановки Eclipse GlassFish Server, а также для управления пользователями, ресурсами и приложениями.
`appclient`	Утилита командной строки, которое запускает контейнер клиентского приложения и вызывает клиентское приложение, упакованное в файл JAR.
`capture-schema`	Утилита командной строки для извлечения информации о схеме базы данных и создания файла схемы, который Eclipse GlassFish Server может использовать для управляемой контейнером персистентности.
`package-appclient`	Утилита командной строки для упаковки библиотек клиентского приложения и файлов JAR.
Apache Derby	Копия базы данных Apache Derby.
`xjc`	Утилита командной строки для преобразования или связывания исходной схемы XML с набором классов JAXB на Java.
`schemagen`	Утилита командной строки для создания файла схемы для каждого пространства имён, указанного в ваших классах Java.
`wsimport`	Утилита командной строки для создания переносимых артефактов XML веб-сервисов Jakarta для заданного файла WSDL. После генерации эти артефакты, как и WSDL, документы схемы и реализация конечной точки могут быть упакованы в WAR-файл а затем развёрнуты.
`wsgen`	Утилита командной строки для чтения класса конечной точки веб-сервиса и создания всех необходимых для развёртывания и вызова переносимых артефактов XML веб-сервиса Jakarta.

Глава 2. Использование примеров учебника

В этой главе рассказывается обо всём, что необходимо знать, чтобы установить, собрать и запустить учебные примеры.

Для дополнительных примеров см. https://github.com/eclipse-ee4j/glassfish-samples/tree/master/ws/jakartaee9

Требуемое программное обеспечение

Для запуска примеров требуется следующее программное обеспечение:

- Java Platform, Standard Edition
- Eclipse Glassfish Server
- Примеры учебника Jakarta EE
- Apache NetBeans IDE
- Apache Maven

Java Platform, Standard Edition

Для сборки, развёртывания и запуска примеров требуется Java Platform, Standard Edition Development Kit (JDK). Требуется использовать JDK 8 Update 20 или выше. Программное обеспечение JDK можно загрузить с веб-сайта https://www.oracle.com/technetwork/java/javase/downloads/index.html.

Eclipse Glassfish Server

Целевой средой сборки и выполнения примеров учебника служит GlassFish Server 6.0. Для сборки, развёртывания и запуска примеров требуется GlassFish Server и, возможно, IDE NetBeans. GlassFish Server можно загрузить с веб-сайта https://glassfish.org/download.

Рекомендации по установке GlassFish Server

GlassFish Server устанавливается из ZIP-файла. Имя пользователя для администрирования по умолчанию устанавливается как `admin` без пароля. Порт администрирования установлен в 4848, а порт HTTP — в 8080.

Это руководство ссылается на *`as-install-parent`* — каталог, в который вы устанавливаете GlassFish Server. Например, каталог установки по умолчанию в Microsoft Windows — `C:\glassfish6`, поэтому *`as-install-parent`* — это `C:\glassfish6`. Сам GlassFish Server устанавливается в *`as-install`* — каталог `glassfish` в *`as-install-parent`*. Итак, в Microsoft Windows *`as-install`* — это `C:\glassfish6\glassfish`.

После установки GlassFish Server добавьте следующие каталоги в `PATH`, чтобы не указывать полный путь при использовании команд:

```
as-install-parent/bin
as-install/bin
```

Примеры учебника Jakarta EE

Код примеров учебника расположен по ссылке https://github.com/eclipse-ee4j/jakartaee-tutorial-examples.

Загрузите примеры и распакуйте их в следующее место:

```
tut-install/examples/
```

Apache NetBeans IDE

Интегрированная среда разработки (IDE) NetBeans — это бесплатная IDE с открытым исходным кодом для разработки приложений Java, в том числе корпоративных приложений. IDE NetBeans поддерживает платформу Jakarta EE. Вы можете создавать, упаковывать, развёртывать и запускать учебные примеры из среды IDE NetBeans.

Для запуска примеров учебных пособий нужна последняя версия IDE NetBeans. Вы можете загрузить IDE NetBeans со страницы https://netbeans.apache.org/download/index.html.

Добавление GlassFish Server в IDE NetBeans

Чтобы запустить учебные примеры в IDE NetBeans, необходимо добавить GlassFish Server в качестве сервера в IDE NetBeans. Следуйте этим инструкциям, чтобы добавить GlassFish Server в IDE NetBeans.

1. В меню **Инструменты** выберите **Серверы**.
2. В мастере серверов нажмите **Добавить сервер**.
3. В разделе «Выбор сервера» выберите **GlassFish Server** и нажмите **Next**.
4. В разделе "Расположение сервера" выберите каталог установки GlassFish Server и нажмите **Далее**.
5. В разделе "Местоположение домена" выберите **Зарегистрировать локальный домен**.
6. Нажмите **Готово**.

Apache Maven

Maven — это инструмент сборки на основе Java, разработанный Apache Software Foundation и используемый для сборки, упаковки и развёртывания примеров учебника. Чтобы запустить учебные примеры из командной строки, требуется Maven версии 3.0 или выше. Если у вас ещё нет Maven, вы можете установить его с:

https://maven.apache.org

Обязательно добавьте каталог `maven-install/bin` в PATH.

Если вы используете среду IDE NetBeans для сборки и запуска примеров, она уже включает в себя копию Maven.

Запуск и остановка GlassFish Server

Сервер GlassFish можно запускать и останавливать с IDE NetBeans или командной строки.

Запуск GlassFish Server из IDE NetBeans

1. Кликните по вкладке **Службы**.
2. Разверните **Серверы**.
3. Кликните правой кнопкой мыши **GlassFish Server** и выберите **Запустить**.

Остановка GlassFish Server из IDE NetBeans

Чтобы остановить GlassFish Server в IDE NetBeans, кликните правой кнопкой мыши **GlassFish Server** и выберите **Остановить**.

Запуск GlassFish Server из командной строки

Чтобы запустить GlassFish Server из командной строки, откройте окно терминала или командную строку и выполните следующее:

```
asadmin start-domain --verbose
```
SHELL

Домен — это набор из одного или нескольких объектов GlassFish Server, управляемых одним сервером администрирования. Следующие элементы связаны с доменом:

- Номер порта GlassFish Server: по умолчанию 8080.
- Номер порта сервера администрирования: по умолчанию 4848.
- Имя пользователя и пароль администратора: имя пользователя по умолчанию — `admin` , и по умолчанию пароль не требуется.

Эти значения указываются при установке GlassFish Server. Примеры в этом руководстве предполагают, что вы выбрали порты по умолчанию, а также имя пользователя по умолчанию и без пароля.

Без аргументов команда `start-domain` инициирует домен по умолчанию — `domain1` . Флаг `--verbose` приводит к тому, что все выводимые данные журнала и отладки появляются в окне терминала или командной строке. Вывод также поступает в журнал сервера, который находится в каталоге *`domain-dir`*`/logs/server.log` .

Остановка GlassFish Server из командной строки

Чтобы остановить GlassFish Server, откройте окно терминала или командную строку и выполните:

```
asadmin stop-domain domain1
```
SHELL

Запуск Консоли администрирования

Для администрирования GlassFish Server и управления пользователями, ресурсами и приложениями Jakarta EE пользуйтесь утилитой Консоль администрирования. GlassFish Server должен быть запущен до вызова Консоли администрирования. Для запуска Консоли администрирования откройте браузер по ссылке http://localhost:4848/.

Запуск Консоли администрирования из IDE NetBeans

1. Кликните по вкладке **Службы**.
2. Разверните **Серверы**.
3. Кликните правой кнопкой мыши **GlassFish Server** и выберите **Просмотр консоли администратора домена**.

IDE NetBeans открывает Консоли администрирования в браузере по умолчанию.

Запуск и остановка Apache Derby

Сервер GlassFish включает в себя Apache Derby.

Запуск Derby из командной строки

Чтобы запустить Derby из командной строки, откройте окно терминала или командную строку, перейдите в каталог *`as-install`*`/bin` и выполните:

```
asadmin start-database
```
SHELL

Остановка Derby из командной строки

Чтобы остановить Derby из командной строки, откройте окно терминала или командную строку, перейдите в каталог *as-install*`/bin` и выполните:

```
asadmin stop-database
```
SHELL

Информацию о Apache Derby, включённом в GlassFish Server, см. в заметках о выпуске в каталоге *as-install*`/javadb/` .

Запуск Derby с IDE NetBeans

При запуске GlassFish Server с использованием IDE NetBeans сервер базы данных запускается автоматически. Однако, если когда-либо потребуется запустить сервер вручную, выполните следующие действия.

1. Кликните по вкладке **Службы**.
2. Разверните **Базы данных**.
3. Кликните правой кнопкой мыши **Java DB** и выберите **Запустить сервер**.

Остановка Derby из IDE NetBeans

Чтобы остановить базу данных из IDE NetBeans кликните правой кнопкой мыши **Java DB** и выберите **Остановить сервер**.

Сборка примеров

Примеры учебных материалов распространяются вместе с файлом конфигурации для IDE NetBeans или Maven. Для сборки, упаковки, развёртывания и запуска примеров можно использовать либо IDE NetBeans, либо Maven. Указания для сборки примеров приведены в каждой главе.

Структура каталогов примеров из учебника

Чтобы упростить итеративную разработку и сохранить исходные файлы приложения отдельно от скомпилированных файлов, в примерах учебника используется структура каталогов приложений Maven.

Каждый модуль приложения имеет следующую структуру:

- `pom.xml` : файл сборки Maven
- `src/main/java` : исходные файлы Java для модуля
- `src/main/resources` : файлы конфигурации для модуля, за исключением веб-приложений
- `src/main/webapp` : веб-страницы, таблицы стилей, файлы тегов и изображения (только веб-приложения)
- `src/main/webapp/WEB-INF` : файлы конфигурации для веб-приложений (только веб-приложения)

Если в примере несколько модулей приложения упакованы в файл EAR, его каталоги подмодулей используют следующие соглашения об именах:

- *example-name* `-app-client` : клиентские приложения
- *example-name* `-ejb` : файлы EJB JAR
- *example-name* `-war` : веб-приложения
- *example-name* `-ear` : приложения EAR

- *example-name* `-common` : JAR-библиотеки с компонентами, классами и файлами, используемыми другими модулями

Файлы сборки Maven (`pom.xml`), распространяемые с примерами, содержат цели компиляции и компоновки приложения в каталог `target` и развёртывания полученного архива в GlassFish Server.

Maven-архетипы Jakarta EE в учебнике

В некоторых главах приведены инструкции по сборке примера приложения с использованием архетипов Maven. Архетипы — это шаблоны для генерации конкретного проекта Maven. Учебное пособие включает в себя несколько архетипов Maven для создания проектов Jakarta EE.

Установка архетипов учебника

Вы должны установить включённые архетипы Maven в свой локальный репозиторий Maven, прежде чем сможете создавать новые проекты на основе архетипов. Вы можете установить архетипы, используя IDE NetBeans или Maven.

Установка архетипов учебника с IDE NetBeans

1. В меню **Файл** выберите **Открыть проект**.
2. В диалоговом окне «Открыть проект» перейдите к:

```
tut-install/examples
```

1. Выберите каталог `archetypes` .
2. Нажмите **Открыть проект**.
3. На вкладке **Проекты** кликните правой кнопкой мыши проект `archetypes` и выберите **Сборка**.

Установка архетипов учебника с использованием Maven

1. В окне терминала перейдите в:

```
tut-install/examples/archetypes/
```

1. Введите следующую команду:

```
mvn install
```

Отладка приложений Jakarta EE

В этом разделе объясняется, как определить причину ошибки при развёртывании или выполнении приложения.

Использование логов сервера

Одним из способов отладки приложений является просмотр журнала сервера в *`domain-dir`*`/logs/server.log` . Журнал содержит выходные данные GlassFish Server и ваших приложений. Сообщения из любого класса Java в приложении могут быть зарегистрированы с помощью `System.out.println` , Java Logging API (документация на https://docs.oracle.com/javase/8/docs /technotes/guides/logging/index.html) и из веб-компонентов с помощью метода `ServletContext.log` .

Если вы используете IDE NetBeans, выходные данные журнала отображаются в окне «Вывод», а также в журнале сервера.

Если вы запустите GlassFish Server с флагом `--verbose`, все выходные данные журнала и отладки появятся в окне терминала или командной строке и в журнале сервера. Если вы запустите GlassFish Server в фоновом режиме, информация об отладке будет доступна только в журнале. Вы можете просмотреть журнал сервера в текстовом редакторе или в Консоли администрирования.

Использование Консоли администрирования для просмотра логов

1. Выберите узел **GlassFish Server**.
2. Кликните **Просмотреть файлы журнала**.
   Открывается программа просмотра журнала и отображаются последние 40 записей.
3. Чтобы отобразить другие записи, выполните следующие действия:
   a. Нажмите **Изменить поиск**.
   b. Укажите любые ограничения на записи, которые вы хотите увидеть.
   c. Нажмите Поиск в верхней части окна просмотра журнала.

Использование отладчика

GlassFish Server поддерживает архитектуру отладчика платформы Java (JPDA). С помощью JPDA вы можете настроить GlassFish Server для передачи отладочной информации через сокет.

Отладка приложения с помощью отладчика

1. Выполните следующие действия, чтобы включить отладку в GlassFish Server в Консоли администрирования:
   a. Разверните узел **Configurations**, затем разверните узел **server-config**.
   b. Выберите узел **Настройки JVM**. Параметры отладки по умолчанию установлены на:

      ```
      -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=9009
      ```

      Как видите, порт сокета отладчика по умолчанию — 9009. Вы можете изменить его на порт, который не используется GlassFish Server или другим сервисом.
   c. Установите флажок **Debug Enabled**.
   d. Нажмите **Сохранить**.
2. Остановите GlassFish Server и перезапустите его.

Часть II: Основы платформы

Часть II знакомит с основами платформы.

Глава 3. Создание ресурса

Ресурс — это программный объект, обеспечивающий соединение с такими системами, как серверы баз данных и системы обмена сообщениями. Компоненты Jakarta EE могут получать доступ к различным ресурсам, включая базы данных, почтовые сессии, объекты Jakarta Messaging и URL-ы. Платформа Jakarta EE предоставляет механизмы, которые позволяют получить доступ ко всем этим ресурсам однообразно. В этой главе рассматриваются несколько типов ресурсов и объясняется, как их создавать.

Ресурсы и именование JNDI

В распределённом приложении компоненты должны иметь доступ к другим компонентам и ресурсам, таким как базы данных. Например, сервлет может вызывать удалённые методы для Enterprise-бина, который получает информацию из базы данных. В платформе Jakarta EE служба именования и интерфейса каталогов (JNDI) позволяет компонентам обращаться к другим компонентам и ресурсам.

Ресурс — это программный объект, который обеспечивает соединения с системами, такими как серверы баз данных и системы обмена сообщениями. Ресурс подключения к базе данных иногда называют источником данных. Каждый объект ресурса идентифицируется по уникальному имени, называемому имя JNDI. Например, имя JNDI предустановленного ресурса JDBC для Apache Derby, поставляемого с GlassFish Server — `java:comp/DefaultDataSource`.

Администратор создаёт ресурсы в пространстве имён JNDI. В GlassFish Server вы можете использовать консоль администрирования или команду `asadmin` для создания ресурсов. Приложения затем используют аннотации для добавления ресурсов. Если приложение использует инъецирование ресурсов, API JNDI вызывается GlassFish Server, а не приложением. Однако приложение также может находить ресурсы, совершая прямые вызовы JNDI API.

Объект ресурса и его имя JNDI связаны между собой сервисом имён и каталогов. Чтобы создать новый ресурс, в пространство имён JNDI вводится новая связь имени с объектом. Инъецировать ресурсы можно аннотацией `@Resource`.

Вы можете использовать дескриптор развёртывания, чтобы переопределить назначение ресурсов, указанное в аннотации. Использование дескриптора развёртывания позволяет изменять приложение только переупаковывая его, без перекомпиляции исходных файлов. Однако для большинства приложений дескриптор развёртывания не требуется.

Объекты источника данных и пулы соединений

Для хранения, организации и извлечения данных большинство приложений используют реляционную базу данных. Компоненты Jakarta EE могут обращаться к реляционным базам данных через API JDBC. Для получения информации об этом API см. https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/.

В API JDBC доступ к базам данных осуществляется с помощью объектов `DataSource`. `DataSource` имеет набор свойств, которые идентифицируют и описывают реальный источник данных, который он представляет. Эти свойства включают в себя такую информацию, как местоположение сервера базы данных, имя базы данных, сетевой протокол, используемый для связи с сервером, и так далее. В GlassFish Server источник данных называется ресурсом JDBC.

Приложения получают доступ к источнику данных, используя соединение, а объект `DataSource` можно рассматривать как фабрику для соединений с конкретным источником данных, который представляет объект `DataSource`. В базовой реализации `DataSource` вызов метода `getConnection` возвращает объект

подключения, который является физическим соединением с источником данных.

Объект `DataSource` может быть зарегистрирован сервисом имён JNDI. В этом случае приложение может использовать API JNDI для доступа к этому объекту `DataSource`, который затем можно использовать для подключения к источнику данных, который он представляет.

Объекты `DataSource`, которые реализуют пул соединений, также создают соединение с конкретным источником данных, который представляет класс `DataSource`. Объект соединения, который возвращает метод `getConnection`, является дескриптором объекта `PooledConnection`, а не физическим соединением. Приложение использует объект соединения так же, как оно использует соединение. Пул соединений не влияет на код приложения с тем исключением, что пул соединений, как и все соединения, всегда должен закрываться явно. Когда приложение закрывает соединение, которое находится в пуле, оно возвращается в пул для повторного использования. В следующий раз, когда вызывается `getConnection`, дескриптор одного из этих объединённых соединений будет возвращен, если он доступен. Поскольку пул соединений не создаёт новое физическое соединение при каждом запросе, приложения могут работать значительно быстрее.

Пул соединений JDBC — это группа повторно используемых соединений для конкретной базы данных. Поскольку создание каждого нового физического соединения занимает много времени, сервер поддерживает пул доступных соединений для повышения производительности. Когда приложение запрашивает соединение, оно получает его из пула. Когда приложение закрывает соединение, оно возвращается в пул.

Приложения, использующие Jakarta Persistence, указывают объект `DataSource`, который они используют, в элементе `jta-data-source` файла `persistence.xml`:

```xml
<jta-data-source>jdbc/MyOrderDB</jta-data-source>
```

Обычно это единственная ссылка на объект JDBC для юнита персистентности. Код приложения не ссылается ни на какие объекты JDBC.

Административное создание ресурсов

Прежде чем развёртывать или запускать много приложений, может потребоваться создать ресурсы для них. Приложение может содержать файл `glassfish-resources.xml`, который можно использовать для задания ресурсов этого и других приложений. Затем, для административного создания ресурсов вы можете использовать команду `asadmin`, указав файл `glassfish-resources.xml` как аргумент:

```shell
asadmin add-resources glassfish-resources.xml
```

Файл `glassfish-resources.xml` может быть создан в любом проекте с использованием IDE NetBeans или вручную. Некоторые примеры Jakarta Messaging используют этот подход к созданию ресурсов. Файл для создания ресурсов, необходимых для примера простого производителя сообщений, можно найти в каталоге `jms/simple/producer/src/main/setup`.

Вы также можете использовать команду `asadmin create-jms-resource` для создания ресурсов этого примера. Когда использование ресурса закончено, можно использовать команду `asadmin list-jms-resources` для отображения имён всех ресурсов и команду `asadmin delete-jms-resource` для их удаления независимо от того, как ресурсы были созданы.

Глава 4. Инъецирование

В этой главе даётся обзор инъецирования в Jakarta EE и описываются два механизма инъецирования, предоставляемые платформой: инъецирование ресурсов и инъецирование зависимостей.

Jakarta EE предоставляет механизмы инъецирования, которые позволяют вашим объектам получать ссылки на ресурсы и другие зависимости без их инстанцирования напрямую. Вы объявляете требуемые ресурсы и другие зависимости в классах, аннотируя поля или методы одной из аннотаций, которые помечают поле как точку инъецирования. Затем контейнер предоставляет необходимые объекты во время выполнения. Инъецирование упрощает код и отделяет его от реализации зависимостей.

Инъецирование ресурса

Инъецирование ресурсов позволяет инъецировать любой ресурс, доступный в пространстве имён JNDI, в любой объект, управляемый контейнером — в сервлет, Enterprise-бин или Managed-бин. Например, вы можете использовать инъецирование ресурсов для инъецирования источников данных, коннекторов или кастомных ресурсов, доступных в пространстве имён JNDI.

Обычно для ссылки на инъецируемый объект используется его интерфейс, отделяющий код от реализации ресурса.

Например, следующий код инъецирует объект источника данных, обеспечивающий соединение с базой данных Apache Derby, предустановленной в GlassFish Server:

JAVA

```
public class MyServlet extends HttpServlet {
    @Resource(name="java:comp/DefaultDataSource")
    private javax.sql.DataSource dsc;
    ...
}
```

В дополнение к инъецированию на основе полей, как в предыдущем примере, вы можете использовать инъецирование ресурсов на основе методов:

JAVA

```
public class MyServlet extends HttpServlet {
    private javax.sql.DataSource dsc;
    ...
    @Resource(name="java:comp/DefaultDataSource")
    public void setDsc(java.sql.DataSource ds) {
        dsc = ds;
    }
}
```

Чтобы использовать инъецирование на основе метода, set-метод должен следовать соглашениям JavaBeans для имён свойств: имя метода должно начинаться с `set`, иметь возвращаемый тип `void` и иметь только один параметр.

Аннотация `@Resource` находится в пакете `jakarta.annotation` и определена в спецификации Jakarta Annotations. Инъецирование ресурсов происходит по имени, поэтому оно не является типобезопасным (typesafe): тип объекта ресурса неизвестен на этапе компиляции, поэтому вы можете получить ошибки времени выполнения, если типы объекта и его ссылки не совпадут.

Инъецирование зависимостей

Инъецирование зависимостей позволяет превращать обычные классы Java в Managed-бины и инъецировать их в любой другой Managed-бин. Используя инъецирование зависимостей, код может декларировать зависимости от любого Managed-бина. Контейнер автоматически предоставляет объекты этих зависимостей в точках инъецирования во время выполнения, а также управляет жизненным циклом этих объектов.

Инъецирование зависимостей в Jakarta EE задаёт области, которые управляют жизненным циклом инстанцируемых и инъецируемых контейнером объектов. Так, Managed-бин, необходимый для ответа только на один клиентский запрос (например, конвертер валют), имеет область видимости, отличную от области видимости Managed-бина, необходимого для обработки нескольких клиентских запросов в течение сессии (например, корзины покупок).

Managed-бин может быть задан простым назначением области видимости обычному классу. Это делает возможным его инъецирование:

JAVA

```
@jakarta.enterprise.context.RequestScoped
public class CurrencyConverter { ... }
```

Используйте аннотацию `jakarta.inject.Inject` для инъецирования EJB, например:

JAVA

```
public class MyServlet extends HttpServlet {
    @Inject CurrencyConverter cc;
    ...
}
```

В отличие от инъецирования ресурсов, инъецирование зависимостей является типобезопасным (typesafe), поскольку приведение выполняется к известному типу. Чтобы отделить код от реализации Managed-бина, может быть использован интерфейс для ссылки на инъецируемые объекты, которые реализует этот Managed-бин.

Дополнительные сведения об инъецировании зависимостей см. в главе 25 *Введение в Инъецирование контекстов и зависимостей Jakarta*, а также в спецификации Jakarta Contexts and Dependency Injection.

Основные различия между инъецированием ресурса и инъецированием зависимости

Таблица 4-1 перечисляет основные различия между инъецированием ресурсов и инъецированием зависимостей.

Таблица 4-1 Различия между инъецированием ресурсов и инъецированием зависимостей

Механизм инъецирования	**Может инъецировать ресурсы JNDI напрямую**	**Можно вводить регулярные классы напрямую**	**Как происходит выбор инъецируемого объекта**	**Типобезопасный (typesafe)**
Инъецирование ресурса	да	нет	Название ресурса	нет
Инъецирование зависимостей	нет	да	Тип	да

Глава 5. Упаковка

В этой главе описывается упаковка. Приложение Jakarta EE упаковывается в один или несколько стандартных модулей для развёртывания в любой системе, совместимой с платформой Jakarta EE. Каждый модуль содержит функциональный компонент или компоненты (такие как Enterprise-бин, веб-страница, сервлет или апплет) и необязательный дескриптор развёртывания, который описывает его содержимое.

Упаковка приложений

Приложение Jakarta EE поставляется в виде архива Java (JAR), файла Web Archive (WAR) или Enterprise Archive (EAR). Файл WAR или EAR — это стандартный файл JAR (`.jar`), отличающийся от него только расширением `.war` или `.ear` . Использование файлов и модулей JAR, WAR и EAR позволяет компоновать различные приложения Jakarta EE из одних и тех же компонентов. Никакого дополнительного кодирования не требуется. Это только вопрос сборки (или упаковки) различных модулей Jakarta EE в файлы JAR, WAR или EAR.

EAR-файл (см. рисунок 5-1) содержит модули Jakarta EE и, необязательно, дескрипторы развёртывания. Дескриптор развёртывания — XML-документ с расширением `.xml` — описывает параметры развёртывания приложения, модуля или компонента. Поскольку информация дескриптора развёртывания является декларативной, она может быть изменена без необходимости изменения исходного кода. Во время выполнения сервер Jakarta EE считывает дескриптор развёртывания и соответствующим образом конфигурирует приложение, модуль или компонент.

Информация о развёртывании чаще всего указывается в исходном коде аннотациями. Содержимое дескрипторов развёртывания имеет приоритет перед указаниями в исходном коде.

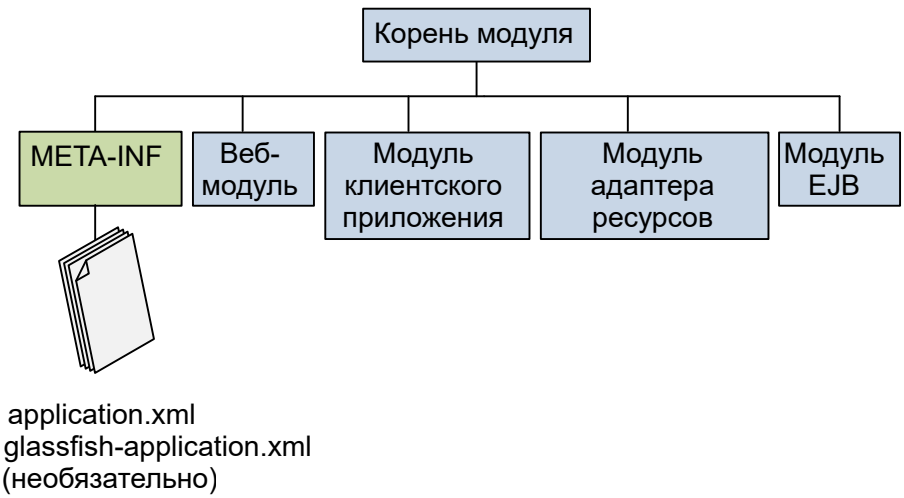

Рисунок 5-1 Структура EAR-файла

Существует два типа дескрипторов развёртывания: Jakarta EE и времени выполнения. Дескриптор развёртывания Jakarta EE определяется спецификацией Jakarta EE и может использоваться для настройки параметров развёртывания в любой совместимой с Jakarta EE реализации. Дескриптор развёртывания времени выполнения используется для настройки параметров реализации Jakarta EE. Например, дескриптор развёртывания среды выполнения GlassFish Server содержит такую информацию, как корневой контекст веб-приложения, а также специфичные для реализации параметры GlassFish Server, например директивы кэширования. Дескрипторы развёртывания GlassFish Server именуются `glassfish-`*`moduleType`*`.xml` и расположены в том же каталоге `META-INF` в качестве дескриптора развёртывания Jakarta EE.

Модуль Jakarta EE состоит из одного или нескольких компонентов Jakarta EE для одного и того же типа контейнера и, необязательно, одного дескриптора развёртывания компонента этого типа. Например, дескриптор развёртывания модуля Enterprise-бина объявляет атрибуты транзакции и полномочия безопасности для Enterprise-бина. Модуль Jakarta EE может быть развёрнут как автономный модуль.

Модули Jakarta EE бывают следующих типов:

- Модули EJB-компонентов, которые содержат файлы классов для EJB-компонентов и, опционально, дескриптор развёртывания EJB-компонентов. Модули EJB упакованы в файлы JAR с расширением `.jar`.
- Веб-модули, которые содержат файлы классов сервлетов, веб-файлы, файлы вспомогательных классов, файлы GIF и HTML и, при необходимости, дескриптор развёртывания веб-приложения. Веб-модули упакованы в JAR-файлы с расширением `.war` (веб-архив).
- Клиентские модули приложения, которые содержат файлы классов и, необязательно, дескриптор развёртывания клиентского приложения. Клиентские модули приложения упакованы в JAR-файлы с расширением `.jar`.
- Модули адаптеров ресурсов, содержащие интерфейсы Java, классы, библиотеки и, необязательно, дескриптор развёртывания адаптера ресурса. Вместе они реализуют архитектуру Connector-ов (см. Jakarta Connectors) для конкретной информационной системы. Модули адаптера ресурсов упакованы в JAR-файлы с расширением `.rar` (архив адаптера ресурса).

Упаковка Enterprise-бинов

В этом разделе объясняется, как EJB-компоненты могут быть упакованы в модули JAR или WAR компонента EJB. Он включает в себя следующие разделы:

- Упаковка компонентов EJB в модули JAR
- Упаковка Enterprise-бинов в модули WAR

Упаковка компонентов EJB в модули JAR

JAR-файл компонента EJB является переносимым и может использоваться в различных приложениях.

Для сборки приложения Jakarta EE упакуйте один или несколько модулей, таких как JAR-файлы EJB, в файл EAR — архив, содержащий приложение. При развёртывании файла EAR, который содержит файл EJB JAR, GlassFish Server развёртывает также и Enterprise-бины. Можно также развернуть JAR EJB-компонента, который не содержится в файле EAR. На рисунке 5-2 показано содержимое JAR-файла EJB-компонента.

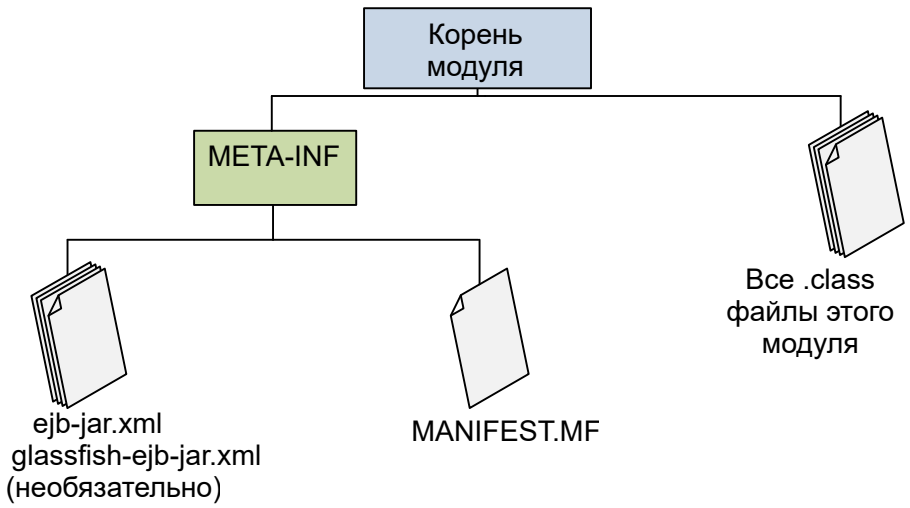

Рисунок 5-2. Структура корпоративного JAR-компонента

Упаковка Enterprise-бинов в модули WAR

Enterprise-бины часто предоставляют бизнес-логику веб-приложения. В этих случаях упаковка Enterprise-бина в модуль WAR веб-приложения упрощает развёртывание и организацию приложения. Enterprise-бины могут быть упакованы в модуль WAR как файлы классов Java или в JAR-файл, входящий в состав модуля WAR.

Чтобы включить файлы классов Enterprise-бина в модуль WAR, файлы классов должны находиться в каталоге `WEB-INF/classes`.

Чтобы включить JAR-файл, содержащий Enterprise-бины, в модуль WAR, добавьте JAR в каталог `WEB-INF/lib` модуля WAR.

Модули WAR, содержащие Enterprise-бины, не требуют дескриптора развёртывания `ejb-jar.xml`. Если приложение использует `ejb-jar.xml`, оно должно находиться в каталоге `WEB-INF` модуля WAR.

Файлы JAR, содержащие классы EJB-компонентов, упакованные в модуль WAR, не считаются файлами JAR EJB-компонентов, даже если объединённый файл JAR соответствует формату файла JAR EJB-компонента. Компоненты EJB, содержащиеся в JAR-файле, семантически эквивалентны компонентам EJB, расположенным в каталоге `WEB-INF/classes` модуля WAR, а пространство имён среды всех компонентов EJB распространяется на модуль WAR.

Для примера предположим, что веб-приложение состоит из Enterprise-бина корзины покупок, Enterprise-бина обработки кредитных карт и внешнего интерфейса сервлета Java. Компонент корзины покупок предоставляет локальное представление без интерфейса и определяется следующим образом:

JAVA

```java
package com.example.cart;

@Stateless
public class CartBean { ... }
```

Компонент обработки кредитной карты упакован в свой собственный JAR-файл `cc.jar`, предоставляет локальное представление без интерфейса и определяется следующим образом:

JAVA

```java
package com.example.cc;

@Stateless
public class CreditCardBean { ... }
```

Сервлет `com.example.web.StoreServlet` обрабатывает веб-интерфейс и использует как `CartBean`, так и `CreditCardBean`. Компоновка модуля WAR для этого приложения выглядит следующим образом:

```
WEB-INF/classes/com/example/cart/CartBean.class
WEB-INF/classes/com/example/web/StoreServlet
WEB-INF/lib/cc.jar
WEB-INF/ejb-jar.xml
WEB-INF/web.xml
```

Упаковка веб-архивов

В архитектуре Jakarta EE веб-модуль — это наименьший развёртываемый и используемый элемент веб-ресурсов. Веб-модуль содержит веб-компоненты и файлы статического веб-содержимого (например, изображения), которые называются веб-ресурсами. Веб-модуль Jakarta EE соответствует веб-приложению, определённому в спецификации сервлетов Jakarta.

В дополнение к веб-компонентам и веб-ресурсам веб-модуль может содержать другие файлы:

- Серверные утилитные классы
- Клиентские утилитные классы

Веб-модуль имеет специфическую структуру. Каталог верхнего уровня веб-модуля является корневым каталогом приложения. В корне документа хранятся страницы XHTML, клиентские классы и архивы, а также статические веб-ресурсы (например, изображения).

Корень документа содержит подкаталог `WEB-INF`, который может содержать следующие файлы и каталоги:

- `classes`, каталог, содержащий серверные классы: сервлеты, файлы классов Enterprise-бинов, служебные классы и компоненты JavaBeans
- `lib`, каталог, содержащий JAR-файлы, содержащие Enterprise-бины, и JAR-архивы библиотек, вызываемых серверными классами
- Дескрипторы развёртывания, такие как `web.xml` (дескриптор развёртывания веб-приложения) и `ejb-jar.xml` (дескриптор развёртывания EJB-компонента)

Веб-модулю нужен файл `web.xml`, если он использует Jakarta Faces, если он должен указывать определённые виды информации о безопасности или если вы хотите переопределить информацию, указанную в аннотациях веб-компонента.

Вы также можете создавать специфичные для приложения подкаталоги (то есть каталоги пакетов) либо в корневом каталоге документа, либо в каталоге `WEB-INF/classes/`.

Веб-модуль может быть развёрнут в виде распакованной файловой структуры или может быть упакован в файл JAR, известный как файл веб-архива (WAR). Поскольку содержимое и использование файлов WAR отличаются от файлов JAR, имена файлов WAR имеют расширение `.war`. Только что описанный веб-модуль является переносимым. Он может быть развернут в любом веб-контейнере, соответствующем спецификации сервлетов Jakarta.

Вы можете предоставить дескриптор развёртывания во время выполнения при развёртывании WAR в GlassFish Server, но в большинстве случаев этого не требуется. Дескриптор развёртывания времени выполнения — это файл XML, который может содержать такую информацию, как корневой контекст веб-приложения, назначение переносимых имён ресурсов приложения ресурсам GlassFish Server и назначение ролей безопасности приложения пользователям, группам и принципалам, определённым в GlassFish Server. Дескриптор развёртывания времени выполнения веб-приложения GlassFish Server, если он используется, имеет имя `glassfish-web.xml` и находится в каталоге `WEB-INF`. Структура веб-модуля, который можно развернуть в GlassFish Server, показана на рисунке 5-3.

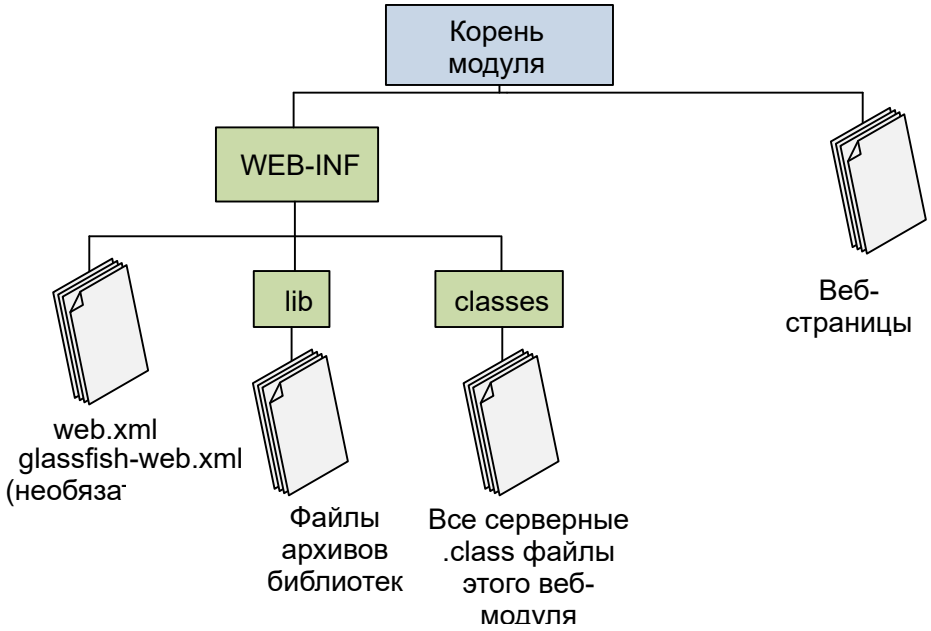

Рисунок 5-3. Структура веб-модуля

Упаковка архивов адаптера ресурсов

Файл архива адаптера ресурсов (RAR) хранит файлы XML, классы Java и другие ресурсы для приложений Jakarta EE Connector. Адаптер ресурса может быть развёрнут на любом сервере Jakarta EE, в основном как приложение Jakarta EE. RAR-файл быть включён в состав EAR-файла или существовать отдельно от него.

RAR-файл содержит

- JAR-файл с классами реализации адаптера ресурсов
- Необязательный каталог `META-INF/`, в котором может храниться файл `ra.xml` и/или специфичный для сервера и используемый для настройки дескриптор развёртывания

RAR-файл может быть развёрнут на сервере приложений как отдельный компонент или как часть более крупного приложения. В любом случае, адаптер доступен для всех приложений через пространство имён JNDI.

Часть III: Веб-слой

Часть III исследует технологии веб-слоя.

Глава 6. Начало работы с веб-приложениями

В этой главе рассматриваются веб-приложения, которые обычно используют Jakarta Faces и/или Jakarta Servlet.

Веб-приложения

Веб-приложение — это динамическое расширение веб-сервера или сервера приложений. Веб-приложения бывают следующих типов:

Ориентированные на отображение

Веб-приложение, ориентированное на отображение, создаёт интерактивные веб-страницы, содержащие различные типы разметки (HTML, XHTML, XML и т. д.) и динамическое содержимое в ответ на запросы. Разработка веб-приложений, ориентированных на отображение, рассматривается с главы 1.8.3 Технология Jakarta Faces по главу 18 *Технология Jakarta Servlet*

Сервис-ориентированные

Сервис-ориентированное веб-приложение реализует конечную точку веб-сервиса. Ориентированные на отображение приложения часто являются клиентами сервис-ориентированных веб-приложений. Разработка сервис-ориентированных веб-приложений рассматривается в главе 31 *Создание веб-сервисов с помощью Jakarta XML Web Services* и главе 32 *Создание RESTful веб-сервисов с помощью Jakarta REST* Части VI «Веб-сервисы»

В платформе Jakarta EE веб-компоненты предоставляют возможности динамического расширения веб-сервера. Веб-компоненты могут быть сервлетами, веб-страницами Jakarta Faces, конечными точками веб-сервисов или веб-страницами Jakarta Server Pages Рисунок 6-1 иллюстрирует взаимодействие между веб-клиентом и веб-приложением, использующим сервлет. Клиент отправляет HTTP-запрос на веб-сервер. Веб-сервер, реализованный на сервлетах Jakarta и Jakarta Server Pages, конвертирует запрос в объект `HTTPServletRequest` . Этот объект доставляется веб-компоненту, который может взаимодействовать с компонентами JavaBeans или базой данных для создания динамического содержимого. Затем веб-компонент может сгенерировать `HTTPServletResponse` или передать запрос другому веб-компоненту. В конечном итоге веб-компонент генерирует объект `HTTPServletResponse` . Веб-сервер преобразует этот объект в ответ HTTP и возвращает его клиенту.

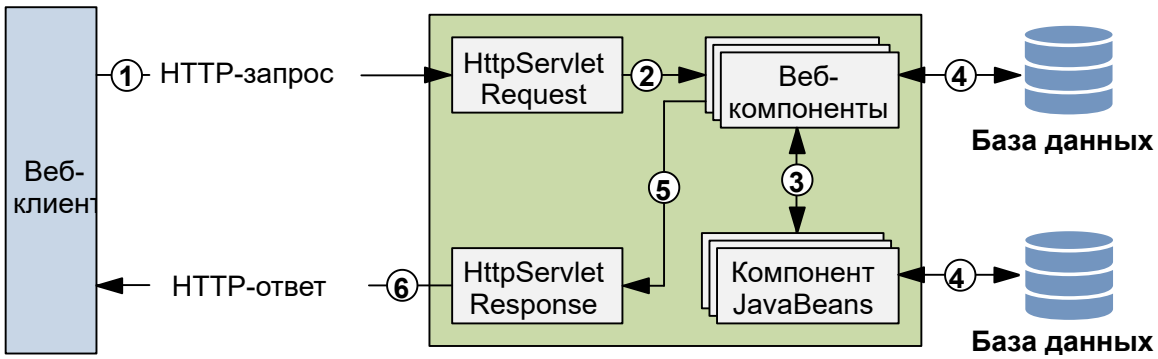

Рис. 6-1. Обработка запросов веб-приложения Jakarta

Сервлеты — это программные классы Java, которые динамически обрабатывают запросы и выдают ответы. Технологии Jakarta Faces и Facelets используются для создания интерактивных веб-приложений. (Фреймворки также могут быть использованы для этой цели.) Хотя сервлеты и страницы Jakarta Faces и Facelets могут использоваться для выполнения похожих задач, у каждого из них есть свои сильные стороны. Сервлеты лучше всего подходят для сервис-ориентированных приложений (конечные точки веб-сервисов могут быть

реализованы в виде сервлетов) и функций управления приложениями, ориентированными на отображение данных, таких как диспетчеризация запросов и обработка бинарных данных. Страницы Jakarta Faces и Facelets больше подходят для создания текстовой разметки, такой как XHTML, и обычно используются для приложений, ориентированных на отображение.

Веб-компоненты поддерживаются сервисами платформы времени выполнения. Эта платформа называется веб-контейнером. Веб-контейнер предоставляет такие сервисы, как диспетчеризация запросов, безопасность, параллелизм и управление жизненным циклом. Веб-контейнер также предоставляет веб-компонентам доступ к таким API, как пространство имён JNDI, транзакции и электронная почта.

Определённые аспекты поведения веб-приложения можно настроить, когда приложение развёрнуто в веб-контейнере. Информация о конфигурации может быть указана аннотациями Jakarta EE или храниться в текстовом файле формата XML, который называется дескриптором развёртывания веб-приложения. Дескриптор развёртывания веб-приложения должен соответствовать схеме, описанной в спецификации Jakarta Servlet.

В этой главе даётся краткий обзор действий, связанных с разработкой веб-приложений. Во-первых, обобщается жизненный цикл веб-приложения и объясняется, как упаковать и развернуть элементарные веб-приложения в GlassFish Server. Затем приводится информация по настройке веб-приложений и обсуждается, как задать наиболее часто используемые параметры конфигурации.

Жизненный цикл веб-приложения

Веб-приложение состоит из веб-компонентов: файлов статических ресурсов (например, изображения и каскадные таблицы стилей CSS), вспомогательные классы и библиотеки. Веб-контейнер предоставляет множество вспомогательных сервисов, которые расширяют возможности веб-компонентов и упрощают их разработку. Однако, поскольку веб-приложение должно учитывать эти сервисы, процесс создания и запуска веб-приложения отличается от такового для традиционных автономных классов Java.

Процесс создания, развёртывания и выполнения веб-приложения можно обобщить следующим образом:

1. Разработайте код веб-компонента.
2. При необходимости разработайте дескриптор развёртывания веб-приложения.
3. Скомпилируйте компоненты веб-приложения и вспомогательные классы, на которые ссылаются компоненты.
4. При желании упакуйте приложение в развёртываемый модуль.
5. Разверните приложение в веб-контейнере.
6. Пройдите по URL, который ассоциирован с веб-приложением.

Разработка кода веб-компонента рассматривается в последующих главах. Шаги 2–4 расширены в следующих разделах и проиллюстрированы приложением вида "Hello, World", ориентированным на отображение. Это приложение позволяет пользователю вводить имя в форму HTML, а после отправки имени отображает приветствие.

Приложение Hello содержит два веб-компонента, которые генерируют приветствие и ответ. В этой главе рассматриваются следующие простые приложения:

- `hello1` , приложение на основе Jakarta Faces, которое использует две страницы XHTML и Managed-бин
- `hello2` , веб-приложение на основе сервлетов, в котором компоненты реализованы двумя классами сервлетов

Приложения используются для иллюстрации задач, связанных с упаковкой, развёртыванием, настройкой и запуском приложения, содержащего веб-компоненты.

Веб-модуль с использованием Jakarta Faces: пример hello1

Приложение `hello1` — это веб-модуль, который использует Jakarta Faces для отображения приветствия и ответа. Вы можете использовать текстовый редактор или IDE NetBeans для просмотра файлов приложения.

Исходный код этого приложения находится в каталоге *`tut-install`*`/examples/web/jsf/hello1/`.

Просмотр веб-модуля hello1 в IDE NetBeans

Чтобы просмотреть веб-модуль `hello1` с IDE NetBeans:

1. В меню **Файл** выберите **Открыть проект**.
2. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/web/jsf
   ```

3. Выберите `hello1` и кликните **Открыть проект**.
4. Разверните узел **Веб-страницы** и дважды кликните файл `index.xhtml` чтобы открыть его в редакторе.

   Файл `index.xhtml` является страницей входа по умолчанию для приложения Facelets. В типичном приложении Facelets веб-страницы создаются в формате XHTML. В этом приложении используются простые теги разметки для отображения формы с графическим изображением, заголовком, полем и двумя командными кнопками:

   XML

   ```
   <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
   <html lang="en"
         xmlns="http://www.w3.org/1999/xhtml"
         xmlns:h="http://xmlns.jcp.org/jsf/html">
       <h:head>
           <title>Facelets Hello Greeting</title>
       </h:head>
       <h:body>
           <h:form>
               <h:graphicImage url="#{resource['images:duke.waving.gif']}"
                               alt="Duke waving his hand"/>
               <h2>Hello, my name is Duke. What's yours?</h2>
               <h:inputText id="username"
                            title="My name is: "
                            value="#{hello.name}"
                            required="true"
                            requiredMessage="Error: A name is required."
                            maxlength="25" />
               <p></p>
               <h:commandButton id="submit" value="Submit" action="response">
               </h:commandButton>
               <h:commandButton id="reset" value="Reset" type="reset">
               </h:commandButton>
           </h:form>
           ...
       </h:body>
   </html>
   ```

   Самым сложным элементом на странице является поле `inputText`. Атрибут `maxlength` указывает максимальную длину поля. Атрибут `required` указывает, что поле должно быть заполнено. Атрибут `requiredMessage` предоставляет сообщение об ошибке, которое будет отображаться, если поле оставить

пустым. Атрибут `title` предоставляет текст, который будет использоваться программами чтения с экрана для инвалидов по зрению. Наконец, атрибут `value` содержит выражение, которое будет предоставлено Managed-бином `Hello`.

Веб-страница подключается к Managed-бину `Hello` через выражение значения языка выражений (EL) `#{hello.name}`, которое извлекает значение свойства `name` Managed-бина. Обратите внимание на использование `hello` для ссылки на Managed-бин `Hello`. Если имя не указано в аннотации `@Named` Managed-бина, то к Managed-бину всегда обращаются с первой буквой имени класса в нижнем регистре.

Элемент Submit `commandButton` определяет действие как `response`, что означает, что при клике кнопки отображается страница `response.xhtml`.

5. Выполните двойной клик на файле `response.xhtml`, чтобы просмотреть его.

   Откроется страница ответа. Устроенная даже проще страницы приветствия, страница ответа содержит графическое изображение, заголовок, который отображает выражение, предоставляемое Managed-бином, и одну кнопку, элемент `action` которой возвращает вас обратно на страницу `index.xhtml`:

   XML

   ```
   <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
   <html lang="en"
         xmlns="http://www.w3.org/1999/xhtml"
         xmlns:h="http://xmlns.jcp.org/jsf/html">
       <h:head>
           <title>Facelets Hello Response</title>
       </h:head>
       <h:body>
           <h:form>
               <h:graphicImage url="#{resource['images:duke.waving.gif']}"
                               alt="Duke waving his hand"/>
               <h2>Hello, #{hello.name}!</h2>
               <p></p>
               <h:commandButton id="back" value="Back" action="index" />
           </h:form>
       </h:body>
   </html>
   ```

6. Разверните узел "Пакеты исходного кода", затем узел `ee.jakarta.tutorial.hello1`.
7. Выполните двойной клик на файле `Hello.java`, чтобы просмотреть его.

   Класс `Hello`, называемый классом Managed-бина, предоставляет get- и set- методы получения и установки свойства `name`, используемого в выражениях страницы Facelets. По умолчанию язык выражений ссылается на имя класса с первой буквой в нижнем регистре (`hello.name`).

JAVA

```java
package ee.jakarta.tutorial.hello1;

import jakarta.enterprise.context.RequestScoped;
import jakarta.inject.Named;

@Named
@RequestScoped
public class Hello {

    private String name;

    public Hello() {
    }

    public String getName() {
        return name;
    }

    public void setName(String user_name) {
        this.name = user_name;
    }
}
```

Если для класса бина используется имя по умолчанию, вы можете указать аннотацию `@Model` вместо `@Named` и `@RequestScoped`. Аннотация `@Model` называется стереотипом. Это термин для аннотации, которая инкапсулирует другие аннотации. Это будет описано позже в Использование стереотипов в приложениях CDI. Некоторые примеры будут использовать `@Model` там, где это уместно.

8. Под узлом веб-страниц разверните узел WEB-INF и выполните двойной клик на файле `web.xml`, чтобы просмотреть его.

   Файл `web.xml` содержит несколько элементов, необходимых для приложения Facelets. Всё перечисленное ниже создаётся автоматически при использовании IDE NetBeans для создания приложения.

   - Контекстный параметр, определяющий этап проекта:

     XML

     ```xml
     <context-param>
         <param-name>jakarta.faces.PROJECT_STAGE</param-name>
         <param-value>Development</param-value>
     </context-param>
     ```

     Контекстный параметр предоставляет информацию о конфигурации, необходимую для веб-приложения. Приложение может определять свои собственные контекстные параметры. Кроме того, Jakarta Faces и Jakarta Servlet задают контекстные параметры, которые могут использоваться приложением.

   - Элемент `servlet` и парный ему элемент `servlet-mapping` определяют сервлет `FacesServlet`. Он будет обрабатывать все запросы, URL которых имеет суффикс `.xhtml`:

     XML

     ```xml
     <servlet>
         <servlet-name>Faces Servlet</servlet-name>
         <servlet-class>jakarta.faces.webapp.FacesServlet</servlet-class>
         <load-on-startup>1</load-on-startup>
     </servlet>
     <servlet-mapping>
         <servlet-name>Faces Servlet</servlet-name>
         <url-pattern>*.xhtml</url-pattern>
     </servlet-mapping>
     ```

   - Элемент `welcome-file-list` указывает местоположение страницы входа:

```xml
    <welcome-file-list>
        <welcome-file>index.xhtml</welcome-file>
    </welcome-file-list>
```

Введение в области видимости

В `Hello.java`, аннотации `jakarta.inject.Named` и `jakarta.enterprise.context.RequestScoped` определяют класс как Managed-бин используя область видимости запроса. Области видимости определяют, как долго данные приложения остаются доступны.

Наиболее часто используемые области в приложениях Jakarta Faces:

Request (`@RequestScoped`)

область видимости запроса хранится в течение одного HTTP-запроса в веб-приложении. область видимости запроса сохраняется в течение одного HTTP-запроса к веб-приложению.

Session (`@SessionScoped`)

область видимости сессии сохраняется для нескольких HTTP-запросов в веб-приложении. Когда приложение состоит из нескольких запросов и ответов, для которых необходимо поддерживать данные, компоненты используют область видимости сессии.

Application (`@ApplicationScoped`)

область видимости приложения хранит данные в течение всего взаимодействия всех пользователей с веб-приложением.

Для получения дополнительной информации об областях видимости в Jakarta Faces см. Использование областей видимости Managed-бинов.

Упаковка и развёртывание веб-модуля hello1

Веб-модуль должен быть упакован в WAR в определённых сценариях развёртывания и всегда, когда вы хотите распространять веб-модуль. Вы можете упаковать веб-модуль в WAR-файл используя Maven или выбранную вами IDE. В этом руководстве показано, как использовать IDE NetBeans и Maven для сборки, упаковки и развёртывания примера приложения `hello1`.

Вы можете развернуть WAR-файл в GlassFish Server:

- Использование IDE NetBeans
- Использование команды `asadmin`
- Использование Консоли администрирования
- Копирование WAR-файла в каталог *`domain-dir`*`/autodeploy/`

В этом руководстве вы будете использовать IDE NetBeans или Maven для упаковки и развёртывания.

Сборка и упаковка веб-модуля hello1 из IDE NetBeans

Чтобы собрать и упаковать веб-модуль `hello1` из IDE NetBeans:

1. Запустите GlassFish Server, как описано в Запуск GlassFish Server с IDE NetBeans, если вы этого ещё не сделали.
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне «Открыть проект» перейдите к:

```
tut-install/examples/web/jsf
```

4. Выберите каталог `hello1` .
5. Нажмите **Открыть проект**.
6. В **Проекты** кликните правой кнопкой мыши вкладку `hello1` и выберите **Сборка**. Эта команда развёртывает проект на сервере.

Сборка и упаковка веб-модуля hello1 с Maven

Сборка и упаковка веб-модуля `hello1` с помощью Maven:

1. Запустите GlassFish Server, как описано в Запуск GlassFish Server из командной строки, если вы этого ещё не сделали.
2. В окне терминала перейдите в:

```
tut-install/examples/web/jsf/hello1/
```

3. Введите следующую команду:

```
mvn install
```
SHELL

Эта команда выполняет компиляцию и создаёт WAR-файл в *`tut-install`*`/examples/web/jsf/hello1/target/` . Затем проект развёртывается на сервере.

Просмотр развёрнутых веб-модулей

GlassFish Server предоставляет два способа просмотра развёрнутых веб-модулей: Консоль администрирования и команда `asadmin` . Вы также можете использовать IDE NetBeans для просмотра развёрнутых модулей.

Просмотр развёрнутых веб-модулей в Консоли администрирования

Чтобы просмотреть развёрнутые веб-модули в Консоли администрирования:

1. Откройте URL http://localhost:4848/ в браузере.
2. Выберите узел Приложения.

   Развёрнутые веб-модули отображаются в таблице «Развёрнутые приложения».

Просмотр развёрнутых веб-модулей с командой asadmin

Введите следующую команду:

```
asadmin list-applications
```
SHELL

Просмотр развёрнутых веб-модулей в IDE NetBeans

Чтобы просмотреть развёрнутые веб-модули в IDE NetBeans:

1. На вкладке **Сервисы** разверните узел **Серверы**, затем узел **GlassFish Server**.
2. Разверните узел **Приложения** для просмотра развёрнутых модулей.

Запуск развёрнутого веб-модуля hello1

Теперь, когда веб-модуль развёрнут, вы можете просмотреть его, открыв приложение в веб-браузере. По умолчанию приложение развёртывается на хосте `localhost` на порту 8080. Корневой контекст веб-приложения — `hello1` .

Чтобы запустить развёрнутый веб-модуль `hello1` :

1. Откройте веб-браузер.
2. Введите следующий URL:

```
http://localhost:8080/hello1/
```

3. В поле введите своё имя и нажмите «Отправить».

   На странице ответа отображается имя, которое вы отправили. Нажмите Назад, чтобы повторить попытку.

Динамическая перезагрузка развёрнутых модулей

Если динамическая перезагрузка включена, не нужно повторно развёртывать приложение или модуль при изменении его кода или дескрипторов развёртывания. Всё, что нужно сделать, это скопировать изменённые страницы или файлы классов в каталог развёртывания приложения или модуля. Каталог развёртывания веб-модуля имеет имя context-root: *`domain-dir`*`/Applications/`*`context-root`* . Сервер периодически проверяет изменения, автоматически и динамически развёртывает приложение с изменениями.

Эта возможность полезна в среде разработки, поскольку позволяет быстро тестировать изменения кода. Однако динамическая перезагрузка не рекомендуется для производственной среды, поскольку она может снизить производительность. Кроме того, всякий раз при перезагрузке сессии становятся недействительными, и клиент должен перезапустить сессию.

В GlassFish Server динамическая перезагрузка включена по умолчанию.

Удаление веб-модуля hello1

Вы можете удалить веб-модули и другие типы корпоративных приложений с IDE NetBeans или Maven.

Удаление веб-модуля hello1 из IDE NetBeans

Чтобы удалить веб-модуль `hello1` из IDE NetBeans:

1. На вкладке **Сервисы** разверните узел **Серверы**, затем узел **GlassFish Server**.
2. Разверните узел **Приложения**.
3. Кликните правой кнопкой мыши модуль `hello1` и выберите **Удаление**.
4. Чтобы удалить файлы классов и другие артефакты сборки, вернитесь к **Проекты**, кликните правой кнопкой мыши проект и выберите **Очистить**.

Удаление веб-модуля hello1 с помощью Maven

Чтобы удалить веб-модуль `hello1` с помощью Maven:

1. В окне терминала перейдите в:

```
tut-install/examples/web/jsf/hello1/
```

2. Введите следующую команду:

SHELL

```
mvn cargo:undeploy
```

3. Чтобы удалить файлы классов и другие артефакты сборки, введите следующую команду:

SHELL

```
mvn clean
```

Веб-модуль с использованием Jakarta Servlet: пример hello2

Приложение `hello2` — это веб-модуль, использующий Jakarta Servlet для отображения приветствия и ответа. Вы можете использовать текстовый редактор или IDE NetBeans для просмотра файлов приложения.

Исходный код этого приложения находится в каталоге *`tut-install`*`/examples/web/servlet/hello2/`.

Маппинг URL на веб-компоненты

Когда веб-контейнер получает запрос, он должен определить, какой веб-компонент этот запрос обрабатывает. Веб-контейнер делает это сопоставлением пути URL, содержащегося в запросе, с веб-приложением и веб-компонентом. Путь URL всегда содержит корень контекста и, необязательно, шаблон URL:

```
http://host:port/context-root[/url-pattern]
```

Вы задаёте шаблон URL для сервлета аннотацией `@WebServlet` в исходном файле сервлета. Например, файл `GreetingServlet.java` в приложении `hello2` содержит следующую аннотацию, в которой шаблон URL указан как `/greeting`:

JAVA

```
@WebServlet("/greeting")
public class GreetingServlet extends HttpServlet {
    ...
}
```

Эта аннотация указывает, что шаблон URL `/greeting` следует за корневым контекстом. Поэтому, когда сервлет развёртывается локально, он доступен по следующему URL:

```
http://localhost:8080/hello2/greeting
```

Чтобы получить доступ к сервлету с помощью только корневого контекста, укажите `"/"` в качестве шаблона URL.

Изучение веб-модуля hello2

Приложение `hello2` ведёт себя почти идентично приложению `hello1`, но оно реализовано с использованием Jakarta Servlet вместо Jakarta Faces. Вы можете использовать текстовый редактор или IDE NetBeans для просмотра файлов приложения.

Просмотр веб-модуля hello2 в IDE NetBeans

Чтобы просмотреть веб-модуль `hello2` с IDE NetBeans:

1. В меню **Файл** выберите **Открыть проект**.
2. В диалоговом окне «Открыть проект» перейдите к:

```
tut-install/examples/web/servlet
```

3. Выберите каталог `hello2` и нажмите **Открыть проект**.
4. Разверните узел **Пакеты исходного кода**, а затем узел `ee.jakarta.tutorial.hello2` .
5. Выполните двойной клик на файле `GreetingServlet.java` чтобы просмотреть его.

   Этот сервлет переопределяет метод `doGet` , реализуя метод `GET` HTTP. Сервлет отображает простую HTML-форму приветствия, кнопка Submit которой, как и в случае с `hello1` , задаёт страницу ответа для своего действия. Следующий фрагмент начинается с аннотации `@WebServlet` , указывающей шаблон URL относительно корневого контекста:

```java
@WebServlet("/greeting")
public class GreetingServlet extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {

        response.setContentType("text/html");
        response.setBufferSize(8192);
        try (PrintWriter out = response.getWriter()) {
            out.println("<html lang=\"en\">"
                    + "<head><title>Servlet Hello</title></head>");

            // запись данных в ответ
            out.println("<body  bgcolor=\"#ffffff\">"
                + "<img src=\"duke.waving.gif\" "
                + "alt=\"Duke waving his hand\">"
                + "<form method=\"get\">"
                + "<h2>Hello, my name is Duke. What's yours?</h2>"
                + "<input title=\"My name is: \"type=\"text\" "
                + "name=\"username\" size=\"25\">"
                + "<p></p>"
                + "<input type=\"submit\" value=\"Submit\">"
                + "<input type=\"reset\" value=\"Reset\">"
                + "</form>");

            String username = request.getParameter("username");
            if (username != null && username.length() > 0) {
                RequestDispatcher dispatcher =
                    getServletContext().getRequestDispatcher("/response");

                if (dispatcher != null) {
                    dispatcher.include(request, response);
                }
            }
            out.println("</body></html>");
        }
    }
    ...
}
```

6. Выполните двойной клик на файле `ResponseServlet.java` чтобы просмотреть его.

   Этот сервлет также переопределяет метод `doGet` , отображая только ответ. Следующий фрагмент начинается с аннотации `@WebServlet` , которая указывает шаблон URL относительно корневого контекста:

JAVA

```
@WebServlet("/response")
public class ResponseServlet extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        try (PrintWriter out = response.getWriter()) {

            // запись данных в ответ
            String username = request.getParameter("username");
            if (username != null && username.length() > 0) {
                out.println("<h2>Hello, " + username + "!</h2>");
            }
        }
    }
    ...
}
```

Запуск hello2

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска примера `hello2`.

Запуск hello2 из IDE NetBeans

Чтобы запустить пример `hello2` с IDE NetBeans:

1. Запустите GlassFish Server, как описано в Запуск GlassFish Server с IDE NetBeans, если вы этого ещё не сделали.
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/web/servlet
   ```

4. Выберите каталог `hello2`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `hello2` и выберите **Сборка**, чтобы упаковать и развернуть проект.
7. В веб-браузере откройте следующий URL:

   ```
   http://localhost:8080/hello2/greeting
   ```

   URL указывает корень контекста, за которым следует шаблон URL.

   Приложение очень похоже на приложение `hello1`. Основное отличие состоит в том, что после клика кнопки «Отправить» ответ появляется под приветствием, а не на отдельной странице.

Запуск hello2 с использованием Maven

Чтобы запустить пример `hello2` с помощью Maven:

1. Запустите GlassFish Server, как описано в Запуск GlassFish Server из командной строки, если вы этого ещё не сделали.
2. В окне терминала перейдите в:

```
tut-install/examples/web/servlet/hello2/
```

3. Введите следующую команду:

```
mvn install
```
SHELL

Она создаёт WAR-файл, копирует его в каталог *tut-install*/examples/web/hello2/target/ и развёртывает его.

4. В веб-браузере откройте следующий URL:

```
http://localhost:8080/hello2/greeting
```

URL указывает корень контекста, за которым следует шаблон URL.

Приложение очень похоже на приложение `hello1`. Основное отличие состоит в том, что после клика кнопки «Отправить» ответ появляется под приветствием, а не на отдельной странице.

Настройка веб-приложений

В этом разделе описываются следующие задачи, связанные с настройкой веб-приложений:

- Установка контекстных параметров
- Объявление файлов приветствия
- Назначение ошибок экранам ошибок
- Объявление ссылок на ресурсы

Установка контекстных параметров

Все веб-компоненты в веб-модуле совместно используют объект, представляющий контекст их приложения. Вы можете передать контекстные параметры в контекст или передать параметры инициализации сервлету. Контекстные параметры доступны для всего приложения. Для получения информации о параметрах инициализации см. Создание и инициализация сервлета.

Добавление контекстного параметра с IDE NetBeans

Эти шаги применяются ко всем веб-приложениям вообще, а не только конкретно к примерам этой главы.

Чтобы добавить контекстный параметр с IDE NetBeans:

1. Откройте проект.
2. Разверните узел проекта в дереве **Проекты**.
3. Разверните узел **Веб-страницы**, а затем узел **WEB-INF**.
4. Выполните двойной клик на `web.xml`.

   Если в проекте нет файла `web.xml`, создайте его, выполнив шаги из Создание файла web.xml в IDE NetBeans.
5. Кликните **Общие** в верхней части окна редактора.
6. Разверните узел **Контекстные параметры**.
7. Кликните **Добавить**.

8. В диалоговом окне **Добавление контекстного параметра**, в поле **Имя параметра** введите имя контексного объекта.
9. В поле **Значение параметра** введите параметр, передающий контекстный объект.
10. Кликните **OK**.

Создание файла web.xml в IDE NetBeans

Чтобы создать файл `web.xml` с IDE NetBeans:

1. В меню **Файл** выберите **Создать файл**.
2. В мастере создания файла выберите категорию **Веб**, затем **Стандартный дескриптор развёртывания** в группе **Типы файлов**.
3. Кликните **Следующий**.
4. Нажмите **Готово**.

   Файл `web.xml` появится в `web/WEB-INF/`.

Объявление файлов приветствия

Механизм файлов приветствия позволяет указать список файлов, которые веб-контейнер может добавить к запросу на URL (называемый допустимым частичным запросом), которому не назначен ни один веб-компонент. Для примера предположим, что вы определяете файл приветствия `welcome.html`. Когда клиент запрашивает URL, например *`host:port/webapp/directory`*, где *`directory`* не соответствует ни одному сервлету или XHTML-странице, клиенту возвращается файл *`host:port/webapp/directory/`*`welcome.html`.

Если веб-контейнер получает валидный частичный запрос, он проверяет список файлов приветствия, добавляет к частичному запросу каждый файл приветствия в указанном порядке и проверяет, назначен ли статический ресурс или сервлет в WAR URL, указанному в запросе. Затем веб-контейнер отправляет запрос первому ресурсу, который есть в WAR.

Если файл приветствия не указан, GlassFish Server будет использовать файл `index.html` в качестве файла приветствия по умолчанию. Если файлов приветствия и файла `index.html` нет, GlassFish Server возвращает список каталогов.

Вы можете указать файлы приветствия в файле `web.xml`. Спецификация файла приветствия для примера `hello1` выглядит следующим образом:

```xml
<welcome-file-list>
    <welcome-file>index.xhtml</welcome-file>
</welcome-file-list>
```

Файл приветствия нужно указывать без начальной и конечной косой черты (`/`).

В примере `hello2` не указан файл приветствия, поскольку запрос URL назначен веб-компоненту `GreetingServlet` через шаблон URL-адреса `/greeting`.

Назначение ошибок экранам ошибок

Если во время выполнения веб-приложения возникает ошибка, вы можете настроить приложение на отображение экрана конкретной ошибки в зависимости от её типа. А именно, можно указать соответствие между кодом состояния, возвращаемым в ответе HTTP, или исключением Java, выбрасываемым любым веб-компонентом, и экраном ошибок любого типа.

В дескрипторе развёртывания может быть указано несколько элементов `error-page`. Каждый элемент идентифицирует отдельную ошибку, которая приводит к открытию страницы ошибки. Эта страница ошибки может быть одинаковой для любого количества элементов `error-page`.

Настройка назначения ошибок экранам ошибок в IDE NetBeans

Эти шаги применяются ко всем веб-приложениям вообще, а не только конкретно к примерам этой главы.

Чтобы настроить отображение ошибок с IDE NetBeans:

1. Откройте проект.
2. Разверните узел проекта на вкладке **Проекты**.
3. Разверните узел **Веб-страницы**, а затем узел **WEB-INF**.
4. Выполните двойной клик на `web.xml`.

   Если в проекте нет файла `web.xml`, создайте его, выполнив шаги из Создание файла web.xml в IDE NetBeans.
5. В верхней части окна редактора выберите **Страницы**.
6. Разверните узел **Страницы ошибок**.
7. Кликните **Добавить**.
8. В диалоговом окне "Добавление страницы ошибки" нажмите кнопку **Просмотр**, чтобы указать страницу, которая будет выступать в качестве страницы ошибки.
9. Укажите либо код ошибки, либо тип исключения.
   - Чтобы указать код ошибки, в поле «Код ошибки» введите код статуса HTTP, который приведёт к открытию страницы ошибки, или оставьте поле пустым, чтобы включить все коды ошибок.
   - Чтобы указать тип исключения, в поле «Тип исключения» введите исключение, которое приведёт к загрузке страницы ошибки. Чтобы указать все генерируемые ошибки и исключения, введите `java.lang.Throwable`.
10. Кликните **ОК**.

Декларирование ссылок на ресурсы

Если ваш веб-компонент использует такие объекты, как Enterprise-бины, источники данных или веб-сервисы, вы можете использовать аннотации Jakarta EE для инъецирования этих ресурсов в ваше приложение. Аннотации позволяют избавиться от большого количества стандартного кода поиска и элементов конфигурации, которые требовались в предыдущих версиях Jakarta EE.

Хотя инъецирование ресурсов с использованием аннотаций может быть более удобным для разработчика, существуют некоторые ограничения на его использование в веб-приложениях. Во-первых, вы можете инъецировать ресурсы только в объекты, управляемые контейнером, так как контейнер должен иметь контроль над созданием компонента, чтобы выполнить инъецирование в него. В результате вы не можете инъецировать ресурсы в такие объекты, как простые JavaBeans. Однако Managed-бины управляются контейнером и, следовательно, для них применимо инъецирование ресурсов.

Компоненты, для которых применимо инъецирование ресурсов, перечислены в таблице 6-1.

В этом разделе объясняется, как использовать несколько поддерживаемых веб-контейнером аннотаций для инъецирования ресурсов. Глава 41 *Запуск примеров персистентности* объясняет, как веб-приложения используют аннотации, поддерживаемые Jakarta Persistence. Глава 51 *Начинаем защиту веб-приложений*

объясняет, как использовать аннотации для защиты веб-приложений. См. главу 56 *Адаптеры и контракты ресурсов*для получения дополнительной информации о ресурсах.

Таблица 6-1. Веб-компоненты, для которых применимо инъецирование ресурсов

Компонент	**Интерфейс/Класс**
Сервлеты	`jakarta.servlet.Servlet`
Фильтры сервлетов	`jakarta.servlet.ServletFilter`

Слушатели событий	`jakarta.servlet.ServletContextListener`
	`jakarta.servlet.ServletContextAttributeListener`
	`jakarta.servlet.ServletRequestListener`
	`jakarta.servlet.ServletRequestAttributeListener`
	`jakarta.servlet.http.HttpSessionListener`
	`jakarta.servlet.http.HttpSessionAttributeListener`
	`jakarta.servlet.http.HttpSessionBindingListener`

Managed-бины	Простые объекты Java

Декларирование ссылки на ресурс

Аннотация `@Resource` используется для объявления ссылки на ресурс, такой как источник данных, Enterprise-бин или другой объект среды выполнения.

Аннотация `@Resource` применяется для класса, метода или поля. Контейнер отвечает за инъецирование ссылок на ресурсы, объявленные аннотацией `@Resource`, и сопоставление их с соответствующими ресурсами JNDI.

В следующем примере аннотация `@Resource` используется для инъецирования источника данных в компонент, которому необходимо установить соединение с источником данных, как это делается при использовании технологии JDBC для доступа к реляционной базе данных:

JAVA

```
@Resource javax.sql.DataSource catalogDS;
public getProductsByCategory() {
    //получение соединения и выполнение запроса
    Connection conn = catalogDS.getConnection();
    ...
}
```

Контейнер инъецирует этот источник данных до того, как компонент станет доступен для приложения. JNDI находит источник данных по его имени поля `catalogDS` и его типу `javax.sql.DataSource`.

Если у вас есть несколько ресурсов, которые нужно инъецировать в один компонент, используйте аннотацию `@Resources`, чтобы содержать их, как показано в следующем примере:

JAVA

```
@Resources ({
    @Resource(name="myDB" type=javax.sql.DataSource.class),
    @Resource(name="myMQ" type=jakarta.jms.ConnectionFactory.class)
})
```

Примеры веб-приложений в данном руководстве используют Jakarta Persistence для доступа к реляционным базам данных. Этот API не требует явного создания соединения с источником данных. Поэтому в примерах не используется аннотация `@Resource` для инъецирования источника данных. Однако этот API поддерживает аннотации `@PersistenceUnit` и `@PersistenceContext` для инъецирования `EntityManagerFactory` и `EntityManager` соответственно. Выполнение примеров персистентности описывает эти аннотации и использование Jakarta Persistence в веб-приложениях.

Декларирование ссылки на веб-сервис

Аннотация `@WebServiceRef` предоставляет ссылку на веб-сервис. В следующем примере показано использование аннотации `@WebServiceRef` для объявления ссылки на веб-сервис. `WebServiceRef` использует элемент `wsdlLocation` для указания URI файла WSDL развёрнутого сервиса:

JAVA

```
...
import jakarta.xml.ws.WebServiceRef;
...
public class ResponseServlet extends HTTPServlet {
@WebServiceRef(wsdlLocation="http://localhost:8080/helloservice/hello?wsdl")
static HelloService service;
```

Дополнительная информация о веб-приложениях

Для получения дополнительной информации о веб-приложениях см.

- Спецификация Jakarta Faces 3.0:
  https://jakarta.ee/specifications/faces/3.0/
- Спецификация Jakarta Servlet 5.0:
  https://jakarta.ee/specifications/servlet/5.0/

Глава 7. Jakarta Faces

Технология Jakarta Faces представляет собой платформу серверных компонентов для создания веб-приложений на Java.

Введение в Jakarta Faces

Jakarta Faces включает в себя следующее:

- API для представления компонентов и управления их состоянием, обработку событий, проверку на стороне сервера и конвертацию данных, определение навигации по страницам, поддержку интернационализации и доступности, обеспечение расширяемости для всех этих функций
- Библиотеки тегов для добавления компонентов на веб-страницы и для подключения компонентов к серверным объектам

Jakarta Faces обеспечивает чётко определённую программную модель и различные библиотеки тегов. Библиотеки тегов содержат обработчики тегов, реализующие теги компонентов. Эти функции значительно облегчают работу по созданию и поддержке веб-приложений с пользовательскими интерфейсами на стороне сервера. Следующие задачи могут быть выполнены с минимальными усилиями.

- Создание веб-страницы.
- Добавление компонентов на веб-страницу с помощью их тегов.
- Связывание компонентов на странице с данными на сервере.
- Создание обработчиков на сервера для генерируемых компонентом событий.
- Сохранение и восстановление состояния приложения после завершения запроса к серверу.
- Повторное использование и расширение компонентов путём кастомизации.

В этой главе представлен обзор Jakarta Faces. После объяснения, что такое приложение Jakarta Faces, и рассмотрения некоторых основных преимуществ использования этой технологии, в этой главе описывается процесс создания простого приложения с её применением. Также эта глава знакомит с жизненным циклом Jakarta Faces, на примере приложения описывая прохождение через этапы жизненного цикла.

Что такое приложение Jakarta Faces?

Функциональные возможности, предоставляемые приложением Jakarta Faces, аналогичны функциям любого другого веб-приложения Java. Типичное приложение Jakarta Faces включает в себя следующие части.

- Набор веб-страниц, на которых размещены компоненты.
- Набор тегов для добавления компонентов на веб-страницу.
- Набор Managed-бинов, которые представляют собой легковесные управляемые контейнером объекты (POJO). В приложении Jakarta Faces Managed-бины служат в качестве вспомогательных бинов, которые определяют свойства и функции для компонентов пользовательского интерфейса на странице.
- Дескриптор развёртывания (файл `web.xml`).
- Необязательно, один или несколько файлов конфигурации приложения, таких как файл `faces-config.xml`, можно использовать для определения правил навигации по страницам и настройки бинов и других кастомных компонентов.

- При желании, набор кастомных объектов, которые могут включать в себя кастомные компоненты, валидаторы, конвертеры или слушатели, созданные разработчиком приложения.
- По желанию, набор кастомных тегов для представления кастомных объектов на странице.

Рисунок 7-1 показывает взаимодействие между клиентом и сервером в типичном приложении Jakarta Faces. В ответ на запрос клиента веб-страница отображается веб-контейнером, который поддерживает Jakarta Faces.

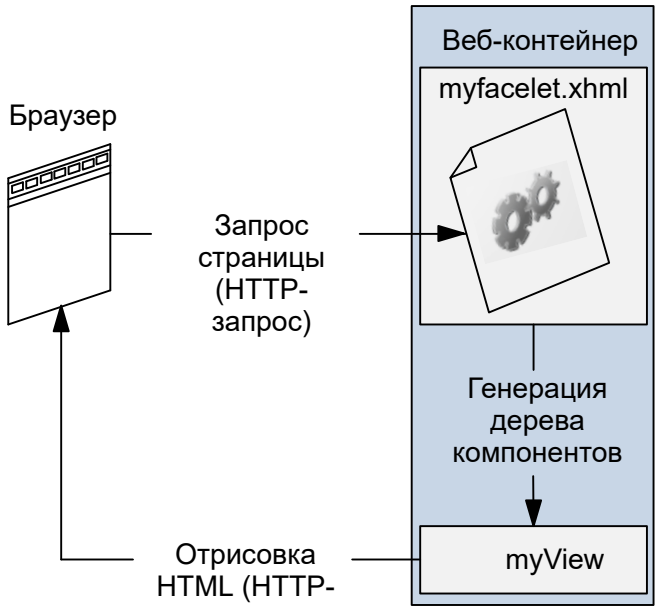

Рис. 7-1. Ответ на запрос клиента к странице Jakarta Faces

Веб-страница `myfacelet.xhtml` построена с использованием тегов Jakarta Faces. Теги используются для добавления компонентов в `view` (представленное на диаграмме `myView`), которое представляет собой представление страницы на стороне сервера. В дополнение к компонентам веб-страница может также ссылаться на объекты, такие как:

- Любые слушатели событий, валидаторы и конвертеры, зарегистрированные с компонентами
- Компоненты JavaBeans, которые собирают данные и выполняют специфичную для приложения обработку.

Представление отрисовывается клиенту в качестве ответа на его запрос. Отрисовка — это процесс, при котором веб-контейнер на основе серверного представления генерирует выходные данные, такие как HTML или XHTML, которые могут быть прочитаны на клиенте, например, браузером.

Преимущества Jakarta Faces

Одним из величайших преимуществ технологии Jakarta Faces является то, что она предлагает чёткое разделение между поведением и представлением для веб-приложений. Приложение Jakarta Faces может сопоставлять HTTP-запросы с обработчиками событий для конкретного компонента и управлять компонентами как объектами с состоянием (stateful) на сервере. Jakarta Faces позволяет создавать веб-приложения, в которых реализовано более тонкое разделение поведения и представления, которое традиционно предлагается клиентскими архитектурами пользовательского интерфейса.

Отделение логики от представления также позволяет каждому участнику команды разработчиков веб-приложений сосредоточиться на одной части процесса разработки и предоставляет простую программную модель для связывания частей. Например, авторы страниц, не имеющие опыта программирования, могут использовать теги Jakarta Faces на веб-странице для ссылки на серверные объекты без написания каких-либо сценариев.

Другая важная цель Jakarta Faces — использовать знакомые концепции компонентов и веб-уровня, не ограничивая вас определённой технологией сценариев или языком разметки. API Jakarta Faces размещаются непосредственно над Servlet API, как показано на рис. 7-2.

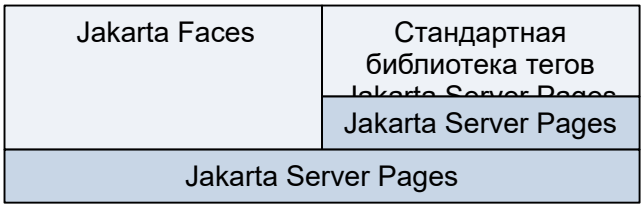

Рисунок 7-2 Технологии веб-приложений

Такое многослойное API позволяет приложению иметь несколько различных вариантов использования, таких как совмещение различных технологий представления, создание собственных кастомных компонентов непосредственно из классов компонентов и генерация вывода для различных клиентских устройств.

Технология Facelets, доступная как часть Jakarta Faces, более предпочтительна для создания веб-приложений на основе Jakarta Faces. Для получения дополнительной информации о функциях технологии Facelets см. главу 8 *Введение в Facelets*.

Технология Facelets предлагает несколько преимуществ.

- Код компонентов может быть повторно использован и расширен с применением шаблонов и составных компонентов.
- Вы можете использовать аннотации для автоматической регистрации Managed-бина в качестве ресурса, доступного для приложений Jakarta Faces. Кроме того, неявные правила навигации позволяют разработчикам быстро настраивать навигацию по страницам (подробности см. Модели навигации). Эти функции упрощают процесс ручной настройки приложений.
- Самое главное, Jakarta Faces предоставляет богатую архитектуру для управления состоянием компонентов, обработки данных компонентов, валидации ввода данных пользователем и обработки событий.

Простое приложение Jakarta Faces

Jakarta Faces обеспечивает простой и удобный процесс создания веб-приложений. Для разработки простого приложения Jakarta Faces обычно требуются следующие шаги, которые уже были описаны в Веб-модуле с использованием Jakarta Faces: пример hello1:

- Создание веб-страниц с использованием тегов
- Разработка Managed-бинов
- Отображение объекта `FacesServlet`

Пример `hello1` включает в себя Managed-бин и две веб-страницы Facelets. При обращении клиента первая веб-страница запрашивает у пользователя его имя, а вторая отвечает сообщением приветствия.

Подробнее о технологии Facelets см. главу 8 *Введение в Facelets*. Для получения подробной информации об использовании выражений EL см. главу 9 *Язык выражений*. Подробнее о модели программирования Jakarta Faces и создании веб-страниц с использованием Jakarta Faces см. главу 10 *Использование Jakarta Faces на веб-страницах*.

Каждое веб-приложение имеет жизненный цикл. Общие задачи, такие как обработка входящих запросов, параметры декодирования, изменение и сохранение состояния и отображение веб-страниц в браузере — все они выполняются в течение жизненного цикла веб-приложения. Некоторые фреймворки веб-приложений скрывают от вас детали жизненного цикла, тогда как другие требуют, чтобы вы управляли ими вручную.

По умолчанию Jakarta Faces автоматически выполняет большинство действий на протяжении жизненного цикла. Тем не менее, он также предоставляет различные этапы жизненного цикла запроса, чтобы можно было изменять или выполнять различные действия, если этого требуют особенности приложения.

Жизненный цикл приложения Jakarta Faces начинается и заканчивается следующим действием: клиент делает запрос на веб-страницу, а сервер отвечает страницей. Жизненный цикл состоит из двух основных фаз: выполнения и отрисовки.

В фазе выполнения могут выполняться несколько действий.

- Собирается или восстанавливается представление приложения.
- Применяются значения параметров запроса.
- Выполняются преобразование и валидация значений компонентов.
- Managed-бины обновляются значениями компонентов.
- Отрабатывает логика приложения.

Для первого (начального) запроса строится только представление. Для последующих запросов могут выполняться некоторые или все другие действия.

На этапе отрисовки запрошенное представление отображается как ответ клиенту. Отрисовка обычно представляет собой процесс генерации вывода, такого как HTML или XHTML, который может быть прочитан клиентом, обычно браузером.

Следующее краткое описание примера приложения Jakarta Faces с прохождением его по жизненному циклу суммирует действия, которые происходят за кулисами.

Приложение `hello1` проходит следующие этапы при его развёртывании в GlassFish Server.

1. Когда приложение `hello1` собрано и развёрнуто в GlassFish Server, приложение находится в неинициализированном состоянии.
2. Когда клиент делает начальный запрос веб-страницы `index.xhtml` , приложение Facelets `hello1` Facelets компилируется.
3. Скомпилированное приложение Facelets выполняется, для приложения `hello1` создаётся новое дерево компонентов и помещается в `FacesContext` .
4. Дерево компонентов заполняется компонентом и связанным с ним свойством Managed-бина, представленным выражением EL `hello.name` .
5. Новый вид строится на основе дерева компонентов.
6. Представление отрисовывается и передаётся запрашивающему клиенту в качестве ответа.
7. Дерево компонентов автоматически уничтожается.
8. При последующих запросах дерево компонентов строится вновь и применяется сохранённое состояние.

Для получения полной информации о жизненном цикле см. Жизненный цикл приложения Jakarta Faces.

Модель компонентов пользовательского интерфейса

Помимо описания жизненного цикла обзор архитектуры Jakarta Faces обеспечивает лучшее понимание технологии.

Компоненты Jakarta Faces являются строительными блоками представления Jakarta Faces. Компонент может быть компонентом пользовательского интерфейса (UI) или компонентом не-UI.

Компоненты пользовательского интерфейса Jakarta Faces — это настраиваемые, повторно используемые элементы, которые составляют пользовательские интерфейсы приложений Jakarta Faces. Компонент может быть простым, например кнопка, или составным, например таблица, состоящая из нескольких компонентов.

Jakarta Faces предоставляет богатую, гибкую компонентную архитектуру, которая включает в себя следующее:

- Набор классов `jakarta.faces.component.UIComponent` для указания состояния и поведения компонентов пользовательского интерфейса
- Модель отрисовки, которая определяет различные способы отрисовки компонента.
- Модель конвертации, которая определяет, как регистрировать конвертеры данных в компоненте.
- Модель события и слушателя, которая определяет, как обрабатывать события компонента
- Модель валидации, которая определяет, как регистрировать валидаторы в компоненте

В этом разделе кратко описывается каждый из этих архитектурных компонентов.

Классы компонентов пользовательского интерфейса

Jakarta Faces предоставляет набор классов компонентов пользовательского интерфейса и связанных поведенческих интерфейсов, которые задают все функциональные возможности компонента пользовательского интерфейса, такие как сохранение состояния компонента, поддержку ссылки на объекты, управление обработкой событий и отрисовкой для набора стандартных компонентов.

Классы компонентов полностью расширяемы, что позволяет разработчику создавать свои собственные кастомные компоненты. См. главу 15 *Создание кастомных компонентов пользовательского интерфейса и других кастомных объектов* для получения дополнительной информации.

`jakarta.faces.component.UIComponent` — абстрактный базовый класс для всех компонентов. Классы компонентов пользовательского интерфейса Jakarta Faces расширяют класс `UIComponentBase` (дочерний класс `UIComponent`), который определяет состояние и поведение компонента по умолчанию. Следующий набор классов компонентов включён в Jakarta Faces.

- `UIColumn`: представляет один столбец данных в компоненте `UIData`.
- `UICommand`: представляет элемент управления, который запускает действия при активации.
- `UIData`: представляет связь для коллекции данных, представленных объектом `jakarta.faces.model.DataModel`.
- `UIForm`: представляет форму ввода для отображения пользователю. Его дочерние компоненты представляют (среди прочего) поля ввода, которые должны быть включены при отправке формы. Этот компонент аналогичен тегу `form` в HTML.
- `UIGraphic`: отображает изображение.
- `UIInput`: принимает данные от пользователя. Этот класс является дочерним классом `UIOutput`.
- `UIMessage`: отображает локализованное сообщение об ошибке.

- `UIMessages` : отображает набор локализованных сообщений об ошибках.
- `UIOutcomeTarget` : отображает ссылку в виде ссылки или кнопки.
- `UIOutput` : отображает вывод данных на странице.
- `UIPanel` : управляет макетом его дочерних компонентов.
- `UIParameter` : представляет параметры замещения.
- `UISelectBoolean` : позволяет пользователю установить значение `boolean` для элемента управления выбором или отменой выбора. Этот класс является дочерним классом `UIInput` .
- `UISelectItem` : представляет отдельный элемент выбора в списке элементов.
- `UISelectItems` : представляет весь список элементов выбора.
- `UISelectMany` : позволяет пользователю множественный выбора из группы элементов. Этот класс является дочерним классом `UIInput` .
- `UISelectOne` : позволяет пользователю выбрать один элемент из группы элементов. Этот класс является дочерним классом `UIInput` .
- `UIViewParameter` : представляет параметры в запросе. Этот класс является дочерним классом `UIInput` .
- `UIViewRoot` : представляет корень дерева компонентов.

В дополнение к расширению `UIComponentBase` классы компонентов также реализуют один или несколько поведенческих интерфейсов, каждый из которых задаёт определённое поведение для набора компонентов, чьи классы реализуют интерфейс.

Эти поведенческие интерфейсы определены в пакете `jakarta.faces.component` , если не указано иное.

- `ActionSource` : указывает, что компонент может инициировать событие действия. Этот интерфейс предназначен для использования с компонентами на основе JavaServer Faces 1.1_01 и более ранних версий. Этот интерфейс считается устаревшим в JavaServer Faces 2.
- `ActionSource2` : расширяет `ActionSource` и, следовательно, предоставляет те же функциональные возможности. Однако он позволяет компонентам использовать язык выражений (EL), когда те ссылаются на методы, которые обрабатывают события действия.
- `EditableValueHolder` : расширяет `ValueHolder` и задаёт дополнительные функции для редактируемых компонентов, такие как валидация и отправка событий изменения значения.
- `NamingContainer` : требует, чтобы каждый расширяющий его компонент имел уникальный идентификатор.
- `StateHolder` : обозначает, что компонент имеет состояние, которое должно быть сохранено между запросами.
- `ValueHolder` : указывает, что компонент поддерживает локальное значение, а также возможность доступа к данным слоя модели.
- `jakarta.faces.event.SystemEventListenerHolder` : поддерживает список объектов `jakarta.faces.event.SystemEventListener` для каждого типа `jakarta.faces.event.SystemEvent` и определяется этим классом.
- `jakarta.faces.component.behavior.ClientBehaviorHolder` : добавляет возможность прикрепить `jakarta.faces.component.behavior.ClientBehavior` , такие как повторно используемые скрипты.

`UICommand` реализует `ActionSource2` и `StateHolder`. `UIOutput` и классы компонентов, которые расширяют `UIOutput`, реализуют `StateHolder` и `ValueHolder`. `UIInput` и классы компонентов, которые расширяют `UIInput`, реализуют `EditableValueHolder`, `StateHolder` и `ValueHolder`. `UIComponentBase` реализует `StateHolder`.

Только составители компонентов должны будут напрямую использовать классы компонентов и поведенческие интерфейсы. Авторы страниц и разработчики приложений будут использовать стандартный компонент, включая тег, который представляет его на странице. Большинство компонентов могут отображаться на странице различными способами. Например, компонент `UICommand` может быть отображен как кнопка или ссылка.

В следующем разделе объясняется, как работает модель отрисовки и как разработчики могут выбирать компоненты для отрисовки, применяя соответствующие теги.

Модель отрисовки компонентов

Компонентная архитектура Jakarta Faces спроектирована таким образом, что функциональность компонентов определяется классами компонентов, тогда как их отрисовка может быть определена отдельным классом. Эта конструкция имеет следующие преимущества.

- Компоненты записи могут определять поведение компонента один раз, но создавать несколько отрисовщиков, каждый из которых определяет свой способ отрисовки компонента одному и тому же клиенту или разным клиентам.
- Разработчики могут изменить внешний вид компонента на странице, выбрав тег, который представляет соответствующую комбинацию компонента и отрисовщика.

Инструмент отрисовки устанавливает соответствие классов компонентов и тегов компонентов, которые подходят для конкретного клиента. Jakarta Faces включает стандартный комплект отрисовки HTML для клиентов HTML.

Инструментарий отрисовки определяет набор `jakarta.faces.render.Renderer` для каждого поддерживаемого компонента. Каждый класс `Renderer` определяет свой способ отрисовки конкретного компонента в вывод, определённый инструментом отрисовки. Например, компонент `UISelectOnen` имеет три разных отрисовщика. Один из них отображает компонент как группу параметров. Другой отрисовывает компонент как поле с выпадающим списком. Третий отрисовывает компонент в виде списка. Аналогично, компонент `UICommand` может быть отображен как кнопка или ссылка, используя тег `h:commandButton` или `h:commandLink`. Часть `command` каждого тега соответствует классу `UICommand`, определяющему функциональность, которая должна выполняться в ответ на действие. Часть `Button` или `Link` каждого тега соответствует отдельному классу `Renderer`, который определяет, как компонент отображается на странице.

Каждый кастомный тег, определённый в стандартном инструменте отрисовки HTML, состоит из функциональных возможностей компонента (определённых в классе `UIComponent`) и атрибутов отрисовки (определённых в классе `Renderer`).

В разделе Добавление компонентов на страницу с использованием библиотеки тегов HTML перечислены все поддерживаемые теги компонентов и показано, как использовать их в примере.

Jakarta Faces предоставляет пользовательскую библиотеку тегов для отображения компонентов в HTML.

Модель конвертации

Приложение Jakarta Faces может при желании связать компонент с данными серверного объекта. Этот объект является компонентом JavaBeans, например Managed-бином. Приложение получает и устанавливает данные объекта для компонента, вызывая соответствующие свойства объекта для этого компонента.

Когда компонент связан с объектом, приложение имеет два представления данных компонента.

- Представление модели, в котором данные представлены в виде типов данных, таких как `int` или `long`.
- Представление даёт возможность пользователю считывать и изменять модель. Например, `java.util.Date` может быть представлен в виде текстовой строки в формате `mm/dd/yy` или в виде набора из трёх текстовых строк.

Jakarta Faces автоматически преобразует данные компонента между этими двумя представлениями, когда свойство компонента, связанное с компонентом, относится к одному из типов, поддерживаемых данными компонента. Например, если компонент `UISelectBoolean` связан со свойством бина типа `java.lang.Boolean`, Jakarta Faces автоматически преобразует данные компонента из `String` в `Boolean`. Кроме того, некоторые данные компонента должны быть связаны со свойствами определённого типа. Например, компонент `UISelectBoolean` должен быть связан со свойством типа `boolean` или `java.lang.Boolean`.

Иногда может потребоваться конвертировать данные компонента в тип, отличный от стандартного, или изменить формат данных. Чтобы облегчить это, технология Jakarta Faces позволяет зарегистрировать реализацию `jakarta.faces.convert.Converter` на компонентах `UIOutput` и компонентах, классы которых являются подклассами `UIOutput`. Если вы зарегистрируете реализацию `Converter` в компоненте, эта реализация `Converter` будет конвертировать данные компонента между двумя представлениями.

Вы можете использовать стандартные конвертеры, поставляемые с Jakarta Faces, или создать свой кастомный конвертер. Создание кастомного конвертера рассматривается в главе 15 *Создание кастомных компонентов пользовательского интерфейса и других кастомных объектов*.

Модель событий и слушателей

Модель событий и слушателелей Jakarta Faces похожа на модель событий JavaBeans в том, что она имеет строго типизированные классы событий и интерфейсы слушателей, которые приложение может использовать для обработки событий, генерируемых компонентами.

Спецификация Jakarta Faces определяет три типа событий: события приложения, системные события и события модели данных.

События приложения связаны с конкретным приложением и генерируются `UIComponent`. Они представляют стандартные события, доступные в предыдущих версиях Jakarta Faces.

Объект события идентифицирует компонент, сгенерировавший событие, и хранит информацию о событии. Чтобы получить уведомление о событии, приложение должно предоставить реализацию класса слушателя и зарегистрировать его в компоненте, который генерирует событие. Когда пользователь активирует компонент, например кликом кнопки, происходит событие. Это заставляет Jakarta Faces вызывать метод слушателя, который обрабатывает событие.

Jakarta Faces поддерживает два вида событий приложения: события действия и события изменения значения.

Событие действия (класс `jakarta.faces.event.ActionEvent`) срабатывает, когда пользователь активирует компонент, реализующий `ActionSource`. Это кнопки и ссылки.

Событие изменения значения (класс `Jakarta.faces.Event.ValueChangeEvent`) срабатывает, когда пользователь изменяет значение компонента, представленного `UIInput` или одним из его подклассов. Например, выбор независимого переключателя (check box) является действием, которое приводит к изменению значения компонента на `true` . Типы компонентов, которые могут генерировать события этих типов, — это компоненты `UIInput` , `UISelectOne` , `UISelectMany` и `UISelectBoolean` . События изменения значения запускаются только в том случае, если не было обнаружено ошибок валидации.

В зависимости от значения свойства `immediate` (см. Атрибут immediate) компонента, генерирующего событие, события действия могут быть обработаны в фазе вызова приложения или применения значений параметров запроса. События изменения значения могут быть обработаны в фазе валидации процесса или применения значений параметров запроса.

Системные события генерируются `Object` -ом, а не `UIComponent` -ом. Они генерируются во время выполнения приложения в заранее определённое время. Они применимы ко всему приложению, а не к конкретному компоненту.

Событие модели данных происходит, когда выбирается новая строка компонента `UIData` .

Есть два способа заставить ваше приложение реагировать на события действия или события изменения значения, которые генерируются стандартным компонентом:

- Реализуйте класс слушателя события для обработки события и зарегистрируйте слушателя в компоненте, вложив тег `f:valueChangeListener` или тег `f:actionListener` в тег компонента.
- Реализуйте метод Managed-бина для обработки события и обратитесь к методу с выражением метода из соответствующего атрибута тега компонента.

Смотрите Реализация слушателя событий для получения информации о том, как реализовать слушатель событий. Смотрите Регистрация слушателей в компонентах для получения информации о том, как зарегистрировать слушателя в компоненте.

Смотрите Пишем метод-обработчик действия и Пишем метод-обработчик изменения значения для получения информации о том, как реализовать методы Managed-бина, которые обрабатывают эти события.

Смотрите Ссылка на метод Managed-бина для получения информации о том, как ссылаться на метод Managed-бина из тега компонента.

При генерации событий кастомными компонентами необходимо реализовать соответствующий класс событий и вручную поставить событие в очередь в дополнение к реализации класса слушателя событий или метода Managed-бина, который обрабатывает событие. Обработка событий для кастомных компонентов объясняет, как это сделать.

Модель валидации

Jakarta Faces поддерживает механизм валидации локальных данных редактируемых компонентов (таких как текстовые поля). Эта валидация происходит перед обновлением соответствующих данных модели в соответствии с локальным значением.

Как и модель конвертации, модель валидации определяет набор стандартных классов для выполнения общих проверок данных. Библиотека основных тегов Jakarta Faces также определяет набор тегов, соответствующих стандартным реализациям `jakarta.faces.validator.Validator` . Смотрите Использование стандартных валидаторов для получения списка всех стандартных классов валидации и соответствующих тегов.

Большинство тегов имеют набор атрибутов для настройки свойств валидатора, таких как минимальные и максимальные допустимые значения для данных компонента. Автор страницы регистрирует валидатор в компоненте, вкладывая тег валидатора в тег компонента.

В дополнение к валидаторам, которые зарегистрированы в компоненте, вы можете объявить валидатор по умолчанию, который зарегистрирован во всех компонентах `UIInput` в приложении. Для получения дополнительной информации о валидаторах по умолчанию см. Использование валидаторов по умолчанию.

Модель валидации также позволяет создавать кастомные валидаторы и соответствующие теги для выполнения кастомной валидации. Модель валидации предоставляет два способа реализации кастомной валидации.

- Реализуйте интерфейс `Validator` , который выполняет валидацию.
- Реализуйте метод Managed-бина, который выполняет валидацию.

При реализации интерфейса `Validator` вы также должны сделать следующее.

- Зарегистрируйте реализацию `Validator` в приложении.
- Создайте кастомный тег или используйте тег `f:validator` для регистрации средства валидации в компоненте.

В ранее описанной стандартной модели валидации валидатор определяется для каждого компонента ввода на странице. Модель Bean Validation позволяет применять валидатор ко всем полям на странице. См. главу 23 *Введение в Jakarta Bean Validation* и главу 24 *Bean Validation: Дополнительные темы* для получения дополнительной информации о Bean Validation.

Модель навигации

Модель навигации Jakarta Faces позволяет легко задать навигацию по страницам и выполнить любую дополнительную обработку, необходимую для определения последовательности загрузки страниц.

В Jakarta Faces навигация — это набор правил для выбора следующей страницы или представления, которое будет отображаться после выполнения обработчика действия, например, клика кнопки или ссылки.

Навигация может быть неявной или задана пользователем. Неявная навигация вступает в игру, когда правила навигации не заданы пользователем в файлах конфигурации приложения.

Когда вы добавляете такой компонент, как `commandButton` на страницу Facelets, и назначаете другую страницу в качестве значения для свойства `action` , обработчик навигации по умолчанию будет пытаться неявно установить ей в соответствие подходящую страница приложения. В следующем примере обработчик навигации по умолчанию попытается найти страницу с именем `response.xhtml` в приложении и перейти к ней:

```xml
<h:commandButton value="submit" action="response">
```

Определяемые пользователем правила навигации объявляются в таких файлах конфигурации приложения, как `faces-config.xml` . Структура правила навигации по умолчанию выглядит следующим образом:

```xml
<navigation-rule>
    <description></description>
    <from-view-id></from-view-id>
    <navigation-case>
        <from-action></from-action>
        <from-outcome></from-outcome>
        <if></if>
        <to-view-id></to-view-id>
    </navigation-case>
</navigation-rule>
```

Пользовательская навигация обрабатывается следующим образом.

- Задайте правила в файле конфигурации приложения.
- Обратитесь к строковому результату из атрибута `action` кнопки или компонента ссылки. Этот результат `String` используется Jakarta Faces для выбора правила навигации.

Вот пример правила навигации:

```xml
<navigation-rule>
    <from-view-id>/greeting.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/response.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

Это правило гласит, что когда компонент command (такой как `h:commandButton` или `h:commandLink`) в `greeting.xhtml` активирован, приложение перейдёт со страницы `greeting.xhtml` на страницу `response.xhtml`, если результатом вычисления обработчика компонента кнопки будет `success`. Вот тег `h:commandButton` из `greeting.xhtml`, который будет указывать логический результат `success`:

```xml
<h:commandButton id="submit" value="Submit" action="success"/>
```

Как показано в примере, каждый элемент `navigation-rule` определяет, как добраться с одной страницы (указанной в элементе `from-view-id`) на другие страницы приложения. Элементы `navigation-rule` могут содержать любое количество элементов `navigation-case`, каждый из которых определяет следующую страницу, которую нужно открыть (определяется `to-view-id`) на основе логического результата (определяется как `from-outcome`).

В более сложных приложениях логический результат также может быть получен из возвращаемого методом-обработчиком значения в Managed-бине. Этот метод выполняет некоторую обработку для определения результата. Например, метод может проверить, совпадает ли пароль, введённый пользователем на странице, с паролем в файле. Если это так, метод может вернуть `success`. В противном случае он может вернуть `failure`. В результате `failure` может произойти перезагрузка страницы входа. В результате `success` может открыться страница, отображающая активность по кредитной карте пользователя. Если вы хотите, чтобы результат был возвращён методом в компоненте, нужно обратиться к методу, используя выражение метода в атрибуте `action`, как показано в этом примере:

```xml
<h:commandButton id="submit" value="Submit"
                 action="#{cashierBean.submit}" />
```

Когда пользователь кликает кнопку, представленную этим тегом, соответствующий компонент генерирует событие действия. Это событие обрабатывается по умолчанию `jakarta.faces.event.ActionListener`, который вызывает метод action, на который ссылается компонент, инициировавший событие. Метод-обработчик возвращает логический результат слушателю.

Слушатель передает результат и ссылку на метод action, который привёл результат по умолчанию `jakarta.faces.application.NavigationHandler`. `NavigationHandler` выбирает следующую страницу для отображения, сопоставляя результат или ссылку на метод-обработчик с правилами навигации в файле конфигурации приложения с помощью следующего процесса.

1. `NavigationHandler` выбирает правило навигации, соответствующее отображаемой странице.
2. Он соответствует результату или ссылке на метод action, которую он получил от `jakarta.faces.event.ActionListener` с теми, которые определены в настройках навигации.
3. Он пытается сопоставить и ссылку на метод, и результат с одним и тем же правилом навигации.
4. Если предыдущий шаг завершается неудачно, обработчик навигации пытается сопоставить результат.
5. Наконец, обработчик навигации пытается сопоставить ссылку на метод-обработчик, если предыдущие две попытки потерпели неудачу.
6. Если ни один из вариантов навигации не совпадает, он снова отображает тот же вид.

Когда `NavigationHandler` устанавливает соответствия, начинается фаза отрисовки ответа. В этой фазе будет отображена страница, выбранная `NavigationHandler`.

Приложение-пример Duke's Tutoring использует правила навигации в бизнес-методах, которые обрабатывают создание, редактирование и удаление пользователей приложения. Например, форма для создания студента имеет тег `h:commandButton`:

XML

```
<h:commandButton id="submit"
        action="#{adminBean.createStudent(studentManager.newStudent)}"
        value="#{bundle['action.submit']}"/>
```

Событие действия вызывает метод `dukestutoring.ejb.AdminBean.createStudent`:

JAVA

```
public String createStudent(Student student) {
    em.persist(student);
    return "createdStudent";
}
```

Возвращаемое значение `creationStudent` имеет соответствующее правило навигации в файле конфигурации `faces-config.xml`:

XML

```
<navigation-rule>
    <from-view-id>/admin/student/createStudent.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>createdStudent</from-outcome>
        <to-view-id>/admin/index.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

После создания студента пользователь возвращается на начальную страницу администрирования.

Для получения дополнительной информации о том, как задать правила навигации, см. Настройка правил навигации.

Для получения дополнительной информации о том, как реализовать методы-обработчики для обработки навигации, смотрите Пишем метод-обработчик действия.

Для получения дополнительной информации о том, как ссылаться на результаты или методы действий из тегов компонента, см. Ссылка на метод, который выполняет навигацию.

Жизненный цикл приложения Jakarta Faces

Жизненный цикл приложения описывает различные этапы обработки этого приложения, от его инициирования до завершения. Жизненный цикл имеют все приложения. В течение жизненного цикла веб-приложения выполняются общие задачи, включая следующие.

- Обработка входящих запросов
- Декодирование параметров
- Изменение и сохранение состояния
- Отображение веб-страниц в браузере

Фреймворк Jakarta Faces автоматически управляет фазами жизненного цикла для простых приложений или позволяет управлять ими вручную (при необходимости) для более сложных приложений.

Приложения Jakarta Faces, использующие расширенные функции, могут требовать взаимодействия с жизненным циклом в определённых фазах. Например, приложения Ajax используют функции частичной обработки жизненного цикла (см. Частичная обработка и частичная отрисовка). Более чёткое понимание фаз жизненного цикла является ключом к созданию хорошо спроектированных компонентов.

Упрощённое представление жизненного цикла Jakarta Faces, состоящего из двух основных этапов веб-приложения Jakarta Faces, представлено в Простом приложении Jakarta Faces. В этом разделе более подробно рассматривается жизненный цикл Jakarta Faces.

Обзор жизненного цикла Jakarta Faces

Жизненный цикл приложения Jakarta Faces начинается, когда клиент делает HTTP-запрос на страницу, и заканчивается, когда сервер отвечает HTML-страницей.

Жизненный цикл можно разделить на две основные фазы: выполнение и отрисовку. Фаза выполнения дополнительно разделена на подфазы для поддержки дерева компонентов. Эта структура требует, чтобы данные компонента были конвертированы и провалидированы, события компонента были обработаны, а данные компонента были переданы в объекты бинов определённым образом.

Страница Jakarta Faces представлена деревом компонентов, которое называется представлением. В течение жизненного цикла Jakarta Faces должна создавать представление с учётом состояния, сохранённого во время предыдущей отправки страницы. Когда клиент запрашивает страницу, Jakarta Faces выполняет несколько задач, таких как валидация ввода данных компонентов в представлении и конвертацию входных данных в типы, указанные на стороне сервера.

Jakarta Faces выполняет все эти задачи по шагам в жизненном цикле запрос-ответ. Рисунок 7-3 иллюстрирует эти шаги.

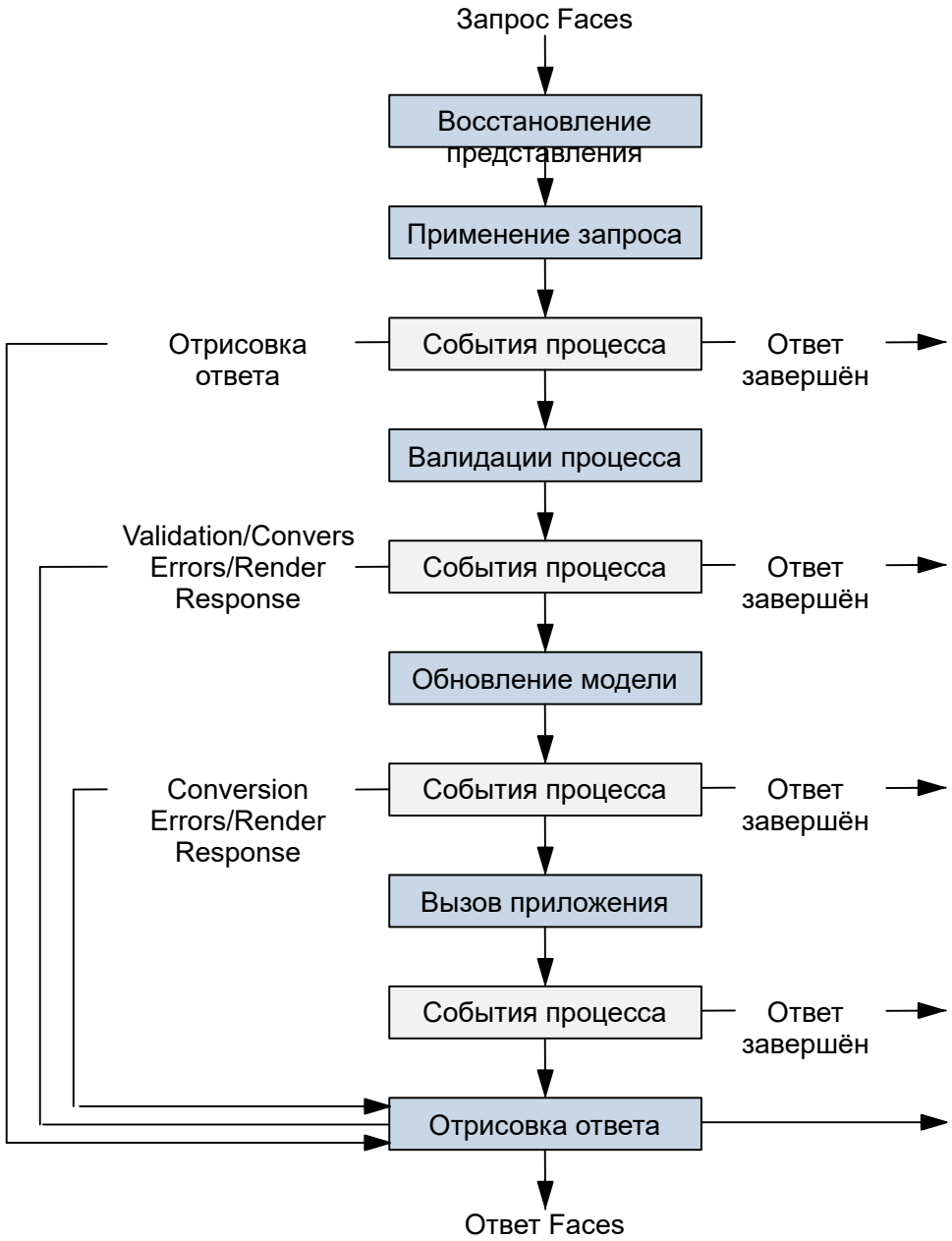

Рисунок 7-3. Стандартный жизненный цикл запроса-ответа Jakarta Faces

Жизненный цикл запрос-ответ обрабатывает два вида запросов: начальные и повторные. Начальный запрос возникает, когда пользователь впервые запрашивает страницу. Повторный запрос возникает, когда пользователь отправляет форму, содержащуюся на странице, которая ранее была загружена в браузер в результате выполнения начального запроса.

Когда жизненный цикл обрабатывает начальный запрос, он выполняет только фазы восстановления представления и отрисовки ответа, потому что отсутствует пользовательский ввод или действия для обработки. И наоборот, когда жизненный цикл обрабатывает повторный запрос, он выполняет все фазы.

Обычно первый запрос на страницу Jakarta Faces поступает от клиента в результате клика ссылки или кнопки на странице Jakarta Faces. Для отрисовки ответа, являющегося другой страницей Jakarta Faces, приложение создаёт новое представление и сохраняет его в `jakarta.faces.context.FacesContext<`, который представляет всю информацию, связанную с обработкой входящего запроса и созданием ответа. Затем приложение получает ссылки на объекты, необходимые представлению, и вызывает метод `FacesContext.renderResponse`, который вызывает немедленную отрисовку представления путём перехода к фазе отрисовки ответа, как показано стрелками с надписью Render Response на рис. 7-3.

Иногда приложению может потребоваться перенаправить на другой ресурс веб-приложения, например веб-сервис, или сгенерировать ответ, который не содержит компонентов Jakarta Faces. В этих ситуациях разработчик должен пропустить фазу отрисовки ответа, вызвав метод `FacesContext.responseComplete`. Эта ситуация также показана — стрелками, помеченными как Response Complete.

Наиболее распространена ситуация, когда компонент Jakarta Faces отправляет запрос на другую страницу Jakarta Faces. В этом случае Jakarta Faces обрабатывает запрос и автоматически проходит фазы жизненного цикла, чтобы выполнить любые необходимые конвертации, валидации и обновления модели, а также сгенерировать ответ.

Существует одно исключение из жизненного цикла, описанного в этом разделе. Когда для атрибута `immediate` компонента установлено значение `true`, валидация, конвертация и события, связанные с этими компонентами, обрабатываются в фазе применения значений запроса, а не в более поздней фазе.

Детали жизненного цикла, описанные в следующих разделах, в первую очередь предназначены для разработчиков, которым необходимо знать информацию, например, когда обычно обрабатываются валидации, конвертации и события, а также способы изменения того, как и когда они обрабатываются. Для получения дополнительной информации о каждой из фаз жизненного цикла загрузите последнюю версию документации Jakarta Faces Specification по ссылкеhttps://jakarta.ee/specifications/faces/.

Фаза выполнения жизненного цикла приложения Jakarta Faces содержит следующие подфазы:

- Фаза восстановления представления
- Фаза применения значений запроса
- Фаза валидации процесса
- Фаза обновления значений модели
- Фаза вызова приложения
- Фаза отрисовки ответа

Фаза восстановления представления

Когда выполняется запрос на страницу Jakarta Faces, обычно с помощью какого-либо действия, например, при клике по ссылке или кнопке, Jakarta Faces начинает фазу восстановления представления.

На этом этапе Jakarta Faces создаёт представление страницы, связывает обработчики событий и средства валидации с компонентами в представлении и сохраняет представление в объекте `FacesContext`, который содержит всю информацию, необходимую для обработки одиночного запроса. Все компоненты приложения, обработчики событий, конвертеры и валидаторы имеют доступ к объекту `FacesContext`.

Если запрос к странице является начальным запросом, Jakarta Faces создаёт пустое представление на этом этапе, и жизненный цикл переходит к фазе отрисовки ответа, во время которого пустое представление заполняется компонентами, на которые ссылаются теги на странице.

Если запрос на страницу — повторный запрос, представление, соответствующее этой странице, уже существует в объекте `FacesContext`. В этой фазе Jakarta Faces восстанавливает представление, используя информацию о состоянии, сохранённую на клиенте или сервере.

Фаза применения значений запроса

После восстановления дерева компонентов во время повторного запроса каждый компонент в дереве извлекает своё новое значение из параметров запроса, используя метод `decode (processDecodes())`. Затем значение сохраняется локально для каждого компонента.

Если какие-либо методы `decode` или слушатели событий вызвали метод `renderResponse` у текущего объекта `FacesContext`, Jakarta Faces перейдёт к фазе отрисовки ответа.

Если какие-либо события в этой фазе были поставлены в очередь, Jakarta Faces передаёт события заинтересованным слушателям.

Если некоторые компоненты на странице имеют атрибуты `immediate` (см. Атрибут immediate), для которых установлено значение `true`, тогда валидации, конвертации и события, связанные с этими компонентами будут обрабатываться в этой фазе. Если какая-либо конвертация завершается неудачно, сообщение об ошибке, связанное с компонентом, генерируется и ставится в очередь в `FacesContext`. Это сообщение будет отображаться в фазе отрисовки ответа вместе с любыми ошибками валидации, возникшими в фазе валидации процесса.

На этом этапе, если приложению необходимо перенаправить на другой ресурс или сгенерировать ответ, который не содержит компонентов Jakarta Faces, оно может вызвать метод `FacesContext.responseComplete`.

В конце этой фазы для компонентов устанавливаются новые значения, а сообщения и события помещаются в очередь.

Если текущий запрос идентифицирован как частичный запрос, частичный контекст извлекается из `FacesContext` и применяется метод частичной обработки.

Фаза валидации процесса

На этом этапе Jakarta Faces обрабатывает все валидаторы, зарегистрированные в компонентах дерева, используя метод `validate (processValidators)`. Он проверяет атрибуты компонента, которые определяют правила валидации, и сравнивает эти правила с локальным значением, сохранённым для компонента. Jakarta Faces также завершает конвертацию для компонентов ввода, для которых атрибут `immediate` не установлен в значение true.

Если локальное значение не прошло валидацию или какая-либо конвертация завершается неудачно, Jakarta Faces добавляет сообщение об ошибке в объект `FacesContext`, и жизненный цикл переходит непосредственно к фазе отрисовки ответа, чтобы отобразить страница повторно с сообщениями об ошибках. Если в фазе применения значений запроса произошли ошибки конвертации, также отображаются сообщения и об этих ошибках.

Если какие-либо методы `validate` или слушатели событий вызвали метод `renderResponse` в текущем `FacesContext`, Jakarta Faces перейдёт к фазе отрисовки ответа.

На этом этапе, если приложению необходимо перенаправить на другой ресурс или сгенерировать ответ, который не содержит компонентов Jakarta Faces, оно может вызвать метод `FacesContext.responseComplete`.

Если события в этой фазе были поставлены в очередь, Jakarta Faces передаёт их заинтересованным слушателям.

Если текущий запрос идентифицирован как частичный запрос, частичный контекст извлекается из `FacesContext` и применяется метод частичной обработки.

Фаза обновления значений модели

После того, как Jakarta Faces определит, что данные валидны, он обходит дерево компонентов и устанавливает соответствующие свойства серверного объекта в локальные значения компонентов. Jakarta Faces обновляет только свойства компонента, на которые указывает атрибут `value` входного компонента.

Если локальные данные не могут быть конвертированы в типы, указанные в свойствах бина, жизненный цикл переходит непосредственно к фазе отрисовки ответа и страница повторно отображается с ошибками. Это похоже на то, что происходит с ошибками валидации.

Если какие-либо методы `updateModels` или какие-либо слушатели вызвали метод `renderResponse` в текущем объекте `FacesContext`, Jakarta Faces перейдёт к фазе отрисовки ответа.

На этом этапе, если приложению необходимо перенаправить на другой ресурс или сгенерировать ответ, который не содержит компонентов Jakarta Faces, оно может вызвать метод `FacesContext.responseComplete`.

Если какие-либо события в этой фазе были поставлены в очередь, Jakarta Faces передаёт их заинтересованным слушателям.

Если текущий запрос идентифицирован как частичный запрос, частичный контекст извлекается из `FacesContext` и применяется метод частичной обработки.

Фаза вызова приложения

В этой фазе Jakarta Faces обрабатывает любые события уровня приложения, такие как отправка формы или ссылки на другую страницу.

На этом этапе, если приложению необходимо перенаправить на другой ресурс или сгенерировать ответ, который не содержит компонентов Jakarta Faces, оно может вызвать метод `FacesContext.responseComplete`.

Если обрабатываемое представление было восстановлено из информации о состоянии из предыдущего запроса и если компонент вызвал событие, эти события передаются заинтересованным слушателям.

Наконец, Jakarta Faces передаёт управление фазе отрисовка ответа.

Фаза отрисовки ответа

На этом этапе Jakarta Faces создаёт представление и делегирует полномочия соответствующему ресурсу для отображения страниц.

Если это начальный запрос, компоненты, представленные на странице, будут добавлены в дерево компонентов. Если это не начальный запрос, компоненты уже добавлены в дерево и не нуждаются в повторном добавлении.

Если запрос является повторным и в фазе применения значений запроса, валидаций процесса или в фазе обновления значений модели возникли ошибки, исходная страница будет отображена снова. Если страницы содержат теги `h:message` или `h:messages`, любые сообщения об ошибках в очереди отображаются на странице.

После отрисовки содержимого представления состояние ответа сохраняется, чтобы последующие запросы могли получить к нему доступ. Сохранённое состояние доступно для фазы восстановления представления.

Частичная обработка и частичное отображение

Жизненный цикл Jakarta Faces охватывает все процессы выполнения и отрисовки приложения. Также возможно обрабатывать и отрисовывать только части приложения, такие как одиночный компонент. Например, Ajax фреймворк Jakarta Faces может генерировать запросы, содержащие информацию о том, какой конкретный компонент может быть обработан и какой конкретный компонент может быть возвращён клиенту.

После того, как такой частичный запрос входит в жизненный цикл Jakarta Faces, информация идентифицируется и обрабатывается объектом `jakarta.faces.context.PartialViewContext`. Жизненный цикл Jakarta Faces знает о таких Ajax-запросах и соответствующим образом модифицирует дерево компонентов.

Атрибуты `execute` и `render` тега `f:ajax` используются для определения того, какие компоненты могут быть выполнены и отрисованы. Для получения дополнительной информации об этих атрибутах см. главу 13 *Использование Ajax с Jakarta Faces*.

Дополнительная информация о Jakarta Faces

Для получения дополнительной информации о Jakarta Faces см.

- Спецификация Jakarta Faces 3.0:
  https://jakarta.ee/specifications/faces/3.0/
- Сайт Mojarra:
  https://eclipse-ee4j.github.io/mojarra/

Дополнительные примеры см. в примерах GlassFish по ссылке https://github.com/eclipse-ee4j/glassfish-samples/tree/master/ws/jakartaee9.

Глава 8. Введение в Facelets

Термин Facelets относится к языку объявления представлений для Jakarta Faces. Facelets является частью спецификации Jakarta Faces, а также предпочтительной технологией представления для создания приложений на основе Jakarta Faces. Jakarta Server Pages, ранее использовавшаяся в качестве технологии презентаций для Jakarta Faces, поддерживает не все новые функции платформы Jakarta EE, доступные в Jakarta Faces. Jakarta Server Pages считается устаревшей по сравнению с Jakarta Faces.

Что такое Facelets?

Facelets — это мощный, но легковесный язык объявлений страниц, который используется для создания представлений Jakarta Faces с использованием HTML-шаблонов и построения деревьев компонентов. Особенности Facelets включают в себя следующее:

- Использование XHTML для создания веб-страниц
- Поддержка библиотек тегов Facelets в дополнение к библиотекам тегов Jakarta Faces и JSTL
- Поддержка языка выражений (EL)
- Шаблонизация для компонентов и страниц

Преимущества Facelets для разработки крупномасштабных проектов:

- Поддержка повторного использования кода с помощью шаблонов и составных компонентов
- Функциональная расширяемость компонентов и других серверных объектов за счёт кастомизации
- Ускорение компиляции
- Валидация EL во время компиляции
- Высокопроизводительная отрисовка

Короче говоря, использование Facelets сокращает время и усилия, которые необходимо затратить на разработку и развёртывание.

Представления Facelets обычно создаются как страницы XHTML. Реализации Jakarta Faces поддерживают страницы XHTML, созданные в соответствии с определением типа переходного документа XHTML (DTD), как указано в https://www.w3.org/TR/xhtml1/#a_dtd_XHTML-1.0-Transitional. По соглашению веб-страницы, созданные с XHTML, имеют расширение `.xhtml` .

Jakarta Faces поддерживает различные библиотеки тегов для добавления компонентов на веб-страницу. Для поддержки механизма библиотеки тегов Jakarta Faces Facelets использует пространства имён XML. Таблица 8-1 содержит список библиотек тегов, поддерживаемых Facelets.

Таблица 8-1 Библиотеки тегов, поддерживаемые Facelets

Библиотека тегов	**URI**	**Префикс**	**Пример**	**Содержание**

Библиотека тегов Facelets Jakarta Faces	http://xmlns.jcp.org/jsf/facelets	`ui:`	`ui:component`
			
			`ui:insert`
				Теги для шаблонов

Библиотека тегов	URI	Префикс	Пример	Содержание

Библиотека тегов HTML для Jakarta Faces	http://xmlns.jcp.org/jsf/html	`h:`	`h:head`
			`h:body`
			`h:outputText`
			`h:inputText`
				Теги Jakarta Faces для всех объектов `UIComponent`

Базовая библиотека тегов Jakarta Faces	http://xmlns.jcp.org/jsf/core	`f:`	`f:actionListener` `f:attribute`	Теги для кастомных действий Jakarta Faces, которые не зависят от какого-либо конкретного инструмента отрисовки
Библиотека сквозных (pass-through) тегов	http://xmlns.jcp.org/jsf	`jsf:`	`jsf:id`	Теги поддержки HTML5-совместимой разметки
Библиотека сквозных (pass-through) атрибутов тегов	http://xmlns.jcp.org/jsf/passthrough	`p:`	`p:type`	Теги поддержки HTML5-совместимой разметки
Библиотека тегов составных компонентов	http://xmlns.jcp.org/jsf/composite	`cc:`	`cc:interface`	Теги поддержки составных компонентов
Базовая библиотека тегов JSTL	http://xmlns.jcp.org/jsp/jstl/core	`c:`	`c:forEach` `c:catch`	Основные теги JSTL 1.2
Библиотека тегов функций JSTL	http://xmlns.jcp.org/jsp/jstl/functions	`fn:`	`fn:toUpperCase` `fn:toLowerCase`	Теги функций JSTL 1.2

Facelets предоставляет два пространства имён для поддержки HTML5-совместимой разметки. Подробнее см. HTML5-совместимая разметка.

Facelets поддерживает теги составных компонентов, для которых вы можете объявить кастомные префиксы. Для получения дополнительной информации о составных компонентах см. Составные компоненты.

Префиксы пространства имён, показанные в таблице, являются условными, не обязательными. Как всегда, когда вы объявляете пространство имён XML, можете указать любой префикс на своей странице Facelets. Например, вы можете объявить префикс для библиотеки тегов составного компонента как

```
xmlns:composite="http://xmlns.jcp.org/jsf/composite"
```

вместо

```
xmlns:cc="http://xmlns.jcp.org/jsf/composite"
```

Основываясь на поддержке Jakarta Faces для синтаксиса языка выражений (EL), Facelets использует выражения EL для ссылки на свойства и методы Managed-бинов. Выражения EL можно использовать для связывания объектов компонентов или значений с методами или свойствами Managed-бинов, которые используются в качестве вспомогательных бинов. Для получения дополнительной информации об использовании выражений EL см. Использование EL для ссылки на Managed-бины.

Жизненный цикл приложения Facelets

Спецификация Jakarta Faces определяет жизненный цикл приложения Jakarta Faces. Для получения дополнительной информации о жизненном цикле см. Жизненный цикл приложения Jakarta Faces. Следующие шаги описывают этот процесс применительно к приложению на основе Facelets.

1. Когда клиент, например браузер, делает новый запрос на страницу, созданную средствами Facelets, создаётся новое дерево компонентов или `jakarta.faces.component.UIViewRoot` и помещается в `FacesContext`.
2. `UIViewRoot` применяется к Facelets, а представление заполняется компонентами для отрисовки.
3. Недавно построенное представление отображается как ответ клиенту.
4. При отрисовке состояние этого представления сохраняется для следующего запроса. Состояние компонентов ввода и данные формы сохраняются.
5. Клиент может взаимодействовать с представлением и запрашивать другое представление или изменение из приложения Jakarta Faces. В это время сохранённое представление восстанавливается из сохранённого состояния.
6. Восстановленное представление снова проходит через жизненный цикл Jakarta Faces, который в конечном итоге либо сгенерирует новое представление, либо повторно отобразит текущее представление, если не было проблем с валидацией и не было выполнено никаких действий.
7. Если запрашивается то же самое представление, сохранённое представление отрисовывается ещё раз.
8. Если запрашивается новое представление, то процесс, описанный в Шаг 2 продолжается.
9. Затем новое представление отображается как ответ клиенту.

Разработка простого приложения Facelets: пример guessnumber-jsf

В этом разделе описываются основные этапы разработки приложения Jakarta Faces. Обычно требуются следующие задачи:

- Разработка Managed-бинов
- Создание страниц с использованием тегов компонентов
- Определение навигации по страницам

- Отображение объекта `FacesServlet`
- Добавление объявлений Managed-бинов

Создание приложения Facelets

В этом учебнике реализован пример приложения `guessnumber-jsf`. Приложение представляет страницу, которая предлагает угадать число от 0 до 10, проверяет ввод по случайному числу и отвечает другой страницей, которая сообщает, правильно угадано число или нет.

Исходный код для этого приложения находится в каталоге *`tut-install`*`/examples/web/jsf/guessnu-jsf/`.

Разработка Managed-бина

В типичном приложении Jakarta Faces каждая страница приложения подключается к вспомогательному Managed-бину. Этот бин определяет методы и свойства, связанные с компонентами. В этом примере обе страницы используют один и тот же вспомогательный компонент.

Следующий класс Managed-бинов, `UserNumberBean.java`, генерирует случайное число от 0 до 10 включительно:

JAVA

```java
package ee.jakarta.tutorial.guessnumber;

import java.io.Serializable;
import java.util.Random;
import jakarta.enterprise.context.SessionScoped;
import jakarta.inject.Named;

@Named
@SessionScoped
public class UserNumberBean implements Serializable {

    private static final long serialVersionUID = 5443351151396868724L;
    Integer randomInt = null;
    Integer userNumber = null;
    String response = null;
    private int maximum = 10;
    private int minimum = 0;

    public UserNumberBean() {
        Random randomGR = new Random();
        randomInt = new Integer(randomGR.nextInt(maximum + 1));
        // Запись номера в лог сервера
        System.out.println("Duke's number: " + randomInt);
    }

    public void setUserNumber(Integer user_number) {
        userNumber = user_number;
    }

    public Integer getUserNumber() {
        return userNumber;
    }

    public String getResponse() {
        if ((userNumber == null) || (userNumber.compareTo(randomInt) != 0)) {
            return "Sorry, " + userNumber + " is incorrect.";
        } else {
            return "Yay! You got it!";
        }
    }

    public int getMaximum() {
        return (this.maximum);
    }

    public void setMaximum(int maximum) {
        this.maximum = maximum;
    }

    public int getMinimum() {
        return (this.minimum);
    }

    public void setMinimum(int minimum) {
        this.minimum = minimum;
    }
}
```

Обратите внимание на использование аннотации `@Named`, которая делает Managed-бин доступным через EL. Аннотация `@SessionScoped` регистрирует компонент в области видимости сессии, чтобы можно было сделать несколько предположений при запуске приложения.

Создание представлений Facelets

Чтобы создать страницу или представление, вы добавляете компоненты на страницы, подключаете компоненты к значениям и свойствам поддерживающего бина и регистрируете конвертеры, валидаторы или слушатели в компонентах.

В этом примере пользовательский интерфейс выполнен веб-страницами XHTML. Первая страница примера приложения — это страница `greeting.xhtml` . Внимательный взгляд на различные разделы этой веб-страницы даёт много информации.

Первый раздел веб-страницы объявляет тип её содержимого. Это XHTML:

```xml
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

В следующем разделе указывается язык страницы, а затем объявляется пространство имён XML для библиотек тегов, используемых на веб-странице:

```xml
<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
```

В следующем разделе используются различные теги для вставки компонентов на веб-страницу:

```xml
    <h:head>
        <h:outputStylesheet library="css" name="default.css"/>
        <title>Guess Number Facelets Application</title>
    </h:head>
    <h:body>
        <h:form>
            <h:graphicImage value="#{resource['images:wave.med.gif']}"
                            alt="Duke waving his hand"/>
            <h2>
                Hi, my name is Duke. I am thinking of a number from
                #{userNumberBean.minimum} to #{userNumberBean.maximum}.
                Can you guess it?
            </h2>
            <p><h:inputText id="userNo"
                            title="Enter a number from 0 to 10:"
                            value="#{userNumberBean.userNumber}">
                   <f:validateLongRange minimum="#{userNumberBean.minimum}"
                                        maximum="#{userNumberBean.maximum}"/>
                </h:inputText>
                <h:commandButton id="submit" value="Submit"
                                 action="response"/>
            </p>
            <h:message showSummary="true" showDetail="false"
                       style="color: #d20005;
                       font-family: 'New Century Schoolbook', serif;
                       font-style: oblique;
                       text-decoration: overline"
                       id="errors1"
                       for="userNo"/>
        </h:form>
    </h:body>
```

Обратите внимание на использование следующих тегов:

- HTML-теги Facelets (начинающиеся с `h:`) для добавления компонентов

- Стандартный тег Facelets `f:validateLongRange` для валидации пользовательского ввода

Тег `h:inputText` принимает пользовательский ввод и устанавливает значение свойства Managed-бина `userNumber` через выражение EL `#{userNumberBean.userNumber}`. Введённое значение валидацируется на принадлежность к диапазону значений с использованием стандартного тега валидатора Jakarta Faces `f:validateLongRange`.

Файл изображения `wave.med.gif` добавляется на страницу в качестве ресурса, как и таблица стилей. Для получения дополнительной информации об объекте ресурсов см. Веб-ресурсы.

Тег `h:commandButton` с идентификатором `submit` запускает валидацию входных данных, когда пользователь кликает кнопку. Используя неявную навигацию, тег перенаправляет клиента на другую страницу, `response.xhtml`, которая показывает ответ ввод. На странице указан только `response`, что по умолчанию заставляет сервер искать `response.xhtml`.

Теперь вы можете создать вторую страницу, `response.xhtml`, со следующим содержанием:

XML

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">

    <h:head>
        <h:outputStylesheet library="css" name="default.css"/>
        <title>Guess Number Facelets Application</title>
    </h:head>
    <h:body>
        <h:form>
            <h:graphicImage value="#{resource['images:wave.med.gif']}"
                            alt="Duke waving his hand"/>
            <h2>
                <h:outputText id="result" value="#{userNumberBean.response}"/>
            </h2>
            <h:commandButton id="back" value="Back" action="greeting"/>
        </h:form>
    </h:body>
</html>
```

Эта страница также использует неявную навигацию, устанавливая атрибут `action` для кнопки «Назад», чтобы отправлять пользователя на страницу `greeting.xhtml`.

Настройка приложения

Конфигурирование приложения Jakarta Faces включает в себя назначение шаблона URL для сервлета Faces файле дескриптора развёртывания `web.xml` и, возможно, добавление объявлений Managed-бинов, правил навигации и объявлений ресурсов bundle в файл конфигурации приложения `faces-config.xml`.

Если вы используете IDE NetBeans, файл дескриптора развёртывания создаётся автоматически. В таком созданном IDE файле `web.xml` измените страницу входа по умолчанию — `index.xhtml` — на `greeting.xhtml`. Вот пример файла `web.xml`, где это изменение показано жирным шрифтом.

XML

```xml
<web-app version="5.0"
    xmlns="https://jakarta.ee/xml/ns/jakartaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee https://jakarta.ee/xml/ns/jakartaee/web-
app_5_0.xsd">
    <context-param>
        <param-name>jakarta.faces.PROJECT_STAGE</param-name>
        <param-value>Development</param-value>
    </context-param>
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>jakarta.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.xhtml</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>greeting.xhtml</welcome-file>
    </welcome-file-list>
</web-app>
```

Обратите внимание на использование контекстного параметра `PROJECT_STAGE`. Этот параметр определяет статус приложения Jakarta Faces в жизненном цикле ПО.

Этап приложения может повлиять на поведение приложения. Например, если этап проекта определён как `Development`, для пользователя автоматически создаётся информация об отладке. Если пользователь не определил его, этап проекта по умолчанию `Production`.

Выполнение guessnumber-jsf

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска примера `guessnumber-jsf`.

Сборка, упаковка и развёртывание guessnumber-jsf в IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/web/jsf
   ```

4. Выберите каталог `guessnumber-jsf`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `guessnumber-jsf` и выберите **Сборка**.

   Эта команда собирает приложение и развёртывает его в GlassFish Server.

Сборка, упаковка и развёртывание guessnumber-jsf с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).

2. В окне терминала перейдите в:

```
tut-install/examples/web/jsf/guessnumber-jsf/
```

3. Введите следующую команду:

```
mvn install
```
SHELL

Эта команда собирает и упаковывает приложение в WAR-файл `guessnumber-jsf.war`, который находится в каталоге `target`. Затем он развёртывается на сервере.

Запуск guessnumber-jsf

1. Откройте веб-браузер.
2. Введите следующий URL в браузере:

```
http://localhost:8080/guessnumber-jsf
```

3. В поле введите число от 0 до 10 и нажмите «Отправить».

   Появится другая страница, сообщающая, верно ваше предположение или нет.

4. Если вы угадали неправильно, нажмите Назад, чтобы вернуться на главную страницу.

   Вы можете продолжать угадывать, пока не получите правильный ответ, или вы можете посмотреть в журнале сервера, где конструктор `UserNumberBean` отображает правильный ответ.

Использование шаблонов Facelets

Jakarta Faces предоставляет инструменты для реализации пользовательских интерфейсов, которые легко расширять и повторно использовать. Шаблонизация — это полезная функция Facelets, позволяющая создать страницу, которая будет выступать в качестве базовой страницы, или шаблона, для других страниц приложения. Используя шаблоны, вы можете повторно использовать код и избежать создания похожих страниц. Шаблонизация также помогает поддерживать стандартный внешний вид в приложении с большим количеством страниц.

Таблица 8-2 перечисляет теги Facelets, которые используются для шаблонов, и их функциональные возможности.

Таблица 8-2 Теги шаблонов Facelets

Тег	Функция
`ui:component`	Определяет компонент, который создаётся и добавляется в дерево компонентов.
`ui:composition`	Определяет композицию страницы, которая, возможно, использует шаблон. Содержимое вне этого тега игнорируется.
`ui:debug`	Определяет компонент отладки, который создаётся и добавляется в дерево компонентов.
`ui:decorate`	Аналогичен тегу композиции, но не игнорирует содержимое вне этого тега.
`ui:define`	Определяет содержимое, которое вставляется на страницу с помощью шаблона.

Тег	Функция
`ui:fragment`	Аналогично тегу компонента, но не игнорирует содержимое вне этого тега.
`ui:include`	Инкапсулирует и повторно использует контент для нескольких страниц.
`ui:insert`	Вставляет содержимое в шаблон.
`ui:param`	Используется для передачи параметров во включаемый файл.
`ui:repeat`	Используется в качестве альтернативы для тегов цикла, таких как `c:forEach` или `h:dataTable`.
`ui:remove`	Удаляет контент со страницы.

Для получения дополнительной информации о тегах шаблонов Facelets см. документацию библиотеки тегов Facelets Jakarta Faces.

Библиотека тегов Facelets включает основной шаблонный тег `ui:insert`. Страница шаблона, созданная с помощью этого тега, позволяет определить структуру страницы по умолчанию. Страница шаблона используется в качестве шаблона для других страниц, обычно называемых страницами клиента.

Вот пример шаблона, сохранённого как `template.xhtml`:

XML

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html">

    <h:head>
        <meta http-equiv="Content-Type"
              content="text/html; charset=UTF-8" />
        <h:outputStylesheet library="css" name="default.css"/>
        <h:outputStylesheet library="css" name="cssLayout.css"/>
        <title>Facelets Template</title>
    </h:head>

    <h:body>
        <div id="top" class="top">
            <ui:insert name="top">Top Section</ui:insert>
        </div>
        <div>
        <div id="left">
             <ui:insert name="left">Left Section</ui:insert>
        </div>
        <div id="content" class="left_content">
             <ui:insert name="content">Main Content</ui:insert>
        </div>
        </div>
    </h:body>
</html>
```

Страница примера определяет страницу XHTML, которая разделена на три раздела: верхний раздел, левый раздел и основной раздел. С разделами связаны таблицы стилей. Такая же структура может быть повторно использована для других страниц приложения.

Страница клиента вызывает шаблон с помощью тега `ui:composition`. В следующем примере клиентская страница `templateclient.xhtml` вызывает страницу шаблона с именем `template.xhtml` из предыдущего примера. Страница клиента позволяет вставлять содержимое с помощью тега `ui:define`.

XML

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html">

    <h:body>
        <ui:composition template="./template.xhtml">
            <ui:define name="top">
                Welcome to Template Client Page
            </ui:define>

            <ui:define name="left">
                <h:outputLabel value="You are in the Left Section"/>
            </ui:define>

            <ui:define name="content">
                <h:graphicImage value="#{resource['images:wave.med.gif']}"/>
                <h:outputText value="You are in the Main Content Section"/>
            </ui:define>
        </ui:composition>
    </h:body>
</html>
```

Вы можете использовать IDE NetBeans для создания шаблона Facelets и страниц клиента. Дополнительная информация о создании этих страниц приведена по ссылке https://netbeans.org/kb/docs/web/jsf20-intro.html.

Составные компоненты

Jakarta Faces предлагает концепцию составных компонентов с Facelets. Составной компонент — это специальный тип шаблона, который действует как компонент.

Любой компонент — это, по сути, фрагмент кода, который можно использовать многократно и который ведёт себя определённым образом. Например, компонент ввода принимает пользовательский ввод. С компонентом также могут быть связаны валидаторы, конвертеры и слушатели для выполнения определённых действий.

Составной компонент состоит из набора тегов разметки и других существующих компонентов. Этот повторно используемый компонент, созданный пользователем, имеет определённую функциональность и может иметь валидаторы, конвертеры и слушатели, связанные с ним, как с любым другим компонентом.

С помощью Facelets любая страница XHTML, содержащая теги разметки и другие компоненты, может быть конвертирована в составной компонент. Используя ресурсы, составной компонент может храниться в библиотеке, которая доступна приложению из определённого местоположения.

Таблица 8-3 перечисляет наиболее часто используемые составные теги и их функции.

Таблица 8-3. Составные теги компонентов

Тег	Функция

Тег	Функция
`composite:interface`	Объявляет контракт на использование составного компонента. Составной компонент может использоваться как отдельный компонент, набор функций которого представляет собой объединение функций, объявленных контрактом на использование.
`composite:implementation`	Определяет реализацию составного компонента. Если появляется элемент `composite:interface`, должен быть соответствующий ему элемент `composite:implementation`.
`composite:attribute`	Объявляет атрибут, который может быть задан объекту составного компонента, в котором объявлен этот тег.
`composite:insertChildren`	Любые дочерние компоненты или текст шаблона в теге составного компонента на странице использования будут переопределены в составном компоненте в точке, указанной размещением этого тега в разделе `composite:implementation`.
`composite:valueHolder`	Объявляет, что составной компонент, чей контракт объявлен `composite:interface`, в который вложен этот элемент, предоставляет реализацию `ValueHolder`, подходящую для использования в качестве цели для прикреплённых объектов на используемой странице.
`composite:editableValueHolder`	Объявляет, что составной компонент, чей контракт объявлен `composite:interface`, в который вложен этот элемент, предоставляет реализацию `EditableValueHolder`, подходящую для использования в качестве цели присоединяемых объектов на используемой странице.
`composite:actionSource`	Объявляет, что составной компонент, чей контракт объявлен `composite:interface`, в который вложен этот элемент, предоставляет реализацию `ActionSource2`, подходящую для использования в качестве цели присоединяемых объектов на используемой странице.

Для получения дополнительной информации и полного списка составных тегов Facelets см. документацию библиотеки тегов Facelet Jakarta Faces.

В следующем примере показан составной компонент, который принимает на вход адрес электронной почты:

```XML
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:composite="http://xmlns.jcp.org/jsf/composite"
  xmlns:h="http://xmlns.jcp.org/jsf/html">

    <h:head>
        <title>This content will not be displayed</title>
    </h:head>
    <h:body>
        <composite:interface>
            <composite:attribute name="value" required="false"/>
        </composite:interface>

        <composite:implementation>
            <h:outputLabel value="Email id: "></h:outputLabel>
            <h:inputText value="#{cc.attrs.value}"></h:inputText>
        </composite:implementation>
    </h:body>
</html>
```

Обратите внимание на использование `cc.attrs.value` при определении значения компонента `inputText`. Слово `cc` в Jakarta Faces является зарезервированным словом для составных компонентов. `#{cc.attrs.`*`attribute-name`*`}` используется для доступа к атрибутам, определённым для интерфейса составного компонента, который в этом случае `value`.

Содержимое предыдущего примера хранится в виде файла `email.xhtml` в каталоге `resources/emcomp` корневого каталога веб-приложения. Jakarta Faces считает этот каталог библиотекой, и компонент может быть доступен из такой библиотеки. Для получения дополнительной информации о ресурсах см. Веб-ресурсы.

Веб-страница, которая использует этот составной компонент, обычно называется страницей использования. Страница использования включает ссылку на составной компонент в объявлениях пространства имён `xml`:

```XML
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:h="http://xmlns.jcp.org/jsf/html"
  xmlns:em="http://xmlns.jcp.org/jsf/composite/emcomp">

    <h:head>
        <title>Using a sample composite component</title>
    </h:head>

    <body>
        <h:form>
            <em:email value="Enter your email id" />
        </h:form>
    </body>
</html>
```

Локальная библиотека составных компонентов определяется в пространстве имён `xmlns` с объявлением `xmlns:em="http://xmlns.jcp.org/jsf/composite/emcomp"`. Доступ к самому компоненту осуществляется через тег `em:email`. Содержимое предыдущего примера можно сохранить как веб-страницу с именем `emuserpage.xhtml` в корневом веб-каталоге. При компиляции и развёртывании на сервере доступ к нему можно получить по следующему URL:

```
http://localhost:8080/application-name/emuserpage.xhtml
```

См. главу 14 *Составные компоненты: дополнительные темы и пример* для получения дополнительной информации и примера.

Веб-ресурсы

Веб-ресурсы — это любые программные артефакты, которые требуются веб-приложению для правильной отрисовки, включая изображения, файлы сценариев и любые библиотеки компонентов, созданные пользователем. Ресурсы должны быть размещены в стандартном месте, которое может быть одним из следующих.

- Ресурс, упакованный в корневой каталог веб-приложения, должен находиться в подкаталоге `resources` в корне веб-приложения: `resources/`*`resource-identifier`*.
- Ресурс, упакованный в путь к классам веб-приложения, должен находиться в подкаталоге `META-INF/resources` в веб-приложении: `META-INF/resources/`*`resource-identifier`*. Вы можете использовать эту файловую структуру для упаковки ресурсов в файл JAR, размещённый в веб-приложении.

Среда выполнения Jakarta Faces будет искать ресурсы в предыдущих перечисленных местах в указанном порядке.

Идентификаторы ресурса — это уникальные строки, которые соответствуют следующему формату (все в одной строке):

```
[locale-prefix/][library-name/][library-version/]resource-name[/resource-version]
```

Элементы идентификатора ресурса в скобках (`[]`) являются необязательными, что указывает на то, что обязательным элементом является только имя ресурса, которое обычно является именем файла. Например, наиболее распространенный способ указать таблицу стилей, изображение или сценарий — использовать атрибуты `library` и `name`, как в следующем теге из примера `guessnumber-jsf`:

```xml
<h:outputStylesheet library="css" name="default.css"/>
```

Этот тег указывает, что таблица стилей `default.css` находится в каталоге `web/resources/css`.

Можно также указать местоположение изображения, используя следующий синтаксис, также взятый из примера `guessnumber-jsf`:

```xml
<h:graphicImage value="#{resource['images:wave.med.gif']}"/>
```

Этот тег указывает, что изображение `wave.med.gif` находится в каталоге `web/resources/images`.

Ресурсы можно рассматривать как место расположения библиотеки. Любой артефакт, такой как составной компонент или шаблон, который хранится в каталоге `resources`, становится доступным для других компонентов приложения, которые могут использовать его для создания объекта ресурса.

Перемещаемые ресурсы

Вы можете разместить тег ресурса в одной части страницы и указать, что он будет отображаться в другой части страницы. Для этого вы используете атрибут `target` тега, который указывает ресурс. Допустимые значения для этого атрибута следующие.

- “head” отображает ресурс в элементе `head`.

- “body” отображает ресурс в элементе `body`.
- “form” отображает ресурс в элементе `form`.

Например, следующий тег `h:outputScript` помещён в элемент `h:form`, но он отображает JavaScript в элементе `head`:

```xml
<h:form>
    <h:outputScript name="myscript.js" library="mylibrary" target="head"/>
</h:form>
```

Тег `h:outputStylesheet` также поддерживает перемещение ресурсов аналогичным образом.

Перемещаемые ресурсы необходимы для составных компонентов, которые используют таблицы стилей, а также могут быть полезны для составных компонентов, использующих JavaScript. См. Пример compositecomponentexample.

Контракты библиотеки ресурсов

Контракты с библиотеками ресурсов позволяют определять различный внешний вид для разных частей одного или нескольких приложений, вместо того, чтобы либо использовать один и тот же внешний вид для всех, либо указывать разный внешний вид для каждой страницы.

Для этого вы создаёте раздел `contracts` вашего веб-приложения. В разделе `contracts` вы можете указать любое количество именованных областей, каждая из которых называется контрактом. В каждом контракте вы можете указать ресурсы, такие как файлы шаблонов, таблиц стилей, JavaScript и изображения.

Например, можно указать два контракта `c1` и `c2`, каждый из которых использует шаблон и другие файлы:

```
src/main/webapp
    WEB-INF/
    contracts
        c1
            template.xhtml
            style.css
            myImg.gif
            myJS.js
        c2
            template.xhtml
            style2.css
            img2.gif
            JS2.js
    index.xhtml
    ...
```

Одна часть приложения может использовать `c1`, а другая — `c2`.

Другой вариант использования контрактов — указать один контракт, который содержит несколько шаблонов:

```
src/main/webapp
    contracts
        myContract
            template1.xhtml
            template2.xhtml
            style.css
            img.png
            img2.png
```

Контракт библиотеки ресурсов можно упаковать в JAR-файл для повторного использования в различных приложениях. Если вы это сделаете, контракты должны находиться в `META-INF/contracts`. Затем вы можете поместить файл JAR в каталог `WEB-INF/lib` приложения. Это означает, что приложение будет организовано следующим образом:

```
src/main/webapp/
    WEB-INF/lib/myContract.jar
    ...
```

Вы можете указать использование контракта в файле `faces-config.xml` приложения в элементе `resource-library-contracts`. Однако можно использовать этот элемент только если приложение использует более одного контракта.

Приложение hello1-rlc

Пример `hello1-rlc` изменяет простой пример `hello1` из Веб-модуля с использованием Jakarta Faces: пример hello1 для использования двух контрактов библиотеки ресурсов. На каждой из двух страниц приложения используются разные контракты.

Managed-бин для `hello1-rlc` — `Hello.java` — идентичен Managed-бину для `hello1` (за исключением того, что в нём аннотации `@Named` и `@RequestScoped` заменены на `@Model`).

Исходный код этого приложения находится в каталоге *`tut-install`*`/examples/web/jsf/hello1-rlc/`.

Настройка hello1-rlc

Файл `faces-config.xml` для примера `hello1-rlc` содержит следующие элементы:

XML

```xml
<resource-library-contracts>
    <contract-mapping>
        <url-pattern>/reply/*</url-pattern>
        <contracts>reply</contracts>
    </contract-mapping>
    <contract-mapping>
        <url-pattern>*</url-pattern>
        <contracts>hello</contracts>
    </contract-mapping>
</resource-library-contracts>
```

Элементы `contract-mapping` в элементе `resource-library-contracts` назначают каждому контракту различный набор страниц. Один контракт с именем `reply` используется для всех страниц в области `reply` приложения (`/reply/*`). Другой контракт, `hello`, используется для всех остальных страниц приложения (`*`).

Приложение организовано следующим образом:

```
hello1-rlc
    pom.xml
    src/main/java/ee/jakarta/tutorial/hello1rlc/Hello.java
    src/main/webapp
        WEB-INF
            faces-config.xml
            web.xml
        contracts
            hello
                default.css
                duke.handsOnHips.gif
                template.xhtml
            reply
                default.css
                duke.thumbsup.gif
                template.xhtml
        reply
            response.xhtml
        greeting.xhtml
```

В файле `web.xml` указан `welcome-file` как `greeting.xhtml`. Поскольку он не находится в `src/main/webapp/reply`, эта страница Facelets использует контракт `hello`, тогда как `src/main/webapp/reply/response.xhtml` использует контракт `reply`.

Страницы Facelets для hello1-rlc

Страницы `greeting.xhtml` и `response.xhtml` имеют идентичный код, вызываемый в их шаблонах

XML

```xml
<ui:composition template="/template.xhtml">
```

Файлы `template.xhtml` в контрактах `hello` и `reply` имеют только два отличия: в тексте заголовка `title` («Hello Template» и «Reply Template») и изображение, которое указано для каждого из них.

Таблицы стилей `default.css` в двух контрактах отличаются только цветом фона, указанном для элемента `body`.

Сборка, упаковка и развёртывание hello1-rlc в IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/web/jsf
   ```

4. Выберите каталог `hello1-rlc`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `hello1-rlc` и выберите **Сборка**.

   Эта команда собирает приложение и развёртывает его в GlassFish Server.

Сборка, упаковка и развёртывание hello1-rlc с помощью Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

```
tut-install/examples/web/jsf/hello1-rlc/
```

3. Введите следующую команду:

```
mvn install
```
SHELL

Эта команда собирает и упаковывает приложение в WAR-файл `hello1-rlc.war`, который находится в каталоге `target`. Затем он развёртывается в GlassFish Server.

Запуск hello1-rlc

1. Введите следующий URL в браузере:

```
http://localhost:8080/hello1-rlc
```

2. Страница `greeting.xhtml` выглядит так же, как страница из `hello1`, за исключением цвета фона и графики.
3. В текстовом поле введите своё имя и нажмите «Отправить».
4. Страница ответов также выглядит точно так же, как страница из `hello1`, за исключением её цвета фона и графики.

   На странице отображается имя, которое вы отправили. Нажмите Назад, чтобы вернуться на страницу `greeting.xhtml`.

HTML5-совместимая разметка

Если при создании пользовательского интерфейса не хватает тегов HTML, можно создать кастомный компонент Jakarta Faces и вставить его на страницу Facelets. Этот механизм может заставить простой элемент создавать сложный код. Однако создание такого компонента — важная задача (см. главу 15 *Создание кастомных компонентов пользовательского интерфейса и других кастомных объектов*).

HTML5 предлагает новые элементы и атрибуты, которые могут сделать ненужным написание собственных компонентов. Он также предоставляет множество новых возможностей для существующих компонентов. Jakarta Faces поддерживает HTML5 не за счёт введения новых компонентов пользовательского интерфейса, имитирующих HTML5, а за счёт возможности непосредственного использования разметки HTML5. Это также позволяет использовать атрибуты Jakarta Faces в элементах HTML5. Поддержка HTML5 в Jakarta Faces делится на две категории:

- Сквозные (pass-through) элементы
- Сквозные (pass-through) атрибуты

Эффект HTML5-совместимой разметки состоит в том, чтобы предоставить разработчику страницы Facelets практически полный контроль над отрисовкой страницы, вместо того, чтобы передавать этот элемент управления разработчикам компонентов. Вы можете сочетать Jakarta Faces и элементы HTML5 по своему усмотрению.

Использование сквозных (pass-through) элементов

Сквозные (pass-through) элементы позволяют использовать теги и атрибуты HTML5, но рассматривать их как эквивалент компонентов Jakarta Faces, связанных с серверным объектом `UIComponent`.

Чтобы создать элемент, который не является элементом Jakarta Faces, а будет сквозным элементом, укажите хотя бы один из его атрибутов, используя пространство имён `http://xmlns.jcp.org/jsf`. Например, следующий код объявляет пространство имён с префиксом `jsf`:

```xml
<html ... xmlns:jsf="http://xmlns.jcp.org/jsf"
...
    <input type="email" jsf:id="email" name="email"
           value="#{reservationBean.email}" required="required"/>
```

Здесь префикс `jsf` размещается у атрибута `id`, поэтому атрибуты тега HTML5 обрабатываются как часть страницы Facelets. Это означает, что, например, вы можете использовать выражения EL для получения свойств Managed-бина.

Таблица 8-4 показывает, как сквозные (pass-through) элементы отрисовываются как теги Facelets. Реализация Faces использует элемент name и идентифицирующий атрибут для определения соответствующего тега Facelets, который будет использоваться на серверной стороне. Браузер, однако, интерпретирует разметку, написанную автором страницы.

Таблица 8-4. Как Facelets отображает элементы HTML5

Имя элемента HTML5	**Идентифицирующий атрибут**	**Тег Facelets**
a	jsf:action	h:commandLink
a	jsf:actionListener	h:commandLink
a	jsf:value	h:outputLink
a	jsf:outcome	h:link
body		h:body
button		h:commandButton
button	jsf:outcome	h:button
form		h:form
head		h:head
img		h:graphicImage
input	type="button"	h:commandButton
input	type="checkbox"	h:selectBooleanCheckbox
input	type="color"	h:inputText
input	type="date"	h:inputText
input	type="datetime"	h:inputText

Имя элемента HTML5	Идентифицирующий атрибут	Тег Facelets
input	type="datetime-local"	h:inputText
input	type="email"	h:inputText
input	type="month"	h:inputText
input	type="number"	h:inputText
input	type="range"	h:inputText
input	type="search"	h:inputText
input	type="time"	h:inputText
input	type="url"	h:inputText
input	type="week"	h:inputText
input	type="file"	h:inputFile
input	type="hidden"	h:inputHidden
input	type="password"	h:inputSecret
input	type="reset"	h:commandButton
input	type="submit"	h:commandButton
input	type="*"	h:inputText
label		h:outputLabel
link		h:outputStylesheet
script		h:outputScript
select	multiple="*"	h:selectManyListbox
select		h:selectOneListbox
textarea		h:inputTextArea

Использование сквозных атрибутов

Сквозные (pass-through) атрибуты — противоположность сквозных элементов. Они позволяют передавать атрибуты, которые не являются атрибутами Jakarta Faces, в браузер без интерпретации. Если вы укажете сквозной (pass-through) атрибут в Jakarta Faces `UIComponent` , имя и значение атрибута будут напрямую переданы в браузер без интерпретации компонентами или средствами отрисовки Jakarta Faces. Есть несколько способов указать сквозные (pass-through) атрибуты.

- Используйте пространство имён Jakarta Faces для сквозных (pass-through) атрибутов, чтобы добавлять префикс к именам атрибутов в компоненте Jakarta Faces. Например, следующий код объявляет пространство имён с префиксом `p`, а затем передаёт атрибуты `type`, `min`, `max`, `required` и `title` в компонент HTML5 `input`:

```xml
<html ... xmlns:p="http://xmlns.jcp.org/jsf/passthrough"
...

<h:form prependId="false">
<h:inputText id="nights" p:type="number" value="#{bean.nights}"
             p:min="1" p:max="30" p:required="required"
             p:title="Enter a number between 1 and 30 inclusive.">
        ...
```

  Это приведёт к отрисовке следующей разметки (при условии, что `bean.nights` имеет значение по умолчанию, равное 1):

```xml
<input id="nights" type="number" value="1" min="1" max="30"
       required="required"
       title="Enter a number between 1 and 30 inclusive.">
```

- Чтобы передать одиночный атрибут, вложите тег `f:passThroughAttribute` в тег компонента. Например:

```xml
<h:inputText value="#{user.email}">
    <f:passThroughAttribute name="type" value="email" />
</h:inputText>
```

  Этот код будет отрисован аналогично следующему:

```xml
<input value="me@me.com" type="email" />
```

- Чтобы передать группу атрибутов, вложите тег `f:passThroughAttributes` в тег компонента, указав значение EL, которое должно быть приводимо к `Map<String, Object>`. Например:

```xml
<h:inputText value="#{bean.nights}">
    <f:passThroughAttributes value="#{bean.nameValuePairs}" />
</h:inputText>
```

  Если бин использовал отображение `Map` и инициализирует его в конструкторе следующим образом, разметка будет аналогична выводу кода, который использует пространство имён сквозного атрибута:

```java
private Map<String, Object> nameValuePairs;
...
public Bean() {
    this.nameValuePairs = new HashMap<>();
    this.nameValuePairs.put("type", "number");
    this.nameValuePairs.put("min", "1");
    this.nameValuePairs.put("max", "30");
    this.nameValuePairs.put("required", "required");
    this.nameValuePairs.put("title",
            "Enter a number between 1 and 4 inclusive.");
}
```

Пример reservation

Приложение `reservation` предоставляет собой набор различных типов HTML5-тегов `input` для покупки билетов в театр. Оно состоит из двух страниц Facelets, `reservation.xhtml` и `confirmation.xhtml`, и вспомогательного бина `ReservationBean.java`. На страницах используются как сквозные (pass-through) атрибуты, так и сквозные элементы.

Исходный код этого приложения находится в каталоге *`tut-install`*`/examples/web/jsf/reservation/`.

Страницы Facelets для reservation

Первой важной особенностью страниц Facelets для приложения `reservation` является заголовок `DOCTYPE`. Большинство страниц Facelets в приложениях Jakarta Faces ссылаются на XHTML DTD. Страницы Facelets для этого приложения начинаются со следующего заголовка `DOCTYPE`, который обозначает страницу HTML5:

```xml
<!DOCTYPE html>
```

В объявлении пространства имён, в элементе `html` страницы `reservation.xhtml` указываются оба пространства имён: `jsf` и `passthrough`:

```xml
<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:p="http://xmlns.jcp.org/jsf/passthrough"
      xmlns:jsf="http://xmlns.jcp.org/jsf">
```

Затем пустой тег `h:head`, за которым следует тег `h:outputStylesheet` внутри тега `h:body`, иллюстрирует использование перемещаемого ресурса (как описано в Перемещаемые ресурсы):

```xml
<h:head>
</h:head>
<h:body>
    <h:outputStylesheet name="css/stylesheet.css" target="head"/>
```

На странице `reservation.xhtml` используются сквозные (pass-through) элементы для большинства полей формы на странице. Это позволяет ему использовать некоторые специфичные для HTML5 типы элементов `input`, такие как `date` и `email`. Например, следующий элемент отображает как формат даты, так и календарь, из которого вы можете выбрать дату. Префикс `jsf` у атрибута `id` делает элемент сквозным:

```xml
    <input type="date" jsf:id="date" name="date"
           value="#{reservationBean.date}" required="required"
           title="Enter or choose a date."/>
```

Однако в поле для количества билетов используется тег `h:passThroughAttributes` для передачи `Map`, определённой в Managed-бине. Он также пересчитывает сумму в ответ на изменение в поле:

```xml
    <h:inputText id="tickets" value="#{reservationBean.tickets}">
        <f:passThroughAttributes value="#{reservationBean.ticketAttrs}"/>
        <f:ajax event="change" render="total"
                listener="#{reservationBean.calculateTotal}"/>
    </h:inputText>
```

В поле для цены указывается тип `number` в качестве сквозного (pass-through) атрибута элемента `h:inputText`, предлагающего диапазон из четырёх цен на билеты. Здесь префикс `p` в атрибутах HTML5 передаёт их браузеру без интерпретации Jakarta Faces:

```xml
<h:inputText id="price" p:type="number"
             value="#{reservationBean.price}"
             p:min="80" p:max="120"
             p:step="20" p:required="required"
             p:title="Enter a price: 80, 100, 120, or 140.">
    <f:ajax event="change" render="total"
            listener="#{reservationBean.calculateTotal}"/>
</h:inputText>
```

Выходные данные метода `calculateTotal`, указанного в качестве слушателя для события Ajax, отрисовываются в элементе вывода, значение которого `id` и `name` равно `total`. См. Использование Ajax с Jakarta Faces для получения дополнительной информации.

Вторая страница Facelets, `confirmation.xhtml`, использует сквозной (pass-through) элемент `output` для отображения значений, введённых пользователем, и предоставляет тег Facelets `h:commandButton`, позволяющий пользователю вернуться на страницу `reservation.xhtml`.

Managed-бин для reservation

`ReservationBean.java`, сессионный Managed-бин для reservation, содержит свойства для всех элементов на страницах Facelets. Он также содержит два метода: `calculateTotal` и `clear`, которые действуют как слушатели событий Ajax на странице `reservation.xhtml`.

Сборка, упаковка и развёртывание reservation в IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/web/jsf
   ```

4. Выберите каталог `reservation`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `reservation` и выберите **Сборка**.

   Эта команда собирает приложение и развёртывает его в GlassFish Server.

Сборка, упаковка и развёртывание reservation с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/web/jsf/reservation/
   ```

3. Введите следующую команду:

   ```shell
   mvn install
   ```

Эта команда собирает и упаковывает приложение в WAR-файл, `reservation.war`, который находится в каталоге `target`. Затем он развёртывается в GlassFish Server.

Запуск reservation

На момент публикации этого руководства браузером, наиболее полно реализующим HTML5, был Google Chrome, и его рекомендуется использовать для запуска этого примера. Однако другие браузеры навёрстывают упущенное и к тому времени, когда вы будете это читать, они могут работать одинаково хорошо.

1. Введите следующий URL в браузере:

   ```
   http://localhost:8080/reservation
   ```

2. Введите информацию в поля страницы `reservation.xhtml`.

   Поле «Performance Date» содержит поле даты со стрелками вверх и вниз, которые позволяют увеличивать и уменьшать месяц, день и год, а также большую стрелку вниз, которая вызывает редактор даты в форме календаря.

   В полях «Number of Tickets» и «Ticket Price» также есть стрелки вверх и вниз, которые позволяют увеличивать и уменьшать значения в пределах допустимого диапазона и шагов. Estimated Total изменяется при изменении любого из этих двух полей.

   Адреса электронной почты и даты проверяются на соблюдение формата, но не на валидность (например, вы можете сделать заказ на прошедшую дату).

3. Нажмите «Make Reservation», чтобы завершить резервирование, или «Clear», чтобы восстановить значения полей по умолчанию.

4. Если вы нажмёте «Make Reservation», откроется страница `confirmation.xhtml`, отображающая отправленные значения.

   Нажмите Назад, чтобы вернуться на страницу `reservation.xhtml`.

Глава 9. Язык выражений (EL)

В этой главе представлен язык выражений (также называемый EL), который предоставляет важный механизм, позволяющий уровню представления (веб-страницам) взаимодействовать с логикой приложения (Managed-бинами). EL используется несколькими технологиями Jakarta EE, такими как Jakarta Faces, Jakarta Server Pages и инъецирование контекстов и зависимостей Jakarta EE (CDI). EL также может использоваться и отдельно. Эта глава охватывает только использование EL в контейнерах Jakarta EE.

Обзор EL

EL позволяет авторам страниц использовать простые выражения для динамического доступа к данным из компонентов JavaBeans. Например, атрибут `test` следующего условного тега поставляется с выражением EL, которое сравнивает 0 с количеством элементов в сессионном компоненте с именем `cart`.

```xml
<c:if test="${sessionScope.cart.numberOfItems > 0}">
  ...
</c:if>
```

См. Использование EL для ссылки на Managed-бины для получения дополнительной информации о том, как использовать EL в приложениях Jakarta Faces.

Подводя итог, EL предлагает способ использовать простые выражения для выполнения следующих задач:

- Динамическое чтение данных приложения, хранящихся в компонентах JavaBeans, различных структурах данных и неявных объектах
- Динамичесая запись данных, таких как пользовательский ввод в формы, в компоненты JavaBeans
- Вызов статических и публичных методов
- Динамическое выполнение арифметических, логических и строковых операций
- Динамическое конструирование коллекций и выполнение операций над ними

На странице Jakarta Faces выражение EL можно использовать либо в статическом тексте, либо в атрибуте кастомного тега или стандартного действия.

Наконец, EL предоставляет API для реализиации кастомных распознавателей, которые могут обрабатывать выражения, ещё не поддерживаемые EL.

Немедленное и отложенное выполнение

EL поддерживает как немедленное, так и отложенное выполнение выражений. Немедленное выполнение означает, что выражение вычисляется и результат возвращается, как только страница будет впервые отображена. Отложенное выполнение означает, что технология, использующая язык выражений, может использовать свой собственный механизм для выполнения выражения спустя некоторое время в течение жизненного цикла страницы, когда это будет уместно.

Выражения немедленного выполнения используют синтаксис `${}`. Выражения отложенного выполнения используют синтаксис `#{}`.

Из-за своего многофазного жизненного цикла Jakarta Faces использует в основном выражения отложенного выполнения. В течение жизненного цикла события компонента обрабатываются, данные валидируются, а остальные задачи выполняются в определённом порядке. Следовательно, Jakarta Faces должна отложить

оценку выражений до соответствующей точки в жизненном цикле.

Другие технологии, использующие EL, могут иметь другие причины для использования отложенных выражений.

Немедленное выполнение

Все выражения, использующие синтаксис `${}`, вычисляются немедленно. Эти выражения могут отображаться как часть текста шаблона или как значение атрибута тега, который может принимать выражения во время выполнения.

В следующем примере показан тег, чей атрибут `value` ссылается на выражение немедленного выполнения, которое обновляет количество книг, извлечённых из вспомогательного бина с именем `catalog`:

```xml
<h:outputText value="${catalog.bookQuantity}" />
```

Jakarta Faces вычисляет выражение `${catalog.bookQuantity}`, конвертирует его и передаёт полученное значение обработчику тега. Значение обновляется на странице.

Отложенное выполнение

Выражения отложенного выполнения принимают форму `#{expr}` и могут быть выполнены в других фазах жизненного цикла страницы, как это определено для каждой технологии, использующей выражения EL. В случае Jakarta Faces контроллер может выполнять выражение на разных этапах жизненного цикла, в зависимости от того, как выражение используется на странице.

В следующем примере показан тег Jakarta Faces `h:inputText`, представляющий текстовое поле, в который пользователь вводит значение. Атрибут `value` тега `h:inputText` ссылается на выражение отложенного выполнения, которое указывает на свойство `name` бина `customer`:

```xml
<h:inputText id="name" value="#{customer.name}" />
```

Для первоначального запроса страницы, содержащей этот тег, Jakarta Faces выполняет выражение `#{customer.name}` в фазе отрисовки ответа. В этой фазе выражение просто получает доступ к значению `name` из бина `customer`, как это делается при немедленном выполнении.

Для повторного запроса Jakarta Faces вычисляет выражение в разных фазах жизненного цикла, в течение которого значение извлекается из запроса, валидируется и подставляется в бин `customer`.

Как показывает этот пример, выражениями отложенного выполнения могут быть:

- Выражения значений, которые можно использовать как для чтения, так и для записи данных
- Выражения методов

Выражения значений (как немедленного, так и отложенного выполнения) и выражения методов объясняются в следующем разделе.

Выражения значений и методов

EL определяет два вида выражений: выражения значений и выражения методов. Выражения значений могут быть вычислены для получения значений, а выражения метода используются для ссылки на метод.

Выражения значений

Выражения значений можно дополнительно классифицировать на выражения *rvalue* и *lvalue*. Выражение *lvalue* может указывать конечный объект, например объект, свойство компонента или элементы коллекции, которым может быть присвоено значение. Выражение *rvalue* не может указывать такую цель.

Все выражения немедленного выполнения используют разделители `${}` и, хотя выражение может быть выражением *lvalue*, никаких присваиваний никогда не произойдёт. Выражения отложенного выполнения используют разделители `#{}` и могут действовать как выражения *rvalue* и *lvalue*. Если выражение является выражением *lvalue*, ему может быть присвоено новое значение. Рассмотрим следующие два выражения значения:

JAVA

```
${customer.name}

#{customer.name}
```

Первое использует синтаксис немедленного выполнения, тогда как второй использует синтаксис отложенного выполнения. Первое выражение обращается к свойству `name`, получает его значение и передаёт значение в обработчик тега. Со вторым выражением обработчик тега может отложить выполнение выражения на более позднее время в жизненном цикле страницы, если технология, использующая этот тег, позволяет.

В случае Jakarta Faces выражение последнего тега вычисляется непосредственно во время первоначального запроса страницы. Во время повторного запроса это выражение может использоваться для установки значения свойству `name` из пользовательского ввода.

Ссылка на объекты

Идентификатор верхнего уровня (например, `customer` в выражении `customer.name`) может ссылаться на следующие объекты:

- Лямбда-параметры
- Переменные EL
- Managed-бины
- Неявные объекты
- Классы статических полей и методов

Для обращения к этим объектам из выражения используется переменная имени объекта. Следующее выражение ссылается на Managed-бин с именем `customer`:

JAVA

```
${customer}
```

Для изменения алгоритма вычисления значений переменных может использоваться кастомный распознаватель EL. Например, можно предоставить распознаватель EL, который перехватывает объекты с именем `customer`, так что `${customer}` возвращает значение в распознавателе EL. (Jakarta Faces использует распознаватель EL для обработки Managed-бинов.)

Константа `enum` является частным случаем статического поля, и вы можете напрямую ссылаться на такую константу. Например, рассмотрим этот `enum`:

JAVA

```
public enum Suit {hearts, spades, diamonds, clubs}
```

В следующем выражении, в котором `mySuit` является объектом `Suit`, вы можете сравнить `suit.hearts` с объектом:

```java
${mySuit == suit.hearts}
```

Ссылка на свойства объекта или элементы коллекции

Чтобы ссылаться на свойства бина, статические поля или методы класса или элементы коллекции, используйте нотацию `.` или `[]`. Тот же синтаксис можно использовать для атрибутов неявного объекта, поскольку атрибуты размещены в отображении Map.

Для ссылки на свойство `name` компонента `customer` используйте либо выражение `${customer.name}`, либо выражение `${customer["name"]}`. Здесь часть внутри скобок является литералом `String`, который является именем свойства для ссылки. Синтаксис `[]` является более общим, чем синтаксис `.`, потому что часть внутри скобок может быть любым выражением `String`, а не только литералами.

Для строковых литералов могут использоваться как двойные, так и одинарные кавычки. Вы также можете объединить нотации `[]` и `.`, как показано здесь:

```java
${customer.address["street"]}
```

Вы можете ссылаться на статическое поле или метод, используя синтаксис classname.field, как в следующем примере:

```java
Boolean.FALSE
```

Имя класса — это имя класса без имени пакета. По умолчанию импортируются все пакеты `java.lang`. При необходимости вы можете импортировать другие пакеты, классы и статические поля.

Обращаясь к элементу в массиве или списке, вы должны использовать нотацию `[]` и указывать индекс в массиве или списке. Индекс — это выражение, которое можно конвертировать в `int`. Следующий пример ссылается на первый из заказов клиента, предполагая, что `customer.orders` имеет тип `List`:

```java
${customer.orders[1]}
```

Если вы обращаетесь к элементу в `Map`, вы должны указать ключ для `Map`. Если ключ является литералом `String`, можно использовать точку (`.`). Предполагая, что `customer.orders` является `Map` с ключом `String`, следующие примеры ссылаются на элемент с ключом «socks»:

```java
${customer.orders["socks"]}

${customer.orders.socks}
```

Ссылка на литералы

EL определяет следующие литералы:

- Логический: `true` и `false`
- Integer: как в Java
- Числа с плавающей запятой: как в Java

- String: с одинарными и двойными кавычками. `"` экранируется как `\"`, `'` — как `\'` и `\` — как `\\`
- Null: `null`

Вот некоторые примеры:

- `${"literal"}`
- `${true}`
- `${57}`

Параметризованные вызовы методов

EL предлагает поддержку параметризованных вызовов методов.

Операторы `.` и `[]` могут использоваться для вызова вызовов методов с параметрами, как показано в следующем синтаксисе выражения:

- *`expr-a`*`[`*`expr-b`*`](`*`parameters`*`)`
- *`expr-a`*`.`*`identifier-b`*`(`*`parameters`*`)`

В синтаксисе первого выражения *expr-a* выполняется для представления объекта компонента. Выражение *expr-b* выполняется и преобразуется в строку, которая представляет метод в компоненте, представленном *expr-a*. Во втором синтаксисе выражение *expr-a* выполняется для представления объекта компонента, а *identifier-b* — это строка, представляющая метод в объекте компонента. *parameters* в скобках являются аргументами для вызова метода. Параметрами могут быть любое количество выражений значений, разделённых запятыми.

Параметры поддерживаются как для выражений значений, так и для выражений методов. В следующем примере, который является изменённым тегом из приложения `guessnumber` , случайное число предоставляется в качестве аргумента, а не из пользовательского ввода для вызова метода:

```xml
<h:inputText value="#{userNumberBean.userNumber('5')}">
```

В предыдущем примере используется выражение значения.

Рассмотрим следующий пример тега компонента Jakarta Faces, который использует выражение метода:

```xml
<h:commandButton action="#{trader.buy}" value="buy"/>
```

Выражение EL `trader.buy` вызывает метод `buy` бина `trader` . Вы можете изменить тег для передачи параметра. Вот исправленный тег, в котором передаётся параметр:

```xml
<h:commandButton action="#{trader.buy('SOMESTOCK')}" value="buy"/>
```

В предыдущем примере вы передаёте строку `'SOMESTOCK'` (символ акции) в качестве параметра методу `buy` .

Где могут использоваться выражения значения

Можно использовать выражения значений с использованием разделителей `${}`

- В статическом тексте
- В любом стандартном или кастомном атрибуте тега, который может принимать выражение

Значение выражения в статическом тексте вычисляется и вставляется в текущий вывод. Вот пример выражения, встроенного в статический текст:

```xml
<some:tag>
    some text ${expr} some text
</some:tag>
```

Атрибут тега можно установить следующими способами.

- С единственной конструкцией выражения:

  ```xml
  <some:tag value="${expr}"/>

  <another:tag value="#{expr}"/>
  ```

  Эти выражения вычисляются, а результат конвертируется к ожидаемому типу атрибута.

- С одним или несколькими выражениями, разделёнными или окружёнными текстом:

  ```xml
  <some:tag value="some${expr}${expr}text${expr}"/>

  <another:tag value="some#{expr}#{expr}text#{expr}"/>
  ```

  Эти виды выражений, называемые составными выражениями, вычисляются слева направо. Каждое выражение, встроенное в составное выражение, конвертируется в `String` и затем объединяется с любым промежуточным текстом. Результирующая `String` затем конвертируется к ожидаемому типу атрибута.

- Только с текстом:

  ```xml
  <some:tag value="sometext"/>
  ```

  Значение атрибута `String` конвертируется к ожидаемому типу атрибута.

Вы можете использовать оператор конкатенации строк `+=` чтобы создать одно выражение из того, что иначе было бы составным выражением. Например, вы можете изменить составное выражение

```xml
<some:tag value="sometext ${expr} moretext"/>
```

на

```xml
<some:tag value="${sometext += expr += moretext}"/>
```

Все выражения, используемые для установки значений атрибутов, вычисляются в контексте ожидаемого типа. Если результат вычисления выражения не соответствует в точности ожидаемому типу, будет выполнена конвертация. Например, выражение `${1.2E4}`, предоставленное как значение атрибута типа `float`, приведёт к следующей конвертации:

```java
Float.valueOf("1.2E4").floatValue()
```

Выражения методов

Ещё одна особенность EL — поддержка выражений отложенного выполнения методов. Выражение метода используется для ссылки на публичный метод компонента и имеет тот же синтаксис, что и выражение *lvalue*.

В Jakarta Faces тег компонента представляет компонент на странице. Тег компонента использует выражения метода для указания методов, которые могут быть вызваны для выполнения некоторой обработки для компонента. Эти методы необходимы для обработки событий, которые генерируют компоненты, и для валидации данных компонентов, как показано в этом примере:

```xml
<h:form>
    <h:inputText id="name" value="#{customer.name}" validator="#{customer.validateName}"/>
    <h:commandButton id="submit" action="#{customer.submit}" />
</h:form>
```

Тег `h:inputText` отображается в виде поля. Атрибут `validator` этого тега `h:inputText` ссылается на метод `validateName` в бине `customer`.

Поскольку метод может быть вызван в разных фазах жизненного цикла, выражения метода всегда должны использовать синтаксис отложенного выполнения.

Как и выражения lvalue, выражения методов могут использовать операторы `.` и `[]`. Например, `#{object.method}` эквивалентно `#{object["method"]}`. Литерал внутри `[]` конвертируется в `String` и используется для поиска имени метода, который ему соответствует.

Выражения метода могут использоваться только в атрибутах тега и только следующими способами:

- С единственной конструкцией выражения, где бин ссылается на компонент JavaBeans, а метод ссылается на метод компонента JavaBeans:

  ```xml
  <some:tag value="#{bean.method}"/>
  ```

  Выражение вычисляется как выражение метода, которое передаётся обработчику тега. Метод, представленный выражением метода, может быть вызван позже.

- Только с текстом:

  ```xml
  <some:tag value="sometext"/>
  ```

  Выражения метода поддерживают литералы в первую очередь для поддержки атрибутов `action` в Jakarta Faces. Когда вызывается метод, на который ссылается выражение этого метода, метод возвращает литерал `String`, который затем конвертируется к ожидаемому возвращаемому типу, как определено в дескрипторе библиотеки тегов.

Лямбда-выражения

Лямбда-выражение — это выражение значения с параметрами. Синтаксис аналогичен лямбда-выражениям в Java с тем исключением, что в EL тело лямбда-выражения является выражением EL.

Для получения основной информации о лямбда-выражениях см. https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html.

Лямбда-выражения являются частью Java SE 8

Лямбда-выражение использует токен стрелки (`->`). Идентификаторы слева от оператора называются лямбда-параметрами. Тело справа от оператора должно быть выражением EL. Параметры лямбда заключены в скобки. Круглые скобки могут быть опущены, если есть только один параметр. Вот некоторые примеры:

JAVA

```
x -> x+1
(x, y) -> x + y
() -> 64
```

Лямбда-выражение ведёт себя как функция. Оно может быть вызвано немедленно. Например, следующий вызов при вычислении даёт результат 7:

JAVA

```
((x, y) -> x + y)(3, 4)
```

Вы можете использовать лямбда-выражение в сочетании с операторами присваивания и точки с запятой. Например, следующий код назначает предыдущее лямбда-выражение переменной, а затем вызывает его. Результат снова 7:

JAVA

```
v = (x, y) -> x + y; v(3, 4)
```

Лямбда-выражение также можно передавать в качестве аргумента методу и вызывать в методе. Он также может быть вложен в другое лямбда-выражение.

Операции над коллекциями

EL поддерживает операции с объектами коллекции: наборы (Set), списки (List) и отображения (Map). Это позволяет динамически создавать объекты коллекции, которыми затем можно управлять, используя потоки и конвейеры.

Как и лямбда-выражения, операции над объектами коллекции являются частью Java SE 8.

Например, вы можете построить набор следующим образом:

JAVA

```
{1,2,3}
```

Вы можете построить список следующим образом. Список может содержать элементы различных типов:

JAVA

```
[1,2,3]
[1, "two", [three,four]]
```

Вы можете создать отображение (Map), используя двоеточие для определения записей следующим образом:

JAVA

```
{"one":1, "two":2, "three":3}
```

Вы работаете с объектами коллекции, используя вызовы методов для потока элементов, полученных из коллекции. Некоторые операции возвращают другой поток, что позволяет выполнять дополнительные операции. Таким образом, вы можете объединить эти операции в очередь.

Очередь потока состоит из следующего:

- Источник (объект `Stream`)
- Любое количество промежуточных операций, которые возвращают поток (например, `filter` и `map`)
- Терминальная операция, которая не возвращает поток (например, `toList()`)

Метод `stream` получает `Stream` из `java.util.Collection` или массива Java. Операции потока не изменяют исходный объект коллекции.

Например, вы можете создать список названий книг по истории следующим образом:

```java
books.stream().filter(b->b.category == "history")
              .map(b->b.title)
              .toList()
```

Следующий простой пример возвращает отсортированную версию исходного списка:

```java
[1,3,5,2].stream().sorted().toList()
```

Потоки и потоковые операции задокументированы в API Java SE 8, доступной по ссылке https://docs.oracle.com/javase/8/docs/api/. EL поддерживает следующее подмножество операций:

`allMatch`	`anyMatch`	`average`	`count`
`distinct`	`filter`	`findFirst`	`flatMap`
`forEach`	`iterator`	`limit`	`map`
`max`	`min`	`noneMatch`	`peek`
`reduce`	`sorted`	`substream`	`sum`
`toArray`	`toList`		

Подробные сведения об этих операциях см. в спецификации языка выражений по ссылке https://jakarta.ee/specifications/expression-language/4.0/.

Операторы

В дополнение к операторам `.` и `[]`, обсуждаемым в выражениях значений и методов, EL предоставляет следующие операторы, которые можно использовать только в выражениях rvalue.

- Арифметические: `+`, `-` (бинарная операция), `*`, `/` и `div`, `%` и `mod`, `-` (унарная операция).
- Конкатенация строк: `+=`.
- Логический: `and`, `&&`, `or`, `||`, `not`, `!`.
- Реляционный: `==`, `eq`, `!=`, `ne`, `<`, `lt`, `>`, `gt`, `<=`, `ge`, `>=`, `le`. Сравнения могут быть сделаны с другими значениями или с логическими, строковыми, целочисленными литералами или литералами с плавающей точкой.
- Пустой: оператор `empty` является префиксной операцией, которую можно использовать для определения, является ли значение `null` или пустым.
- Условный: `A ? B : C`. Выполнение `B` или `C`, в зависимости от результата `A`.
- Лямбда-выражение: `->`, токен стрелки.
- Назначение: `=`.
- Точка с запятой: `;`.

Приоритет операторов, сверху вниз, слева направо, выглядит следующим образом:

- `[].`
- `()` (используется для изменения приоритета операторов)
- `-` (одинарный) `not ! empty`
- `* / div % mod`
- `+ -` (бинарный)
- `+=`
- `<> <= >= lt gt le ge`
- `== != eq ne`
- `&& and`
- `|| or`
- `? :`
- `->`
- `=`
- `;`

Зарезервированные слова

Следующие слова зарезервированы для EL и не должны использоваться в качестве идентификаторов:

`and`	`or`	`not`	`eq`
`ne`	`lt`	`gt`	`le`
`ge`	`true`	`false`	`null`
`instanceof`	`empty`	`div`	`mod`

Примеры выражений EL

Таблица 9-1 содержит примеры выражений EL и результат их вычисления.

Таблица 9-1. Примеры выражений

EL Expression	**Результат**
`${1 > (4/2)}`	`false`
`${4.0 >= 3}`	`true`
`${100.0 == 100}`	`true`
`${(10*10) ne 100}`	`false`
`${'a' > 'b'}`	`false`
`${'hip' lt 'hit'}`	`true`

EL Expression	Результат
`${4 > 3}`	`true`
`${1.2E4 + 1.4}`	`12001.4`
`${3 div 4}`	`0.75`
`${10 mod 4}`	`2`
`${((x, y) → x + y)(3, 5.5)}`	`8.5`
`[1,2,3,4].stream().sum()`	`10`
`[1,3,5,2].stream().sorted().toList()`	`[1, 2, 3, 5]`
`${!empty param.Add}`	`False` , если параметр запроса с именем `Add` равен `null` или пустой строке
`${pageContext.request.contextPath}`	Контекстный путь
`${sessionScope.cart.numberOfItems}`	Значение свойства `numberOfItems` атрибута области видимости сессии с именем `cart`
`${param['mycom.productId']}`	Значение параметра запроса с именем `mycom.productId`
`${header["host"]}`	Сервер
`${departments[deptName]}`	Значение записи с именем `deptName` в карте `департаментов`
`${requestScope['jakarta.servlet.forward.servlet_path']}`	Значение атрибута области видимости запроса с именем `jakarta.servlet.forward.servlet_path`
`#{customer.lName}`	Получает значение свойства `lName` из компонента `customer` при первоначальном запросе. Устанавливает значение `lName` при повторный передаче
`#{customer.calcTotal}`	Возвращаемое значение метода `calcTotal` компонента `customer`

Дополнительная информация о EL

Дополнительные сведения о языке выражений см. в разделе

- Спецификация Expression Language 4.0: https://jakarta.ee/specifications/expression-language/4.0/
- Сайт спецификаци EL: https://github.com/eclipse-ee4j/el-ri/tree/master/spec

Глава 10. Использование Jakarta Faces на веб-страницах

Веб-страницы (в большинстве случаев страницы Facelets) представляют собой уровень представления для веб-приложений. Процесс создания веб-страниц для приложения Jakarta Faces включает использование тегов компонентов для добавления компонентов на страницу и их связывания со вспомогательными бинами, валидаторами, слушателями, конвертерами и другими серверными объектами, ассоциированными со страницей.

В этой главе объясняется, как создавать веб-страницы, используя различные типы компонентов и основные теги. В следующей главе вы узнаете о добавлении конвертеров, валидаторов и слушателей к тегам компонентов, чтобы обеспечить дополнительную функциональность для компонентов.

Многие из примеров этой главы взяты из главы 61 *Пример Duke's Bookstore*

Настройка страницы

Типичная веб-страница Jakarta Faces включает следующие элементы:

- Набор объявлений пространства имён, которые объявляют библиотеки тегов Jakarta Faces
- При желании, теги HTML head (`h:head`) и body (`h:body`)
- Тег формы (`h:form`), который представляет компоненты пользовательского ввода

Чтобы добавить компоненты Jakarta Faces на веб-страницу, необходимо предоставить доступ к двум стандартным библиотекам тегов: библиотеке тегов набора отрисовки HTML Jakarta Faces и библиотеке основных тегов Jakarta Faces. Jakarta Faces standard HTML tag library (https://jakarta.ee/specifications/faces/3.0/renderkitdoc/) определяет теги, представляющие общие компоненты интерфейса пользователя HTML. Библиотека основных тегов Jakarta Faces определяет теги, которые выполняют основные действия и не зависят от конкретного инструментария отрисовки.

Для получения полного списка тегов Facelets Jakarta Faces и их атрибутов обратитесь к документации Jakarta Faces Facelets Tag Library (https://jakarta.ee/specifications/faces/3.0/vdldoc/).

Чтобы использовать любой из тегов Jakarta Faces, необходимо включить соответствующие директивы вверху каждой страницы, определяющие библиотеки тегов.

Для приложений Facelets директивы пространства имён XML однозначно идентифицируют URI библиотеки тегов и префикс тега.

Например, когда вы создаёте страницу Facelets XHTML, включите директивы пространства имён следующим образом:

```xml
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
```

URI пространства имён XML идентифицирует расположение библиотеки тегов, а значение префикса используется для различения тегов, принадлежащих этой конкретной библиотеке тегов. Вы также можете использовать другие префиксы вместо стандартных `h` или `f` . Однако при включении тега на страницу вы должны использовать префикс, выбранный для библиотеки тегов. Например, на следующей веб-странице ссылка на тег `form` должна использоваться с префиксом `h` , поскольку в предыдущей директиве библиотеки тегов используется префикс `h` для различайте теги, определённые в библиотеке тегов HTML:

```xml
<h:form ...>
```

В разделах Добавление компонентов на страницу с использованием библиотеки тегов HTML и Использование основных тегов описано, как использовать теги компонентов из стандартной библиотеки тегов HTML Jakarta Faces и основные теги из библиотека тегов ядра Jakarta Faces.

Добавление компонентов на страницу с помощью библиотеки тегов HTML

Теги, определённые стандартной библиотекой тегов HTML Jakarta Faces, представляют компоненты формы HTML и другие основные элементы HTML. Эти компоненты отображают данные или принимают данные от пользователя. Эти данные собираются как часть формы и отправляются на сервер. Обычно, это происходит когда пользователь кликает кнопку. В этом разделе объясняется, как использовать каждый из тегов компонента, показанных в таблице 10-1.

Таблица 10-1 Теги компонентов

Тег	**Функции**	**Представление**	**Внешний вид**
`h:column`	Представляет столбец данных в компоненте данных	Столбец данных в HTML таблице	Столбец в таблице
`h:commandButton`	Передаёт форму с данными на сервер	HTML-элемент `<input type=value>` для которого `type` может принимать значения «submit», «reset» или «image»	Кнопка
`h:commandLink`	Ссылки на другую страницу или местоположение на странице	HTML-элемент `<a href>`	Ссылка
`h:dataTable`	Представляет обёртку (wrapper) данных	HTML-элемент `<table>`	Таблица, которая может быть динамически обновлена
`h:form`	Представляет форму ввода (внутренние теги формы получают данные, которые будут отправлены вместе с формой)	HTML-элемент `<form>`	Нет визуального представления
`h:graphicImage`	Отображает изображение	HTML-элемент `<img>`	Изображение
`h:inputFile`	Позволяет пользователю загрузить файл	HTML-элемент `<input type="file">`	Поле с кнопкой Обзор ...

Тег	Функции	Представление	Внешний вид
`h:inputHidden`	Позволяет разработчику страницы добавить скрытую переменную на страницу	HTML-элемент `<input type="hidden">`	Нет визуального представления
`h:inputSecret`	Позволяет пользователю безопасно ввести пароль	HTML-элемент `<input type="password">`	Поле, которое отображает строку символов вместо фактически введённой строки
`h:inputText`	Позволяет пользователю вводить строку	HTML-элемент `<input type="text">`	Текстовое поле
`h:inputTextarea`	Позволяет пользователю вводить несколько строк	HTML-элемент `<textarea>`	Многострочное поле
`h:message`	Отображает локализованное сообщение	HTML-тег `<span>`, если используются стили	Текстовая строка
`h:messages`	Отображает локализованные сообщения	Набор тегов HTML `<span>`, если используются стили	Текстовая строка
`h:outputFormat`	Отображает отформатированное сообщение	Простой текст	Простой текст
`h:outputLabel`	Отображает вложенный компонент в качестве метки для указанного поля ввода	HTML-элемент `<label>`	Простой текст
`h:outputLink`	Ссылки на другую страницу или местоположение на странице без создания события действия	HTML-элемент `<a>`	Ссылка
`h:outputText`	Отображает строку текста	Простой текст	Простой текст
`h:panelGrid`	Отображает таблицу	HTML-элемент `<table>` с элементами `<tr>` и `<td>`	Таблица
`h:panelGroup`	Группирует набор компонентов под одним родителем	HTML-элемент `<div>` или `<span>`	Строка в таблице

Тег	Функции	Представление	Внешний вид
`h:selectBooleanCheckbox`	Позволяет пользователю изменять логическое значение	HTML-элемент `<input type="checkbox">`	Флажок
`h:selectManyCheckbox`	Отображает набор флажков, из которых пользователь может выбрать несколько значений	Набор HTML `<input>` элементов типа `checkbox`	Группа флажков
`h:selectManyListbox`	Позволяет пользователю выбрать несколько элементов из набора элементов	HTML-элемент `<select>`	Список выбора
`h:selectManyMenu`	Позволяет пользователю выбрать несколько элементов из набора элементов	HTML-элемент `<select>`	Меню
`h:selectOneListbox`	Позволяет пользователю выбрать один элемент из набора элементов	HTML-элемент `<select>`	Список выбора
`h:selectOneMenu`	Позволяет пользователю выбрать один элемент из набора элементов	HTML-элемент `<select>`	Меню
`h:selectOneRadio`	Позволяет пользователю выбрать один элемент из набора элементов	HTML-элемент `<input type="radio">`	Группа вариантов Для автономного переключателя используйте атрибут `group`.

Теги соответствуют компонентам в пакете `jakarta.faces.component`. Компоненты обсуждаются более подробно в главе 12 *Разработка с использованием технологии Jakarta Faces*

В следующем разделе объясняются важные атрибуты, общие для большинства тегов компонентов. Для каждого из компонентов, обсуждаемых в следующих разделах, Запись свойств бина объясняет, как записать свойство бина, связанное с этим конкретным компонентом или его значением.

Справочные сведения о тегах и их атрибутах см. в разделе Документация Jakarta Faces Facelets Tag Library (https://jakarta.ee/specifications/faces/3.0/vdldoc/).

Общие атрибуты тегов компонентов

Большинство тегов компонентов поддерживают атрибуты, показанные в табл. 10-2.

Таблица 10-2 Общие атрибуты тегов компонентов

Атрибут	Описание
`binding`	Идентифицирует свойство компонента и связывает его с объектом компонента.
`id`	Уникально идентифицирует компонент.
`immediate`	Если установлено значение `true`, указывает, что для значений этого компонента должны применяться события, валидация и конвертация, связанные с компонентом.
`rendered`	Задаёт условие отображения компонента. Если условие не выполняется, компонент не отображается.
`style`	Задаёт стиль каскадной таблицы стилей (CSS) для тега.
`styleClass`	Определяет класс CSS, который содержит css-стили.
`value`	Определяет значение компонента в форме выражения.

Все атрибуты тегов, кроме `id`, могут принимать выражения, определённые языком выражений, описанным в главе 9 *Язык выражений*.

Атрибут, такой как `render` или `value`, может быть установлен на странице и затем изменён во вспомогательном бине.

Атрибут id

Атрибут `id` обычно не требуется для тега компонента, но используется, когда другой компонент или класс на стороне сервера должен ссылаться на компонент. Если вы не включите атрибут `id`, Jakarta Faces автоматически сгенерирует ID компонента. В отличие от большинства других атрибутов тегов Jakarta Faces, атрибут `id` принимает выражения, используя только синтаксис немедленного выполнения, в котором используются разделители `${}`. Для получения дополнительной информации о синтаксисе выражений см. Выражения значений.

Атрибут immediate

Компоненты ввода и команд (т.е. реализующие интерфейс `ActionSource`, такие как кнопки и ссылки) могут установить атрибут `immediate` в значение `true`, чтобы форсировать обработку событий, валидации и конвертации, которые будут срабатывать при применении параметров запроса.

Необходимо тщательно продумать, как комбинация значения `immediate` компонента ввода и значения `immediate` компонента команды определяет, что происходит при активации компонента команды.

Предположим, что у вас есть страница с кнопкой и полем ввода количества книг в корзине. Если для атрибутов `immediate` кнопки и поля установлено значение `true`, новое значение, введённое в поле, будет доступно для любой обработки, связанной с генерируемым событием когда кнопка нажата. Событие, связанное с кнопкой, а также события, валидация и конвертация, связанные с полем, обрабатываются при применении параметров запроса.

Если для атрибута `immediate` кнопки установлено значение `true`, а для атрибута `immediate` поля установлено значение `false`, событие связывается с кнопкой обрабатывается без обновления локального значения поля для слоя модели. Причина заключается в том, что любые события, конвертации и валидации, связанные с полем, происходят после применения параметров запроса.

Страница `bookshowcart.xhtml` приложения Duke's Bookstore содержит примеры компонентов, использующих атрибут `immediate` для управления тем, какие данные компонента обновляются при клике определённых кнопок. Поле `quantity` для каждой книги не устанавливает атрибут `immediate`, поэтому значение равно `false` (по умолчанию).

```xml
<h:inputText id="quantity"
             size="4"
             value="#{item.quantity}"
             title="#{bundle.ItemQuantity}">
    <f:validateLongRange minimum="0"/>
    ...
</h:inputText>
```

Атрибут `immediate` гиперссылки «Continue Shopping» установлен в `true`, а атрибут `immediate` гиперссылки «Update Quantities» установлен в `false`:

```xml
<h:commandLink id="continue"
               action="bookcatalog"
               immediate="true">
    <h:outputText value="#{bundle.ContinueShopping}"/>
</h:commandLink>
...
<h:commandLink id="update"
               action="#{showcart.update}"
               immediate="false">
    <h:outputText value="#{bundle.UpdateQuantities}"/>
</h:commandLink>
```

Если кликнуть гиперссылку «Continue Shopping», ни одно из изменений, введённых в поле ввода `quantity`, не будет обработано. Если кликнуть гиперссылку «Update Quantities», значение в поле `quantity` будет обновлено в корзине покупок.

Атрибут rendered

Тег компонента использует логическое выражение EL вместе с атрибутом `rendered`, чтобы определить, будет ли отображаться компонент. Например, компонент `commandLink` в следующем разделе страницы не отображается, если в корзине нет элементов:

```xml
<h:commandLink id="check" ... rendered="#{cart.numberOfItems > 0}">
    <h:outputText value="#{bundle.CartCheck}"/>
</h:commandLink>
```

В отличие от почти любого другого атрибута тега Jakarta Faces, атрибут `rendered` ограничен использованием выражений rvalue. Как объяснено в выражениях значений и методов, выражения с rvalue могут только читать данные. Они не могут записать данные обратно в источник данных. Следовательно, выражения, используемые в атрибуте `rendered`, могут использовать арифметические операторы и литералы, которые могут использовать выражения rvalue, но не выражения lvalue. Например, выражение в предыдущем примере использует оператор `>`.

В этом и других примерах `bundle` ссылается на файл `java.util.ResourceBundle`, который отображает строки, зависящие от установленной локали. Bundle-ресурсы обсуждаются в главе 22 *Интернационализация и локализация веб-приложений*.

Атрибуты style и styleClass

Атрибуты `style` и `styleClass` позволяют указать стили CSS для отрисовки ваших тегов. Отображение сообщений об ошибках с тегами h:message и h:messages описывает пример использования атрибута `style` для указания стилей непосредственно в атрибуте. Вместо этого тег компонента может ссылаться на класс CSS.

В следующем примере показано использование тега `dataTable`, который ссылается на класс стиля `list-background`:

XML

```
<h:dataTable id="items"
             ...
             styleClass="list-background"
             value="#{cart.items}"
             var="book">
```

Этот класс определяется таблицей стилей `stylesheet.css`, которую нужно включить в состав приложения. Дополнительные сведения об определении стилей см. в спецификациях и черновиках Cascading Style Sheets по ссылке https://www.w3.org/Style/CSS/.

Атрибуты value и binding

Тег, представляющий компонент вывода, использует атрибуты `value` и `binding`, чтобы связать значение или объект компонента с соответствующим объектом данных. Атрибут `value` используется чаще, чем атрибут `binding`. Примеры его использования приведены в этой главе. Для получения дополнительной информации об этих атрибутах см. Создание Managed-бина, Запись свойств объектов, связанных со значениями компонентов, и Запись свойств, связанных с объектами компонентов.

Добавление в HTML тегов Head и Body

HTML-теги head (`h:head`) и body (`h:body`) добавляют структуру HTML-страниц к веб-страницам Jakarta Faces.

- Тег `h:head` представляет элемент head страницы HTML.
- Тег `h:body` представляет элемент body страницы HTML.

Ниже приведён пример страницы XHTML с использованием обычных тегов разметки заголовка и тела:

XML

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>Add a title</title>
    </head>
    <body>
        Add Content
    </body>
</html>
```

Ниже приведён пример страницы XHTML с использованием тегов `h:head` и `h:body`:

```xml
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        Add a title
    </h:head>
    <h:body>
        Add Content
    </h:body>
</html>
```

Оба предыдущих сегмента кода примера отображают одинаковые элементы HTML. Теги head и body полезны в основном для перемещения ресурсов. Для получения дополнительной информации о перемещении ресурсов см. Перемещение ресурсов с использованием тегов h:outputScript и h:outputStylesheet.

Добавление компонента Form

Тег `h:form` представляет форму ввода, включающую дочерние компоненты, которые могут содержать данные для представления пользователю или отправки на сервер.

На рисунке 10-1 показана типичная форма входа в систему, в которой пользователь вводит имя пользователя и пароль, а затем отправляет форму, кликая кнопку входа в систему.

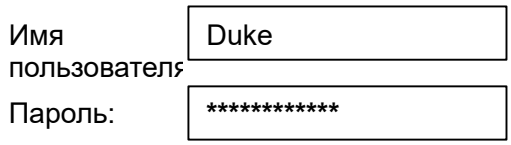

Рисунок 10-1 Типичная форма

Тег `h:form` представляет форму на странице и включает в себя все компоненты, которые отображают или собирают данные от пользователя, как показано здесь:

```xml
<h:form>
... другие теги Jakarta Faces и остальное содержимое...
</h:form>
```

Тег `h:form` также может включать разметку HTML для размещения компонентов на странице. Обратите внимание, что сам тег `h:form` не выполняет никакой разметки: его целью является сбор данных и объявление атрибутов, которые могут использоваться другими компонентами на форме.

Страница может содержать несколько тегов `h:form`, но только значения из формы, отправленной пользователем, будут включены в повторный запрос для отправки на сервер.

Использование компонентов Text

Текстовые компоненты позволяют пользователям просматривать и редактировать текст в веб-приложениях. Основные типы текстовых компонентов:

- Метка, которая отображает текст только для чтения
- Текстовое поле, которое позволяет пользователям вводить текст (в одну или несколько строк), часто для отправки в составе формы
- Поле пароля, представляющего собой тип поля, которое отображает набор символов, таких как звездочки, вместо текста пароля, вводимого пользователем

На рисунке 10-2 показаны примеры этих текстовых компонентов.

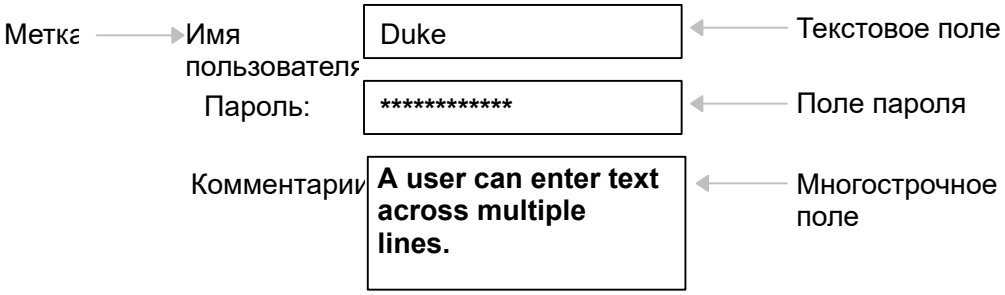

Рисунок 10-2 Пример текстовых компонентов

Текстовые компоненты могут быть классифицированы как входные или выходные данные. Компонент вывода Jakarta Faces, такой как метка, отображается как текст только для чтения. Компонент ввода Jakarta Faces, например текстовое поле, отображается как редактируемый текст.

Компоненты ввода и вывода могут отрисовываться различными способами для отображения более специализированного текста.

Таблица 10-3 перечисляет теги, которые представляют компоненты ввода.

Таблица 10-3 Входные теги

Тег	Функция
`h:inputHidden`	Позволяет разработчику страницы добавить скрытую переменную на страницу
`h:inputSecret`	Стандартное поле пароля: принимает одну строку текста без пробелов и отображает её в виде набора звёздочек при вводе
`h:inputText`	Стандартное текстовое поле: принимает одну строку текста
`h:inputTextarea`	Стандартное многострочное поле: принимает несколько строк текста

Теги ввода поддерживают атрибуты, показанные в таблице 10-4 в дополнение к описанным в Общих атрибутах тегов компонентов. Обратите внимание, что эта таблица включает не все атрибуты, поддерживаемые тегами ввода, а только наиболее часто используемые. Полный список атрибутов см. в разделе Документация Jakarta Faces Facelets Tag Library (https://jakarta.ee/specifications/faces/3.0/vdldoc/).

Таблица 10-4 Атрибуты входных тегов

Атрибут	Описание
`converter`	Определяет конвертер, который будет использоваться для конвертирования локальных данных компонента. Смотрите Использование стандартных конвертеров для получения дополнительной информации о том, как использовать этот атрибут.
`converterMessage`	Указывает сообщение об ошибке, отображаемое в случае её возникновения при работе конвертера.

Атрибут	Описание
`dir`	Определяет направление текста, отображаемого этим компонентом. Допустимые значения: `ltr` — слева направо, `rtl` — справа налево.
`label`	Задаёт имя, которое можно использовать для идентификации этого компонента в сообщениях об ошибках.
`lang`	Указывает код для языка (естественного), используемого на странице, например `en` или `pt-BR`.
`required`	Принимает `boolean` значение, которое указывает обязательность заполнения этого компонента обязательным.
`requiredMessage`	Указывает сообщение об ошибке, которое отображается, если пользователь оставляет компонент незаполненным.
`validator`	Определяет выражение метода, указывающее на метод Managed-бина, который выполняет валидацию данных компонента. См. Ссылка на метод, который выполняет валидацию для примера использования тега `f:validator`.
`validatorMessage`	Указывает сообщение об ошибке, отображаемое в случае её возникновения при работе валидатора.
`valueChangeListener`	Определяет выражение метода, указывающее на метод Managed-бина, который обрабатывает событие ввода значения в этот компонент. См. Ссылка на метод-обработчик изменения значения для примера использования `valueChangeListener`.

Таблица 10-5 перечисляет теги, которые представляют компоненты вывода.

Таблица 10-5 Теги вывода

Тег	Функция
`h:outputFormat`	Отображает отформатированное сообщение
`h:outputLabel`	Стандартная метка "только для чтения": отображает компонент-метку для указанного поля ввода
`h:outputLink`	Отображает тег `<a href>`, который ссылается на другую страницу без генерации события действия
`h:outputText`	Отображает одиночную строку текста

Выходные теги поддерживают атрибут тега `converter` в дополнение к перечисленным в Общих атрибутах тегов компонентов.

В оставшейся части этого раздела объясняется, как использовать некоторые из тегов, перечисленных в таблице 10-5. Другие теги написаны аналогичным образом.

Отображение поля тегом h:inputText

Тег `h:inputText` используется для отображения текстового поля. Аналогичный тег, `h:outputText`, отображает одиночную строку текста "только для чтения". В этом разделе показано, как использовать тег `h:inputText`. Тег `h:outputText` записывается аналогичным образом.

Вот пример тега `h:inputText`:

```xml
<h:inputText id="name"
             label="Customer Name"
             size="30"
             value="#{cashierBean.name}"
             required="true"
             requiredMessage="#{bundle.ReqCustomerName}">
    <f:valueChangeListener
        type="ee.jakarta.tutorial.dukesbookstore.listeners.NameChanged" />
</h:inputText>
```

Атрибут `label` указывает понятное имя, которое будет использоваться в параметрах замещения сообщений об ошибках, отображаемых для этого компонента.

Атрибут `value` ссылается на свойство `name` Managed-бина с именем `CashierBean`. Это свойство содержит данные для компонента `name`. После того, как пользователь отправит форму на сервер, значение свойства `name` в `CashierBean` будет установлено в текст, введённый в поле, соответствующее этому тегу.

Атрибут `required` вызывает перерисовку страницы с отображением ошибки, если пользователь не указал значение в поле `name`. Jakarta Faces проверяет, является ли значение компонента null или пустой строкой.

Если ваш компонент должен иметь значение не-null и `String` как минимум длиной в один символ, вы должны добавить атрибут `required` в тег и установить для него значение `true`. Если ваш тег имеет атрибут `required`, для которого установлено значение `true`, а значение равно null или строке нулевой длины, никакие другие валидаторы, зарегистрированные в теге, не вызываются. Если у вашего тега нет атрибута `required`, установленного в `true`, вызываются другие валидаторы, зарегистрированные в теге, но эти валидаторы должны обрабатывать возможность null или строки нулевой длины. Смотрите Валидация строк на null и пустоту для получения дополнительной информации.

Отображение поля пароля тегом h:inputSecret

Тег `h:inputSecret` отображает HTML-тег `<input type="password">`. Когда пользователь вводит строку в это поле, вместо вводимого текста отображается строка звёздочек. Вот пример:

```xml
<h:inputSecret redisplay="false" value="#{loginBean.password}" />
```

В этом примере атрибуту `redisplay` присвоено значение `false`. Это предотвращает отображение пароля в строке запроса или в исходном файле получившейся HTML-страницы.

Отображение метки тегом h:outputLabel

Тег `h:outputLabel` используется для прикрепления метки к указанному полю ввода с целью сделать его доступным. На следующей странице используется тег `h:outputLabel` для отображения метки флажка:

```xml
<h:selectBooleanCheckbox id="fanClub"
                         rendered="false"
                         binding="#{cashierBean.specialOffer}" />
<h:outputLabel for="fanClub"
               rendered="false"
               binding="#{cashierBean.specialOfferText}">
    <h:outputText id="fanClubLabel"
                  value="#{bundle.DukeFanClub}" />
</h:outputLabel>
...
```

Теги `h:selectBooleanCheckbox` и `h:outputLabel` имеют атрибуты `rendered`, для которых на странице установлено значение `false`, но могут иметь значение true в `CashierBean` при определённых обстоятельствах. Атрибут `for` тега `h:outputLabel` соответствует `id` поля ввода, к которому прикреплена метка. Тег `h:outputText`, вложенный в тег `h:outputLabel`, представляет компонент метки. Атрибут `value` в теге `h:outputText` указывает текст, который отображается рядом с полем ввода.

Вместо использования тега `h:outputText` для текста, отображаемого в качестве метки, вы можете просто использовать атрибут `value` тега `h:outputLabel`. Следующий фрагмент кода показывает, как будет выглядеть предыдущий код, если он будет использовать атрибут `value` тега `h:outputLabel` для указания текста метки:

```xml
<h:selectBooleanCheckbox id="fanClub"
                         rendered="false"
                         binding="#{cashierBean.specialOffer}" />
<h:outputLabel for="fanClub"
               rendered="false"
               binding="#{cashierBean.specialOfferText}"
               value="#{bundle.DukeFanClub}" />
</h:outputLabel>
...
```

Отображение ссылки тегом h:outputLink

Тег `h:outputLink` используется для отображения ссылки, которая при клике загружает другую страницу, но не генерирует событие действия. Вам следует использовать этот тег вместо тега `h:commandLink`, если вы хотите, чтобы всегда открывался URL, указанный атрибутом `value` тега `h:outputLink` и не хотите, чтобы выполнялась какая-либо обработка, когда пользователь кликает ссылку. Вот пример:

```xml
<h:outputLink value="javadocs">
    Documentation for this demo
</h:outputLink>
```

Текст в теле тега `h:outputLink` задаёт текст ссылки, по которой пользователь может перейти на следующую страницу.

Отображение форматированного сообщения тегом h:outputFormat

Тег `h:outputFormat` позволяет отображать составные сообщения в виде шаблона `MessageFormat`, как описано в документации API для `java.text.MessageFormat`. Вот пример тега `h:outputFormat`:

```xml
<h:outputFormat value="Hello, {0}!">
    <f:param value="#{hello.name}"/>
</h:outputFormat>
```

Атрибут `value` указывает шаблон `MessageFormat`. Тег `f:param` определяет параметры замещения для сообщения. Значение параметра заменяет `{0}` в предложении. Если значение `"#{hello.name}"` равно "Bill", то сообщение, отображаемое на странице, выглядит следующим образом:

```
Hello, Bill!
```

Тег `h:outputFormat` может включать более одного тега `f:param` для тех сообщений, которые имеют более одного параметра, который необходимо объединить в сообщение. Если у вас есть более одного параметра для одного сообщения, убедитесь, что вы поместили теги `f:param` в правильном порядке, чтобы данные вставлялись в правильное место в сообщении. Вот предыдущий пример, модифицированный дополнительным параметром:

XML

```xml
<h:outputFormat value="Hello, {0}! You are visitor number {1} to the page.">
    <f:param value="#{hello.name}" />
    <f:param value="#{bean.numVisitor}"/>
</h:outputFormat>
```

Значение `{1}` заменяется вторым параметром. Параметр является выражением EL `bean.numVisitor`, в котором свойство `numVisitor` Managed-бина `bean` отслеживает посетителей страницы. Это пример атрибута тега, поддерживающего выражения значения и принимающего выражения EL. Сообщение, отображаемое на странице, теперь выглядит следующим образом:

```
Hello, Bill! You are visitor number 10 to the page.
```

Использование тегов командных компонентов для выполнения действий и навигации

В приложениях Jakarta Faces теги компонента кнопки и ссылки используются для выполнения таких действий, как отправка формы и переход на другую страницу. Эти теги называются тегами командных компонентов, потому что они выполняют действие при активации.

Тег `h:commandButton` отображается как кнопка. Тег `h:commandLink` — как ссылка.

В дополнение к атрибутам тегов, перечисленным в Общих атрибутах тегов компонентов, теги `h:commandButton` и `h:commandLink` могут использовать следующие атрибуты.

- `action`, который является либо значением `String`, либо выражением метода, указывающим на метод бина, который возвращает значение `String`. В любом случае значение `String` используется для определения того, к какой странице обращаться, когда активирован тег командного компонента.
- `actionListener`, который является выражением метода, указывающим на метод бина, обрабатывающего событие действия, инициируемое тегом командного компонента.

Смотрите Ссылка на метод, который выполняет навигацию для получения дополнительной информации об использовании атрибута `action`. Смотрите Ссылка на метод-обработчик действия для получения подробной информации об использовании атрибута `actionListener`.

Отображение кнопки тегом h:commandButton

Если вы используете тег компонента `h:commandButton`, данные с текущей страницы обрабатываются, когда пользователь кликает кнопку, после чего открывается следующая страница. Вот пример тега `h:commandButton`:

```xml
<h:commandButton value="Submit"
                 action="#{cashierBean.submit}"/>
```

Клик на кнопку вызовет метод `submit` бина `CashierBean`, поскольку атрибут `action` ссылается на этот метод. Метод `submit` выполняет обработку и возвращает результат.

Атрибут `value` примера тега `h:commandButton` ссылается на метку кнопки. Для получения информации о том, как использовать атрибут `action`, смотрите Ссылка на метод, который выполняет навигацию.

Отображение ссылки тегом h:commandLink

Тег `h:commandLink` представляет HTML-ссылку и отображается как HTML-элемент `<a>`.

Тег `h:commandLink` должен включать вложенный тег `h:outputText`, представляющий текст, по которому пользователь кликает для генерации события. Вот пример:

```xml
<h:commandLink id="Duke" action="bookstore">
    <f:actionListener
        type="ee.jakarta.tutorial.dukesbookstore.listeners.LinkBookChangeListener" />
    <h:outputText value="#{bundle.Book201}"/>
</h:commandLink>
```

Этот тег будет отображать HTML, который выглядит примерно так:

```html
<a id="_idt16:Duke" href="#"
     onclick="mojarra.jsfcljs(document.getElementById('j_idt16'),
     {'j_idt16:Duke':'j_idt16:Duke'},'');
     return false;">My Early Years: Growing Up on Star7, by Duke</a>
```

Тег `h:commandLink` отобразит вызов функции на языке JavaScript. Если вы используете этот тег, убедитесь, что в вашем браузере включена поддержка JavaScript.

Добавление графики и изображений тегом h:graphicImage

В приложении Jakarta Faces используйте тег `h:graphicImage`, чтобы отобразить изображение на странице:

```xml
<h:graphicImage id="mapImage" url="/resources/images/book_all.jpg"/>
```

В этом примере атрибут `url` указывает путь к изображению. URL тега начинается с косой черты (`/`), которая добавляет относительный контекстный путь веб-приложения в начало пути до изображения.

Кроме того, вы можете использовать средство, описанное в Веб-ресурсах, чтобы указать местоположение изображения. Вот два примера:

```xml
<h:graphicImage id="mapImage"
                name="book_all.jpg"
                library="images"
                alt="#{bundle.ChooseBook}"
                usemap="#bookMap" />

<h:graphicImage value="#{resource['images:wave.med.gif']}"/>
```

Вы можете использовать аналогичный синтаксис для ссылки на изображение в таблице стилей. Следующий синтаксис в таблице стилей указывает, что изображение находится в `resources/img/top-background.jpg`:

```css
header {
    position: relative;
    height: 150px;
    background: #fff url(#{resource['img:top-background.jpg']}) repeat-x;
    ...
}
```

Управление расположением компонентов с помощью тегов h:panelGrid и h:panelGroup

В приложении Jakarta Faces панели используются в качестве контейнера для других компонентов. Панели отображается в виде HTML-таблицы. Таблица 10-6 перечисляет теги, используемые для создания панелей.

Таблица 10-6 Метки компонентов панели

Тег	Атрибуты	Функция
`h:panelGrid`	`columns`, `columnClasses`, `footerClass`, `headerClass`, `panelClass`, `rowClasses`, `role`	Отображает таблицу
`h:panelGroup`	`layout`	Группирует набор компонентов под одним родителем

Тег `h:panelGrid` используется для представления всей таблицы. Тег `h:panelGroup` используется для представления строк в таблице. Другие теги используются для представления отдельных ячеек в строках.

Атрибут `columns` определяет, как группировать данные в таблице, и поэтому необходим, если вы хотите, чтобы ваша таблица имела более одного столбца. Тег `h:panelGrid` также имеет набор необязательных атрибутов, которые определяют CSS-классы: `columnClasses`, `footerClass`, `headerClass`, `panelClass` и `rowClasses`. Атрибут `role` может иметь значение `"presentation"`, чтобы указать, что целью таблицы является форматирование отображения, а не отображение данных.

Если указано значение атрибута `headerClass`, тег `h:panelGrid` должен иметь заголовок первым дочерним элементом. Точно так же, если указано значение атрибута `footerClass`, тег `h:panelGrid` должен иметь нижний колонтитул последним дочерним элементом.

Вот пример:

XML

```
<h:panelGrid columns="2"
             headerClass="list-header"
             styleClass="list-background"
             rowClasses="list-row-even, list-row-odd"
             summary="#{bundle.CustomerInfo}"
             title="#{bundle.Checkout}"
             role="presentation">
    <f:facet name="header">
        <h:outputText value="#{bundle.Checkout}"/>
    </f:facet>

    <h:outputLabel for="name" value="#{bundle.Name}" />
    <h:inputText id="name" size="30"
                 value="#{cashierBean.name}"
                 required="true"
                 requiredMessage="#{bundle.ReqCustomerName}">
         <f:valueChangeListener
             type="ee.jakarta.tutorial.dukesbookstore.listeners.NameChanged" />
    </h:inputText>
    <h:message styleClass="error-message" for="name"/>

    <h:outputLabel for="ccno" value="#{bundle.CCNumber}"/>
    <h:inputText id="ccno"
                 size="19"
                 converterMessage="#{bundle.CreditMessage}"
                 required="true"
                 requiredMessage="#{bundle.ReqCreditCard}">
    <f:converter converterId="ccno"/>
    <f:validateRegex
        pattern="\d{16}|\d{4} \d{4} \d{4} \d{4}|\d{4}-\d{4}-\d{4}-\d{4}" />
    </h:inputText>
    <h:message styleClass="error-message"  for="ccno"/>
    ...
</h:panelGrid>
```

Предыдущий тег `h:panelGrid` отображается в виде таблицы, содержащей компоненты, в которые клиент вводит информацию. Этот тег `h:panelGrid` использует классы таблицы стилей для форматирования таблицы. Следующий код показывает определение `list-header`:

CSS

```
.list-header {
    background-color: #ffffff;
    color: #000000;
    text-align: center;
}
```

Поскольку тег `h:panelGrid` определяет `headerClass`, тег `h:panelGrid` должен содержать заголовок. В примере тега `h:panelGrid` используется тег `f:facet` для заголовка. У фасетов может быть только один дочерний элемент, поэтому тег `h:panelGroup` необходим, если вы хотите сгруппировать более одного компонента в `f:facet`. В примере тега `h:panelGrid` есть только одна ячейка данных, поэтому тег `h:panelGroup` не требуется. (Для получения дополнительной информации о фасетах см. Использование связанных с данными компонентов таблицы.

Тег `h:panelGroup` имеет атрибут `layout` в дополнение к перечисленным в Общих атрибутах тегов компонентов. Если атрибут `layout` имеет значение `block`, отображается HTML-элемент `div`. В противном случае отображается HTML-элемент `span`. Если указываются стили для тега `h:panelGroup`, следует установить атрибуту `layout` значение `block`, чтобы стили могли быть применены к компонентам внутри тега `h:panelGroup`. Вы должны сделать это, потому что стили, устанавливающие ширину и высоту, не применяются к внутренним элементам HTML-элемента `span`.

Тег `h:panelGroup` также можно использовать для инкапсуляции вложенного дерева компонентов, чтобы дерево компонентов отображалось как одиночный компонент для родительского компонента.

Данные, представленные вложенными тегами, группируются в строки в соответствии со значением атрибута `columns` тега `h:panelGrid`. Атрибут `columns` в примере установлен на `2`, и поэтому таблица будет иметь два столбца. Столбец, в котором отображается каждый компонент, определяется порядком, в котором компонент указан на странице по модулю 2. Таким образом, если компонент является пятым в списке компонентов, этот компонент будет находиться в столбце 5 по модулю 2, т. е. в столбце 1.

Компоненты выбора одного из значений

Другим часто используемым компонентом является тот, который позволяет пользователю выбрать одно значение, будь то единственное доступное значение или один из набора вариантов. Наиболее распространённые теги для этого вида компонентов следующие:

- Тег `h:selectBooleanCheckbox`, отображаемый в виде флажка, который представляет логическое состояние
- Тег `h:selectOneRadio`, отображаемый в виде набора параметров
- Тег `h:selectOneMenu`, отображаемый в виде прокручиваемого списка
- Тег `h:selectOneListbox`, отображаемый в виде непрокручиваемого списка

На рисунке 10-3 показаны примеры этих компонентов.

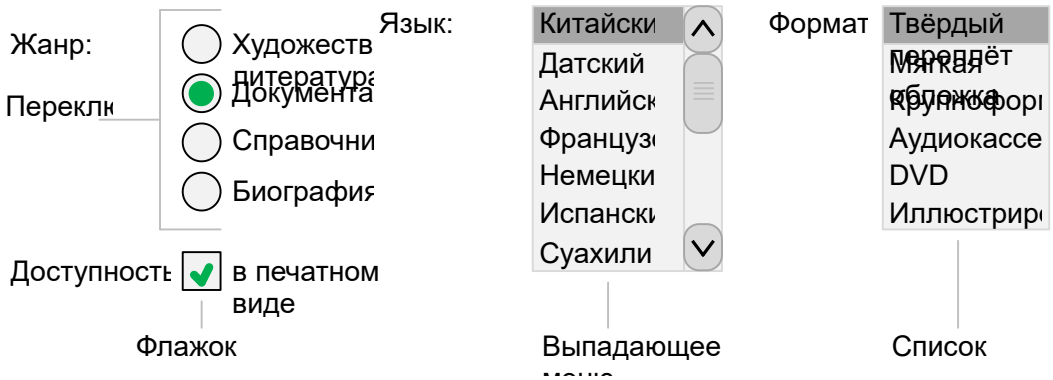

Рис. 10-3. Пример компонентов для выбора одного элемента

Отображение флажка с помощью тега h:selectBooleanCheckbox

Тег `h:selectBooleanCheckbox` является единственным тегом, который Jakarta Faces предоставляет для выбора логического состояния.

Вот пример, который показывает, как использовать тег `h:selectBooleanCheckbox`:

```xml
<h:selectBooleanCheckbox id="fanClub"
                         rendered="false"
                         binding="#{cashierBean.specialOffer}" />
<h:outputLabel for="fanClub"
               rendered="false"
               binding="#{cashierBean.specialOfferText}"
               value="#{bundle.DukeFanClub}" />
```

Теги `h:selectBooleanCheckbox` и `h:outputLabel` имеют атрибуты `rendered`, для которых на странице установлено значение `false`, но могут иметь значение true в `CashierBean` при определённых обстоятельствах. Тег `h:selectBooleanCheckbox` отображает флажок, позволяющий пользователям указать, хотят ли они присоединиться к фан-клубу Дюка. Тег `h:outputLabel` отображает метку для флажка. Текст метки представлен атрибутом `value`.

Отображение меню тегом h:selectOneMenu

Компонент, который позволяет пользователю выбрать одно значение из набора, может быть представлен как блок или набор параметров. В этом разделе описывается тег `h:selectOneMenu`. Теги `h:selectOneRadio` и `h:selectOneListbox` используются аналогичным образом. Тег `h:selectOneListbox` аналогичен тегу `h:selectOneMenu` с тем исключением, что `h:selectOneListbox` определяет атрибут `size`, который задаёт, сколько элементов отображается одновременно.

Тег `h:selectOneMenu` представляет компонент, содержащий список элементов, из которого пользователь может выбрать один элемент. Этот компонент меню иногда называют раскрывающимся списком или полем со списком. В следующем фрагменте кода показано, как используется тег `h:selectOneMenu`, чтобы позволить пользователю выбрать метод доставки:

```xml
<h:selectOneMenu id="shippingOption" required="true" value="#{cashierBean.shippingOption}">
    <f:selectItem itemValue="2" itemLabel="#{bundle.QuickShip}"/>
    <f:selectItem itemValue="5" itemLabel="#{bundle.NormalShip}"/>
    <f:selectItem itemValue="7" itemLabel="#{bundle.SaverShip}"/>
 </h:selectOneMenu>
```

Атрибут `value` тега `h:selectOneMenu` сопоставляется со свойством, которое содержит значение выбранного элемента. В этом случае значение устанавливается вспомогательным бином. Вы не обязаны указывать значение для выбранного в данный момент элемента. Если вы не предоставите значение, браузер определит, какое из них выбрано.

Как и тег `h:selectOneRadio`, тег `h:selectOneMenu` должен содержать тег `f:selectItems` или набор `f:selectItem` теги для представления элементов в списке. Опишем их с спользование тегов f:selectItem и f:selectItems.

Компонент выбора нескольких значений

В некоторых случаях нужно разрешить пользователям выбирать несколько значений, а не только одно значение из списка вариантов. Вы можете сделать это, используя один из следующих тегов компонента:

- Тег `h:selectManyCheckbox`, отображаемый в виде набора флажков
- Тег `h:selectManyMenu` отображается как меню
- Тег `h:selectManyListbox` отображается в виде блока

На рисунке 10-4 показаны примеры этих компонентов.

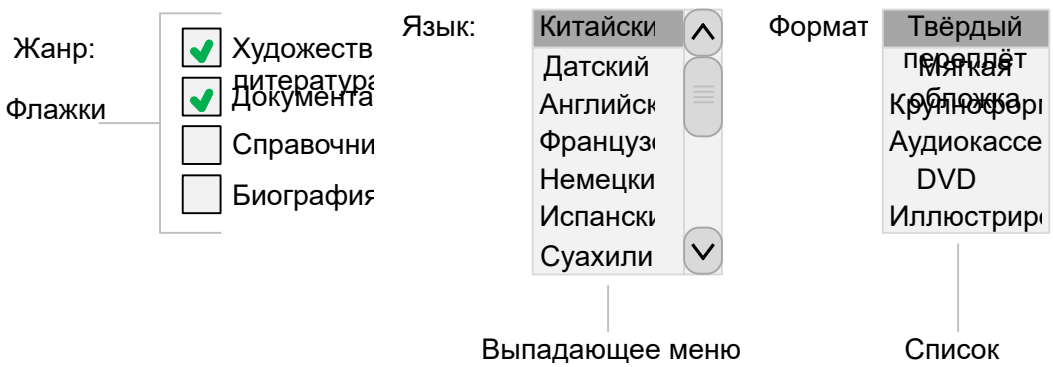

Рис. 10-4. Пример компонентов для выбора нескольких значений

Эти теги позволяют пользователю выбирать любое количество значений из набора значений. В этом разделе описывается тег `h:selectManyCheckbox`. Теги `h:selectManyListbox` и `h:selectManyMenu` используются аналогичным образом.

В отличие от меню, список отображает подмножество элементов в блоке. Меню отображает только один элемент за раз, когда пользователь не выбирает меню. Атрибут `size` тега `h:selectManyListbox` определяет количество элементов, отображаемых одновременно. Блок содержит полосу прокрутки для прокрутки всех оставшихся элементов в списке.

Тег `h:selectManyCheckbox` отображает группу флажков, каждый из которых представляет одно значение, которое можно выбрать:

```xml
<h:selectManyCheckbox id="newslettercheckbox"
                      layout="pageDirection"
                      value="#{cashierBean.newsletters}">
    <f:selectItems value="#{cashierBean.newsletterItems}"/>
</h:selectManyCheckbox>
```

Атрибут `value` тега `h:selectManyCheckbox` сопоставляется со свойством `newsletters` Managed-бина EJB `CashierBean`. Это свойство содержит значения выбранных в данный момент элементов из набора флажков. Вы не обязаны предоставлять значение для выбранных в данный момент элементов. Если вы не предоставите значение, первый элемент в списке будет выбран по умолчанию. В Managed-бине `CashierBean` это значение равно 0, поэтому по умолчанию элементы не выбираются.

Атрибут `layout` указывает, как устроен набор флажков на странице. Поскольку layout установлен в `pageDirection`, флажки расположены вертикально. По умолчанию используется значение `lineDirection`, которое выравнивает флажки по горизонтали.

Тег `h:selectManyCheckbox` также должен содержать тег или набор тегов, представляющих набор флажков. Для представления набора элементов вы используете тег `f:selectItems`. Чтобы представлять каждый элемент отдельно, вы используете тег `f:selectItem`. Следующий раздел объясняет эти теги более подробно.

Использование тегов f:selectItem и f:selectItems

Теги `f:selectItem` и `f:selectItems` представляют компоненты, которые могут быть вложены в компонент, позволяющий выбрать один или несколько элементов. Тег `f:selectItem` содержит значение, метку и описание одного элемента. Тег `f:selectItems` содержит значения, метки и описания всего списка элементов.

Вы можете использовать либо набор тегов `f:selectItem`, либо один тег `f:selectItems` внутри тега компонента.

Преимущества использования тега `f:selectItems` заключаются в следующем.

- Элементы могут быть представлены с использованием различных структур данных, включая `Array`, `Map` и `Collection`. Значение тега `f:selectItems` может представлять даже Generic-коллекцию POJO.
- Различные списки могут быть объединены в один компонент, они также могут быть сгруппированы внутри компонента.
- Значения могут генерироваться динамически во время выполнения.

Преимущества использования `f:selectItem` заключаются в следующем.

- Элементы в списке могут быть определены со страницы.
- Во вспомогательном бине требуется меньше кода для свойств `f:selectItem`.

В оставшейся части этого раздела показано, как использовать теги `f:selectItems` и `f:selectItem`.

Использование тега f:selectItems

В следующем примере из Отображение компонентов для выбора нескольких значений показано, как использовать тег `h:selectManyCheckbox` :

```xml
<h:selectManyCheckbox id="newslettercheckbox"
                      layout="pageDirection"
                      value="#{cashierBean.newsletters}">
    <f:selectItems value="#{cashierBean.newsletterItems}"/>
</h:selectManyCheckbox>
```

Атрибут `value` тега `f:selectItems` связан со свойством Managed-бина `cashierBean.newsletterItems` . Отдельные объекты `SelectItem` создаются программно в Managed-бине.

Смотрите Свойства UISelectItems для получения информации о том, как написать свойство Managed-бина для одного из этих тегов.

Использование тега f:selectItem

Тег `f:selectItem` представляет отдельный элемент в списке элементов. Вот пример из Отображения меню, использующего тег h:selectOneMenu ещё раз:

```xml
<h:selectOneMenu id="shippingOption"
                 required="true"
                 value="#{cashierBean.shippingOption}">
    <f:selectItem itemValue="2"
                  itemLabel="#{bundle.QuickShip}"/>
    <f:selectItem itemValue="5"
                  itemLabel="#{bundle.NormalShip}"/>
    <f:selectItem itemValue="7"
                  itemLabel="#{bundle.SaverShip}"/>
</h:selectOneMenu>
```

Атрибут `itemValue` представляет значение для тега `f:selectItem` . Атрибут `itemLabel` представляет `String` , которая появляется в компоненте списка на странице.

Атрибуты `itemValue` и `itemLabel` поддерживают привязку значений, то есть могут использовать выражения привязки значений для ссылки на значения во внешних объектах. Эти атрибуты также могут определять литеральные значения, как показано в примере тега `h:selectOneMenu` .

Отображение результатов компонентов выбора

При отображении компонентов , которые позволяют пользователю выбирать значения, вы также можете захотеть отобразить результат выбора.

Например, вы можете поблагодарить пользователя, который установил флажок, чтобы присоединиться к фан-клубу Дюка, как описано в Отображение флажка с использованием тега h:selectBooleanCheckbox. Поскольку флажок связан со свойством `specialOffer` в `CashierBean` значением `UISelectBoolean` , вы можете вызвать метод `isSelected` свойства, чтобы определить, следует ли отображать сообщение:

```xml
<h:outputText value="#{bundle.DukeFanClubThanks}"
              rendered="#{cashierBean.specialOffer.isSelected()}"/>
```

Точно так же вы можете захотеть подтвердить, что пользователь подписался на новостную рассылку, используя тег `h:selectManyCheckbox` , как описано в Отображение компонентов для выбора нескольких значений. Для этого вы можете получить значение свойства `newsletters` , массива `String` , который содержит выбранные элементы:

XML

```
<h:outputText value="#{bundle.NewsletterThanks}"
              rendered="#{!empty cashierBean.newsletters}"/>
<ul>
    <ui:repeat value="#{cashierBean.newsletters}" var="nli">
        <li><h:outputText value="#{nli}" /></li>
    </ui:repeat>
</ul>
```

Вступительное благодарственное сообщение отображается только в том случае, если массив `newsletters` не пустой. Затем тег `ui:repeat` — простой способ вывести значения в цикле — отображает содержимое выбранных элементов в подробном списке. (Этот тег указан в таблице 8-2.)

Использование таблицы Data-Bound

Компоненты таблицы с привязкой к данным отображают реляционные данные в табличном формате. В приложении Jakarta Faces тег компонента `h:dataTable` поддерживает привязку к коллекции объектов данных и отображает данные в виде таблицы HTML. Тег `h:column` представляет столбец данных в таблице, перебирая каждую запись в источнике данных, которая отображается в виде строки. Вот пример:

XML

```xml
<h:dataTable id="items"
             captionClass="list-caption"
             columnClasses="list-column-center, list-column-left,
             list-column-right, list-column-center"
             footerClass="list-footer"
             headerClass="list-header"
             rowClasses="list-row-even, list-row-odd"
             styleClass="list-background"
             summary="#{bundle.ShoppingCart}"
             value="#{cart.items}"
             border="1"
             var="item">
    <h:column>
        <f:facet name="header">
            <h:outputText value="#{bundle.ItemQuantity}" />
        </f:facet>
        <h:inputText id="quantity"
                     size="4"
                     value="#{item.quantity}"
                     title="#{bundle.ItemQuantity}">
            <f:validateLongRange minimum="1"/>
            <f:valueChangeListener
                type="ee.jakarta.tutorial.dukesbookstore.listeners.QuantityChanged"/>
        </h:inputText>
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="#{bundle.ItemTitle}"/>
        </f:facet>
        <h:commandLink action="#{showcart.details}">
            <h:outputText value="#{item.item.title}"/>
        </h:commandLink>
    </h:column>
    ...
    <f:facet name="footer">
        <h:panelGroup>
            <h:outputText value="#{bundle.Subtotal}"/>
            <h:outputText value="#{cart.total}" />
                <f:convertNumber currencySymbol="$" type="currency" />
            </h:outputText>
        </h:panelGroup>
    </f:facet>
    <f:facet name="caption">
        <h:outputText value="#{bundle.Caption}"/>
    </f:facet>
</h:dataTable>
```

В примере тег `h:dataTable` отображает книги в корзине с указанием количества каждой книги, цены и набора кнопок, которые пользователь может нажать, чтобы удалить книги из корзины.

Теги `h:column` представляют столбцы данных в таблице. Пока компонент выполняет итерацию по строкам данных, он обрабатывает компонент столбца, связанный с каждым тегом `h:column` для каждой строки в таблице.

Тег `h:dataTable` , показанный в предыдущем примере кода, перебирает список книг (`cart.items`) в корзине покупок и отображает их названия, авторов и цены. Каждый раз, когда тег `h:dataTable` просматривает список книг, он отображает одну ячейку в каждом столбце.

В тегах `h:dataTable` и `h:column` используются фасеты для представления частей таблицы, которые не повторяются или не обновляются. Эти части включают верхние и нижние колонтитулы и подписи.

В предыдущем примере теги `h:column` включают теги `f:facet` для представления верхних или нижних колонтитулов столбцов. Тег `h:column` позволяет управлять стилями верхних и нижних колонтитулов, поддерживая атрибуты `headerClass` и `footerClass`. Эти атрибуты принимают разделённые пробелами списки CSS-классов, которые будут применяться к ячейкам верхнего и нижнего колонтитула соответствующего столбца в отображаемой таблице.

У фасетов может быть только один дочерний элемент. Если требуется сгруппировать более одного компонента в `f:facet`, необходим тег `h:panelGroup`. Поскольку тег фасета, представляющий нижний колонтитул, включает в себя более одного тега, тег `h:panelGroup` необходим для группировки этих тегов. Наконец, этот же тег `h:dataTable` включает тег `f:facet` с его атрибутом `name`, установленным в `caption`, что представляет собой заголовок таблицы.

Эта таблица — классический вариант использования компонента данных, поскольку число книг может быть неизвестно разработчикам приложения и страницы при разработке приложения. Компонент данных может динамически регулировать количество строк в таблице для размещения данных.

Атрибут `value` тега `h:dataTable` ссылается на данные, которые будут включены в таблицу. Эти данные могут принимать следующие формы:

- Список бинов
- Массив бинов
- Одиночный бин
- Объект `jakarta.faces.model.DataModel`
- Объект `java.sql.ResultSet`
- Объект `jakarta.servlet.jsp.jstl.sql.Result`
- Объект `javax.sql.RowSet`

Все источники данных для компонентов данных имеют обёртку (wrapper) `DataModel`. Если вы явно не создадите обёртку (wrapper) `DataModel`, Jakarta Faces сделает это за вас с данными одного из допустимых типов. Смотрите Запись свойств бина для получения дополнительной информации о том, как писать свойства для использования компонентом данных.

Атрибут `var` указывает имя, которое используется компонентами в теге `h:dataTable` в качестве псевдонима данных, на которые ссылается атрибут `value` из `h:dataTable`.

В примере тега `h:dataTable` атрибут `value` указывает на список книг. Атрибут `var` указывает на одну книгу в этом списке. Поскольку тег `h:dataTable` выполняет итерацию по списку, каждая ссылка на `item` указывает на текущую книгу в списке.

Тег `h:dataTable` также может отображать только подмножество данных. Эта функция не показана в предыдущем примере. Чтобы отобразить подмножество данных, вы используете необязательные атрибуты `first` и `rows`.

Атрибут `first` указывает первую строку для отображения. Атрибут `lines` определяет количество строк, начиная с первой строки, которые будут отображаться. Например, если вы хотите отобразить записи со 2 по 10 базовых данных, то должны установить для `first` значение 2, а для `rows` — 9. Когда вы отображаете подмножество данных на своих страницах, можете подумать о добавлении ссылки или кнопки, которая приводит к отображению последующих строк при клике. По умолчанию и `first`, и `rows` установлены в ноль, что приводит к отображению всех строк данных.

Таблица 10-7 показывает необязательные атрибуты для тега `h:dataTable`.

Таблица 10-7. Необязательные атрибуты для тега h:dataTable

Атрибут	Задаёт стиль для
`captionClass`	Заголовок таблицы
`columnClasses`	Все столбцы
`footerClass`	Нижний колонтитул
`headerClass`	Заголовок
`rowClasses`	Строки
`styleClass`	Таблица в целом

Каждый из атрибутов в таблице 10-7 может указать более одного стиля. Если в `columnClasses` или `rowClasses` указано более одного стиля, они применяются к столбцам или строкам в том порядке, в котором перечислены в атрибуте. Например, если `columnClasses` указывает стили `list-column-center` и `list-column-right`, и если таблица имеет два столбца, первый столбец будет иметь стиль `list-column-center`, а второй столбец будет иметь стиль `list-column-right`.

Если атрибут style задаёт больше стилей, чем столбцов или строк, остальные стили будут назначены столбцам или строкам, начиная с первого столбца или строки. Аналогичным образом, если атрибут style определяет меньше стилей, чем столбцов или строк, оставшимся столбцам или строкам будут назначаться стили, начиная с первого стиля.

Отображение сообщений об ошибках тегами h:message и h:messages

Теги `h:message` и `h:messages` используются для отображения сообщений об ошибках в случае, если таковые были выданы конвертерами или валидаторами. Тег `h:message` отображает сообщения об ошибках, связанных с конкретным компонентом ввода, а тег `h:messages` отображает сообщения об ошибках для всей страницы.

Вот пример тега `h:message` из приложения `guessnumber-jsf`:

XML

```
<p>
    <h:inputText id="userNo"
                 title="Type a number from 0 to 10:"
                 value="#{userNumberBean.userNumber}">
        <f:validateLongRange minimum="#{userNumberBean.minimum}"
                             maximum="#{userNumberBean.maximum}"/>
   </h:inputText>
   <h:commandButton id="submit" value="Submit"
                    action="response"/>
</p>
<h:message showSummary="true" showDetail="false"
           style="color: #d20005;
           font-family: 'New Century Schoolbook', serif;
           font-style: oblique;
           text-decoration: overline"
           id="errors1"
           for="userNo"/>
```

Атрибут `for` ссылается на идентификатор компонента, который сгенерировал сообщение об ошибке. Сообщение об ошибке отображается в том же месте, что и тег `h:message` на странице. В этом случае сообщение об ошибке появится под кнопкой «Отправить».

Атрибут `style` позволяет указать стиль текста сообщения. В примере в этом разделе текст будет отрисован наклонным шрифтом с засечками, красного цвета, New Century Schoolbook, а над текстом появится линия. Теги message и messages поддерживают множество других атрибутов для определения стилей. Дополнительные сведения об этих атрибутах см. в Документация Jakarta Faces Facelets Tag Library (https://jakarta.ee/specifications/faces/3.0/vdldoc/).

Другим атрибутом, поддерживаемым тегом `h:messages`, является атрибут `layout`. Его значение по умолчанию — `list`, которое указывает, что сообщения отображаются в списке маркеров с использованием элементов HTML `ul` и `li`. Если для атрибута установлено значение `table`, сообщения будут отображаться в таблице с использованием элемента HTML `table`.

В предыдущем примере показан стандартный валидатор, который зарегистрирован в компоненте ввода. Тег отображает сообщение об ошибке, связанное с этим валидатором, когда тот сигнализирует об ошибке во введённом значении. В общем, при регистрации конвертера или валидатора в компоненте вы тем самым создаёте очередь сообщений об ошибках для этого компонента. Теги `h:message` и `h:messages` отображают соответствующие сообщения об ошибках, помещённые в эту очередь компонента, когда валидаторы или конвертеры, не могут сконвертировать или провалидировать значения компонента.

Стандартные сообщения об ошибках предоставляются со стандартными конвертерами и валидаторами. Разработчик приложения может переопределить эти стандартные сообщения и предоставить сообщения об ошибках для кастомных конвертеров и валидаторов путём регистрации кастомных сообщений об ошибках в приложении.

Создание URL для закладок с помощью тегов h:button и h:link

Возможность создавать закладки для URL относится к возможности создавать ссылки на основе заданного результата навигации и параметров компонента.

Большинство браузеров по умолчанию отправляют запросы GET для получения URL и запросы POST для обработки данных. Запросы GET могут иметь параметры запроса и могут кэшироваться, что не рекомендуется для запросов POST, которые отправляют данные на сервер для обработки. Другие теги Jakarta Faces, генерирующие ссылки, используют либо простые запросы GET, как в случае `h:outputLink`, либо запросы POST, как в случае `h:commandLink` или `h:commandButton`. GET-запросы с параметрами запроса обеспечивают более высокую степень детализации для URL. Эти URL создаются с одним или несколькими параметрами `name=value`, добавляемыми к простому URL после символа `?` и разделяемыми `&;` либо `&`.

Чтобы создать URL для закладок, используйте тег `h:link` или `h:button`. Оба эти тега могут генерировать ссылку с помощью атрибута `outcome`. Например:

```xml
<h:link outcome="somepage" value="Message" />
```

Тег `h:link` сгенерирует URL-ссылку, указывающую на файл `somepage.xhtml`. Из этого тега генерируется следующий HTML-фрагмент (предполагаем, что имя приложения `simplebookmark`):

```html
<a href="/simplebookmark/somepage.xhtml">Message</a>
```

Это простой GET-запрос, который не может передавать данные со страницы на страницу. Чтобы создавать более сложные GET-запросы и полнее использовать возможности тега `h:link`, обратите внимание на параметры представления.

Использование параметров представления для настройки URL для закладок

Чтобы передать параметр с одной страницы на другую, используйте атрибут `includeViewParams` в теге `h:link` и, кроме того, используйте тег `f:param` для указания передаваемых имени и значения. Здесь тег `h:link` определяет страницу результатов как `personal.xhtml` и предоставляет параметр с именем `Result`, значение которого является свойством Managed-бина:

XML

```
<h:body>
    <h:form>
        <h:graphicImage url="#{resource['images:duke.waving.gif']}"
                        alt="Duke waving his hand"/>
        <h2>Hello, #{hello.name}!</h2>
        <p>I've made your
            <h:link outcome="personal" value="personal greeting page!"
                    includeViewParams="true">
                <f:param name="Result" value="#{hello.name}"/>
            </h:link>
        </p>
        <h:commandButton id="back" value="Back" action="index" />
    </h:form>
</h:body>
```

Если для компонента установлен атрибут `includeViewParams`, параметры представления добавляются в гиперссылку. Если значение `hello.name` равно `Timmy`, то результирующий URL будет выглядеть так:

```
http://localhost:8080/bookmarks/personal.xhtml?Result=Timmy
```

На странице результатов укажите основные теги `f:metadata` и `f:viewparam` в качестве источника параметров для настройки URL. Параметры представления объявляются как часть `f:metadata` для страницы, как показано в следующем примере:

XML

```
<f:metadata>
    <f:viewParam name="Result" value="#{hello.name}"/>
</f:metadata>
```

Это позволяет указать значение свойства бина на странице:

XML

```
<h:outputText value="Howdy, #{hello.name}!" />
```

В качестве параметра просмотра имя также отображается в URL страницы. При редактировании URL изменяется и вывод на странице.

Поскольку URL может быть результатом различных значений параметров, порядок создания URL был предопределён. Различные значения параметров считываются в следующем порядке:

1. Компонент
2. Параметры навигации
3. Параметры представления

Приложение bookmarks

Приложение `bookmarks` изменяет приложение `hello1`, описанное в Веб-модуле с использованием Jakarta Faces: пример hello1, чтобы использовать URL для закладки с применением параметров представления.

Как и `hello1`, приложение bookmarks включает в себя Managed-бин `Hello.java`, страницы `index.xhtml` и `response.xhtml`. Кроме того, оно включает в себя страницу `personal.xhtml`, на которую указывает URL для закладок и передаются параметры просмотра со страницы `response.xhtml`, как описано в Использование параметров представления для настройки URL для закладок.

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска `bookmarks`. Исходный код для этого примера находится в каталоге *`tut-install`*`/examples/web/jsf/bookmarks/`.

Сборка, упаковка и развёртывание bookmarks с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/web/jsf
   ```

4. Выберите каталог `bookmarks`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `bookmarks` и выберите **Сборка**.

   Эта команда собирает приложение и развёртывает его в GlassFish Server.

Сборка, упаковка и развёртывание bookmarks с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/web/jsf/bookmarks/
   ```

3. Введите следующую команду:

   ```
   mvn install
   ```
   SHELL

   Эта команда собирает и упаковывает приложение в WAR-файл `bookmarks.war`, который находится в каталоге `target`. Затем он развёртывается в GlassFish Server.

Запуск bookmarks

1. Введите следующий URL в браузере:

   ```
   http://localhost:8080/bookmarks
   ```

2. В текстовом поле введите имя и нажмите «Отправить».
3. На странице ответа наведите указатель мыши на ссылку «personal greeting page», чтобы просмотреть URL с параметром представления, а затем кликните ссылку.

   Откроется страница `personal.xhtml` с приветствием для введённого вами имени.

4. В поле URL измените значение параметра Result и нажмите Return.

Имя в приветствии изменится на введённое вами.

Перемещение ресурсов с использованием тегов h:outputScript и h:outputStylesheet

Перемещение ресурса относится к способности приложения Jakarta Faces указывать местоположение, где ресурс может быть отображён. Перемещение ресурса может быть задано с помощью следующих тегов HTML:

- `h:outputScript`
- `h:outputStylesheet`

Эти теги имеют атрибуты `name` и `target`, которые можно использовать для указания местоположения отрисовки. Полный список атрибутов для этих тегов см. в Документация Jakarta Faces Facelets Tag Library (https://jakarta.ee/specifications/faces/3.0/vdldoc/).

Для тега `h:outputScript` атрибуты `name` и `target` определяют, где могут отображаться выходные данные ресурса. Вот пример:

```xml
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head id="head">
        <title>Resource Relocation</title>
    </h:head>
    <h:body id="body">
        <h:form id="form">
            <h:outputScript name="hello.js"/>
            <h:outputStylesheet name="hello.css"/>
        </h:form>
    </h:body>
</html>
```

Поскольку атрибут `target` не определён в тегах, таблица стилей `hello.css` отображается в элементе head страницы, а `hello.js` отображается в теле страницы.

Вот HTML, сгенерированный предыдущим кодом:

```xml
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>Resource Relocation</title>
        <link type="text/css" rel="stylesheet"
              href="/context-root/jakarta.faces.resource/hello.css"/>
    </head>
    <body>
        <form id="form" name="form" method="post"
              action="..." enctype="...">
            <script type="text/javascript"
                    src="/context-root/jakarta.faces.resource/hello.js">
            </script>
        </form>
    </body>
</html>
```

Если вы установите атрибут `target` для тега `h:outputScript`, входящий запрос GET предоставит параметр location. Вот пример:

```XML
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head id="head">
        <title>Resource Relocation</title>
    </h:head>
    <h:body id="body">
        <h:form id="form">
            <h:outputScript name="hello.js" target="#{param.location}"/>
            <h:outputStylesheet name="hello.css"/>
        </h:form>
    </h:body>
</html>
```

В этом случае, если во входящем запросе не указан параметр местоположения, будут применяться местоположения по умолчанию: таблица стилей отображается в заголовке, а сценарий встроен в форму. Однако, если во входящем запросе в качестве значения параметра location указан head, таблица стилей и сценарий будут отображаться в элементе `head`.

HTML, сгенерированный предыдущим кодом, выглядит следующим образом:

```HTML
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>Resource Relocation</title>
        <link type="text/css" rel="stylesheet"
              href="/context-root/jakarta.faces.resource/hello.css"/>
        <script type="text/javascript"
                src="/context-root/jakarta.faces.resource/hello.js">
          </script>
    </head>
    <body>
        <form id="form" name="form" method="post"
              action="..." enctype="...">
        </form>
    </body>
</html>
```

Аналогично, если во входящем запросе предоставлено body в качестве значения параметра location, скрипт будет отображаться в элементе body.

В предыдущем разделе описан простой пример использования перемещения ресурсов. Эта функция может добавить компонентам и страницам ещё больше функциональности. Автору страницы не обязательно знать местоположение ресурса или его размещение.

Используя аннотацию `@ResourceDependency` для компонентов, авторы компонентов могут задавать ресурсы для компонента, такие как таблицы стилей и сценарии. Это избавляет авторов страниц от указания конкретного местоположения ресурса.

Использование основных тегов

Теги, включённые в базовую библиотеку тегов Jakarta Faces, используются для выполнения основных действий, которые не выполняются тегами HTML.

Таблица 10-8 перечисляет основные теги обработки событий.

Таблица 10-8 Основные теги обработки событий

Тег	Функция

Тег	Функция
`f:actionListener`	Добавляет слушатель действия в родительский компонент
`f:phaseListener`	Добавляет `PhaseListener` на страницу
`f:setPropertyActionListener`	Регистрирует слушателя специальных действий, единственная цель которого — передать значение в Managed-бин при отправке формы
`f:valueChangeListener`	Добавляет слушатель изменения значения в родительский компонент

Таблица 10-9 перечисляет основные теги преобразования данных.

Таблица 10-9 Основные теги конвертации данных

Тег	Функция
`f:converter`	Добавляет произвольный конвертер в родительский компонент
`f:convertDateTime`	Добавляет объект `DateTimeConverter` в родительский компонент
`f:convertNumber`	Добавляет объект `NumberConverter` в родительский компонент

Таблица 10-10 перечисляет основные теги фасетов.

Таблица 10-10 Основные теги фасетов

Тег	Функция
`f:facet`	Добавляет вложенный компонент, имеющий особое отношение к тегу включения
`f:metadata`	Регистрирует `facet` в родительском компоненте

Таблица 10-11 перечисляет основные теги, которые представляют элементы в списке.

Таблица 10-11 Основные теги, которые представляют элементы в списке

Тег	Функция
`f:selectItem`	Представляет один элемент в списке элементов
`f:selectItems`	Представляет набор элементов

Таблица 10-12 перечисляет основные теги валидаторов.

Таблица 10-12 Основные теги валидаторов

Тег	Функция
`f:validateDoubleRange`	Добавляет `DoubleRangeValidator` к компоненту
`f:validateLength`	Добавляет `LengthValidator` к компоненту
`f:validateLongRange`	Добавляет `LongRangeValidator` к компоненту

Тег	Функция
`f:validator`	Добавляет кастомный валидатор к компоненту
`f:validateRegEx`	Добавляет `RegExValidator` к компоненту
`f:validateBean`	Делегирует валидацию локального значения `BeanValidator` -у
`f:validateRequired`	Обеспечивает наличие значения в компоненте

Таблица 10-13 перечисляет основные теги, которые попадают в другие категории.

Таблица 10-13 Остальные основные теги

Категория тегов	Тег	Функция
Конфигурация атрибута	`f:attribute`	Добавляет настраиваемые атрибуты в родительский компонент
Локализация	`f:loadBundle`	Задаёт `ResourceBundle` , который отображается как `Map`
Подстановка параметров	`f:param`	Подставляет параметры в объект `MessageFormat` и добавляет пары имя-значение в строку запроса URL
Ajax	`f:ajax`	Связывает действие Ajax с одним компонентом или группой компонентов в зависимости от размещения тега
Событие	`f:event`	Позволяет установить `ComponentSystemEventListener` на компонент
Веб-сокеты	`f:websocket`	Позволяет передавать серверные сообщения всем сокетам, содержащим то же имя канала.

Эти теги используются совместно с тегами компонентов и описаны в других разделах данного учебника.

Таблица 10-14 перечисляет разделы, в которых разъясняется, как использовать некоторые из тегов.

Таблица 10-14. Где разъясняются теги

Теги	Где объясняется
Теги обработки событий	Регистрация слушателей в компонентах
Теги конвертации данных	Использование стандартных конвертеров
`f:facet`	Использование таблицы Data-Bound и Управление расположением компонентов с помощью тегов h:panelGrid и h:panelGroup
`f:loadBundle`	Настройка bundle-ресурса
`f:metadata`	Использование параметров представления для настройки URL для закладок
`f:param`	Отображение форматированного сообщения тегом h:outputFormat

Теги	Где объясняется
`f:selectItem` and `f:selectItems`	Использование тегов f:selectItem и f:selectItems
Теги валидаторов	Использование стандартных валидаторов
`f:ajax`	Глава 13 *Использование Ajax с Jakarta Faces*
`f:websocket`	Глава 17 *Использование веб-сокетов в Jakarta Faces*

Глава 11. Использование конвертеров, слушателей и валидаторов

В предыдущей главе описывались компоненты и объяснялось, как добавить их на веб-страницу. В этой главе содержится информация о добавлении дополнительной функциональности компонентам с помощью конвертеров, слушателей и валидаторов.

- Конвертеры используются для конвертации данных, полученных от компонентов ввода. Конвертеры позволяют приложению перенести строго типизированные функции языка Java в ориентированный на строковые данные мир сервлетов.
- Слушатели используются для выполнения определённых разработчиком действий как реакция на события, происходящие на странице.
- Валидаторы используются для валидации данных, полученных от компонентов ввода. Валидаторы позволяют приложению наложить ограничения на введённые пользователем в форму данные, чтобы обеспечить выполнение необходимых проверок перед обработкой этих данных.

Использование стандартных конвертеров

Jakarta Faces предоставляет набор реализаций `Converter` , которые можно использовать для конвертации данных компонента. Цель конвертации — взять строковые данные, поступающие из Servlet API, и преобразовать их в строго типизированные объекты Java, которые можно использовать в компонентах бизнес-логики. Для получения дополнительной информации о концептуальных деталях модели этих преобразований см. Модель конвертации.

Стандарт `Converter` находятся в пакете `jakarta.faces.convert` . Обычно конвертеры присваиваются неявно в зависимости от типа выражения EL, на которое указывает значение компонента. Однако к этим конвертерам также можно получить доступ по идентификатору конвертера. Таблица 11-1 показывает классы конвертеров и связанные с ними идентификаторы конвертеров.

Таблица 11-1 Классы конвертеров и их идентификаторы

Класс в пакете jakarta.faces.convert	**Идентификатор конвертера**
`BigDecimalConverter`	`jakarta.faces.BigDecimal`
`BigIntegerConverter`	`jakarta.faces.BigInteger`
`BooleanConverter`	`jakarta.faces.Boolean`
`ByteConverter`	`jakarta.faces.Byte`
`CharacterConverter`	`jakarta.faces.Character`
`DateTimeConverter`	`jakarta.faces.DateTime`
`DoubleConverter`	`jakarta.faces.Double`
`EnumConverter`	`jakarta.faces.Enum`
`FloatConverter`	`jakarta.faces.Float`
`IntegerConverter`	`jakarta.faces.Integer`

Класс в пакете jakarta.faces.convert	Идентификатор конвертера
`LongConverter`	`jakarta.faces.Long`
`NumberConverter`	`jakarta.faces.Number`
`ShortConverter`	`jakarta.faces.Short`

Стандартное сообщение об ошибке связано с каждым из этих конвертеров. Если вы зарегистрировали какой-либо из этих конвертеров в компоненте на своей странице, и конвертер не может конвертировать значение компонента, на странице появится сообщение об ошибке конвертера. Например, следующее сообщение об ошибке появляется, если `BigIntegerConverter` не может преобразовать значение:

```
{0} должен быть числом, состоящим из одной или нескольких цифр
```
JAVA

В этом случае параметр замещения `{0}` будет заменён именем компонента ввода, на котором зарегистрирован конвертер.

Два стандартных конвертера `DateTimeConverter` и `NumberConverter` имеют собственные теги, которые позволяют настраивать формат данных компонента с помощью атрибутов тега. Для получения дополнительной информации об использовании `DateTimeConverter` см. Использование DateTimeConverter. Для получения дополнительной информации об использовании `NumberConverter` см. Использование NumberConverter. В следующем разделе объясняется, как преобразовать значение компонента, в том числе как зарегистрировать другие стандартные конвертеры в компоненте.

Конвертация значения компонента

Чтобы использовать конкретный конвертер для преобразования значения компонента, необходимо зарегистрировать конвертер в компоненте. Вы можете зарегистрировать любой из стандартных конвертеров одним из следующих способов.

- Вложите один из стандартных тегов конвертера в тег компонента. Это теги `f:convertDateTime` и `f:convertNumber` , которые описаны в Использование NumberConverter.
- Свяжите значение компонента со свойством Managed-бина того же типа, что и конвертер. Эта техника наиболее распространена.
- Обратитесь к конвертеру из атрибута `converter` тега компонента, указав идентификатор класса конвертера.
- Вложите тег `f:converter` внутрь тега компонента и используйте либо атрибут `converterId` тега `f:converter` , либо его атрибут `binding` для ссылки на конвертер.

В качестве примера второго метода, когда вы хотите преобразовать данные компонента в `Integer` , вы можете просто связать значение компонента со свойством Managed-бина. Вот пример:

```
Integer age = 0;
public Integer getAge(){ return age;}
public void setAge(Integer age) {this.age = age;}
```
JAVA

Данные из тега `h:inputText` в этом примере будут преобразованы в значение `java.lang.Integer`. Тип `Integer` поддерживается типом конвертера `NumberConverter`. Если не требуется указывать какие-либо инструкции форматирования с помощью атрибутов тега `f:convertNumber`, и если одного из стандартных конвертеров будет достаточно, можно просто ссылаться на этот конвертер с помощью атрибута `converter`.

Вы также можете вложить тег `f:converter` в тег компонента и использовать атрибут `converterId` тега конвертера или его атрибут `binding` для ссылки на конвертер.

Атрибут `converterId` должен ссылаться на идентификатор конвертера. Вот пример, который использует один из идентификаторов конвертера, перечисленных в таблице 11-1:

XML

```
<h:inputText value="#{loginBean.age}">
    <f:converter converterId="jakarta.faces.Integer" />
</h:inputText>
```

Вместо использования атрибута `converterId` тег `f:converter` может использовать атрибут `binding`. Атрибут `binding` должен разрешать свойство бина, которое принимает и возвращает соответствующий объект `Converter`.

Вы также можете создавать собственные конвертеры и регистрировать их в компонентах, используя тег `f:converter`. Подробнее см. Создание и использование кастомного конвертера.

Использование DateTimeConverter

Вы можете преобразовать данные компонента в `java.util.Date`, вложив тег `convertDateTime` в тег компонента. Тег `convertDateTime` имеет несколько атрибутов, которые позволяют указать формат и тип данных. Таблица 11-2 перечисляет атрибуты.

Вот простой пример тега `convertDateTime`:

XML

```
<h:outputText value="#{cashierBean.shipDate}">
    <f:convertDateTime type="date" dateStyle="full" />
</h:outputText>
```

При связывании `DateTimeConverter` с компонентом убедитесь, что свойство Managed-бина, с которым связан компонент, имеет тип `java.util.Date`. В предыдущем примере `cashierBean.shipDate` должен иметь тип `java.util.Date`.

Тег примера может отображать следующий вывод:

```
Saturday, September 21, 2013
```

Вы также можете отобразить ту же дату и время, используя следующий тег, в котором указан формат даты:

XML

```
<h:outputText value="#{cashierBean.shipDate}">
    <f:convertDateTime pattern="EEEEEEEE, MMM dd, yyyy" />
</h:outputText>
```

Если вы хотите отобразить пример даты на испанском языке, вы можете использовать атрибут `locale`:

```xml
<h:outputText value="#{cashierBean.shipDate}">
    <f:convertDateTime dateStyle="full"
                       locale="es"
                       timeStyle="long" type="both" />
</h:outputText>
```

Этот тег будет отображать следующий вывод:

```
jueves 24 de octubre de 2013 15:07:04 GMT
```

Обратитесь к уроку «Настройка форматов» учебника Java по ссылке https://docs.oracle.com/javase/tutorial/i18n/format/simpleDateFormat.html для получения дополнительной информации о том, как отформатировать вывод с помощью атрибута `pattern` тега `convertDateTime`.

Таблица 11-2 Атрибуты для тега f:convertDateTime

Атрибут	Тип	Описание
`binding`	`DateTimeConverter`	Используется для связывания конвертера со свойством Managed-бина.
`dateStyle`	`String`	Определяет формат, указанный в `java.text.DateFormat`, даты или части даты строки `date`. Применяется, только если `type` равен `date` или `both` и если `pattern` не определён. Допустимые значения: `default`, `short`, `medium`, `long` и `full`. Если значение не указано, используется `default`.
`for`	`String`	Используется с составными компонентами. Относится к одному из объектов в составном компоненте, внутрь которого этот тег вложен.
`locale`	`String` или `Locale`	`Locale`, чьи предопределённые стили для дат и времени используются во время форматирования или анализа. Если не указано, будет использоваться `Locale`, возвращаемый `FacesContext.getLocale`.
`pattern`	`String`	Пользовательский шаблон форматирования, который определяет, как строка даты/времени должна быть отформатирована и проанализирована. Если этот атрибут указан, атрибуты `dateStyle` и `timeStyle` игнорируются. Смотрите таблицу 11-3 для значений по умолчанию, когда `pattern` не указан.

Атрибут	Тип	Описание
timeStyle	String	Определяет формат, указанный в java.text.DateFormat, time или части времени строки date. Применяется, только если type имеет значение time и pattern не определён. Допустимые значения: default, short, medium, long и full. Если значение не указано, используется default.
timeZone	String или TimeZone	Часовой пояс, в котором можно интерпретировать любую информацию о времени в строке date.
type	String	Указывает, будет ли строковое значение содержать дату, время или оба. Допустимые значения: date, time, both, LocalDate, LocalTime, LocalDateTime, OffsetTime, OffsetDateTime или ZonedDateTime. Если значение не указано, используется date. Смотрите таблицу 11-3 для дополнительной информации.

Таблица 11-3 Тип атрибута и значения шаблона по умолчанию

Тип атрибута	Класс	По умолчанию, когда шаблон не указан
both	java.util.Date	DateFormat.getDateTimeInstance(dateStyle,timeStyle)
date	java.util.Date	DateFormat.getDateTimeInstance(dateStyle)
time	java.util.Date	DateFormat.getDateTimeInstance(timeStyle)
localDate	java.time.LocalDate	DateTimeFormatter.ofLocalizedDate(dateStyle)
localTime	java.time.LocalTime	DateTimeFormatter.ofLocalizedTime(dateStyle)
localDateTime	java.time.LocalDateTime	DateTimeFormatter.ofLocalizedDateTime(dateStyle)
offsetTime	java.time.OffsetTime	DateTimeFormatter.ISO_OFFSET_TIME
offsetDateTime	java.time.OffsetDateTime	DateTimeFormatter.ISO_OFFSET_DATE_TIME
zonedDateTime	java.time.ZonedDateTime	DateTimeFormatter.ISO_ZONED_DATE_TIME

Использование NumberConverter

Вы можете преобразовать данные компонента в java.lang.Number, вложив тег convertNumber в тег компонента. Тег convertNumber имеет несколько атрибутов, которые позволяют указать формат и тип данных. Таблица 11-4 перечисляет атрибуты.

В следующем примере тег convertNumber используется для отображения общей цены содержимого корзины покупок:

```xml
<h:outputText value="#{cart.total}">
    <f:convertNumber currencySymbol="$" type="currency"/>
</h:outputText>
```

При связывании `NumberConverter` с компонентом убедитесь, что свойство Managed-бина, с которым связан компонент, имеет примитивный тип или тип `java.lang.Number`. В предыдущем примере `cart.total` имеет тип `double`.

Вот пример числа, которое может отображать этот тег:

```
$934
```

Этот результат также можно отобразить с помощью следующего тега, в котором указан шаблон валюты:

```xml
<h:outputText id="cartTotal" value="#{cart.total}">
    <f:convertNumber pattern="$####" />
</h:outputText>
```

См. урок «Настройка форматов» учебника Java на странице https://docs.oracle.com/javase/tutorial/i18n/format/decimalFormat.html для получения дополнительной информации о том, как отформатировать вывод с помощью атрибута `pattern` тега `convertNumber`.

Таблица 11-4 Атрибуты для тега f:convertNumber

Атрибут	**Тип**	**Описание**
`binding`	`NumberConverter`	Используется для связывания конвертера со свойством Managed-бина.
`currencyCode`	`String`	Код валюты ISO 4217, используется только при форматировании валют.
`currencySymbol`	`String`	Символ валюты, применяется только при форматировании валют.
`for`	`String`	Используется с составными компонентами. Относится к одному из объектов в составном компоненте, внутрь которого этот тег вложен.
`groupingUsed`	`Boolean`	Указывает, содержит ли отформатированный вывод разделители группировки.
`integerOnly`	`Boolean`	Указывает, будет ли анализироваться только целая часть значения.
`locale`	`String` или `Locale`	`Locale`, число стилей которого используется для форматирования или анализа данных.
`maxFractionDigits`	`int`	Максимальное количество цифр, отформатированных в дробной части вывода.

Атрибут	Тип	Описание
`maxIntegerDigits`	`int`	Максимальное количество цифр, отформатированных в целочисленной части вывода.
`minFractionDigits`	`int`	Минимальное количество цифр, отформатированных в дробной части вывода.
`minIntegerDigits`	`int`	Минимальное количество цифр, отформатированных в целочисленной части вывода.
`pattern`	`String`	Пользовательский шаблон форматирования, определяющий способ форматирования и синтаксического анализа числовой строки.
`type`	`String`	Указывает, анализируется ли строковое значение и форматируется ли он как `number` , `currency` или `percentage` . Если не указано, используется `number` .

Регистрация слушателей в компонентах

Разработчик приложения может реализовывать слушатели как классы или как методы Managed-бина. Если слушатель является методом Managed-бина, автор страницы ссылается на метод либо из атрибута `valueChangeListener` компонента, либо из его атрибута `actionListener` . Если слушатель является классом, автор страницы может ссылаться на слушателя либо из тега `f:valueChangeListener` , либо из тега `f:actionListener` и вкладывать внутрь тега компонента для регистрирации слушателя в компоненте.

Ссылка на метод-обработчик действия и Ссылка на метод-обработчик изменения значения объясняют, как автор страницы использует атрибуты `valueChangeListener` и `actionListener` для ссылки на методы Managed-бина, которые обрабатывают события.

В этом разделе объясняется, как зарегистрировать слушатель изменения значения `NameChanged` и слушатель действия `BookChange` в компонентах. В примере Duke's Bookstore участвуют оба этих слушателя.

Регистрация слушателя изменения значения в компоненте

Автор страницы может зарегистрировать реализацию `ValueChangeListener` в компоненте, который реализует `EditableValueHolder` , вложив тег `f:valueChangeListener` в тег компонента на странице. Тег `f:valueChangeListener` поддерживает атрибуты, показанные в таблице 11-5, один из которых обязательно должен использоваться.

Таблица 11-5 Атрибуты для тега f:valueChangeListener

Атрибут	Описание
`type`	Ссылается на полное имя класса реализации `ValueChangeListener`. Может принимать литерал или выражение значения.

Атрибут	Описание
`binding`	Ссылается на объект, который реализует `ValueChangeListener`. Может принимать только выражение значения, которое должно указывать на свойство Managed-бина, принимающее и возвращающее реализацию `ValueChangeListener`.

В следующем примере показан слушатель изменения значения, зарегистрированный в компоненте:

```xml
<h:inputText id="name"
             size="30"
             value="#{cashierBean.name}"
             required="true"
             requiredMessage="#{bundle.ReqCustomerName}">
    <f:valueChangeListener
        type="ee.jakarta.tutorial.dukesbookstore.listeners.NameChanged" />
</h:inputText>
```

В этом примере атрибут `type` основного тега указывает кастомный слушатель `NameChanged` как реализацию `ValueChangeListener`, зарегистрированную в `name` компонента.

После обработки этого тега и валидации локальных значений соответствующий ему компонент поставит `ValueChangeEvent` в очередь для компонента, связанного с указанным `ValueChangeListener`.

Атрибут `binding` используется для связывания реализации `ValueChangeListener` со свойством Managed-бина. Этот атрибут работает аналогично атрибуту `binding`, поддерживаемому стандартными тегами конвертера. См. Связывание значений компонентов и объектов со свойствами Managed-бина для получения дополнительной информации.

Регистрация слушателя действий в компоненте

Автор страницы может зарегистрировать реализацию `ActionListener` в командном компоненте, вложив тег `f:actionListener` в тег компонента на странице. Аналогично тегу `f:valueChangeListener`, тег `f:actionListener` поддерживает оба атрибута: `type` и `binding`. Один из этих атрибутов должен использоваться для ссылки на слушатель действия.

Вот пример тега `h:commandLink`, который ссылается на реализацию `ActionListener`:

```xml
<h:commandLink id="Duke" action="bookstore">
    <f:actionListener
        type="ee.jakarta.tutorial.dukesbookstore.listeners.LinkBookChangeListener" />
    <h:outputText value="#{bundle.Book201}"/>
</h:commandLink>
```

Атрибут `type` тега `f:actionListener` указывает полное имя класса реализации `ActionListener`. Аналогично тегу `f:valueChangeListener`, тег `f:actionListener` также поддерживает атрибут `binding`. Смотрите Связывание конвертеров, слушателей и валидаторов со свойствами Managed-бинов для получения дополнительной информации о связывании слушателей со свойствами Managed-бинов.

В дополнение к тегу `actionListener`, который позволяет зарегистрировать пользовательский слушатель в компоненте, библиотека основных тегов включает тег `f:setPropertyActionListener`. Этот тег используется для регистрации специального слушателя действий в объекте `ActionSource`, связанном с компонентом.

Когда компонент активирован, слушатель сохраняет объект, на который ссылается атрибут `value`, в объект, на который ссылается атрибут `target`.

Страница `bookcatalog.xhtml` приложения Duke's Bookstore использует `f:setPropertyActionListener` с двумя компонентами: компонентом `h:commandLink`, используемым для ссылки на страницу `bookdetails.xhtml` и компонентом `h:commandButton`, используемым для добавления книги в корзину:

XML

```
<h:dataTable id="books"
    value="#{store.books}"
    var="book"
    headerClass="list-header"
    styleClass="list-background"
    rowClasses="list-row-even, list-row-odd"
    border="1"
    summary="#{bundle.BookCatalog}" >
    ...
    <h:column>
        <f:facet name="header">
            <h:outputText value="#{bundle.ItemTitle}"/>
        </f:facet>
        <h:commandLink action="#{catalog.details}"
                       value="#{book.title}">
            <f:setPropertyActionListener target="#{requestScope.book}"
                                         value="#{book}"/>
        </h:commandLink>
    </h:column>
    ...
    <h:column>
        <f:facet name="header">
            <h:outputText value="#{bundle.CartAdd}"/>
        </f:facet>
        <h:commandButton id="add"
                         action="#{catalog.add}"
                         value="#{bundle.CartAdd}">
            <f:setPropertyActionListener target="#{requestScope.book}"
                                         value="#{book}"/>
        </h:commandButton>
    </h:column>
    ...
</h:dataTable>
```

Теги `h:commandLink` и `h:commandButton` находятся внутри тега `h:dataTable`, который перебирает список книг. Атрибут `var` указывает на одну книгу в списке.

Объект, на который ссылается атрибут `var` тега `h:dataTable`, находится в области видимости страницы. Однако в этом случае нужно поместить этот объект в область запроса, чтобы при активации пользователем компонента `commandLink` для перехода на `bookdetails.xhtml` или компонента `commandButton` для перехода на `bookcatalog.xhtml` данные книги стали доступны для этих страниц. Следовательно, тег `f:setPropertyActionListener` используется для установки объекта текущей книги в область запроса, когда активирован компонент `commandLink` или `commandButton`.

В предыдущем примере атрибут `value` тега `f:setPropertyActionListener` ссылается на объект `book`. Атрибут `target` тега `f:setPropertyActionListener` ссылается на выражение значения `requestScope.book`, в котором содержится объект `book` атрибута `value`, когда активируется компонент `commandLink` или `commandButton`.

Использование стандартных валидаторов

Jakarta Faces предоставляет набор стандартных классов и связанных с ними тегов, которые авторы страниц и разработчики приложений могут использовать для валидации данных компонента. Таблица 11-6 перечисляет все стандартные классы валидаторов и теги, которые позволяют использовать валидаторы со страницы.

Таблица 11-6. Классы валидаторов

Класс валидатора	**Тег**	**Функция**
`BeanValidator`	`validateBean`	Регистрирует валидатор бина в компоненте.
`BeanValidator`	`validateWholeBean`	Позволяет валидацию нескольких полей путём включения валидацию бина на уровне класса вспомогательного бина CDI.
`DoubleRangeValidator`	`validateDoubleRange`	Проверяет, находится ли локальное значение компонента в определённом диапазоне. Значение должно быть числом с плавающей точкой или конвертируемым в число с плавающей точкой.
`LengthValidator`	`validateLength`	Проверяет, находится ли длина локального значения компонента в определённом диапазоне. Значение должно быть `java.lang.String`.
`LongRangeValidator`	`validateLongRange`	Проверяет, находится ли локальное значение компонента в определённом диапазоне. Значение должно быть любого числового типа или объектом типа `String`, конвертируемым в `long`.
`RegexValidator`	`validateRegex`	Проверяет, соответствует ли локальное значение компонента регулярному выражению из пакета `java.util.regex`.
`RequiredValidator`	`validateRequired`	Гарантирует, что локальное значение не является пустым в компоненте `EditableValueHolder`.

Все эти классы валидатора реализуют интерфейс `Validator`. Авторы компонентов и разработчики приложений также могут реализовать этот интерфейс, чтобы задать собственный набор ограничений для значения компонента.

Подобно стандартным конвертерам, каждый из этих валидаторов имеет одно или несколько стандартных сообщений об ошибках, связанных с ним. Если вы зарегистрировали один из этих валидаторов в компоненте на своей странице, и валидатор не может проверить значение компонента, на странице появится сообщение об ошибке валидатора. Например, сообщение об ошибке, которое отображается, когда значение компонента превышает максимально допустимое значение `LongRangeValidator`, выглядит следующим образом:

```
{1}: Валидация Error: значение больше допустимого максимума "{0}"
```
JAVA

В этом случае параметр замещения `{1}` заменяется меткой компонента или `id`, а параметр замещения `{0}` заменяется на максимальное значение, разрешённое валидатором.

См. Отображение сообщений об ошибках тегами h:message и h:messages для получения информации о том, как отображать сообщения об ошибках на странице при сбоях валидации.

Вместо использования стандартных валидаторов вы можете использовать Bean Validation для проверки данных. Если вы указываете ограничения Bean Validation на свойствах вашего Managed-бина, эти ограничения автоматически помещаются в соответствующие поля на веб-страницах ваших приложений. См. главу 23 *Введение в Jakarta Bean Validation* для получения дополнительной информации. Вам не нужно указывать тег `validateBean` для использования Bean Validation, но тег позволяет использовать расширенные функции Bean Validation. Например, можно использовать атрибут `validationGroups` , чтобы указать группы ограничений.

Вы также можете создавать и регистрировать кастомные валидаторы, хотя Bean Validation делает эту функцию менее полезной. Для получения дополнительной информации см. Создание и использование кастомного валидатора.

Валидация значения компонента

Чтобы проверить значение компонента с помощью определённого валидатора, необходимо зарегистрировать этот валидатор в компоненте. Вы можете сделать это одним из следующих способов.

- Вложите соответствующий тег валидатора (показан в таблице 11-6) внутрь тега компонента. Использование тегов валидатора объясняет, как использовать тег `validateLongRange` . Вы можете использовать другие стандартные теги таким же образом.
- Обратитесь к методу, который выполняет проверку из атрибута `validator` тега компонента.
- Вложите тег валидатора в тег компонента и используйте для ссылки на валидатор либо атрибут `validatorId` тега валидатора, либо атрибут `binding` компонента.

Смотрите Ссылка на метод, который выполняет валидацию для получения дополнительной информации об использовании атрибута `validator` .

Атрибут `validatorId` работает аналогично атрибуту `converterId` тега `converter` , как описано в Конвертация значения компонента.

Помните, что проверка может быть выполнена только для компонентов, которые реализуют `EditableValueHolder` , потому что только эти компоненты принимают валидируемые значения.

Использование тегов валидатора

В следующем примере показано, как использовать тег валидатора `f:validateLongRange` для компонента ввода с именем `quantity` :

```xml
<h:inputText id="quantity" size="4" value="#{item.quantity}">
    <f:validateLongRange minimum="1"/>
</h:inputText>
<h:message for="quantity"/>
```

Этот тег требует, чтобы пользователь ввёл число не меньше 1 (единицы). Тег `validateLongRange` также имеет атрибут `maximum` , который устанавливает максимальное значение для ввода.

Атрибуты всех стандартных тегов валидатора принимают выражения значений EL. Это означает, что атрибуты могут ссылаться на свойства Managed-бина, а не указывать литеральные значения. Например, тег `f:validateLongRange` в предыдущем примере может ссылаться на свойства `minimum` и `maximum` Managed-

бина, чтобы получить минимальное и максимальное допустимые значения валидатора, как показано в этом фрагменте из примера `guessnumber-jsf`:

```xml
<h:inputText id="userNo"
             title="Type a number from 0 to 10:"
             value="#{userNumberBean.userNumber}">
    <f:validateLongRange minimum="#{userNumberBean.minimum}"
                         maximum="#{userNumberBean.maximum}"/>
</h:inputText>
```

Следующий тег `f:validateRegex` показывает, как можно убедиться, что пароль имеет длину от 4 до 10 символов и содержит как минимум одну цифру, как минимум одну строчную букву и как минимум одну заглавную букву:

```xml
<f:validateRegex pattern="((?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{4,10})"
                 for="passwordVal"/>
```

Ссылка на метод Managed-бина

Тег компонента имеет набор атрибутов для ссылки на методы Managed-бина, которые могут выполнять определённые функции для компонента, связанного с тегом. Эти атрибуты приведены в таблице 11-7.

Таблица 11-7 Атрибуты тегов компонентов, которые ссылаются на методы Managed-бина

Атрибут	Функция
`action`	Указывает на метод Managed-бина, который выполняет обработку навигации для компонента и возвращает результат типа `String`
`actionListener`	Указывает на метод Managed-бина, который обрабатывает события действия
`validator`	Указывает на метод Managed-бина, который выполняет проверку значения компонента
`valueChangeListener`	Указывает на метод Managed-бина, который обрабатывает события изменения значения

Только компоненты, реализующие `ActionSource`, могут использовать атрибуты `action` и `actionListener`. Только компоненты, реализующие `EditableValueHolder`, могут использовать атрибуты `validator` или `valueChangeListener`.

Тег компонента ссылается на метод Managed-бина, использующий выражение метода в качестве значения одного из атрибутов. Метод, на который ссылается атрибут, должен иметь определённую сигнатуру, которая определяется атрибутом тега в документации Jakarta Faces Facelets Tag Library (https://jakarta.ee/specifications/faces/3.0/vdldoc/). Например, определение атрибута `validator` тега `inputText` следующее:

```java
void validate(jakarta.faces.context.FacesContext,
              jakarta.faces.component.UIComponent, java.lang.Object)
```

В следующих разделах приведены примеры использования атрибутов.

Ссылка на метод, который выполняет навигацию

Если ваша страница содержит компонент, такой как кнопка или ссылка, который инициирует переход на другую страницу при активации компонента, то тег, соответствующий этому компоненту, должен содержать атрибут `action`. Этот атрибут выполняет одно из следующих действий:

- Определяет результат типа `String`, который сообщает приложению, к какой странице перейти дальше
- Ссылается на метод Managed-бина, который выполняет некоторую обработку и возвращает результат типа `String`

В следующем примере показано, как ссылаться на метод навигации:

```xml
<h:commandButton value="#{bundle.Submit}"
                 action="#{cashierBean.submit}" />
```

Смотрите Пишем метод для обработки навигации для получения информации о том, как написать такой метод.

Ссылка на метод-обработчик действия

Если компонент на вашей странице генерирует событие действия и если это событие обрабатывается методом Managed-бина, вы обращаетесь к этому методу с помощью атрибута `actionListener` компонента.

В следующем примере показано, как можно ссылаться на такой метод:

```xml
<h:commandLink id="Duke" action="bookstore"
               actionListener="#{actionBean.chooseBookFromLink}">
```

Атрибут `actionListener` этого тега компонента ссылается на метод `chooseBookFromLink`, используя выражение метода. Метод `chooseBookFromLink` обрабатывает событие, когда пользователь кликает ссылку, созданную этим компонентом. Смотрите Пишем метод-обработчик действия для получения информации о том, как написать такой метод.

Ссылка на метод, который выполняет валидацию

Если вход одного из компонентов на вашей странице проверен методом Managed-бина, обратитесь к методу из тега компонента с помощью атрибута `validator`.

В следующем упрощённом примере из примера CDI guessnumber-cdi показано, как ссылаться на метод, который выполняет проверку на `inputGuess`, компоненте ввода:

```xml
<h:inputText id="inputGuess"
    value="#{userNumberBean.userNumber}"
    required="true" size="3"
    disabled="#{userNumberBean.number eq userNumberBean.userNumber ...}"
    validator="#{userNumberBean.validateNumberRange}">
</h:inputText>
```

Метод Managed-бина `validateNumberRange` проверяет, что входное значение находится в допустимом диапазоне, который изменяется каждый раз, когда делается другое предположение. Смотрите Пишем метод для выполнения валидации для получения информации о том, как написать такой метод.

Ссылка на метод-обработчик изменения значения

Если вы хотите, чтобы компонент на странице генерировал событие изменения значения и чтобы это событие обрабатывалось методом Managed-бина вместо реализации `ValueChangeListener`, обратитесь к методу с помощью атрибута компонента `valueChangeListener`:

XML

```
<h:inputText id="name"
             size="30"
             value="#{cashierBean.name}"
             required="true"
             valueChangeListener="#{cashierBean.processValueChange}" />
</h:inputText>
```

Атрибут `valueChangeListener` этого тега компонента ссылается на метод `processValueChange` бина `CashierBean` с помощью выражения метода. Метод `processValueChange` обрабатывает событие, когда пользователь вводит имя в поле ввода, отображаемое этим компонентом.

Пишем метод-обработчик изменения значения описывает, как реализовать метод, который обрабатывает `ValueChangeEvent`.

Глава 12. Разработка с использованием Jakarta Faces

В этой главе даётся обзор Managed-бинов и объясняется, как писать методы и свойства Managed-бинов, используемые приложением Jakarta Faces. В этой главе также представлена функциональность Bean Validation.

Managed-бины в Jakarta Faces

Типичное приложение Jakarta Faces включает в себя один или несколько Managed-бинов, каждый из которых может быть связан с компонентами, используемыми на конкретной странице. В этом разделе представлены основные концепции создания, настройки и использования Managed-бинов в приложении.

Глава 10 *Использование Jakarta Faces на веб-страницах* и глава 11 *Использование конвертеров, слушателей и валидаторов* показывают, как добавлять компоненты на страницу и связывать их с объектами на стороне сервера с помощью тегов компонентов и основных тегов. В этих главах также показано, как предоставить дополнительную функциональность компонентам через конвертеры, слушатели и валидаторы. Разработка приложения Jakarta Faces также включает в себя задачу программирования серверных объектов: Managed-бинов, конвертеров, обработчиков событий и валидаторов.

Создание Managed-бина

Managed-бин создаётся с помощью конструктора без аргументов, набора свойств и методов, необходимых для выполнения функций компонента. Каждое из свойств Managed-бина может быть связано с одним из следующих:

- Значение компонента
- Объект компонента
- Объект конвертера
- Объект слушателя
- Объект валидатора

Наиболее распространённые функции, выполняющиеся методами Managed-бина, включают в себя следующие:

- Валидация данных компонента
- Обработка события, инициированного компонентом
- Выполнение обработки для определения следующей страницы, на которую должен быть выполнен переход

Как и во всех компонентах JavaBeans, свойство состоит из приватного поля данных и набора методов доступа, как показано в следующем коде:

JAVA

```
private Integer userNumber = null;
...
public void setUserNumber(Integer user_number) {
    userNumber = user_number;
}
public Integer getUserNumber() {
    return userNumber;
}
```

При связывании со значением компонента свойство бина может быть любым примитивным числовым типом или любого типа Java, для которого приложение имеет соответствующий конвертер. Например, свойство может иметь тип `java.util.Date`, если приложение имеет конвертер, который может преобразовать тип `Date` в `String` и обратно. См. Запись свойств бина для получения информации о том, какие типы принимаются тегами компонентов.

Когда свойство бина связано с объектом компонента, тип свойства должен совпадать с типом компонента. Например, если компонент `jakarta.faces.component.UISelectBoolean` связан со свойством, свойство должно принимать и возвращать объект `UISelectBoolean`. Аналогично, если свойство связано с объектом конвертера, валидатора или слушателя, свойство должно иметь соответствующий тип конвертера, валидатора или слушателя.

Для получения дополнительной информации о написании бинов и их свойств см. Запись свойств бина.

Использование EL для ссылки на Managed-бины

Чтобы связать значения и объекты компонента со свойствами Managed-бина или ссылаться на методы Managed-бина из тегов компонента, авторы страниц используют синтаксис языка выражений. Как объясняется в Обзоре EL, ниже перечислены некоторые функции, которые предлагает EL:

- Отложенное выполнение выражений
- Возможность использовать выражение значения для чтения и записи данных
- Выражения методов

Отложенное выполнение выражений важно, поскольку жизненный цикл Jakarta Faces разделён на несколько фаз, на которых обработка событий компонента, конвертация и валидация данных, а также передача данных во внешние объекты выполняются упорядоченным образом. Реализация должна иметь возможность задержать выполнение выражений до тех пор, пока не будет достигнута надлежащая фаза жизненного цикла. Поэтому реализация атрибутов тега всегда используют синтаксис отложенного выполнения, которое имеет разделитель `#{}`.

Для хранения данных во внешних объектах почти все атрибуты тега Jakarta Faces используют выражения lvalue — выражениями, позволяющими как получать, так и устанавливать данные для внешних объектов.

Наконец, некоторые атрибуты принимают выражения метода, которые ссылаются на методы, обрабатывающие события компонента, валидирующие или конвертирующие данные компонента.

Чтобы проиллюстрировать тег Jakarta Faces с использованием EL, следующий тег ссылается на метод, валидирующий пользовательский ввод:

```XML
<h:inputText id="inputGuess"
    value="#{userNumberBean.userNumber}"
    required="true" size="3"
    disabled="#{userNumberBean.number eq userNumberBean.userNumber ...}"
    validator="#{userNumberBean.validateNumberRange}">
</h:inputText>
```

Этот тег связывает значение компонента `inputGuess` со свойством Managed-бина `UserNumberBean.userNumber` используя выражение lvalue. В теге используется выражение метода для ссылки на метод `UserNumberBean.validateNumberRange`, который валидирует локальное значение компонента. Локальное значение — это то, что пользователь вводит в поле, соответствующем этому тегу. Этот метод вызывается при вычислении выражения.

Почти все атрибуты тега Jakarta Faces принимают выражения значений. В дополнение к ссылкам на свойства бинов выражения значений могут ссылаться на списки (List), отображения (Map), массивы, неявные объекты и bundle-ресурсы.

Другое использование выражений значений — это связывание объекта компонента со свойством Managed-бина. Автор страницы делает это, ссылаясь на свойство из атрибута `binding`:

```XML
<h:outputLabel for="fanClub"
               rendered="false"
               binding="#{cashierBean.specialOfferText}"
               value="#{bundle.DukeFanClub}"/>
</h:outputLabel>
```

Помимо использования выражений со стандартными тегами компонентов, можно настроить кастомные свойства компонента на приём выражений путем создания объектов `jakarta.el.ValueExpression` или `jakarta.el.MethodExpression`.

Для получения информации о EL см. главу 9 *Язык выражений*.

Для получения информации о ссылках на методы Managed-бина из тегов компонента см. Ссылка на метод Managed-бина.

Запись свойств бина

Как объясняется в Managed-бинах в Jakarta Faces, свойство Managed-бина может быть связано с одним из следующих элементов:

- Значение компонента
- Объект компонента
- Реализация конвертера
- Реализация слушателя
- Реализация валидатора

Эти свойства соответствуют соглашениям компонентов JavaBeans (также называемых бинами). Дополнительные сведения о компонентах JavaBeans см. в учебном пособии JavaBeans по ссылке https://docs.oracle.com/javase/tutorial/javabeans/index.html.

Тег компонента связывает значение компонента со свойством Managed-бина с помощью его атрибута `value` и объект компонента со свойством Managed-бина с помощью его атрибута `binding`. Аналогично, все теги конвертера, слушателя и валидатора используют свои атрибуты `binding` для связывания их реализаций со свойствами Managed-бина. См. Связывание значений и объектов компонентов со свойствами Managed-бинов и Связывание конвертеров, слушателей и валидаторов со свойствами Managed-бинов для получения дополнительной информации.

Чтобы связать значение компонента со свойством Managed-бина, тип свойства должен соответствовать типу значения компонента. Например, если свойство Managed-бина связано со значением компонента `UISelectBoolean`, свойство должно принимать и возвращать значение `boolean` или объект-обёртку (wrapper) `Boolean`.

Чтобы связать объект компонента со свойством Managed-бина, свойство должно соответствовать типу компонента. Например, если свойство Managed-бина связано с объектом `UISelectBoolean`, свойство должно принять и вернуть значение `UISelectBoolean`.

Аналогично, чтобы связать реализацию конвертера, слушателя или валидатора со свойством Managed-бина, свойство должно принимать и возвращать объект конвертера, слушателя или валидатора того же типа. Например, если используется тег `convertDateTime` для связывания `jakarta.faces.convert.DateTimeConverter` со свойством, это свойство должно принимать и возвращать объект `DateTimeConverter`.

В оставшейся части этого раздела объясняется, как записывать свойства, которые можно связать со значениями компонентов и с объектами компонентов для компонентов, описанных в Добавление компонентов на страницу с использованием библиотеки тегов HTML, а также с реализацией конвертера, слушателя и валидатора.

Запись свойств объектов, связанных со значениями компонентов

Чтобы написать свойство Managed-бина, связанное со значением компонента, необходимо сопоставить тип свойства значению компонента.

Таблица 12-1 перечисляет классы пакета `jakarta.faces.component` и допустимые типы их значений.

Таблица 12-1. Допустимые типы значений компонентов

Класс компонента	**Допустимые типы значений компонентов**
`UIInput`, `UIOutput`, `UISelectItem`, `UISelectOne`	Любой из основных примитивных и числовых типов или любой объектный тип Java, для которого доступна соответствующая реализация `jakarta.faces.convert.Converter`
`UIData`	`array` бинов, `List` бинов, отдельный бин, `java.sql.ResultSet`, `jakarta.servlet.jsp.jstl.sql.Result`, `javax.sql.RowSet`
`UISelectBoolean`	`boolean` или `Boolean`
`UISelectItems`	`java.lang.String`, `Collection`, `Array`, `Map`
`UISelectMany`	`array` или `List`, а также элементы `array` или `List` могут быть любого стандартного типа

Когда они связывают компоненты со свойствами с помощью атрибутов `value`, авторам страниц необходимо убедиться, что соответствующие свойства соответствуют типам значений компонентов.

Свойства UIInput и UIOutput

Классы компонентов `UIInput` и `UIOutput` представлены тегами компонентов, которые начинаются с `h:input` и `h:output` соответственно (например, `h:inputText` и `h:outputText`).

В следующем примере тег `h:inputText` связывает компонент `name` со свойством `name` Managed-бина `CashierBean`.

XML

```
<h:inputText id="name"
             size="30"
             value="#{cashierBean.name}"
    ...>
</h:inputText>
```

В следующем фрагменте кода из Managed-бина `CashierBean` показан тип свойства компонента, связанный с предыдущим тегом компонента:

JAVA

```
protected String name = null;

public void setName(String name) {
    this.name = name;
}
public String getName() {
    return this.name;
}
```

Как описано в Использование стандартных конвертеров, для преобразования значения входного или выходного компонента вы можете либо применить конвертер, либо создать свойство бина, связанное с компонентом соответствующего типа. Вот пример тега Использование DateTimeConverter, который отображает дату отправки товаров.

XML

```
<h:outputText value="#{cashierBean.shipDate}">
    <f:convertDateTime type="date" dateStyle="full" />
</h:outputText>
```

Свойство бина, представленное этим тегом, должно иметь тип `java.util.Date`. В следующем фрагменте кода показано свойство `shipDate` из Managed-бина `CashierBean`, которое связано со значением тега в предыдущем примере:

JAVA

```
private Date shipDate;

public Date getShipDate() {
    return this.shipDate;
}
public void setShipDate(Date shipDate) {
    this.shipDate = shipDate;
}
```

Свойства UIData

Класс компонента `UIData` представлен тегом `h:dataTable`.

Компоненты `UIData` должны быть связаны с одним из типов свойств Managed-бина, перечисленных в таблице 12-1. Компоненты данных обсуждаются в Использование таблицы Data-Bound. Вот открывающая часть тега `dataTable` из этого раздела:

```xml
<h:dataTable id="items"
    ...
    value="#{cart.items}"
    ...
    var="item">
```

Выражение значения указывает на свойство `items` бина `cart` корзины покупок. Бин `cart` содержит отображение (Map) бинов `ShoppingCartItem`.

Метод `getItems` бина `cart` заполняет `List` объектами `ShoppingCartItem`, которые сохраняются в отображении (Map) `items`, когда клиент добавляет книги в корзину, как показано в следующем фрагменте кода:

```java
public synchronized List<ShoppingCartItem> getItems() {
    List<ShoppingCartItem> results = new ArrayList<ShoppingCartItem>();
    results.addAll(this.items.values());
    return results;
}
```

Все компоненты, содержащиеся в компоненте `UIData`, связаны со свойствами компонента `cart`, который связан со всем компонентом `UIData`. Например, вот тег `h:outputText`, который отображает название книги в таблице:

```xml
<h:commandLink action="#{showcart.details}">
    <h:outputText value="#{item.item.title}"/>
</h:commandLink>
```

Название на самом деле является ссылкой на страницу `bookdetails.xhtml`. Тег `h:outputText` использует выражение значения `#{item.item.title}`, чтобы связать его компонент `UIOutput` со свойством `title` объекта `Book`. Первым элементом в выражении является объект `ShoppingCartItem`, на который ссылается тег `h:dataTable` при отображении текущей строки. Второй элемент в выражении ссылается на свойство `item` `ShoppingCartItem`, которое возвращает `Object` (в данном случае, `Book`). Часть выражения `title` ссылается на свойство `title` в `Book`. Значение компонента `UIOutput`, соответствующего этому тегу, связано со свойством `title` объекта `Book`:

```java
private String title;
...
public String getTitle() {
    return title;
}

public void setTitle(String title) {
    this.title = title;
}
```

Компонент UIData (и UIRepeat) поддерживает интерфейсы `Map` и `Iterable`, а также кастомные типы.

Для UIData и UIRepeat поддерживаются следующие типы:

- `null` (становится пустым списком)
- `jakarta.faces.model.DataMode`
- `java.util.List`

- `java.lang.Object[]`
- `java.sql.ResultSet`
- `jakarta.servlet.jsp.jstl.sql.Result`
- `java.util.Collection`
- `java.lang.Iterable`
- `java.util.Map`
- `java.lang.Object` (становится ScalarDataModel)

Свойства UISelectBoolean

Класс компонента `UISelectBoolean` представлен тегом компонента `h:selectBooleanCheckbox`.

Свойства Managed-бина, которые содержат данные компонента `UISelectBoolean`, должны иметь тип `boolean` или `Boolean`. Пример тега `selectBooleanCheckbox` из раздела Компоненты выбора одного из значений связывает компонент со свойством. В следующем примере показан тег, который связывает значение компонента со свойством `boolean`:

XML
```
<h:selectBooleanCheckbox title="#{bundle.receiveEmails}"
                         value="#{custFormBean.receiveEmails}">
</h:selectBooleanCheckbox>
<h:outputText value="#{bundle.receiveEmails}">
```

Вот пример свойства, которое можно связать с компонентом, представленным тегом примера:

JAVA
```
private boolean receiveEmails = false;
...
public void setReceiveEmails(boolean receiveEmails) {
    this.receiveEmails = receiveEmails;
}
public boolean getReceiveEmails() {
    return receiveEmails;
}
```

Свойства UISelectMany

Класс компонента `UISelectMany` представлен тегами компонента, которые начинаются с `h:selectMany` (например, `h:selectManyCheckbox` и `h:selectManyListbox`).

Поскольку компонент `UISelectMany` позволяет пользователю выбирать один или несколько элементов из списка, этот компонент должен сопоставляться со свойством бина типа `List` или `array`. Это свойство бина представляет набор выбранных в данный момент элементов из списка доступных.

Следующий пример тега `selectManyCheckbox` взят из Компонент выбора нескольких значений:

XML
```
<h:selectManyCheckbox id="newslettercheckbox"
                      layout="pageDirection"
                      value="#{cashierBean.newsletters}">
    <f:selectItems value="#{cashierBean.newsletterItems}"/>
</h:selectManyCheckbox>
```

Вот свойство бина, которое отображается на `value` тега `selectManyCheckbox` из предыдущего примера:

JAVA
```
private String[] newsletters;

public void setNewsletters(String[] newsletters) {
    this.newsletters = newsletters;
}
public String[] getNewsletters() {
    return this.newsletters;
}
```

Компоненты `UISelectItem` и `UISelectItems` используются для представления всех значений в компоненте `UISelectMany`. См. Свойства UISelectItems для получения информации о записи свойств для компонентов `UISelectItem` и `UISelectItems`.

Свойства UISelectOne

Класс компонента `UISelectOne` представлен тегами компонента, которые начинаются с `h:selectOne` (например, `h:selectOneRadio` и `h:selectOneListbox`).

Свойства `UISelectOne` принимают те же типы, что и свойства `UIInput` и `UIOutput`, поскольку компонент `UISelectOne` представляет выбор одного элемента из набора. Этот элемент может быть любого из примитивных типов и всего, для чего вы можете применить конвертер.

Вот пример тега `h:selectOneMenu` из Отображение меню тегом h:selectOneMenu:

XML
```
<h:selectOneMenu id="shippingOption"
                 required="true"
                 value="#{cashierBean.shippingOption}">
    <f:selectItem itemValue="2"
                  itemLabel="#{bundle.QuickShip}"/>
    <f:selectItem itemValue="5"
                  itemLabel="#{bundle.NormalShip}"/>
    <f:selectItem itemValue="7"
                  itemLabel="#{bundle.SaverShip}"/>
 </h:selectOneMenu>
```

Вот свойство бина, соответствующее этому тегу:

JAVA
```
private String shippingOption = "2";

public void setShippingOption(String shippingOption) {
    this.shippingOption = shippingOption;
}
public String getShippingOption() {
    return this.shippingOption;
}
```

Обратите внимание, что `shippingOption` представляет текущий выбранный элемент из списка в компоненте `UISelectOne`.

Компоненты `UISelectItem` и `UISelectItems` используются для представления всех значений в компоненте `UISelectOne`. Это объясняется в Отображение меню тегом h:selectOneMenu.

Для получения информации о том, как записать свойства Managed-бина для компонентов `UISelectItem` и `UISelectItems`, см. Свойства UISelectItems.

Свойства UISelectItem

Компонент `UISelectItem` представляет одно значение в наборе в компоненте `UISelectMany` или `UISelectOne`. Компонент `UISelectItem` должен быть связан со свойством Managed-бина типа `jakarta.faces.model.SelectItem`. Объект `SelectItem` состоит из `Object`, представляющего значение, и двух `Strings`, представляющих метку и описание объекта `UISelectItem`.

Пример тега `selectOneMenu` из Свойства UISelectOne содержит теги `selectItem`, которые задают значения списка элементов на странице. Вот пример свойства компонента, который может устанавливать значения для этого списка в компоненте:

JAVA

```
SelectItem itemOne = null;

SelectItem getItemOne(){
    return itemOne;
}
void setItemOne(SelectItem item) {
    itemOne = item;
}
```

Свойства UISelectItems

Компоненты `UISelectItems` являются дочерними для компонентов `UISelectMany` и `UISelectOne`. Каждый компонент `UISelectItems` состоит из набора объектов `UISelectItem` или любой коллекции объектов, таких как массив, список или даже POJO.

В следующем фрагменте кода из `CashierBean` показано, как записать свойства для тегов `selectItems`, содержащих объекты `SelectItem`.

JAVA

```
private String[] newsletters;
private static final SelectItem[] newsletterItems = {
    new SelectItem("Duke's Quarterly"),
    new SelectItem("Innovator's Almanac"),
    new SelectItem("Duke's Diet and Exercise Journal"),
    new SelectItem("Random Ramblings")
};
...
public void setNewsletters(String[] newsletters) {
    this.newsletters = newsletters;
}

public String[] getNewsletters() {
    return this.newsletters;
}

public SelectItem[] getNewsletterItems() {
    return newsletterItems;
}
```

Здесь свойство `newsletters` представляет объект `SelectItems`, а свойство `newsletterItems` представляет статический массив объектов `SelectItem`. Класс `SelectItem` имеет несколько конструкторов. В этом примере первым аргументом является `Object`, представляющий значение элемента, тогда как вторым аргументом является `String`, представляющий метку, которая появляется на странице в компоненте `UISelectMany`.

Запись свойств, связанных с объектами компонентов

Свойство, связанное с объектом компонента, возвращает и принимает объект компонента, а не его значение. Следующие компоненты связывают объект компонента со свойством Managed-бина:

```xml
<h:selectBooleanCheckbox id="fanClub"
                         rendered="false"
                         binding="#{cashierBean.specialOffer}" />
<h:outputLabel for="fanClub"
               rendered="false"
               binding="#{cashierBean.specialOfferText}"
               value="#{bundle.DukeFanClub}" />
</h:outputLabel>
```

Тег `selectBooleanCheckbox` отображает флажок и связывает `fanClub` компонента `UISelectBoolean` со свойством `specialOffer` в `CashierBean`. Тег `outputLabel` связывает значение атрибута `value`, представляющего метку флажка, со свойством `specialOfferText` элемента `CashierBean`. Если пользователь заказывает книги на сумму более 100 долларов США и кликает кнопку «Отправить», метод `submit` бина `CashierBean` устанавливает свойства `rendered` обоих компонентов в `true`, в результате чего флажок и метка отображаются при повторном отображении страницы.

Поскольку компоненты, соответствующие тегам из примера, связаны со свойствами Managed-бина, эти свойства должны соответствовать типам компонентов. Это означает, что свойство `specialOfferText` должно иметь тип `UIOutput`, а свойство `specialOffer` должно иметь тип `UISelectBoolean`:

```java
UIOutput specialOfferText = null;
UISelectBoolean specialOffer = null;

public UIOutput getSpecialOfferText() {
    return this.specialOfferText;
}
public void setSpecialOfferText(UIOutput specialOfferText) {
    this.specialOfferText = specialOfferText;
}

public UISelectBoolean getSpecialOffer() {
    return this.specialOffer;
}
public void setSpecialOffer(UISelectBoolean specialOffer) {
    this.specialOffer = specialOffer;
}
```

Для получения общей информации о связывании компонентов см. Managed-бины в Jakarta Faces.

Для получения информации о том, как ссылаться на метод Managed-бина, который выполняет навигацию при клике кнопки, см. Ссылка на метод, который выполняет навигацию.

Для получения дополнительной информации о написании методов Managed-бина, которые обрабатывают навигацию, смотрите Пишем метод для обработки навигации.

Запись свойств объектов, связанных с конвертерами, слушателями или валидаторами

Все стандартные теги конвертеров, слушателей и валидаторов, включённые в Jakarta Faces, поддерживают атрибуты binding, которые позволяют связывать реализации конвертеров, слушателей или валидаторов со свойствами Managed-бинов.

В следующем примере показан стандартный тег `convertDateTime`, использующий выражение значения с его атрибутом `binding` для связывания объекта `jakarta.faces.convert.DateTimeConverter` со свойством `convertDate` бина `LoginBean`:

XML

```xml
<h:inputText value="#{loginBean.birthDate}">
    <f:convertDateTime binding="#{loginBean.convertDate}" />
</h:inputText>
```

Поэтому свойство `convertDate` должно принимать и возвращать объект `DateTimeConverter`, как показано здесь:

JAVA

```java
private DateTimeConverter convertDate;
public DateTimeConverter getConvertDate() {
        ...
    return convertDate;
}
public void setConvertDate(DateTimeConverter convertDate) {
    convertDate.setPattern("EEEEEEEE, MMM dd, yyyy");
    this.convertDate = convertDate;
}
```

Поскольку конвертер связан со свойством Managed-бина, свойство Managed-бина может изменять атрибуты конвертера или добавлять к нему новые функциональные возможности. В случае предыдущего примера свойство устанавливает шаблон даты, который конвертер использует для анализа ввода пользователя в объект `Date`.

Свойства Managed-бина, связанные с реализациями валидатора или слушателя, записываются одинаково и имеют одинаковое общее назначение.

Пишем методы Managed-бинов

Методы Managed-бина могут выполнять несколько специфичных для приложения функций. Эти функции включают

- Выполнение обработки, связанной с навигацией
- Обработка событий действия
- Выполнение валидации значения компонента
- Обработка событий изменения значения

Зачем использовать Managed-бины

Используя Managed-бин для выполнения этих функций, вы устраняете необходимость реализации `jakarta.faces.validator.Validator` для обработки валидации или одного из интерфейсов слушателя для обработки событий. Кроме того, используя Managed-бин вместо реализации `Validator` для выполнения валидации, вы устраняете необходимость создания кастомного тега для реализации `Validator`.

Как правило, рекомендуется включать эти методы в один и тот же Managed-бин, определяющий свойства компонентов, ссылающихся на эти методы. Причина заключается в том, что методам может потребоваться доступ к данным компонента, чтобы определить, как обрабатывать событие, или выполнить валидацию, связанную с компонентом.

В следующих разделах объясняется, как писать различные типы методов Managed-бина.

Пишем метод для обработки навигации

Метод действия — метод Managed-бина, который обрабатывает навигацию — должен быть публичным методом, который не принимает параметров и возвращает `Object`, являющийся сигналом навигационной системе для определения следующей отображаемой страницы. На этот метод ссылается атрибут `action` тега

компонента.

Следующий метод действия взят из Managed-бина `CashierBean`, который вызывается, когда пользователь кликает кнопку «Отправить» на странице. Если пользователь заказал книги стоимостью более 100 долларов, этот метод устанавливает свойства `rendered` компонентов `fanClub` и `specialOffer` в `true`, заставляя их отображаться на странице при следующем отображении страницы.

После установки свойств `rendered` компонентов в `true` этот метод возвращает результат `null`. Это приводит к тому, что Jakarta Faces повторно отображает страницу без создания нового представления, сохраняя введённые пользователем данные. Если бы этот метод возвратил `purchase`, что является сигналом для перехода на страницу оплаты, страница была бы повторно отображена без сохранения ввода клиента. В этом случае вы перерисовываете страницу без очистки данных.

Если пользователь не покупает книг на сумму более 100 долларов США или компонент `thankYou` уже отрисован, метод возвращает `bookreceipt`. Jakarta Faces загружает страницу `bookreceipt.xhtml` после возврата из этого метода:

JAVA

```
public String submit() {
    ...
    if ((cart().getTotal() > 100.00) && !specialOffer.isRendered()) {
        specialOfferText.setRendered(true);
        specialOffer.setRendered(true);
        return null;
    } else if (specialOffer.isRendered() && !thankYou.isRendered()) {
        thankYou.setRendered(true);
        return null;
    } else {
        ...
        cart.clear();
        return ("bookreceipt");
    }
}
```

Как правило, метод действия возвращает результат `String`, как показано в предыдущем примере. Кроме того, вы можете определить класс `Enum`, который инкапсулирует все возможные строки результата, а затем заставить метод действия возвращать константу `enum`, которая представляет конкретный результат `String`, определяемый классом `Enum`.

В следующем примере используется класс `Enum` для инкапсуляции всех результатов:

JAVA

```
public enum Navigation  {
    main, accountHist, accountList, atm, atmAck, transferFunds,
     transferAck, error
}
```

Когда он возвращает результат, метод действия использует точечную нотацию для ссылки на результат из класса `Enum`:

JAVA

```
public Object submit(){
    ...
    return Navigation.accountHist;
}
```

В разделе Ссылка на метод, который выполняет навигацию объясняется, как тег компонента ссылается на этот метод. В разделе Запись свойств, связанных с объектами компонентов объясняется, как записать свойства бинов, с которыми связаны компоненты.

Пишем метод-обработчик действия

Метод Managed-бина, который обрабатывает событие действия, должен быть публичным методом, который принимает событие действия и возвращает `void`. На этот метод ссылается атрибут `actionListener` тега компонента. Только компоненты, реализующие `jakarta.faces.component.ActionSource`, могут ссылаться на этот метод.

В следующем примере метод из Managed-бина с именем `ActionBean` обрабатывает событие, когда пользователь кликает одну из ссылок на странице:

JAVA

```
public void chooseBookFromLink(ActionEvent event) {
    String current = event.getComponent().getId();
    FacesContext context = FacesContext.getCurrentInstance();
    String bookId = books.get(current);
    context.getExternalContext().getSessionMap().put("bookId", bookId);
}
```

Этот метод получает из объекта события компонент, который сгенерировал это событие. Затем он получает идентификатор компонента, который является кодом книги. Метод сопоставляет код с объектом `HashMap`, который содержит коды книг и соответствующие значения идентификаторов книг. Наконец, метод устанавливает идентификатор книги, используя выбранное значение из объекта `HashMap`.

Ссылка на метод-обработчик действия объясняет, как тег компонента ссылается на этот метод.

Пишем метод для выполнения валидации

Вместо реализации `jakarta.faces.validator.Validator` для валидации компонента, вы можете включить метод в Managed-бин, чтобы позаботиться о валидации ввода для компонента. Метод Managed-бина, выполняющий валидацию, должен принимать `jakarta.faces.context.FacesContext`, компонент, данные которого должны быть проверены, и данные, подлежащие проверке, так же, как метод `validate` интерфейса `Validator`. Компонент ссылается на метод Managed-бина с использованием его атрибута `validator`. Провалидированы могут быть только значения компонентов `UIInput` или его дочерних.

Вот пример метода Managed-бина, который проверяет пользовательский ввод, из примера CDI guessnumber-cdi:

JAVA

```
public void validateNumberRange(FacesContext context,
                                UIComponent toValidate,
                                Object value) {
    if (remainingGuesses <= 0) {
        ((UIInput) toValidate).setValid(false);
        FacesMessage message = new FacesMessage("No guesses left!");
        context.addMessage(toValidate.getClientId(context), message);
        return;
    }

    int input = (Integer) value;
    if (input < minimum || input> maximum) {
        ((UIInput) toValidate).setValid(false);

        FacesMessage message = new FacesMessage("Invalid guess");
        context.addMessage(toValidate.getClientId(context), message);
    }
}
```

Метод `validateNumberRange` выполняет две разные валидации.

- Если у пользователя закончились предположения, метод устанавливает свойству `valid` компонента `UIInput` значение `false`. Затем он помещает сообщение в очередь на объект `FacesContext`, связывая сообщение с идентификатором компонента, и возвращает результат.
- Если у пользователя ещё остались предположения, метод получает локальное значение компонента. Если входное значение находится за пределами допустимого диапазона, метод снова устанавливает свойство `valid` компонента `UIInput` в `false`, помещая другое сообщение в очередь объекта `FacesContext` и возвращает результат.

Смотрите Ссылка на метод, который выполняет валидацию для получения информации о том, как тег компонента ссылается на этот метод.

Пишем метод-обработчик изменения значения

Managed-бин, который обрабатывает событие изменения значения, должен использовать публичный метод, который принимает событие изменения значения и возвращает `void`. На этот метод ссылается атрибут `valueChangeListener` компонента. В этом разделе объясняется, как написать метод Managed-бина для замены реализации `jakarta.faces.event.ValueChangeListener`.

Следующий пример тега взят из Регистрация слушателя изменения значения в компоненте, где тег `h:inputText` с `id` из `name` имеет зарегистрированный объект `ValueChangeListener`. Этот объект `ValueChangeListener` обрабатывает событие ввода значения в поле, соответствующее компоненту. Когда пользователь вводит значение, генерируется событие изменения значения и вызывается метод `processValueChange(ValueChangeEvent)` класса `ValueChangeListener`:

XML

```
<h:inputText id="name"
             size="30"
             value="#{cashierBean.name}"
             required="true"
             requiredMessage="#{bundle.ReqCustomerName}">
     <f:valueChangeListener
         type="ee.jakarta.tutorial.dukesbookstore.listeners.NameChanged" />
</h:inputText>
```

Вместо реализации `ValueChangeListener` вы можете написать метод Managed-бина для обработки этого события. Для этого вы перемещаете метод `processValueChange(ValueChangeEvent)` из класса `ValueChangeListener`, называемого `NameChanged`, в Managed-бин.

Вот метод Managed-бина, который обрабатывает событие ввода значения в поле `name` на странице:

JAVA

```
public void processValueChange(ValueChangeEvent event)
        throws AbortProcessingException {
    if (null != event.getNewValue()) {
        FacesContext.getCurrentInstance().getExternalContext().
                getSessionMap().put("name", event.getNewValue());
    }
}
```

Чтобы этот метод обрабатывал `ValueChangeEvent`, сгенерированный компонентом ввода, обратитесь к этому методу из атрибута `valueChangeListener` тега компонента. Смотрите Ссылка на метод-обработчик изменения значения для получения дополнительной информации.

Глава 13. Использование Ajax с Jakarta Faces

В этой главе описывается использование Ajax в веб-приложениях Jakarta Faces. Ajax — это сокращение от Asynchronous JavaScript и XML, группы веб-технологий, которые позволяют создавать динамические и высокочувствительные веб-приложения. Используя Ajax, веб-приложения могут извлекать контент с сервера, не мешая отображению на клиенте. Jakarta Faces в платформе Jakarta EE обеспечивает поддержку Ajax "из коробки".

Обзор Ajax

Ранние веб-приложения создавались в основном как статические веб-страницы. Когда клиент обновляет статическую веб-страницу, вся страница должна перезагрузиться, чтобы отразить обновление. По сути, каждое обновление требует перезагрузки страницы, чтобы отразить это изменение. Повторная перезагрузка страницы может привести к чрезмерной загрузке сети и повлиять на производительность приложения. Такие технологии, как Ajax, были созданы для преодоления этих недостатков.

Ajax относится к JavaScript и XML — технологиям, которые широко используются для создания динамического и асинхронного веб-контента. Хотя Ajax не ограничивается технологиями JavaScript и XML, чаще всего они используются вместе веб-приложениями. Основное внимание в этом руководстве уделяется использованию функций Ajax на основе JavaScript в веб-приложениях Jakarta Faces.

JavaScript — это динамический язык сценариев для веб-приложений. Это позволяет добавлять расширенные функциональные возможности в пользовательские интерфейсы и позволяет веб-страницам асинхронно взаимодействовать с клиентами. JavaScript работает в основном на стороне клиента (в браузере) и тем самым уменьшает загрузку сервера.

Когда функция JavaScript отправляет асинхронный запрос от клиента на сервер, сервер отправляет ответ, который используется для обновления объектной модели документа (DOM) страницы. Этот ответ часто имеет формат XML-документа. Термин Ajax относится к этому взаимодействию между клиентом и сервером.

Ответ сервера не обязан имет формат XML. Он также может иметь и другой формат, например JSON (see Введение в JSON и https://www.json.org/). В этом руководстве не рассматриваются форматы ответов.

Ajax обеспечивает асинхронное и частичное обновление веб-приложений. Такая функциональность позволяет быстро реагировать на веб-страницы, которые отображаются почти в реальном времени. Веб-приложения на основе Ajax могут получать доступ к серверу и обрабатывать информацию, а также извлекать данные, не влияя на отображение текущей веб-страницы на клиенте (в браузере).

Некоторые из преимуществ использования Ajax следующие:

- Валидация данных формы в режиме реального времени, что исключает необходимость отправки формы для верификации
- Расширенная функциональность для веб-страниц, такая как ввод имени пользователя и пароля
- Частичное обновление веб-контента, без полной перезагрузки страницы

Использование Ajax в Jakarta Faces

Функциональность Ajax может быть добавлена в приложение Jakarta Faces одним из следующих способов:

- Добавление необходимого кода JavaScript в приложение
- Использование встроенной библиотеки Ajax

В более ранних выпусках платформы Jakarta EE приложения Jakarta Faces обеспечивали функциональность Ajax, добавляя необходимый JavaScript на веб-страницу. В платформе Jakarta EE стандартная поддержка Ajax обеспечивается встроенной библиотекой ресурсов JavaScript.

Благодаря этой библиотеке JavaScript функции Ajax могут быть включены в стандартные компоненты пользовательского интерфейса Jakarta Faces, такие как кнопки, метки или текстовые поля. Вы также можете загрузить эту библиотеку ресурсов и использовать её методы непосредственно из кода Managed-бина. В следующих разделах руководства описывается использование встроенной библиотеки Ajax.

Кроме того, поскольку модель компонентов Jakarta Faces может быть расширена, кастомные компоненты могут быть созданы с функциональностью Ajax.

Примеры учебника включают Ajax-версию приложения `guessnumber` — `ajaxguessnumber`. Смотрите Приложение ajaxguessnumber для получения дополнительной информации.

Специфичный для Ajax тег `f:ajax` и его атрибуты описаны в следующих разделах.

Использование Ajax с Facelets

Как упоминалось в предыдущем разделе, Jakarta Faces поддерживает Ajax с помощью встроенной библиотеки JavaScript, входящей в состав базовых библиотек Jakarta Faces. Эта встроенная библиотека Ajax может использоваться в веб-приложениях Jakarta Faces одним из следующих способов.

- Используя тег `f:ajax` вместе с другим стандартным компонентом в приложении Facelets. Этот метод добавляет функциональность Ajax к любому компоненту пользовательского интерфейса без дополнительного кодирования и настройки.
- Используя метод JavaScript API `jsf.ajax.request()` непосредственно в приложении Facelets. Этот метод обеспечивает прямой доступ к методам Ajax и позволяет кастомизировать поведение компонентов.
- Используя компонент `<h:commandScript>` для выполнения произвольных серверных методов из представления. Компонент генерирует функцию JavaScript с заданным именем, которая при вызове, в свою очередь, вызывает указанный серверный метод через Ajax.

Использование тега f:ajax

Тег `f:ajax` — это основной тег Jakarta Faces, который обеспечивает функциональность Ajax для любого обычного компонента пользовательского интерфейса при использовании вместе с этим компонентом. В следующем примере поведение Ajax добавляется к компоненту ввода путём включения основного тега `f:ajax`:

```xml
<h:inputText value="#{bean.message}">
    <f:ajax />
</h:inputText>
```

В этом примере, хотя Ajax включён, другие атрибуты тега `f:ajax` не определены. Если событие не определено, выполняется действие по умолчанию для компонента. Для компонента `inputText`, если не указан атрибут `event`, событием по умолчанию является `valueChange`. Таблица 13-1 перечисляет атрибуты тега `f:ajax` и их действия по умолчанию.

Таблица 13-1 Атрибуты тега f:ajax

Название	Тип	Описание

Название	Тип	Описание
disabled	jakarta.el.ValueExpression, который принимает значение Boolean	Значение Boolean, которое идентифицирует статус тега. Значение true указывает, что поведение Ajax не должно выполняться. Значение false указывает, что поведение Ajax должно быть выполнено. По умолчанию false.
event	jakarta.el.ValueExpression, который вычисляется как String	String, определяющий тип события, к которому будет применяться действие Ajax. Если указано, должно быть одним из событий, поддерживаемых компонентом. Если не указано иное, для компонента определяется событие по умолчанию (событие, инициирующее запрос Ajax). Событие по умолчанию — action для компонентов jakarta.faces.component.ActionSource и valueChange для компонентов jakarta.faces.component.EditableValueHolder.
execute	jakarta.el.ValueExpression, который выполняет Object	Collection с идентификаторами компонентов, которые должны быть выполнены на сервере. Если указан литерал, это должен быть разделённый пробелами объект типа String идентификаторов компонентов и/или одно из ключевых слов. Если указано ValueExpression, оно должно ссылаться на свойство, которое возвращает объекты Collection объектов типа String. Если не указано, значением по умолчанию является @this.
immediate	jakarta.el.ValueExpression, который принимает значение Boolean	Значение Boolean, указывающее, нужно ли обрабатывать введённые данные в начале жизненного цикла. Если true, события поведения, сгенерированные из этого поведения, передаются в фазе применения значений запроса. В противном случае события будут выполняться в фазе вызова приложения.
listener	jakarta.el.MethodExpression	Имя метода слушателя, который вызывается при событии jakarta.faces.event.AjaxBehaviorEvent и транслируется слушателю.
onevent	jakarta.el.ValueExpression, который вычисляется как String	Имя функции JavaScript, которая обрабатывает события пользовательского интерфейса.
onerror	jakarta.el.ValueExpression, который вычисляется как String	Имя функции JavaScript, которая обрабатывает ошибки.

Название	Тип	Описание
render	jakarta.el.ValueExpression, который выполняет Object	Collection, идентифицирующий список компонентов, которые будут отображаться на клиенте. Если указан литерал, это должен быть разделённый пробелами объект типа String идентификаторов компонентов и/или одно из ключевых слов. Если указано ValueExpression, оно должно ссылаться на свойство, которое возвращает объекты Collection объектов типа String. Если не указано, по умолчанию используется значение @none

Ключевые слова, перечисленные в таблице 13-2, могут использоваться с атрибутами execute и render тега f:ajax.

Таблица 13-2 Ключевые слова Execute и Render

Ключевое слово	Описание
@all	Все идентификаторы компонентов
@form	Форма, которая включает компонент
@none	Нет идентификаторов компонентов
@this	Элемент, который вызвал запрос

Обратите внимание, что когда вы используете тег f:ajax на странице Facelets, библиотека ресурсов JavaScript загружается неявно. Эта библиотека ресурсов также может быть загружена явно, как описано в Загрузка JavaScript как ресурса.

Отправка запроса Ajax

Чтобы задействовать функциональность Ajax, веб-приложение должно создать запрос Ajax и отправить его на сервер. Затем сервер обрабатывает запрос.

Приложение использует атрибуты тега f:ajax, перечисленные в табл. 13-1, для создания запроса Ajax. В следующих разделах объясняется процесс создания и отправки Ajax-запроса с использованием некоторых из этих атрибутов.

За кулисами метод jsf.ajax.request() библиотеки ресурсов JavaScript собирает данные, предоставленные тегом f:ajax, и отправляет запрос в жизненный цикл Jakarta Faces.

Использование атрибута event

Атрибут event определяет событие, которое запускает действие Ajax. Вот некоторые из возможных значений этого атрибута: click, keyup, mouseover, focus и blur.

Если событие не указано, будет применено событие по умолчанию на основе родительского компонента. Событие по умолчанию — action для компонентов jakarta.faces.component.ActionSource, например commandButton и valueChange для компонентов jakarta.faces.component.EditableValueHolder, например

`inputText`. В следующем примере тег Ajax связан с компонентом кнопки, а событие, запускающее действие Ajax, представляет собой клик мыши:

```xml
<h:commandButton id="submit" value="Submit">
    <f:ajax event="click" />
</h:commandButton>
<h:outputText id="result" value="#{userNumberBean.response}" />
```

Возможно, вы заметили, что перечисленные события очень похожи на события JavaScript. Фактически, это они и есть, разве что не имеют префикса `on`.

Для командной кнопки событие по умолчанию — `click`, поэтому не обязательно указывать `event="click"`, чтобы получить желаемое поведение.

Использование атрибута execute

Атрибут `execute` определяет компонент или компоненты, которые должны быть выполнены на сервере. Компонент идентифицируется его атрибутом `id`. Вы можете указать более одного исполняемого компонента. Если необходимо выполнить более одного компонента, укажите разделённый пробелами список компонентов.

Когда компонент выполняется, он участвует во всех фазах жизненного цикла обработки запросов, кроме фазы отрисовки ответа.

Значением атрибута `execute` также может быть ключевое слово, например `@all`, `@none`, `@this` или `@form`. Значением по умолчанию является `@this`, которое относится к компоненту, внутрь которого вложен тег `f:ajax`.

Следующий код указывает, что компонент `h:inputText` со значением `id`, равным `userNo`, должен выполняться при клике кнопки:

```xml
<h:inputText id="userNo"
             title="Type a number from 0 to 10:"
             value="#{userNumberBean.userNumber}">
    ...
</h:inputText>
<h:commandButton id="submit" value="Submit">
    <f:ajax event="click" execute="userNo" />
</h:commandButton>
```

Использование атрибута immediate

Атрибут `immediate` указывает, должен ли пользовательский ввод обрабатываться в ранних фазах жизненного цикла приложения или позже. Если для атрибута установлено значение `true`, события, сгенерированные из этого компонента, передаются в фазе применения значений запроса. В противном случае события будут выполняться в фазе вызова приложения.

Если он не определён, значением этого атрибута по умолчанию является `false`.

Использование атрибута listener

Атрибут `listener` относится к выражению метода, которое выполняется на стороне сервера в ответ на действие Ajax на клиенте. Слушатель `jakarta.faces.event.AjaxBehaviorListener.processAjaxBehavior` вызывается один раз во время фазы жизненного цикла "Вызов приложения". В следующем коде из примера

`reservation` (см. Пример reservation) атрибут `listener` определяется с помощью тега `f:ajax`, который ссылается на метод из бина:

```xml
<f:ajax event="change" render="total"
        listener="#{reservationBean.calculateTotal}"/>
```

Всякий раз, когда изменяется цена или количество заказанных билетов, метод `calculateTotal` бина `ReservationBean` пересчитывает общую стоимость билетов и отображает её в выходном компоненте с именем `total`.

Мониторинг событий на клиенте

Чтобы отслеживать текущие запросы Ajax, используйте атрибут `onevent` тега `f:ajax`. Значением этого атрибута является имя функции JavaScript. Jakarta Faces вызывает функцию `onevent` на каждом этапе обработки Ajax-запроса: начало, завершение и успех.

При вызове функции JavaScript, назначенной свойству `onevent`, Jakarta Faces передаёт ему объект данных. Объект данных содержит свойства, перечисленные в табл. 13-3.

Таблица 13-3 Свойства объекта данных onevent

Свойство	**Описание**
`responseXML`	Ответ на вызов Ajax в формате XML
`responseText`	Ответ на вызов Ajax в текстовом формате
`responseCode`	Ответ на вызов Ajax в числовом коде
`source`	Источник текущего события Ajax: элемент DOM
`status`	Статус текущего вызова Ajax: `begin`, `complete` или `success`
`type`	Тип вызова Ajax: `event`

Используя свойство `status` объекта данных, вы можете определить текущее состояние запроса Ajax и отслеживать его выполнение. В следующем примере `monitormyajaxevent` — это функция JavaScript, которая отслеживает Ajax-запрос, отправленный событием:

```xml
<f:ajax event="click" render="statusmessage" onevent="monitormyajaxevent"/>
```

Обработка ошибок

Jakarta Faces обрабатывает ошибки Ajax посредством использования атрибута `onerror` тега `f:ajax`. Значением этого атрибута является имя функции JavaScript.

Когда при обработке Ajax-запроса возникает ошибка, Jakarta Faces вызывает определённую функцию JavaScript `onerror` и передаёт ей объект данных. Объект данных содержит все свойства, доступные для атрибута `onevent`, и, кроме того, следующие свойства:

- `description`

- `errorName`
- `errorMessage`

`type` — это `error`. Свойство `status` объекта данных содержит одно из допустимых значений ошибок, перечисленных в табл. 13-4.

Таблица 13-4 Допустимые значения ошибок для свойства статуса объекта данных

Значения	Описание
`emptyResponse`	Нет ответа Ajax от сервера.
`httpError`	Одна из допустимых ошибок HTTP: `request.status==null` или `request.status==undefined` или `request.status<200` или `request.status>=300`.
`malformedXML`	Ответ Ajax не является корректным XML-документом.
`serverError`	Ответ Ajax содержит элемент `error`.

В следующем примере любые ошибки, возникшие при обработке запроса Ajax, обрабатываются функцией JavaScript `handlemyajaxerror`:

```xml
<f:ajax event="click" render="errormessage" onerror="handlemyajaxerror"/>
```

Получение ответа Ajax

После того, как приложение отправляет запрос Ajax, он обрабатывается на стороне сервера и ответ отправляется обратно клиенту. Как было описано ранее, Ajax допускает частичное обновление веб-страниц. Чтобы задействовать такое частичное обновление, Jakarta Faces позволяет частично обрабатывать представление. Обработка ответа определяется атрибутом `render` тега `f:ajax`.

Подобно атрибуту `execute`, атрибут `render` определяет, какие разделы страницы будут обновляться. Значением атрибута `render` может быть одно или несколько значений `id` компонентов, одно из ключевых слов `@this`, `@all`, `@none` или `@form` или выражение EL. В следующем примере атрибут `render` идентифицирует компонент вывода, который будет отображаться при клике на кнопку (событие по умолчанию для командной кнопки):

```xml
<h:commandButton id="submit" value="Submit">
    <f:ajax execute="userNo" render="result" />
</h:commandButton>
<h:outputText id="result" value="#{userNumberBean.response}" />
```

За кулисами ещё раз метод `jsf.ajax.request()` обрабатывает ответ. Он регистрирует Callback-обработчик ответа при создании исходного запроса. Когда ответ отправляется обратно клиенту, вызывается Callback-метод. Этот вызов автоматически обновляет DOM на стороне клиента, чтобы отобразить предоставленный ответ.

Жизненный цикл запроса Ajax

Запрос Ajax отличается от других запросов Jakarta Faces, и его обработка жизненным циклом Jakarta Faces также отличается.

Как описано в разделе Частичная обработка и частичная визуализация, при получении Ajax-запроса состояние, связанное с этим запросом, фиксируется `jakarta.faces.context.PartialViewContext`. Этот объект обеспечивает доступ к информации, например о том, какие компоненты должны быть обработаны и/или перерисованы. Метод `processPartial` в `PartialViewContext` использует эту информацию для выполнения обработки и отрисовки дерева частичных компонентов.

Атрибут `execute` тега `f:ajax` определяет, какие сегменты дерева компонентов на стороне сервера следует обрабатывать. Поскольку компоненты могут быть однозначно идентифицированы в дереве компонентов Jakarta Faces, легко идентифицировать и обработать один компонент, несколько компонентов или целое дерево. Это стало возможным благодаря методу `visitTree` класса `UIComponent`. Затем идентифицированные компоненты проходят через фазы жизненного цикла запроса Jakarta Faces.

Подобно атрибуту `execute`, атрибут `render` определяет, какие сегменты дерева компонентов Jakarta Faces необходимо отображать в фазе отрисовки ответа.

В фазе отрисовки ответа проверяется атрибут `render`. Выявленные компоненты находятся и для них и их дочерних элементов запрашивается отрисовка. Затем компоненты упаковываются и отправляются клиенту как ответ.

Группировка компонентов

В предыдущих разделах описано, как связать отдельный компонент пользовательского интерфейса с функциональностью Ajax. Вы также можете связать Ajax с более чем одним компонентом одновременно, сгруппировав их на странице. В следующем примере показано, как можно сгруппировать несколько компонентов с помощью тега `f:ajax`:

```xml
<f:ajax>
    <h:form>
        <h:inputText id="input1" value="#{user.name}"/>
        <h:commandButton id="Submit"/>
    </h:form>
</f:ajax>
```

В этом примере ни один из компонентов ещё не связан с какими-либо атрибутами Ajax: `event` или `render`. Следовательно, в случае ввода пользователем никаких действий не будет. Вы можете связать вышеуказанные компоненты с атрибутами `event` или `render` следующим образом:

```xml
<f:ajax event="click" render="@all">
    <h:form>
        <h:inputText id="input1" value="#{user.name}"/>
        <h:commandButton id="Submit"/>
    </h:form>
</f:ajax>
```

В обновлённом примере, когда пользователь кликает любой компонент, обновлённые результаты будут отображаться для всех компонентов. Вы можете дополнительно отрегулировать действие Ajax, добавив определённые события для каждого из компонентов, и в этом случае функциональность Ajax станет накопительной. Рассмотрим следующий пример:

XML

```
<f:ajax event="click" render="@all">
    ...
    <h:commandButton id="Submit">
        <f:ajax event="mouseover"/>
    </h:commandButton>
    ...
</f:ajax>
```

Теперь компонент кнопки будет запускать действие Ajax при событии `mouseover`, а также при клике мыши.

Загрузка JavaScript как ресурса

Файл JavaScript в составе Jakarta Faces называется `jsf.js` и доступен в библиотеке `jakarta.faces`. Эта библиотека ресурсов поддерживает функциональность Ajax в приложениях Jakarta Faces.

Если вы используете тег `f:ajax` на странице, ресурс `jsf.js` автоматически доставляется клиенту. Нет необходимости использовать тег `h:outputScript` для указания этого ресурса. Вы можете использовать тег `h:outputScript` для указания других библиотек JavaScript.

Чтобы использовать ресурс JavaScript напрямую с `UIComponent`, вы должны явно загрузить ресурс, как описано в одной из следующих тем:

- Использование JavaScript API в приложении Facelets — использует тег `h:outputScript` непосредственно на странице Facelets
- Использование аннотации @ResourceDependency в классе компонента — использует аннотацию `jakarta.faces.application.ResourceDependency` на классе `UIComponent`

Использование JavaScript API в приложении Facelets

Чтобы использовать API JavaScript непосредственно в веб-приложении, например на странице Facelets:

1. Определите ресурс JavaScript по умолчанию для страницы с помощью тега `h:outputScript`.

   Например, рассмотрим следующий раздел страницы Facelets:

   XML

   ```
   <h:form>
       <h:outputScript name="jsf.js" library="jakarta.faces" target="head"/>
   </h:form>
   ```

   Указание цели как `head` заставляет ресурс сценария отображаться в элементе `head` на странице HTML.

2. Определите компонент, к которому вы хотите присоединить функциональность Ajax.
3. Добавьте функциональность Ajax к компоненту с помощью JavaScript API. Например, рассмотрим следующее:

   XML

   ```
   <h:form>
       <h:outputScript name="jsf.js" library="jakarta.faces" target="head">
       <h:inputText id="inputname" value="#{userBean.name}"/>
       <h:outputText id="outputname" value="#{userBean.name}"/>
       <h:commandButton id="submit" value="Submit"
                        onclick="jsf.ajax.request(this, event,
                                 {execute:'inputname',render:'outputname'});
                                 return false;" />
   </h:form>
   ```

Метод `jsf.ajax.request` принимает до трёх параметров, которые указывают источник, событие и параметры. Параметр source определяет элемент DOM, который запустил Ajax-запрос, обычно `this`. Необязательный параметр события определяет событие DOM, которое вызвало этот запрос. Необязательный параметр options содержит набор пар имя/значение из таблицы 13-5.

Таблица 13-5 Возможные значения параметра Options

Название	**Значение**
`execute`	Разделённый пробелами список идентификаторов или одно из ключевых слов, перечисленных в табл. 13-2. Идентификаторы ссылаются на компоненты, которые будут обрабатываться в фазе выполнения жизненного цикла.
`render`	Разделённый пробелами список идентификаторов или одно из ключевых слов, перечисленных в табл. 13-2. Идентификаторы ссылаются на компоненты, которые будут обработаны в фазе отрисовки жизненного цикла.
`onevent`	`String`, который является именем функции JavaScript, вызываемой при возникновении события.
`onerror`	`String`, который является именем функции JavaScript, вызываемой при возникновении ошибки.
`params`	Объект, который может включать в себя дополнительные параметры для включения в запрос.

Если идентификатор не указан, предполагаемым ключевым словом по умолчанию для атрибута `execute` является `@this`, а для атрибута `render` — `@none`.

Можно также поместить метод JavaScript в файл и включить его в качестве ресурса.

Использование аннотации @ResourceDependency в классе бина

Используйте `jakarta.faces.application.ResourceDependency`, чтобы вызвать принудительную загрузку библиотеки `jsf.js` классом бина.

Чтобы загрузить ресурс Ajax со стороны сервера:

1. Используйте метод `jsf.ajax.request` в классе компонента.

> Этот метод обычно используется при создании кастомного компонента или пользовательского отрисовщика для компонента.

В следующем примере показано, как ресурс загружается в класс компонента:

```java
@ResourceDependency(name="jsf.js" library="jakarta.faces" target="head")
```

Приложение ajaxguessnumber

Чтобы продемонстрировать преимущества использования Ajax, вернитесь к примеру `guessnumber` из главы 8 *Введение в Facelets*. Если вы измените этот пример для использования Ajax, ответ не обязательно будет отображаться на странице `response.xhtml`. Вместо этого выполняется асинхронный вызов серверного

компонента, и ответ отображается на исходной странице путём обработки только компонента ввода, а не путём отправки формы.

Исходный код для этого приложения находится в каталоге *tut-install*/examples/web/jsf/ajaxguessnumber/ .

Исходный код ajaxguessnumber

Изменения в приложении `guessnumber` происходят в двух исходных файлах.

Страница ajaxgreeting.xhtml Facelets

Страница Facelets для `ajaxguessnumber` — `ajaxgreeting.xhtml` — почти такая же, как страница `greeting.xhtml` для приложения `guessnumber` :

XML

```
<h:head>
    <h:outputStylesheet library="css" name="default.css"/>
    <title>Ajax Guess Number Facelets Application</title>
</h:head>
<h:body>
    <h:form id="AjaxGuess">
        <h:graphicImage value="#{resource['images:wave.med.gif']}"
                        alt="Duke waving his hand"/>
        <h2>
            Hi, my name is Duke. I am thinking of a number from
            #{dukesNumberBean.minimum} to #{dukesNumberBean.maximum}.
            Can you guess it?
        </h2>
        <p>
            <h:inputText id="userNo"
                         title="Enter a number from 0 to 10:"
                         value="#{userNumberBean.userNumber}">
                <f:validateLongRange minimum="#{dukesNumberBean.minimum}"
                                     maximum="#{dukesNumberBean.maximum}"/>
            </h:inputText>

            <h:commandButton id="submit" value="Submit">
                <f:ajax execute="userNo" render="outputGroup" />
            </h:commandButton>
        </p>
        <p>
            <h:panelGroup layout="block" id="outputGroup">
                <h:outputText id="result" style="color:blue"
                              value="#{userNumberBean.response}"
                              rendered="#{!facesContext.validationFailed}"/>
                <h:message id="errors1"
                           showSummary="true"
                           showDetail="false"
                           style="color: #d20005;
                           font-family: 'New Century Schoolbook', serif;
                           font-style: oblique;
                           text-decoration: overline"
                           for="userNo"/>
            </h:panelGroup>
        </p>
    </h:form>
</h:body>
```

Самое важное изменение в теге `h:commandButton` . Атрибут `action` удалён из тега, а тег `f:ajax` добавлен.

Тег `f:ajax` указывает, что при клике кнопки будет выполнен компонент `h:inputText` со значением `id` `userNo` . Затем отображаются компоненты в группе панелей `outputGroup` . Если возникает ошибка валидации, Managed-бин не выполняется и сообщение об ошибке валидации отображается на панели

сообщений. В противном случае результат предположения отображается в компоненте `result`.

Вспомогательный бин UserNumberBean

Небольшое изменение также внесено в код `UserNumberBean`, чтобы компонент вывода не отображал никаких сообщений для значения по умолчанию (null) свойства `response`. Вот модифицированный код компонента:

JAVA

```
public String getResponse() {
    if ((userNumber != null)
            && (userNumber.compareTo(dukesNumberBean.getRandomInt()) == 0)) {
        return "Yay! You got it!";
    }
    if (userNumber == null) {
        return null;
    } else {
        return "Sorry, " + userNumber + " is incorrect.";
    }
}
```

Managed-бин CDI DukesNumberBean

Managed-бин EJB `DukesNumberBean` с областью видимости сессии хранит диапазон возможных чисел и случайно выбранное числов из этого диапазона. Он инъецируется в `UserNumberBean` аннотацией CDI `@Inject`, так что значение случайного числа можно сравнить с числом, отправленным пользователем:

JAVA

```
@Inject
DukesNumberBean dukesNumberBean;
```

Вы узнаете больше о CDI в главе 25 *Введение в Инъецирование контекстов и зависимостей Jakarta.*

Запуск ajaxguessnumber

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска приложения `ajaxguessnumber`.

Сборка, упаковка и развёртывание ajaxguessnumber в IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/web/jsf
   ```

4. Выберите каталог `ajaxguessnumber`.
5. Нажмите **Открыть проект**.
6. В **Проекты** кликните правой кнопкой мыши вкладку `ajaxguessnumber` и выберите **Сборка**.

   Эта команда собирает и развёртывает проект.

Сборка, упаковка и развёртывание ajaxguessnumber с помощью Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/web/jsf/ajaxguessnumber/
   ```

3. Введите следующую команду:

```
mvn install
```
SHELL

Эта команда собирает и упаковывает приложение в WAR-файл, `ajaxguessnumber.war`, расположенный в каталоге `target`. Затем она развёртывает приложение.

Запуск ajaxguessnumber

1. В веб-браузере введите следующий URL:

```
http://localhost:8080/ajaxguessnumber
```

2. Введите значение в поле и нажмите «Отправить».

Если значение находится в диапазоне от 0 до 10, в сообщении указывается, является ли предположение правильным или неправильным. Если значение выходит за пределы этого диапазона или если это не число, сообщение об ошибке отображается красным цветом.

Дополнительная информация об Ajax в Jakarta Faces

Для получения дополнительной информации об использовании Ajax в Jakarta Faces см.

- Сайт проекта Jakarta Faces:
  https://jakarta.ee/specifications/faces/3.0/
- API библиотеки JavaScript Jakarta Faces (https://jakarta.ee/specifications/faces/3.0/jsdoc/jsf.ajax.html)

Глава 14. Составные компоненты: дополнительные темы и примеры

В этой главе описываются дополнительные возможности составных компонентов в Jakarta Faces.

Атрибуты составного компонента

Составной компонент — это особый тип шаблона Jakarta Faces, который действует как компонент. Если вы новичок в составных компонентах, ознакомьтесь с Составные компоненты, прежде чем приступить к этой главе.

Вы определяете атрибут составного компонента с помощью тега `composite:attribute`. Таблица 14-1 содержит список часто используемых атрибутов этого тега.

Таблица 14-1. Часто используемые атрибуты составного тега

Атрибут	**Описание**
`name`	Задаёт имя атрибута составного компонента, который будет использоваться на странице. В качестве альтернативы, атрибут `name` может указывать стандартные обработчики событий, такие как `action`, `actionListener` и Managed-бин.
`default`	Определяет значение по умолчанию атрибута составного компонента.
`required`	Указывает, является ли атрибут обязательным для заполнения.
`method-signature`	Указывает дочерний класс `java.lang.Object` в качестве типа атрибута составного компонента. Элемент `method-signature` объявляет, что атрибут составного компонента является выражением метода. Атрибут `type` и атрибут `method-signature` являются взаимоисключающими. Если вы укажете оба, `method-signature` игнорируется. Тип атрибута по умолчанию — `java.lang.Object`. Примечание. Выражения методов похожи на выражения значений, но вместо поддержки динамического извлечения и установки свойств выражения метода поддерживают вызов метода произвольного объекта, передачу указанного набора параметров и возврат результата из вызываемого метода (если есть).
`type`	Задаёт полное имя класса в качестве типа атрибута. Атрибуты `type` и `method-signature` являются взаимоисключающими. Если вы укажете оба, `method-signature` игнорируется. Тип атрибута по умолчанию — `java.lang.Object`.

Следующий фрагмент кода определяет атрибут составного компонента и присваивает ему значение по умолчанию:

```xml
<composite:attribute name="username" default="admin"/>
```

В следующем фрагменте кода используется элемент `method-signature` :

```xml
<composite:attribute name="myaction"
                     method-signature="java.lang.String action()"/>
```

В следующем фрагменте кода используется элемент `type` :

```xml
<composite:attribute name="dateofjoining" type="java.util.Date"/>
```

Вызов Managed-бина

Чтобы включить составной компонент для обработки данных на стороне сервера

1. Вызвать Managed-бин одним из следующих способов:
   - Передайте ссылку Managed-бина на составной компонент.
   - Непосредственно используйте свойства Managed-бина.

     Приложение, описанный в Приложение compositecomponentexample, показывает, как использовать Managed-бин с составным компонентом путём передачи ссылки на Managed-бин в компонент.

Валидация значений составного компонента

Jakarta Faces предоставляет следующие теги для валидацит значений компонентов ввода. Эти теги могут быть использованы с тегами `composite:valueHolder` или `composite:editableValueHolder` .

Таблица 14-2 содержит список часто используемых тегов валидатора. Смотрите Использование стандартных валидаторов для получения подробной информации и полного списка.

Таблица 14-2 Теги валидатора

Название тега	**Описание**
`f:validateBean`	Делегирует валидацию локального значения API Bean Validation.
`f:validateRegex`	Использует атрибут `pattern` для валидацию обёрнутого компонента. Весь шаблон сопоставляется со значением `String` компонента. Если значение соответствует шаблону, оно валидно.
`f:validateRequired`	Обеспечивает наличие значения. Имеет тот же эффект, что и установка атрибуту `required` составного компонента значения `true` .

Пример compositecomponentexample

Приложение `compositecomponentexample` создаёт составной компонент, который принимает имя (или любую другую строку). Компонент взаимодействует с Managed-бином, который вычисляет, является ли сумма букв в имени после их преобразования в числовые значения, простым числом. Компонент отображает сумму значений букв и сообщает, является ли она простым числом или нет.

Приложение `compositecomponentexample` содержит файл составного компонента, страницу использования и Managed-бин.

Исходный код для этого приложения находится в каталоге *`tut-install`*`/examples/web/jsf/compositecomponentexample/` .

Файл составного компонента

Файл составного компонента представляет собой XHTML-файл `/web/resources/ezcomp/PrimePanel.xhtml` . Он имеет раздел `composite:interface` , который объявляет метки для имени и кнопки команд. Он также объявляет Managed-бин, который определяет свойства для имени.

XML

```
<composite:interface>
    <composite:attribute name="namePrompt"
                         default="Name, word, or phrase: "/>
    <composite:attribute name="calcButtonText" default="Calculate"/>
    <composite:attribute name="calcAction"
                         method-signature="java.lang.String action()"/>
    <composite:attribute name="primeBean"/>
    <composite:editableValueHolder name="nameVal" targets="form:name"/>
</composite:interface>
```

Реализация составного компонента принимает введённое значение для свойства `name` Managed-бина. Тег `h:outputStylesheet` определяет таблицу стилей как перемещаемый ресурс. Затем реализация указывает формат вывода, используя свойства Managed-бина, а также формат сообщений об ошибках. Значение суммы отображается только после того, как оно было рассчитано, а отчёт о том, является ли сумма простой или нет, отображается только при валидации введённого значения.

XML

```
<composite:implementation>
    <h:form id="form">
        <h:outputStylesheet library="css" name="default.css"
                            target="head"/>
        <h:panelGrid columns="2" role="presentation">
            <h:outputLabel for="name"
                           value="#{cc.attrs.namePrompt}"/>
            <h:inputText id="name"
                         size="45"
                         value="#{cc.attrs.primeBean.name}"
                         required="true"/>
        </h:panelGrid>
        <p>
            <h:commandButton id="calcButton"
                             value="#{cc.attrs.calcButtonText}"
                             action="#{cc.attrs.calcAction}">
                <f:ajax execute="name" render="outputGroup"/>
            </h:commandButton>
        </p>

       <h:panelGroup id="outputGroup" layout="block">
            <p>
                <h:outputText id="result" style="color:blue"
                              rendered="#{cc.attrs.primeBean.totalSum gt 0}"
                              value="Sum is #{cc.attrs.primeBean.totalSum}" />
            </p>
            <p>
                <h:outputText id="response" style="color:blue"
                              value="#{cc.attrs.primeBean.response}"
                              rendered="#{!facesContext.validationFailed}"/>
                <h:message id="errors1"
                           showSummary="true"
                           showDetail="false"
                           style="color: #d20005;
                           font-family: 'New Century Schoolbook', serif;
                           font-style: oblique;
                           text-decoration: overline"
                           for="name"/>
            </p>
        </h:panelGroup>
    </h:form>
</composite:implementation>
```

Страница использования

Страница использования в этом примере приложения, `web/index.xhtml` , представляет собой XHTML-файл, который вызывает файл составного компонента `PrimePanel.xhtml` вместе с Managed-бином. Это валидирует ввод пользователя.

XML

```
<div id="compositecomponent">
    <ez:PrimePanel primeBean="#{primeBean}" calcAction="#{primeBean.calculate}">
    </ez:PrimePanel>
</div>
```

Managed-бин

Managed-бин `PrimeBean.java` определяет метод с именем `calculate` , который выполняет вычисления для введённой строки и соответственно устанавливает свойства. Сначала бин создаёт массив простых чисел. Он вычисляет сумму букв в строке, где `'a'` равно 1 и `'z'` равно 26, и определяет, можно ли найти значение в массиве простых чисел. Заглавная буква во введённой строке имеет то же значение, что и её строчный эквивалент.

Компонент определяет минимальный и максимальный размер строки `name`, которая обеспечивается ограничением `@Size` Bean Validation. Компонент использует `@Model` — ярлык для `@Named` и `@RequestScoped` — как описано в Шаг 7 из Просмотр веб-модуля hello1 в IDE NetBeans.

JAVA

```
@Model
public class PrimeBean implements Serializable {
    ...
    @Size(min=1, max=45)
    private String name;
    ...

    public String calculate() {
        ...
    }
}
```

Запуск compositecomponentexample

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска примера `compositecomponentexample`.

Сборка, упаковка и развёртывание compositecomponentexample в IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/web/jsf
   ```

4. Выберите каталог `compositecomponentexample`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `complexcomponentexample` и выберите **Сборка**.

   Эта команда собирает и развёртывает приложение.

Сборка, упаковка и развёртывание compositecomponentexample с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/web/jsf/compositecomponentexample/
   ```

3. Введите следующую команду, чтобы собрать и развернуть приложение:

   SHELL

   ```
   mvn install
   ```

Запуск compositecomponentexample

1. В веб-браузере введите следующий URL:

   ```
   http://localhost:8080/compositecomponentexample
   ```

2. На появившейся странице введите строку в поле Имя, слово или фразу, затем нажмите Рассчитать.

Страница сообщает сумму букв и является ли эта сумма простой. Ошибка проверки выдается, если значение не введено или строка содержит более 45 символов.

Глава 15. Создание кастомных компонентов интерфейса пользователя и других кастомных объектов

В этой главе описывается создание кастомных компонентов для приложений, имеющих функции, которые не предоставляются стандартными компонентами Jakarta Faces.

Введение в создание кастомных компонентов

Jakarta Faces предлагает набор стандартных, многократно используемых компонентов, которые позволяют быстро и легко создавать пользовательские интерфейсы для веб-приложений. Эти компоненты в основном связаны отношением один-к-одному с элементами в HTML 4. Но часто для приложения требуется компонент, который имеет дополнительные функциональные возможности или совершенно новый компонент. Jakarta Faces позволяет расширять стандартные компоненты для изменения их функциональности или для создания кастомных компонентов. Возможность расширения активно используется обширной экосистемой сторонних библиотек компонентов, но их изучение выходит за рамки данного учебника. Чтобы больше узнать об этом важном аспекте использования Jakarta Faces, хорошей отправной точкой является поиск в Интернете по запросу «Библиотеки компонентов JSF».

Дополнительно к расширению функциональности стандартных компонентов разработчик компонентов может дать автору страницы возможность изменять внешний вид компонента на странице или изменять поведение слушателя. С другой стороны, разработчику компонента может потребоваться отобразить компонент не на настольный компьютер, а на другой тип клиентского устройства — смартфон или планшет. Благодаря гибкой архитектуре Jakarta Faces разработчик компонентов может отделить определение поведения компонента от его внешнего вида, делегировав отрисовку компонента отдельному отрисовщику. Таким образом, разработчик компонента может определить поведение кастомного компонента один раз, но создать несколько отрисовщиков, каждый из которых определяет свой способ отрисовки компонента для определённого типа клиентского устройства.

`jakarta.faces.component.UIComponent` — это класс Java, который отвечает за представление автономной части пользовательского интерфейса в течение фазы обработки запроса жизненного цикла. Он предназначен для представления значения компонента. За визуальное представление компонента ответствен `jakarta.faces.render.Renderer`. В любом представлении Jakarta Faces может быть несколько объектов одного и того же класса `UIComponent`, точно так же, как может быть несколько объектов любого класса Java в любой программе Java.

Jakarta Faces позволяет создавать кастомные компоненты, расширяя класс `UIComponent` — родительский класс для всех стандартных компонентов пользовательского интерфейса. Кастомный компонент может использоваться везде, где может использоваться обычный компонент. Например, в составном компоненте. `UIComponent` идентифицируется двумя именами: `component-family` определяет общее назначение компонента (например, ввод или вывод) и `component-type` указывает конкретное назначение компонента, такого как поле ввода текста или командная кнопка.

`Renderer` является вспомогательным для `UIComponent` и определяет, как этот конкретный класс `UIComponent` должен отображаться на клиентском устройстве каждого типа. Отрисовщики идентифицируются двумя именами: `render-kit-id` и `renderer-type`. Инструментарий отрисовки — это просто набор, в который помещаются заданные отрисовщики, а `render-kit-id` — идентификатор этой группы. Большинство библиотек компонентов Jakarta Faces предоставляют свой собственный инструментарий отрисовки.

`jakarta.faces.view.facelets.Tag` является вспомогательным для объектов `UIComponent` и `Renderer`, которые позволяет автору страницы включать объект `UIComponent` в представление Jakarta Faces. Тег представляет собой комбинацию `component-type` и `renderer-type`.

Смотрите Комбинирование компонента, отрисовщика и тега для получения информации о взаимодействии компонентов, отрисовщиков и тегов.

В этой главе используется компонент карты изображения из примера Duke's Bookstore, чтобы объяснить, как можно создавать простые кастомные компоненты, кастомные отрисовщики и связанные с ними кастомные теги, а также позаботиться обо всех других деталях, связанных с использованием компонентов и отрисовщиков в приложении. См. главу 61 *Пример Duke's Bookstore* для получения дополнительной информации об этом примере.

В этой главе также описывается, как создавать другие кастомные объекты: конвертеры, слушатели и валидаторы. Также описывается, как связать значения и объекты компонентов с объектами данных и как связать кастомные объекты со свойствами Managed-бина.

Определение того, требуется ли кастомный компонент или отрисовщик

Jakarta Faces поддерживает очень простой набор компонентов и связанных отрисовщиков. Этот раздел поможет решить, подходят ли стандартные компоненты и отрисовщики в для конкретного случая или требуется кастомный компонент или кастомный отрисовщик.

Когда использовать кастомный компонент

Класс компонента определяет состояние и поведение компонента пользовательского интерфейса. Это поведение включает преобразование значения компонента в соответствующую разметку, организацию очередей событий на компонентах, выполнение валидации и другое поведение, связанное с взаимодействием компонента с браузером и жизненным циклом обработки запросов.

Необходимо создавать кастомный компонент в следующих ситуациях.

- Необходимо добавить новое поведение в стандартный компонент, создать событие дополнительного типа (например, уведомить другую часть страницы о том, что что-то изменилось в этом компоненте в результате взаимодействия с пользователем).
- При обработке запроса значения компонента необходимо выполнить действие, отличное от того, что доступно в любом из существующих стандартных компонентов.
- Вы хотите воспользоваться возможностями HTML, предлагаемыми браузером, но ни один из стандартных компонентов Jakarta Faces не использует эту возможность так, как вы этого хотите. Текущая версия не содержит стандартных компонентов для сложных компонентов HTML, таких как фреймы. Однако из-за расширяемости архитектуры компонентов вы можете использовать Jakarta Faces для создания таких компонентов. В примере Duke's Bookstore создаются кастомные компоненты, соответствующие тегам HTML `map` и `area`.
- Вы должны выполнить отрисовку в не-HTML клиент, который требует дополнительных компонентов, не поддерживаемых HTML. В конце концов, стандартный инструментарий отрисовки HTML обеспечит поддержку всех стандартных компонентов HTML. Однако, если вы выполняете отрисовку для другого клиента, например для телефона, может потребоваться создать кастомные компоненты для представления элементов управления, поддерживаемых клиентом. Например, некоторые архитектуры компонентов для беспроводных клиентов включают поддержку тикеров и индикаторов выполнения, которые недоступны в клиенте HTML. В этом случае может также потребоваться кастомный отрисовщик вместе с компонентом, или только кастомный отрисовщик.

Вам не нужно создавать кастомный компонент в следующих случаях.

- Вам нужно объединить компоненты, чтобы создать новый компонент со своим уникальным поведением. В этой ситуации вы можете использовать составной компонент для объединения существующих стандартных компонентов. Для получения дополнительной информации о составных компонентах см. Составные компоненты и главу 14 *Составные компоненты: дополнительные темы и пример*.
- Требуется манипулировать данными компонента или добавить ему специфичные для приложения функции. В этой ситуации следует создать Managed-бин и связать его со стандартным компонентом, а не создавать кастомный компонент. См. Managed-бины в Jakarta Faces для получения дополнительной информации о Managed-бинах.
- Необходимо конвертировать данные компонента в тип, не поддерживаемый его отрисовщиком. См. Использование стандартных конвертеров для получения дополнительной информации о конвертации данных компонента.
- Необходимо выполнить валидацию данных компонента. Стандартные и кастомные валидаторы могут быть добавлены к компоненту с помощью тегов validator на странице. Смотрите Использование стандартных валидаторов и Создание и использование кастомного валидатора для получения дополнительной информации о валидации данных компонента.
- Необходимо зарегистрировать слушатели событий в компонентах. Вы можете либо зарегистрировать слушатели событий в компонентах, используя теги `f:valueChangeListener` и `f:actionListener`, либо указать на метод обработки событий в Managed-бине, используя атрибуты `actionListener` или `valueChangeListener`. Смотрите Реализация слушателя событий и Пишем методы Managed-бинов для получения дополнительной информации.

Когда использовать кастомный отрисовщик

Отрисовщик, который генерирует разметку для отображения компонента на веб-странице, позволяет отделить семантику компонента от его внешнего вида. Сохраняя это разделение, вы можете поддерживать различные виды клиентских устройств используя одну и ту же логику обработки. Вы можете думать об отрисовщике как о «клиентском адаптере». Он производит вывод, подходящий для употребления и отображения клиентом, и принимает ввод от клиента, когда пользователь взаимодействует с этим компонентом.

Если создаётся кастомный компонент, необходимо, среди прочего, убедиться, что класс компонента выполняет следующие операции, которые являются центральными для отрисовки компонента:

- Декодирование: конвертация параметров входящего запроса в локальное значение компонента
- Кодирование: конвертация текущего локального значения компонента в соответствующую разметку, которая представляет его в ответе.

Спецификация Jakarta Faces поддерживает две программные модели для обработки кодирования и декодирования.

- Прямая реализация: класс компонента сам реализует декодирование и кодирование.
- Делегированная реализация: класс компонента делегирует реализацию кодирования и декодирования отдельному отрисовщику.

Передав операции отрисовщику, вы можете связать кастомный компонент с различными отрисовщиками, чтобы можно было отрисовать компонент на разных клиентах. Если вы не планируете отрисовывать определённый компонент на разных клиентах, может быть проще позволить классу компонента

обрабатывать отрисовку. Тем не менее, отдельный отрисовщик позволяет сохранить разделение семантики от внешнего вида. Приложение Duke's Bookstore отделяет отрисовщики от компонентов, хотя оно обслуживает только веб-браузеры с поддержкой HTML 4.

Если нет уверенности, необходима ли гибкость, предоставляемая отдельными отрисовщиками, но есть желание сделать по возможности более простую реализацию, можно использовать обе модели. Класс вашего компонента может включать некоторый код отрисовки по умолчанию, но он может и делегировать отрисовку отдельному отрисовщику.

Комбинирование компонента, отрисовщика и тега

Когда вы создаёте кастомный компонент, вы можете создать и парный ему кастомный отрисовщик. Кастомный тег потребуется также, чтобы связать компонент с отрисовщиком и сослаться на компонент со страницы.

Хотя если нужно написать кастомный компонент и отрисовщик, писать код для кастомного тега (обработчик тега) нет необходимости. Если вы укажете комбинацию компонента и отрисовщика, Facelets автоматически создаст обработчик тега.

В редких случаях вы можете использовать кастомный отрисовщик со стандартным компонентом, а не с кастомным компонентом. Или вы можете использовать кастомный тег без отрисовщика или компонента. В этом разделе приводятся примеры таких ситуаций и суммируется то, что требуется для кастомного компонента, отрисовщика и тега.

Стоит использовать кастомный отрисовщик без кастомного компонента, если вы хотите добавить некоторую валидацию на стороне клиента к стандартному компоненту. Код валидации реализуются на JavaScript на клиентской стороне, а затем он отображается с помощью кастомного отрисовщика. В этой ситуации требуется кастомный тег для отрисовщика, чтобы обработчик тега мог зарегистрировать отрисовщик в стандартном компоненте.

Кастомные компоненты, как и кастомные отрисовщики нуждаются в кастомных тегах, связанных с ними. Тем не менее, вы можете иметь кастомный тег без кастомного отрисовщика или кастомного компонента. Для примера предположим, что нужно создать кастомный валидатор, который требует дополнительных атрибутов в теге валидатора. В этом случае кастомный тег соответствует кастомному валидатору, а не кастомному компоненту или кастомному отрисовщику. В любом случае, нужно связать кастомный тег с серверным объектом.

Таблица 15-1 суммирует, что нужно или можно связать с кастомным компонентом, отрисовщиком или тегом.

Таблица 15-1 Требования к кастомным компонентам, кастомным отрисовщикам и кастомным тегам

Что кастомизируется	**Должен иметь**	**Может иметь**
Кастомный компонент	Кастомный тег	Кастомный или стандартный отрисовщик
Кастомный отрисовщик	Кастомный тег	Кастомный или стандартный компонент

Что кастомизируется	Должен иметь	Может иметь
Кастомный тег Jakarta Faces	Некоторые серверные объекты, такие как компонент, кастомный отрисовщик или кастомный валидатор	Кастомный или стандартный компонент, связанный с кастомным отрисовщиком

Объяснение на примере карты изображения

Duke's Bookstore включает в себя кастомный компонент карты изображения на странице `index.xhtml`. Эта карта изображений отображает выбор из шести названий книг. Когда пользователь кликает одно из названий книг на карте изображения, приложение переходит на страницу, на которой отображается название выбранной книги, а также информация о выбранной книге. Страница позволяет пользователю добавить любую книгу в корзину.

Зачем использовать Jakarta Faces для реализации карты изображения?

Jakarta Faces является идеальным фреймворком, используемым для реализации этого вида карты изображения, поскольку он может выполнять работу, которая должна выполняться на сервере, не требуя создания серверной карты изображения.

В целом, карты изображения на стороне клиента предпочтительнее карт изображений на стороне сервера по нескольким причинам. Одна из причин заключается в том, что карта изображений на стороне клиента позволяет браузеру обеспечивать немедленную обратную связь, когда пользователь наводит указатель мыши на точку доступа. Другая причина в том, что карты изображения на стороне клиента работают лучше, потому что они не требуют обращений к серверу. Однако в некоторых ситуациях вашей карте изображения может потребоваться обращение к серверу для получения данных или для изменения внешнего вида элементов управления без формы — задач, которые не может выполнять карта изображений на стороне клиента.

Поскольку кастомный компонент карты изображения использует Jakarta Faces, он обладает лучшим из двух стилей карт изображений: он может обрабатывать части приложения, которые необходимо выполнить на сервере, и в то же время разрешать выполнение других частей приложения на стороне клиента.

Пояснение к отрисованному HTML

Вот сокращённая версия части формы HTML-страницы, которую приложение должно отобразить:

```HTML
<form id="j_idt13" name="j_idt13" method="post"
        action="/dukesbookstore/index.xhtml" ...>
    ...
    <img id="j_idt13:mapImage"
         src="/dukesbookstore/jakarta.faces.resource/book_all.jpg?ln=images"
         alt="Choose a Book from our Catalog"
         usemap="#bookMap" />
    ...
    <map name="bookMap">
       <area alt="Duke"
          coords="67,23,212,268"
          shape="rect"
          onmouseout="document.forms[0]['j_idt13:mapImage'].src='resources/images/book_all.jpg'"
          onmouseover="document.forms[0]['j_idt13:mapImage'].src='resources/images/book_201.jpg'"
          onclick="document.forms[0]['bookMap_current'].value='Duke'; document.forms[0].submit()"
       />
    ...
       <input type="hidden" name="bookMap_current">
    </map>
    ...
</form>
```

Тег `img` связывает изображение (`book_all.jpg`) с картой изображения, указанной в значении атрибута `usemap` .

Тег `map` определяет карту изображения и содержит набор тегов `area` .

Каждый тег `area` определяет область карты изображения. Атрибуты `onmouseover` , `onmouseout` и `onclick` определяют, какой код JavaScript выполняется при возникновении этих событий. Когда пользователь наводит указатель мыши на область, функция `onmouseover` отображает карту с выделенной областью. Когда пользователь перемещает мышь из области, функция `onmouseout` повторно отображает исходное изображение. Если пользователь кликает на область, функция `onclick` устанавливает значение тега `input` в качестве идентификатора выбранной области и отправляет страницу.

Тег `input` представляет собой скрытый элемент управления, в котором хранится значение выбранной в данный момент области между обменами клиент-сервер, чтобы классы серверных компонентов могли получить это значение.

Серверные объекты получают значение `bookMap_current` и устанавливают локаль в объекте `jakarta.faces.context.FacesContext` в соответствии с выбранной областью.

Пояснение к странице Facelets

Вот сокращённая форма страницы Facelets, которую компонент карты изображения использует для создания HTML-страницы, показанной в предыдущем разделе. Он использует кастомные теги `bookstore:map` и `bookstore:area` для представления кастомных компонентов:

```xml
<h:form>
    ...
        <h:graphicImage id="mapImage"
                        name="book_all.jpg"
                        library="images"
                        alt="#{bundle.ChooseBook}"
                        usemap="#bookMap" />
        <bookstore:map id="bookMap"
                       current="map1"
                       immediate="true"
                       action="bookstore">
            <f:actionListener
                type="dukesbookstore.listeners.MapBookChangeListener" />
            <bookstore:area id="map1" value="#{Book201}"
                            onmouseover="resources/images/book_201.jpg"
                            onmouseout="resources/images/book_all.jpg"
                            targetImage="mapImage" />
            <bookstore:area id="map2" value="#{Book202}"
                            onmouseover="resources/images/book_202.jpg"
                            onmouseout="resources/images/book_all.jpg"
                            targetImage="mapImage"/>
            ...
        </bookstore:map>
    ...
</h:form>
```

Атрибут `alt` тега `h:graphicImage` соответствует локализованной строке `«Выбрать книгу в нашем каталоге»`.

Тег `f:actionListener` в теге `bookstore:map` указывает на класс слушателя для события действия. Метод слушателя `processAction` помещает идентификатор книги для выбранной области карты в сессию. Способ обработки этого события объясняется более подробно в Обработка событий для кастомных компонентов.

Атрибут `action` тега `bookstore:map` определяет результат типа `String` — `«bookstore»` — который по неявному правилу навигации отправляет приложение на страницу `bookstore.xhtml`. Для получения дополнительной информации о навигации см. Настройка правил навигации.

Атрибут `immediate` тега `bookstore:map` имеет значение `true`, что указывает на то, что реализация `jakarta.faces.event.ActionListener` по умолчанию должна выполняться в фазе применения значений запроса жизненного цикла, а не ждать фазы вызова приложения. Поскольку запрос, полученный в результате клика на карту, не требует какой-либо валидации, конвертации данных или обновления серверных объектов, имеет смысл перейти непосредственно к фазе вызова приложения.

Атрибут `current` тега `bookstore:map` установлен в область по умолчанию, которая называется `map1` (книга My Early Years: Growing Up on Star7, by Duke).

Обратите внимание, что теги `bookstore:area` не содержат никаких данных JavaScript, координат или формы, отображаемых на странице HTML. JavaScript генерируется классом `dukesbookstore.renderers.AreaRenderer`. Значения атрибутов `onmouseover` и `onmouseout` указывают изображение, которое будет загружено при возникновении этих событий. Как генерируется JavaScript, объясняется подробнее в Выполнение кодирования.

Данные о координатах, форме и альтернативном тексте получают через атрибут `value`, значение которого относится к атрибуту в области видимости приложения. Значением этого атрибута является бин, в котором хранятся данные `coords`, `shape` и `alt`. Как эти бины хранятся в области приложения, объясняется более подробно в следующем разделе.

Конфигурирование модели данных

В приложении Jakarta Faces такие данные, как координаты кликабельных точек карты изображения, извлекаются из атрибута `value` через бин. Тем не менее, форма и координаты кликабельных точек должны быть определены вместе, потому что координаты интерпретируются по-разному в зависимости от формы точек. Поскольку значение компонента может быть связано только с одним свойством, атрибут `value` не может ссылаться как на форму, так и на координаты.

Чтобы решить эту проблему, приложение инкапсулирует всю эту информацию в наборе объектов `ImageArea`. Эти объекты инициализируются в области видимости приложения средством создания Managed-бинов (см. Использование элемента Managed-бина). Вот часть объявления Managed-бина для компонента `ImageArea`, соответствующего кликабельной точке в Южной Америке:

XML

```
<managed-bean eager="true">
    ...
    <managed-bean-name>Book201</managed-bean-name>
    <managed-bean-class>
        ee.jakarta.tutorial.dukesbookstore.model.ImageArea
    </managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
    <managed-property>
        ...
        <property-name>shape</property-name>
        <value>rect</value>
    </managed-property>
    <managed-property>
        ...
        <property-name>alt</property-name>
        <value>Duke</value>
    </managed-property>
    <managed-property>
        ...
        <property-name>coords</property-name>
        <value>67,23,212,268</value>
    </managed-property>
</managed-bean>
```

Для получения дополнительной информации об инициализации Managed-бинов с помощью средства создания Managed-бина см. раздел Файл конфигурации приложения.

Атрибуты `value` тегов `bookstore:area` ссылаются на бины в области видимости приложения, как показано в этом теге `bookstore:area` из `index.xhtml`:

XML

```
<bookstore:area id="map1" value="#{Book201}"
                onmouseover="resources/images/book_201.jpg"
                onmouseout="resources/images/book_all.jpg"
                targetImage="mapImage" />
```

Чтобы ссылаться на значения бина объекта модели `ImageArea` из класса компонента, вы реализуете метод `getValue` в классе компонента. Этот метод вызывает `super.getValue`. Суперкласс *`tut-install`*`/examples/case-studies/dukes-bookstore/src/java/dukesbookstore/components/AreaComponent.java`, `UIOutput`, имеет метод `getValue`, который выполняет поиск объекта `ImageArea`, связанного с `AreaComponent`. Класс `AreaRenderer`, который должен отображать значения `alt`, `shape` и `coords` из `ImageArea` вызывает метод `getValue` у `AreaComponent` для получения объекта `ImageArea`.

JAVA

```
ImageArea iarea = (ImageArea) area.getValue();
```

`ImageArea` — это простой компонент, поэтому вы можете получить доступ к форме, координатам и альтернативным текстовым значениям, вызвав соответствующие методы доступа `ImageArea`. Создание класса отрисовщика объясняет, как это сделать в классе `AreaRenderer`.

Обзор классов карты изображения

Таблица 15-2 суммирует все классы, необходимые для реализации компонента карты изображения.

Таблица 15-2 Классы карты изображения

Класс	Функция
`AreaSelectedEvent`	`jakarta.faces.event.ActionEvent` указывает, что был выбран `AreaComponent` из `MapComponent`.
`AreaComponent`	Класс, определяющий `AreaComponent`, который соответствует кастомному тегу `bookstore:area`.
`MapComponent`	Класс, определяющий `MapComponent`, который соответствует кастомному тегу `bookstore:map`.
`AreaRenderer`	`jakarta.faces.render.Renderer` выполняет делегированную отрисовку для `AreaComponent`.
`ImageArea`	Бин, который хранит форму и координаты кликабельных точек.
`MapBookChangeListener`	Слушатель действия для `MapComponent`.

Исходный каталог Duke's Bookstore, называемый bookstore-dir — *`tut-install`*`/examples/case-studies/dukes-bookstore/src/java/dukesbookstore/`. Классы событий и слушателей расположены в каталоге *`bookstore-dir`*`/listeners/`. Классы компонентов расположены в каталоге *`bookstore-dir`*`/components/`. Классы отрисовки расположены в каталоге *`bookstore-dir`*`/renderers/`. `ImageArea` находится в *`bookstore-dir`*`/model/`.

Шаги для создания кастомного компонента

При разработке кастомного компонента вы можете применить следующие шаги.

1. Создайте класс компонента, который выполняет следующие действия:
   a. Переопределите метод `getFamily` для возврата семейства компонентов, которое используется для поиска отрисовщиков, которые могут отобразить компонент
   b. Добавьте код отрисовщика или делегируйте отрисовку другому отрисовщику (объяснено в Шаге 2)
   c. Добавьте возможность атрибутам компонента принимать выражения
   d. Поместите событие в компонент, если компонент генерирует события

e. Реализуйте сохранение и восстановление состояния компонента

2. Делегируйте отрисовку отрисовщику, если ваш компонент не занимается отрисовкой сам. Для этого:
   a. Создайте кастомный отрисовщик, наследуясь от `jakarta.faces.render.Renderer`.
   b. Зарегистрируйте отрисовщик в инструментарии отрисовки.
3. Зарегистрируйте компонент.
4. Создайте обработчик событий, если ваш компонент генерирует события.
5. Создайте дескриптор библиотеки тегов (TLD), который определяет кастомный тег.

Смотрите Регистрация кастомного компонента и Регистрация кастомного отрисовщика с помощью инструментария отрисовки для получения информации о регистрации кастомного компонента и отрисовщика. В разделе Использование кастомного компонента обсуждается, как использовать кастомный компонент на странице Jakarta Faces.

Создание классов кастомного компонента

Как объясняется в Когда использовать кастомный компонент, класс компонента определяет состояние и поведение компонента пользовательского интерфейса. Информация о состоянии включает тип компонента, идентификатор и локальное значение. Поведение, определённое классом компонента, включает следующее:

- Декодирование (преобразование параметра запроса в локальное значение компонента)
- Кодирование (преобразование локального значения в соответствующую разметку)
- Сохранение состояния компонента
- Обновление значения компонента локальным значением
- Обработка валидации по локальному значению
- Очередь событий

Класс `jakarta.faces.component.UIComponentBase` определяет поведение класса компонента по умолчанию. Все классы, представляющие стандартные компоненты, являются дочерними для `UIComponentBase`. Эти классы переопределяют поведение по умолчанию, как будет делать и класс вашего компонента.

Класс вашего кастомного компонента должен расширять либо `UIComponentBase` непосредственно, либо расширять класс, представляющий один из стандартных компонентов. Эти классы расположены в каталоге `jakarta.faces.component` и их имена начинаются с `UI`.

Если ваш кастомный компонент выполняет те же функции, что и какой-либо стандартный компонент, предпочтительно расширять этот стандартный компонент, а не `UIComponentBase`. Для примера предположим, что вы хотите создать редактируемый компонент меню. Имеет смысл расширять `UISelectOne`, а не `UIComponentBase`, потому что в этом случае вы можете повторно использовать поведение, уже определённое в `UISelectOne`. Единственная новая функциональность, которую нужно определить, — это сделать меню редактируемым.

Независимо от того, решите ли вы наследовать свой компонент от `UIComponentBase` или стандартного компонента, вы также можете захотеть, чтобы ваш компонент реализовал один или несколько из поведенческих интерфейсов, определённых в пакете `jakarta.faces.component`:

- `ActionSource`: указывает, что компонент может генерировать `jakarta.faces.event.ActionEvent`

- `ActionSource2` : расширяет `ActionSource` и позволяет свойствам компонентов, ссылающимся на методы, которые обрабатывают события действия, использовать выражения метода, как определено в EL
- `EditableValueHolder` : расширяет `ValueHolder` и указывает дополнительные функции для редактируемых компонентов, такие как валидация и генерация событий изменения значения
- `NamingContainer` : требует, чтобы каждый компонент, имеющий родителем этот компоненте, имел уникальный идентификатор
- `StateHolder` : обозначает, что компонент имеет состояние, которое должно быть сохранено между запросами
- `ValueHolder` : указывает, что компонент поддерживает локальное значение, а также возможность доступа к данным слоя модели

Если ваш компонент расширяет `UIComponentBase` , он автоматически реализует только `StateHolder` . Поскольку все компоненты прямо или косвенно расширяют `UIComponentBase` , все они реализуют `StateHolder` . Любой компонент, который реализует `StateHolder` , также реализует интерфейс `StateHelper` , который расширяет `StateHolder` и определяет `Map` -подобный контракт, который упрощает для компонентов сохранение и восстановление состояния частичного просмотра.

Если ваш компонент расширяет один из других стандартных компонентов, он может также реализовать другие поведенческие интерфейсы в дополнение к `StateHolder` . Если ваш компонент расширяет `UICommand` , он автоматически реализует `ActionSource2` . Если ваш компонент расширяет `UIOutput` или один из классов компонентов, расширяющих `UIOutput` , он автоматически реализует `ValueHolder` . Если ваш компонент расширяет `UIInput` , он автоматически реализует `EditableValueHolder` и `ValueHolder` . См. документацию API Jakarta Faces, чтобы узнать, что реализуют другие классы компонентов.

Вы также можете явно указать поведенческий интерфейс и реализовать его, если он его ещё не имеет при расширении стандартного компонента. Например, если у вас есть компонент, который расширяет `UIInput` , и вы хотите, чтобы он вызывал события действия, то должны явно указать `ActionSource2` и реализовать его, потому что `UIInput` компонент не реализует этот интерфейс автоматически.

Пример карты изображения Duke's Bookstore состоит из двух классов компонентов: `AreaComponent` и `MapComponent` . Класс `MapComponent` расширяет `UICommand` и поэтому реализует `ActionSource2` , что означает, что он может инициировать события действия, когда пользователь кликает на карту. Класс `AreaComponent` расширяет стандартный компонент `UIOutput` . Аннотация `@FacesComponent` регистрирует компоненты с Jakarta Faces:

```java
@FacesComponent("DemoMap")
public class MapComponent extends UICommand {...}

@FacesComponent("DemoArea")
public class AreaComponent extends UIOutput {...}
```

Класс `MapComponent` представляет компонент, соответствующий тегу `bookstore:map` :

```xml
<bookstore:map id="bookMap"
               current="map1"
               immediate="true"
               action="bookstore">
    ...
</bookstore:map>
```

Класс `AreaComponent` представляет компонент, соответствующий тегу `bookstore:area` :

XML

```
<bookstore:area id="map1" value="#{Book201}"
                onmouseover="resources/images/book_201.jpg"
                onmouseout="resources/images/book_all.jpg"
                targetImage="mapImage"/>
```

Объект `MapComponent` содержит один или несколько дочерних обхектов `AreaComponent`. Его поведение состоит из следующих действий:

- Получение значения текущей выбранной области
- Определение свойств, соответствующих значениям компонента
- Генерация события, когда пользователь кликает на карту изображения
- Очередь на событие
- Сохранение своего состояния
- Отрисовка тегов HTML `map` и `input`

`MapComponent` делегирует отображение HTML-тегов `map` и `input` классу `MapRenderer`.

`AreaComponent` связан с бином, который хранит форму и координаты области карты изображения. Вы увидите, как все эти данные доступны через выражение значения в Создание класса отрисовщика. Поведение `AreaComponent` состоит из:

- Получение данных формы и координат из компонента
- Установка `id` этого компонента значением скрытого тега
- Отрисовка тега `area`, включая JavaScript для функций `onmouseover`, `onmouseout` и `onclick`

Хотя эти задачи фактически выполняются `AreaRenderer`, `AreaComponent` должен делегировать задачи `AreaRenderer`. Смотрите Делегирование отрисовки отрисовщику для получения дополнительной информации.

В оставшейся части этого раздела описываются задачи, которые выполняет `MapComponent`, а также кодирование и декодирование, которые он делегирует `MapRenderer`. Обработка событий для кастомных компонентов подробно описывает, как `MapComponent` обрабатывает события.

Указание семейства компонентов

Если ваш кастомный класс компонентов делегирует отрисовку, ему необходимо переопределить метод `getFamily` класса `UIComponent`, чтобы вернуть идентификатор семейства компонентов, который используется для ссылки на компонент или набор компонентов, которые могут быть предоставлены отрисовщиком или набором отрисовщиков. Семейство компонентов используется вместе с типом отрисовщика для поиска отрисовщиков, которые могут отобразить компонент:

JAVA

```
public String getFamily() {
    return ("Map");
}
```

Идентификатор семейства компонентов `Map` должен совпадать с тем, который определён элементами `component-family`, включёнными в конфигурации компонента и отрисовщика в файле конфигурации приложения. Регистрация кастомного отрисовщика с помощью инструментария отрисовки объясняет, как определить семейство компонентов в конфигурации отрисовщика. Регистрация кастомного компонента объясняет, как определить семейство компонентов в конфигурации компонента.

Выполнение кодирования

В фазе отрисовки ответа Jakarta Faces обрабатывает методы кодирования всех компонентов и связанных с ними отрисовщиков представления. Методы кодирования преобразуют текущее локальное значение компонента в соответствующую разметку, которая представляет его в ответе.

Класс `UIComponentBase` определяет набор методов для отрисовки разметки: `encodeBegin`, `encodeChildren` и `encodeEnd`. Если у компонента есть дочерние компоненты, может потребоваться использовать более одного из этих методов для отрисовки компонента. В противном случае вся отрисовка должна выполняться в `encodeEnd`. Кроме того, вы можете использовать метод `encodeALL`, который охватывает все методы.

Поскольку `MapComponent` является родительским компонентом `AreaComponent`, теги `area` должны отображаться между начальным и конечным тегами `map`. Для этого класс `MapRenderer` отображает начальный тег `map` в `encodeBegin`, а остальную часть тега `map` в `encodeEnd`.

Jakarta Faces автоматически вызывает метод отрисовщика `encodeEnd` у `AreaComponent` после того, как он вызвал метод `encodeBegin` у `MapRenderer` и прежде чем он вызовет метод `encodeEnd` у `MapRenderer`. Если компонент должен выполнить отрисовку своих дочерних элементов, он делает это в методе `encodeChildren`.

Вот методы `encodeBegin` и `encodeEnd` для `MapRenderer`:

JAVA

```
@Override
public void encodeBegin(FacesContext context, UIComponent component)
        throws IOException {
    if ((context == null)|| (component == null)) {
        throw new NullPointerException();
    }
    MapComponent map = (MapComponent) component;
    ResponseWriter writer = context.getResponseWriter();
    writer.startElement("map", map);
    writer.writeAttribute("name", map.getId(), "id");
}

@Override
public void encodeEnd(FacesContext context, UIComponent component)
        throws IOException {
    if ((context == null) || (component == null)){
        throw new NullPointerException();
    }
    MapComponent map = (MapComponent) component;
    ResponseWriter writer = context.getResponseWriter();
    writer.startElement("input", map);
    writer.writeAttribute("type", "hidden", null);
    writer.writeAttribute("name", getName(context,map), "clientId");
    writer.endElement("input");
    writer.endElement("map");
}
```

Обратите внимание, что `encodeBegin` отображает только начальный тег `map`. Метод `encodeEnd` визуализирует тег `input` и конечный тег `map`.

Методы кодирования принимают в аргументах `UIComponent` и `jakarta.faces.context.FacesContext`. Объект `FacesContext` содержит всю информацию, связанную с текущим запросом. Аргумент `UIComponent` — это компонент, который необходимо отобразить.

Остальная часть метода отрисовывает разметку для объекта `jakarta.faces.context.ResponseWriter`, который записывает разметку в ответ. В основном это включает передачу имён тегов HTML и имён атрибутов в объект `ResponseWriter` в виде строк, извлечение значений атрибутов компонента и передачу этих

значений в объект `ResponseWriter`.

Метод `startElement` принимает `String` (имя тега) и компонент, которому соответствует тег (в данном случае, `map`). (Передача этой информации в объект `ResponseWriter` помогает среде разработки понять, какие части сгенерированной разметки связаны с какими компонентами.)

После вызова `startElement` вы можете вызвать `writeAttribute`, чтобы отобразить атрибуты тега. Метод `writeAttribute` принимает имя атрибута, его значение и имя свойства или атрибута содержащего его компонента, соответствующего этому атрибуту. Последний параметр может быть нулевым, и он не будет отображаться.

Значение атрибута `name` тега `map` извлекается с помощью метода `getId` класса `UIComponent`, который возвращает уникальный идентификатор компонента. Значение атрибута `name` тега `input` извлекается с использованием метода `getName(FacesContext, UIComponent)` из `MapRenderer`.

Если вы хотите, чтобы ваш компонент выполнял сам свою собственную отрисовку, но при наличии отрисовщика делегировал её ему, добавьте следующие строки в метод кодирования, чтобы проверить, существует ли отрисовщик, связанный с этим компонентом:

JAVA

```
if (getRendererType() != null) {
    super.encodeEnd(context);
    return;
}
```

Если доступен отрисовщик, этот метод вызывает метод `encodeEnd` родительского класса, который выполняет поиск отрисовщика. Класс `MapComponent` делегирует всю отрисовку `MapRenderer`, поэтому ему не нужно проверять наличие доступных отрисовщиков.

В некоторых кастомных классах компонентов, расширяющих стандартные компоненты, может потребоваться реализовать другие методы в дополнение к `encodeEnd`. Например, если нужно получить значение компонента из параметров запроса, вы также должны реализовать метод `decode`.

Выполнение декодирования

В фазе применения параметров запроса Jakarta Faces обрабатывает методы `decode` всех компонентов в дереве. Метод `decode` извлекает локальное значение компонента из параметров входящего запроса и использует реализацию `jakarta.faces.convert.Converter` для преобразования значения в тип, приемлемый для компонента.

Кастомный класс компонента или его отрисовщик должны реализовывать метод `decode` только в том случае, если ему необходимо получить локальное значение или поставить события в очередь. Компонент ставит событие в очередь, вызывая `queueEvent`.

Вот метод `decode` для `MapRenderer`:

JAVA

```
@Override
public void decode(FacesContext context, UIComponent component) {
    if ((context == null) || (component == null)) {
        throw new NullPointerException();
    }
    MapComponent map = (MapComponent) component;
    String key = getName(context, map);
    String value = (String) context.getExternalContext().
            getRequestParameterMap().get(key);
    if (value != null)
        map.setCurrent(value);
    }
}
```

Метод `decode` сначала получает имя скрытого поля `input`, вызывая `getName(FacesContext, UIComponent)`. Затем он использует это имя как ключ в отображении (Map) параметров запроса, чтобы получить текущее значение поля `input`. Это значение представляет текущую выбранную область. Наконец, он устанавливает значение атрибута `current` класса `MapComponent` в значение поля `input`.

Добавление компоненту возможности принимать выражения

Почти все атрибуты стандартных тегов Jakarta Faces могут принимать выражения, будь то выражения значений или выражения методов. Рекомендуется также использовать атрибуты компонента для принятия выражений, поскольку это даёт гораздо большую гибкость при написании страниц Facelets.

Чтобы атрибуты могли принимать выражения, класс компонента должен реализовывать get- и set- методы для свойств компонента. Эти методы могут использовать средства, предлагаемые интерфейсом `StateHelper`, для хранения и извлечения не только значений для этих свойств, но и состояния компонентов в нескольких запросах.

Поскольку `MapComponent` расширяет `UICommand`, класс `UICommand` уже выполняет работу по получению `ValueExpression` и `MethodExpression` объекты, связанные с каждым из поддерживаемых им атрибутов. Точно так же класс `UIOutput`, который расширяет `AreaComponent`, уже получает объекты `ValueExpression` для своих поддерживаемых атрибутов. Для обоих компонентов get- и set-методы сохраняют и извлекают значения ключей и состояния для атрибутов, как показано в этом фрагменте кода из `AreaComponent`:

JAVA

```
enum PropertyKeys {
    alt, coords, shape, targetImage;
}
public String getAlt() {
    return (String) getStateHelper().eval(PropertyKeys.alt, null);
}
public void setAlt(String alt) {
    getStateHelper().put(PropertyKeys.alt, alt);
}
...
```

Однако если есть кастомный класс компонентов, расширяющий `UIComponentBase`, нужно реализовать методы, получающие объекты `ValueExpression` и `MethodExpression`, связанные с теми атрибутами, которые могут принимать выражения. Например, можно включить метод, который получает объект `ValueExpression` для атрибута `immediate`:

JAVA

```
public boolean isImmediate() {
    if (this.immediateSet) {
        return (this.immediate);
    }
    ValueExpression ve = getValueExpression("immediate");
    if (ve != null) {
        Boolean value = (Boolean) ve.getValue(
            getFacesContext().getELContext());
        return (value.booleanValue());
    } else {
        return (this.immediate);
    }
}
```

Свойства, соответствующие атрибутам компонента, которые принимают выражения методов, должны принимать и возвращать объект `MethodExpression`. Например, если `MapComponent` расширяет `UIComponentBase` вместо `UICommand`, ему потребуется предоставить свойство `action`, которое возвращает и принимает объект `MethodExpression`:

JAVA

```
public MethodExpression getAction() {
    return (this.action);
}
public void setAction(MethodExpression action) {
    this.action = action;
}
```

Сохранение и восстановление состояния

Как описано в Добавление компоненту возможности принимать выражения, использование интерфейса `StateHelper` позволяет сохранять состояние компонента одновременно с тем, как вы устанавливаете и извлекаете значения свойств. Реализация `StateHelper` позволяет частично сохранить состояние. Она сохраняет только изменения состояния с момента первоначального запроса, а не всё состояние, поскольку полное состояние можно восстановить в фазе восстановления представления.

Классы компонентов, которые реализуют `StateHolder`, могут предпочесть реализовать методы `saveState(FacesContext)` и `restoreState(FacesContext, Object)`, чтобы помочь Jakarta Faces и восстановить состояние компонентов по нескольким запросам.

Чтобы сохранить набор значений, вы можете реализовать метод `saveState(FacesContext)`. Этот метод вызывается в фазе отрисовки ответа, во время которой состояние ответа сохраняется для обработки последующих запросов. Вот гипотетический метод из `MapComponent`, который имеет только один атрибут `current`:

JAVA

```
@Override
public Object saveState(FacesContext context) {
    Object values[] = new Object[2];
    values[0] = super.saveState(context);
    values[1] = current;
    return (values);
}
```

Этот метод инициализирует массив, который будет содержать сохранённое состояние. Затем сохраняется всё состояние, связанное с компонентом.

Компонент, который реализует `StateHolder`, может также предоставить реализацию для `restoreState(FacesContext, Object)`, которая восстанавливает состояние компонента до того, которое было сохранено с помощью метода `saveState(FacesContext)`. Метод `restoreState(FacesContext, Object)` вызывается в фазе восстановления представления, во время которой Jakarta Faces проверяет, было ли сохранено какое-либо состояние в последней фазе отрисовки ответа, и должно ли оно быть восстановлено при подготовке к следующему повторному отображению.

Вот гипотетический метод `restoreState(FacesContext, Object)` из `MapComponent`:

```java
public void restoreState(FacesContext context, Object state) {
    Object values[] = (Object[]) state;
    super.restoreState(context, values[0]);
    current = (String) values[1];
}
```

Этот метод принимает объекты `FacesContext` и `Object`. Второй представляет массив, содержащий состояние компонента. Этот метод устанавливает свойства компонента в значения, сохранённые в массиве `Object`.

Независимо от того, реализуете ли вы эти методы в компоненте, вы можете использовать `jakarta.faces.STATE_SAVING_METHOD` чтобы указать в дескрипторе развёртывания, где хотите сохранять состояние: `client` или `server`. Если состояние сохраняется на клиенте, состояние всего представления отображается в скрытом поле на странице. По умолчанию состояние сохраняется на сервере.

Веб-приложения в примере Duke's Forest сохраняют состояние представления на клиенте.

Сохранение состояния на клиенте требует большую пропускную способность сети, а также больше клиентских ресурсов, тогда как сохранение на сервере требует больше серверных ресурсов. Вы также можете сохранять состояние на клиенте в случае, если ожидаете, что ваши пользователи запретят cookie.

Делегирование отрисовки отрисовщику

И `MapComponent`, и `AreaComponent` делегируют всю отрисовку отдельному отрисовщику. В разделе Выполнение кодирования объясняется, как `MapRenderer` выполняет кодирование для `MapComponent`. В этом разделе подробно объясняется процесс делегирования отрисовки отрисовщику с использованием `AreaRenderer`, который выполняет отрисовку для `AreaComponent`.

Чтобы делегировать отрисовку, нужно выполнить описанные в следующих разделах задачи:

- Создание класса отрисовщика
- Определение типа отрисовщика

Создание класса отрисовщика

При делегировании отрисовки отрисовщику вы можете делегировать всё кодирование и декодирование отрисовщику, или сделать часть этого в классе компонента. Класс `AreaComponent` делегирует кодирование классу `AreaRenderer`.

Класс отрисовщика начинается с аннотации `@FacesRenderer`:

```java
@FacesRenderer(componentFamily = "Area", rendererType = "DemoArea")
public class AreaRenderer extends Renderer { }
```

Аннотация `@FacesRenderer` регистрирует класс отрисовщика с Jakarta Faces. Аннотация определяет семейство компонентов и тип отрисовщика.

Чтобы выполнить отрисовку для `AreaComponent`, `AreaRenderer` должен реализовать метод `encodeEnd`. Метод `encodeEnd` в `AreaRenderer` извлекает форму, координаты и альтернативные текстовые значения, хранящиеся в бине `ImageArea`, который связан с `AreaComponent`. Предположим, что тег `area`, отображаемый в данный момент, имеет значение атрибута `value` равное `"book203"`. Следующая строка из `encodeEnd` получает значение атрибута `"book203"` из объекта `FacesContext`:

```java
ImageArea ia = (ImageArea)area.getValue();
```

Значением атрибута является объект компонента `ImageArea`, который содержит значения `shape`, `coords` и `alt`, связанные с `book203` `AreaComponent`. Конфигурирование модели данных описывает, как приложение хранит эти значения.

После извлечения объекта `ImageArea` метод отображает значения для `shape`, `coords` и `alt`, просто вызывая связанные методы доступа и передавая возвращённые значения в объект `ResponseWriter`, как показано этими строками кода, которые записывают форму и координаты:

```java
writer.startElement("area", area);
writer.writeAttribute("alt", iarea.getAlt(), "alt");
writer.writeAttribute("coords", iarea.getCoords(), "coords");
writer.writeAttribute("shape", iarea.getShape(), "shape");
```

Метод `encodeEnd` также отображает JavaScript для атрибутов `onmouseout`, `onmouseover` и `onclick`. Страница Facelets должна содержать только путь к изображениям, которые должны быть загружены во время действия `onmouseover` или `onmouseout`:

```xml
<bookstore:area id="map3" value="#{Book203}"
                onmouseover="resources/images/book_203.jpg"
                onmouseout="resources/images/book_all.jpg"
                targetImage="mapImage"/>
```

Класс `AreaRenderer` заботится о генерации JavaScript для этих действий, как показано в следующем коде из `encodeEnd`. JavaScript, который `AreaRenderer` генерирует для действия `onclick`, устанавливает значение скрытого поля равным значению идентификатора компонента текущей области и отправляет страницу.

JAVA

```
sb = new StringBuffer("document.forms[0]['").append(targetImageId).
        append("'].src='");
sb.append(
        getURI(context,
        (String) area.getAttributes().get("onmouseout")));
sb.append("'");
writer.writeAttribute("onmouseout", sb.toString(), "onmouseout");
sb = new StringBuffer("document.forms[0]['").append(targetImageId).
        append("'].src='");
sb.append(
        getURI(context,
        (String) area.getAttributes().get("onmouseover")));
sb.append("'");
writer.writeAttribute("onmouseover", sb.toString(), "onmouseover");
sb = new StringBuffer("document.forms[0]['");
sb.append(getName(context, area));
sb.append("'].value='");
sb.append(iarea.getAlt());
sb.append("'; document.forms[0].submit()");
writer.writeAttribute("onclick", sb.toString(), "value");
writer.endElement("area");
```

Отправляя страницу, этот код заставляет жизненный цикл Jakarta Faces вернуться к фазе восстановления представления. В этой фазе сохраняется вся информация о состоянии, включая значение скрытого поля, так что создаётся новое дерево компонентов запроса. Это значение извлекается методом `decode` класса `MapComponent`. Этот метод декодирования вызывается Jakarta Faces в фазе применения значений запроса, которая следует за фазой восстановления представления.

В дополнение к методу `encodeEnd`, `AreaRenderer` содержит пустой конструктор. Он используется для создания объекта `AreaRenderer` для добавления его в инструментарий отрисовки.

Аннотация `@FacesRenderer` регистрирует класс отрисовщика с Jakarta Faces. Аннотация определяет семейство компонентов и тип отрисовщика.

Определение типа отрисовщика

Зарегистрируйте отрисовщик с помощью инструментария отрисовки, используя аннотацию `@FacesRenderer` (или используя файл конфигурации приложения, как описано в Регистрация кастомного отрисовщика с помощью инструментария отрисовки). В фазе отрисовки ответа Jakarta Faces вызывает метод `getRendererType` обработчика тега компонента, чтобы определить, какой отрисовщик вызывать, если он есть.

Вы определяете тип, связанный с отрисовщиком, в элементе `rendererType` в аннотации `@FacesRenderer` для `AreaRenderer`, а также в элементе `renderer-type` дескриптора библиотеки тегов.

Реализация слушателя событий

Jakarta Faces поддерживает события действий и события изменения значений для компонентов.

События действия происходят, когда пользователь активирует компонент, реализующий `jakarta.faces.component.ActionSource`. Эти события представлены классом `jakarta.faces.event.ActionEvent`.

События изменения значения происходят, когда пользователь изменяет значение компонента, реализующего `jakarta.faces.component.EditableValueHolder`. Эти события представлены классом `jakarta.faces.event.ValueChangeEvent`.

Одним из способов обработки событий является реализация соответствующих классов слушателей. Классы слушателей, которые обрабатывают события действия в приложении, должны реализовывать интерфейс `jakarta.faces.event.ActionListener`. Точно так же слушатели, которые обрабатывают события изменения значения, должны реализовать интерфейс `jakarta.faces.event.ValueChangeListener`.

В этом разделе объясняется, как реализовать два класса слушателей.

Для обработки событий, генерируемых кастомными компонентами, необходимо реализовать слушатель событий и обработчик событий и вручную поставить событие в очередь на компоненте. Смотрите Обработка событий для кастомных компонентов для получения дополнительной информации.

> Вам не нужно создавать реализацию `ActionListener` для обработки события, которое приводит исключительно к переходу на страницу и не выполняет никакой другой специфичной для приложения обработки. Смотрите Пишем метод для обработки навигации для получения информации о том, как управлять навигацией по страницам.

Реализация слушателей изменения значения

Реализация `jakarta.faces.event.ValueChangeListener` должна содержать метод `processValueChange(ValueChangeEvent)`. Этот метод обрабатывает указанное событие изменения значения и вызывается Jakarta Faces, когда происходит событие изменения значения. Объект `ValueChangeEvent` хранит старые и новые значения компонента, вызвавшего событие.

В примере Duke's Bookstore реализация слушателя `NameChanged` зарегистрирована в `name` компонента `UIInput` на странице `bookcashier.xhtml`. Этот слушатель сохраняет в области видимости сессии имя, введённое пользователем в поле, соответствующем компоненту имени.

`bookreceipt.xhtml` впоследствии получает имя из области видимости сессии:

XML

```
<h:outputFormat title="thanks"
                value="#{bundle.ThankYouParam}">
    <f:param value="#{sessionScope.name}"/>
</h:outputFormat>
```

Когда страница `bookreceipt.xhtml` загружена, в сообщении отображается имя:

JAVA

```
"Thank you, {0}, for purchasing your books from us."
```

Вот часть реализации слушателя `NameChanged`:

JAVA

```
public class NameChanged extends Object implements ValueChangeListener {

    @Override
    public void processValueChange(ValueChangeEvent event)
            throws AbortProcessingException {

        if (null != event.getNewValue()) {
            FacesContext.getCurrentInstance().getExternalContext().
                getSessionMap().put("name", event.getNewValue());
        }
    }
}
```

Когда пользователь вводит имя в поле, генерируется событие изменения значения и вызывается метод `processValueChange(ValueChangeEvent)` реализации слушателя `NameChanged`. Этот метод получает идентификатор компонента, вызвавшего событие, из объекта `ValueChangeEvent` и помещает значение вместе с именем атрибута в сессию объекта `FacesContext`.

Регистрация слушателя изменения значения в компоненте объясняет, как зарегистрировать этот слушатель в компоненте.

Реализация слушателей действий

Реализация `jakarta.faces.event.ActionListener` должна содержать метод `processAction(ActionEvent)`. Метод `processAction(ActionEvent)` обрабатывает указанное событие действия. Jakarta Faces вызывает метод `processAction(ActionEvent)` когда происходит `ActionEvent`.

В примере Duke's Bookstore используются две реализации `ActionListener`, `LinkBookChangeListener` и `MapBookChangeListener`. Смотрите Обработка событий для кастомных компонентов для получения подробной информации о `MapBookChangeListener`.

Регистрация слушателя действия в компоненте объясняет, как зарегистрировать этого слушателя в компоненте.

Обработка событий для кастомных компонентов

Как объяснено в Реализация слушателя событий, события автоматически ставятся в очередь в стандартных компонентах, которые запускают события. Кастомный компонент, с другой стороны, должен вручную ставить события в очередь из своего метода `decode`, если он запускает события.

Выполнение декодирования объясняет, как поставить событие в очередь в `MapComponent`, используя метод `decode`. В этом разделе объясняется, как написать класс, представляющий событие клика на карту, и как написать метод, который обрабатывает это событие.

Как объясняется в Пояснении к странице Facelets, атрибут `actionListener` тега `bookstore:map` указывает на класс `MapBookChangeListener`, Метод `processAction` класса слушателя обрабатывает событие клика на карту изображения. Вот метод `processAction`:

JAVA

```
@Override
public void processAction(ActionEvent actionEvent)
        throws AbortProcessingException {

    AreaSelectedEvent event = (AreaSelectedEvent) actionEvent;
    String current = event.getMapComponent().getCurrent();
    FacesContext context = FacesContext.getCurrentInstance();
    String bookId = books.get(current);
    context.getExternalContext().getSessionMap().put("bookId", bookId);
}
```

Когда Jakarta Faces вызывает этот метод, он передаёт объект `ActionEvent`, который представляет событие, сгенерированное путём клика на карту изображения. Затем он преобразует его в объект `AreaSelectedEvent` (см. *`tut-install`*`/examples/case-studies/dukes-bookstore/src/java/dukesbookstore/listeners/AreaSelectedEvent.java`). Затем этот метод получает `MapComponent`, связанный с событием. Далее он получает значение `current` атрибута объекта `MapComponent`, который указывает выбранную в данный момент область. Затем метод использует значение атрибута `current` для получения значения идентификатора книги из объекта `HashMap`, который создаётся где-то в другом месте класса `MapBookChangeListener`. Наконец, метод помещает идентификатор, полученный из объекта `HashMap`, в отображение (Map) сессии приложения.

В дополнение к методу, который обрабатывает событие, требуется сам класс события. Этот класс очень прост в написании. Вы расширяете `ActionEvent` и предоставляете конструктор, принимающий компонент, для которого событие находится в очереди, и метод, возвращающий компонент. Вот класс `AreaSelectedEvent`, используемый с картой изображения:

```java
public class AreaSelectedEvent extends ActionEvent {
    public AreaSelectedEvent(MapComponent map) {
        super(map);
    }
    public MapComponent getMapComponent() {
        return ((MapComponent) getComponent());
    }
}
```

Как объяснено в разделе Создание классов кастомного компонента, чтобы `MapComponent` в первую очередь вызывал события, он должен реализовывать `ActionSource`. Поскольку `MapComponent` расширяет `UICommand`, он также реализует `ActionSource`.

Определение тега кастомного компонента в дескрипторе библиотеки тегов

Чтобы использовать кастомный тег, вы объявляете его в дескрипторе библиотеки тегов (TLD). Файл TLD определяет, как кастомный тег используется на странице Jakarta Faces. Веб-контейнер использует TLD для валидации тега. Набор тегов, которые являются частью инструментария отрисовки HTML, определены в TLD HTML_BASIC, доступном в стандартной библиотеке HTML-тегов Jakarta Faces (https://jakarta.ee/specifications/faces/3.0/renderkitdoc/).

Имя файла TLD должно заканчиваться на `taglib.xml`. В примере Duke's Bookstore кастомные теги `area` и `map` определены в файле `web/WEB-INF/bookstore.taglib.xml`.

Все определения тегов должны быть вложены в элемент `facelet-taglib` в TLD. Каждый тег определяется элементом `tag`. Вот определения тегов для компонентов `area` и `map`:

```xml
<facelet-taglib xmlns="https://jakarta.ee/xml/ns/jakartaee"
...>
    <namespace>http://dukesbookstore</namespace>
    <tag>
        <tag-name>area</tag-name>
        <component>
            <component-type>DemoArea</component-type>
            <renderer-type>DemoArea</renderer-type>
        </component>
    </tag>
    <tag>
        <tag-name>map</tag-name>
        <component>
            <component-type>DemoMap</component-type>
            <renderer-type>DemoMap</renderer-type>
        </component>
    </tag>
</facelet-taglib>
```

Элемент `component-type` указывает имя, определённое в аннотации `@FacesComponent`, а элемент `renderer-type` определяет `rendererType`, определеный в аннотации `@FacesRenderer`.

Элемент `facelet-taglib` также должен включать элемент `namespace`, который определяет пространство имён, которое должно быть указано на страницах, использующих кастомный компонент. См. Использование кастомного компонента для получения информации об указании пространства имён на страницах.

Файл TLD находится в каталоге `WEB-INF`. Кроме того, в дескриптор развёртывания (`web.xml`) включена запись для идентификации файла дескриптора библиотеки кастомных тегов следующим образом:

XML

```
    <context-param>
        <param-name>jakarta.faces.FACELETS_LIBRARIES</param-name>
        <param-value>/WEB-INF/bookstore.taglib.xml</param-value>
    </context-param>
```

Использование кастомного компонента

Чтобы использовать кастомный компонент на странице, нужно добавить на страницу его тег.

Как описано в Определение тега кастомного компонента в дескрипторе библиотеки тегов, нужно убедиться, что TLD, который определяет любые кастомные теги, упакован в приложение, если вы собираетесь использовать теги на своих страницах. Файлы TLD хранятся в каталоге `WEB-INF/` или подкаталоге файла WAR или в каталоге `META-INF/` или подкаталоге библиотеки тегов, упакованной в файл JAR.

Вам также необходимо включить объявление пространства имён на страницу, чтобы страница имела доступ к тегам. Кастомные теги для примера Duke's Bookstore определены в `bookstore.taglib.xml`. Тег `ui:composition` на странице `index.xhtml` объявляет пространство имён, определённое в библиотеке тегов:

XML

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
                xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
                xmlns:h="http://xmlns.jcp.org/jsf/html"
                xmlns:f="http://xmlns.jcp.org/jsf/core"
                xmlns:bookstore="http://dukesbookstore"
                template="./bookstoreTemplate.xhtml">
```

Наконец, чтобы использовать кастомный компонент на странице, вы добавляете на страницу его тег.

Пример Duke's Bookstore включает в себя кастомный компонент карты изображения на странице `index.xhtml`. Этот компонент позволяет выбрать книгу, нажав на область карты изображения:

XML

```
...
<h:graphicImage id="mapImage"
                name="book_all.jpg"
                library="images"
                alt="#{bundle.chooseLocale}"
                usemap="#bookMap" />
<bookstore:map id="bookMap"
               current="map1"
               immediate="true"
               action="bookstore">
    <f:actionListener
        type="ee.jakarta.tutorial.dukesbookstore.listeners.MapBookChangeListener" />
    <bookstore:area id="map1" value="#{Book201}"
                    onmouseover="resources/images/book_201.jpg"
                    onmouseout="resources/images/book_all.jpg"
                    targetImage="mapImage" />
    ...
    <bookstore:area id="map6" value="#{Book207}"
                    onmouseover="resources/images/book_207.jpg"
                    onmouseout="resources/images//book_all.jpg"
                    targetImage="mapImage" />
</bookstore:map>
```

Стандартный тег `h:graphicImage` связывает изображение (`book_all.jpg`) с картой изображения, на которую ссылается значение атрибута `usemap`.

Кастомный тег `bookstore:map`, представляющий кастомный компонент `MapComponent`, задаёт карту изображения и содержит набор тегов `bookstore:area`. Каждый кастомный тег `bookstore:area` представляет кастомный тег `AreaComponent` и указывает область карты изображения.

На странице атрибуты `onmouseover` и `onmouseout` определяют изображение, которое отображается, когда пользователь выполняет действия, описанные атрибутами. Кастомный отрисовщик также отображает атрибут `onclick`.

На отображаемой HTML-странице атрибуты `onmouseover`, `onmouseout` и `onclick` определяют, какой код JavaScript выполняется при возникновении этих событий. Когда пользователь наводит указатель мыши на область, функция `onmouseover` отображает карту с выделенной областью. Когда пользователь перемещает мышь за пределы области, функция `onmouseout` отображает исходное изображение. Когда пользователь кликает на области, функция `onclick` устанавливает значение идентификатора выбранной области для скрытого тега `input` и отправляет страницу.

Когда кастомный отрисовщик отображает эти атрибуты в HTML, он также отображает код JavaScript. Кастомный отрисовщик также отображает весь атрибут `onclick`, а не позволяет автору страницы установить его.

Кастомный отрисовщик, который отображает тег HTML `map`, также отображает скрытый компонент `input`, который содержит текущую область. Серверные объекты получают значение скрытого поля `input` и задают локаль в объекте `FacesContext` в соответствии с выбранной областью.

Создание и использование кастомного конвертера

Класс конвертера Jakarta Faces преобразует строки в объекты, а объекты в строки по мере необходимости. Для этой цели Jakarta Faces предоставляет несколько стандартных конвертеров. Смотрите Использование стандартных конвертеров для получения дополнительной информации об этих включённых конвертерах.

Как объясняется в Модель конвертации, если стандартные конвертеры, включённые в Jakarta Faces, не могут выполнить нужное преобразование данных, можно создать кастомный конвертер для выполнения этого специализированного преобразования. Эта реализация, как минимум, должна определить, как преобразовывать данные в обоих направлениях между двумя представлениями данных, описанными в Модель преобразования.

Все кастомные конвертеры должны реализовывать интерфейс `jakarta.faces.convert.Converter`. В этом разделе объясняется, как реализовать этот интерфейс для выполнения кастомной конвертации данных.

Пример Duke’s Bookstore использует кастомную реализацию `Converter`, расположенную в *`tut-install`*`/examples/case-studies/dukes-bookstore/src/java/dukesbookstore/converters/CreditCardConverter.java`, для преобразования данных, введённых в поле "Номер кредитной карты" на странице `bookcashier.xhtml`. Он удаляет пробелы и дефисы из текстовой строки и форматирует её так, чтобы пробел разделял каждые четыре символа.

Другой типичный пример использования кастомного конвертера — список нестандартных типов объектов. В примере Duke's Tutoring сущностям `Student` и `Guardian` требуется кастомный конвертер, чтобы их можно было преобразовывать в компонент ввода `UISelectItems` и из него.

Создание кастомного конвертера

Класс кастомного конвертера `CreditCardConverter` создаётся следующим образом:

JAVA

```
@FacesConverter("ccno")
public class CreditCardConverter implements Converter {
    ...
}
```

Аннотация `@FacesConverter` регистрирует класс кастомного конвертера как конвертер с именем `ccno` в реализации Jakarta Faces. Кроме того, вы можете зарегистрировать конвертер с записями в файле конфигурации приложения, как показано в Регистрация кастомного конвертера.

Чтобы определить, как данные преобразуются из представления в модель, реализация `Converter` должна реализовать метод `getAsObject(FacesContext, UIComponent, String)` интерфейса `Converter`. Вот реализация этого метода из `CreditCardConverter`:

JAVA

```
@Override
public Object getAsObject(FacesContext context,
        UIComponent component, String newValue)
        throws ConverterException {

    if (newValue.isEmpty()) {
        return null;
    }
    // Так как это конвертация строки в строку,
    // то ConverterException произойти не может.

    String convertedValue = newValue.trim();
    if ( (convertedValue.contains("-")) || (convertedValue.contains(" "))) {
        char[] input = convertedValue.toCharArray();
        StringBuilder builder = new StringBuilder(input.length);
        for (int i = 0; i < input.length; ++i) {
            if ((input[i] == '-') || (input[i] == ' ')) {
            } else {
                builder.append(input[i]);
            }
        }
        convertedValue = builder.toString();
    }
    return convertedValue;
}
```

В фазе применения значений запроса, когда обрабатываются методы `decode` компонентов, Jakarta Faces ищет локальное значение компонента в запросе и вызывает метод `getAsObject`. При вызове этого метода Jakarta Faces передаёт текущий объект `FacesContext`, компонент, данные которого требуют преобразования, и локальное значение в виде `String`. Затем метод записывает локальное значение в массив символов, обрезает дефисы и пробелы, добавляет остальные символы в `String` и возвращает `String`.

Чтобы определить, как данные преобразуются из модели в представление, реализация `Converter` должна реализовать метод `getAsString(FacesContext, UIComponent, Object)` из интерфейса `Converter`. Вот реализация этого метода:

JAVA

```java
@Override
public String getAsString(FacesContext context,
        UIComponent component, Object value)
        throws ConverterException {

    String inputVal = null;
    if ( value == null ) {
        return "";
    }
    // value должно быть приводимого к String типа.
    try {
        inputVal = (String)value;
    } catch (ClassCastException ce) {
        FacesMessage errMsg = new FacesMessage(CONVERSION_ERROR_MESSAGE_ID);
        FacesContext.getCurrentInstance().addMessage(null, errMsg);
        throw new ConverterException(errMsg.getSummary());
    }
    // добавление пробелов после 4 символов для лучшей
    // читаемости, если их ещё нет.
    char[] input = inputVal.toCharArray();
    StringBuilder builder = new StringBuilder(input.length + 3);
    for (int i = 0; i < input.length; ++i) {
        if ((i % 4) == 0 && (i != 0)) {
            if ((input[i] != ' ') || (input[i] != '-')){
                builder.append(" ");
                // если есть "-", то они заменяются пробелами.
            } else if (input[i] == '-') {
                builder.append(" ");
            }
         }
         builder.append(input[i]);
    }
    String convertedValue = builder.toString();
    return convertedValue;
}
```

В фазе отрисовки ответа, на котором вызываются методы `encode` компонентов, Jakarta Faces вызывает метод `getAsString` для генерации соответствующего вывода. Когда Jakarta Faces вызывает этот метод, он передаёт текущий `FacesContext` компоненту `UIComponent`, значение которого необходимо конвертировать, и значение бина, подлежащего конвертации. Поскольку этот конвертер выполняет преобразование `String` в `String`, этот метод может преобразовывать значение бина в `String`.

Если значение не может быть преобразовано в `String`, метод выдает исключение, передавая сообщение об ошибке из пакета ресурсов, зарегистрированного в приложении. Регистрация сообщений приложения объясняет, как зарегистрировать кастомные сообщения об ошибках в приложении.

Если значение можно преобразовать в `String`, метод считывает `String` в массив символов и циклически перебирает массив, добавляя пробел после каждых четырёх символов.

Вы также можете создать кастомный конвертер с аннотацией `@FacesConverter`, в которой указан атрибут `forClass`, как показано в следующем примере из приложения Duke's Tutoring:

JAVA

```java
@FacesConverter(forClass=Guardian.class, value="guardian")
public class GuardianConverter extends EntityConverter implements Converter { ... }
```

Атрибут `forClass` регистрирует конвертер как конвертер по умолчанию для класса `Guardian`. Поэтому всякий раз, когда этот класс указывается атрибутом `value` компонента ввода, конвертер вызывается автоматически.

Класс конвертера может быть отдельным классом POJO, как в случае с Duke's Bookstore. Однако если ему требуется доступ к объектам, определённым в классе Managed-бина, он может быть дочерним классом Managed-бина Jakarta Faces, как в примере `address-book` , в котором конвертеры используют Enterprise-бин, который инъецируется в класс Managed-бина.

Использование кастомного конвертера

Чтобы применить преобразование данных, выполненное кастомным конвертером, к значению определённого компонента, необходимо выполнить одно из следующих действий.

- Запишите ссылку на конвертер в атрибуте `converter` тега компонента.
- Вложите тег `f:converter` в тег компонента и укажите ссылку на кастомный конвертер из одного из атрибутов тега `f:converter` .

Если вы используете атрибут `converter` тега компонента, этот атрибут должен ссылаться на идентификатор реализации `Converter` или полное имя класса конвертера. Создание и использование кастомного конвертера объясняет, как реализовать кастомный конвертер.

Идентификатор класса конвертера кредитных карт: `ccno` , значение указано в аннотации `@FacesConverter` :

JAVA

```
@FacesConverter("ccno")
public class CreditCardConverter implements Converter {
    ...
}
```

Поэтому объект `CreditCardConverter` можно зарегистрировать в компоненте `ccno` , как показано в следующем примере:

XML

```
<h:inputText id="ccno"
             size="19"
             converter="ccno"
             value="#{cashierBean.creditCardNumber}"
             required="true"
             requiredMessage="#{bundle.ReqCreditCard}">
    ...
</h:inputText>
```

Задавая атрибут `converter` тега компонента для идентификатора конвертера или его имени класса, вы вызываете автоматическое преобразование локального значения этого компонента в соответствии с правилами, указанными в реализации `Converter` .

Вместо ссылки на конвертер из атрибута `converter` тега компонента, вы можете ссылаться на конвертер из тега `f:converter` , вложенного в тег компонента. Чтобы сослаться на кастомный конвертер с помощью тега `f:converter` , выполните одно из следующих действий.

- Задайте для атрибута `converterId` тега `f:converter` идентификатор реализации `Converter` , определённый в аннотации `@FacesConverter` или в файле конфигурации приложения. Этот метод показан в `bookcashier.xhtml` :

```xml
<h:inputText id="ccno"
             size="19"
             value="#{cashierBean.creditCardNumber}"
             required="true"
             requiredMessage="#{bundle.ReqCreditCard}">
    <f:converter converterId="ccno"/>
    <f:validateRegex
       pattern="\d{16}|\d{4} \d{4} \d{4} \d{4}|\d{4}-\d{4}-\d{4}-\d{4}"/>
</h:inputText>
```

- Свяжите реализацию `Converter` со свойством Managed-бина с помощью атрибута `binding` тега `f:converter`, как описано в Связывание конвертеров, слушателей и валидаторов со свойствами Managed-бинов.

Jakarta Faces вызывает метод конвертера `getAsObject` для удаления пробелов и дефисов из введённого значения. Метод `getAsString` вызывается при повторном отображении страницы `bookcashier.xhtml`. Это происходит, если пользователь заказывает книги на сумму более 100 долларов.

В примере Duke's Tutoring каждый конвертер зарегистрирован как конвертер для определённого класса. Конвертер автоматически вызывается всякий раз, когда этот класс определяется атрибутом `value` компонента ввода. В следующем примере атрибут `itemValue` вызывает конвертер для класса `Guardian`:

```xml
<h:selectManyListbox id="selectGuardiansMenu"
                     title="#{bundle['action.add.guardian']}"
                     value="#{guardianManager.selectedGuardians}"
                     size="5"
                     converter="guardian">
    <f:selectItems value="#{guardianManager.allGuardians}"
                   var="selectedGuardian"
                   itemLabel="#{selectedGuardian.name}"
                   itemValue="#{selectedGuardian}" />
</h:selectManyListbox>
```

Создание и использование кастомного валидатора

Если стандартные валидаторы или Bean Validation не выполняют необходимые проверки, можно создать кастомный валидатор для валидации пользовательского ввода. Как объясняется в Модель валидации, существует два способа реализации кода валидации.

- Реализуйте метод Managed-бина, который выполняет валидацию.
- Обеспечить реализацию `jakarta.faces.validator.Validator` для выполнения валидации.

Пишем метод для выполнения валидации объясняет, как реализовать метод Managed-бина для выполнения валидации. В оставшейся части этого раздела объясняется, как реализовать интерфейс `Validator`.

Если вы решили реализовать интерфейс `Validator` и хотите, чтобы автор страницы настраивал атрибуты валидатора со страницы, то также должны указать кастомный тег для регистрации валидатора в компоненте.

Если вы предпочитаете настраивать атрибуты в реализации `Validator`, можете отказаться от указания кастомного тега и вместо этого позволить автору страницы зарегистрировать валидатор в компоненте с помощью тега `f:validator`, как описано в Использование кастомного валидатора.

Вы также можете создать свойство Managed-бина, которое принимает и возвращает созданную реализацию `Validator`, как описано в Запись свойств объектов, связанных с конвертерами, слушателями или валидаторами. Вы можете использовать атрибут binding тега `f:validator`, чтобы связать реализацию

`Validator` со свойством Managed-бина.

Как правило, если данные не проходят валидацию, пользователю на странице отображаются соответствующие сообщения об ошибках. Необходимо хранить эти сообщения об ошибках в bundle-ресурсе.

После создания bundle-ресурса у вас есть два способа сделать сообщения доступными для приложения. Вы можете поместить сообщения об ошибках в `FacesContext` программно или зарегистрировать сообщения об ошибках в файле конфигурации приложения, как описано в Регистрация сообщений приложения.

Например, приложение электронной торговли может использовать кастомный валидатор общего назначения с именем `FormatValidator.java` для валидации введённых данных по шаблону формата, указанному в кастомном теге валидатора. Этот валидатор будет использоваться с полем Номер кредитной карты на странице Facelets. Вот кастомный тег валидатора:

```xml
<mystore:formatValidator
 formatPatterns="9999999999999999|9999 9999 9999 9999|9999-9999-9999-9999"/>
```

Согласно этому валидатору, данные, введённые в поле, должны быть одним из следующих:

- 16-значный номер без пробелов
- 16-значный номер с пробелами между каждыми четырьмя цифрами
- 16-значный номер с дефисами между каждыми четырьмя цифрами

Тег `f:validateRegex` делает ненужным кастомный валидатор в этой ситуации. Однако в оставшейся части этого раздела описывается, как этот валидатор будет реализован и как указать кастомный тег, чтобы автор страницы мог зарегистрировать валидатор в компоненте.

Реализация интерфейса валидатора

Реализация `Validator` должна содержать конструктор, набор методов доступа для любых атрибутов тега и метод `validate`, который переопределяет метод `validate` интерфейса `Validator`.

Гипотетический класс `FormatValidator` также определяет методы доступа для установки атрибута `formatPatterns`, который указывает приемлемые шаблоны формата для ввода в поля. Set-метод вызывает метод `parseFormatPatterns`, который разделяет компоненты строки шаблона в строковый массив `formatPatternsList`.

```java
public String getFormatPatterns() {
    return (this.formatPatterns);
}
public void setFormatPatterns(String formatPatterns) {
    this.formatPatterns = formatPatterns;
    parseFormatPatterns();
}
```

В дополнение к определению методов доступа для атрибутов, класс переопределяет метод `validate` интерфейса `Validator`. Этот метод проверяет ввод и также получает доступ к кастомным сообщениям об ошибках, которые отображаются, когда `String` невалидный.

Метод `validate` выполняет фактическую валидацию данных. Он принимает объект `FacesContext`, компонент, данные которого необходимо проверить, и значение, которое необходимо проверить. Валидатор может проверять только данные компонента, реализующего `jakarta.faces.component.EditableValueHolder`.

Вот реализация метода `validate`:

JAVA

```
@FacesValidator
public class FormatValidator implements Validator, StateHolder {
    ...
    public void validate(FacesContext context, UIComponent component,
                         Object toValidate) {

        boolean valid = false;
        String value = null;
        if ((context == null) || (component == null)) {
            throw new NullPointerException();
        }
        if (!(component instanceof UIInput)) {
            return;
        }
        if ( null == formatPatternsList || null == toValidate) {
            return;
        }
        value = toValidate.toString();
        // валидация значений списка по паттернам.
        Iterator patternIt = formatPatternsList.iterator();
        while (patternIt.hasNext()) {
            valid = isFormatValid(
                ((String)patternIt.next()), value);
            if (valid) {
                break;
            }
        }
        if ( !valid ) {
            FacesMessage errMsg =
                new FacesMessage(FORMAT_INVALID_MESSAGE_ID);
            FacesContext.getCurrentInstance().addMessage(null, errMsg);
            throw new ValidatorException(errMsg);
        }
    }
}
```

Аннотация `@FacesValidator` регистрирует класс `FormatValidator` как валидатор с Jakarta Faces. Метод `validate` получает локальное значение компонента и преобразует его в `String`. Затем он перебирает список `formatPatternsList`, являющийся списком допустимых шаблонов, которые были проанализированы из атрибута `formatPatterns` кастомного тега валидатора.

Во время итерации по списку этот метод проверяет локальное значение компонента на соответствие шаблонам в списке. Если локальное значение не соответствует ни одному шаблону в списке, этот метод генерирует сообщение об ошибке. Затем он создаёт `jakarta.faces.application.FacesMessage` и ставит его в очередь на `FacesContext` для отображения, используя `String`, которая представляет собой ключ в файле `Properties`:

JAVA

```
public static final String FORMAT_INVALID_MESSAGE_ID =
     "FormatInvalid";
}
```

Наконец, метод передаёт сообщение конструктору `jakarta.faces.validator.ValidatorException`.

Когда отображается сообщение об ошибке, `{0}` будет заменён на шаблон формата в сообщении об ошибке, которое выглядит следующим образом:

```
Входные данные должны соответствовать одному из следующих шаблонов: {0}
```
JAVA

Вы можете сохранить и восстановить состояние вашего валидатора, хотя сохранение состояния обычно не требуется. Для этого необходимо реализовать интерфейс `StateHolder`, а также интерфейс `Validator`. Для реализации `StateHolder` нужно реализовать четыре метода: `saveState(FacesContext)`, `restoreState(FacesContext, Object)`, `isTransient` и `setTransient(boolean)`. Смотрите Сохранение и восстановление состояния для получения дополнительной информации.

Указание кастомного тега

Если вы реализовали интерфейс `Validator`, а не метод Managed-бина, который выполняет валидацию, нужно выполнить одно из следующих действий.

- Разрешите автору страницы указывать реализацию `Validator` для использования с тегом `f:validator`. В этом случае реализация `Validator` должна определять свои собственные свойства. Использование кастомного валидатора объясняет, как использовать тег `f:validator`.
- Укажите кастомный тег, который предоставляет атрибуты для настройки свойств валидатора со страницы.

Чтобы создать кастомный тег, необходимо добавить тег в дескриптор библиотеки тегов приложения, `bookstore.taglib.xml`:

```xml
<tag>
    <tag-name>validator</tag-name>
    <validator>
        <validator-id>formatValidator</validator-id>
        <validator-class>
            dukesbookstore.validators.FormatValidator
        </validator-class>
    </validator>
</tag>
```
XML

Элемент `tag-name` определяет имя тега, так как он должен использоваться на странице Facelets. Элемент `validator-id` идентифицирует кастомный валидатор. Элемент `validator-class` связывает кастомный тег с его классом реализации.

Использование кастомного валидатора объясняет, как использовать кастомный тег валидатора на странице.

Использование кастомного валидатора

Чтобы зарегистрировать кастомный валидатор в компоненте, необходимо выполнить одно из следующих действий.

- Вложите кастомный тег валидатора в тег компонента, значение которого вы хотите проверить.
- Вложите стандартный тег `f:validator` в тег компонента и укажите ссылку на реализацию кастомного `Validator` из тега `f:validator`.

Вот гипотетический кастомный тег `formatValidator` для поля Credit Card Number, вложенный в тег `h:inputText`:

```xml
<h:inputText id="ccno" size="19"
  ...
  required="true">
  <mystore:formatValidator
  formatPatterns="9999999999999999|9999 9999 9999 9999|9999-9999-9999-9999"/>
</h:inputText>
<h:message styleClass="validationMessage" for="ccno"/>
```

Этот тег проверяет ввод поля `ccno` по шаблонам, определённым автором страницы в атрибуте `formatPatterns`.

Вы можете использовать один и тот же кастомный валидатор для любого аналогичного компонента, просто вложив кастомный тег валидатора в тег компонента.

Если разработчик приложения, создавший кастомный валидатор, предпочитает настраивать атрибуты в реализации `Validator`, а не разрешать автору страницы настраивать атрибуты со страницы, разработчик не будет создавать кастомный тег для использования с валидатором.

В этом случае автор страницы должен вложить тег `f:validator` в тег компонента, данные которого необходимо валидировать. Затем автору страницы необходимо выполнить одно из следующих действий.

- Задайте для атрибута `validatorId` тега `f:validator` идентификатор валидатора, определённый в файле конфигурации приложения.
- Свяжите реализацию кастомного `Validator` со свойством Managed-бина, используя атрибут `binding` тега `f:validator`, как описано в Связывание конвертеров, слушателей и валидаторов со свойствами Managed-бинов.

Следующий тег регистрирует гипотетический валидатор в компоненте с помощью тега `f:validator` и ссылается на идентификатор валидатора:

```xml
<h:inputText id="name" value="#{CustomerBean.name}"
            size="10" ...>
    <f:validator validatorId="customValidator" />
    ...
</h:inputText>
```

Связывание значений компонентов и объектов со свойствами Managed-бина

Тег компонента может связать свои данные с Managed-бином одним из следующих способов:

- Привязка значения его компонента к свойству бина
- Привязка объекта компонента к свойству бина

Чтобы связать значение компонента со свойством Managed-бина, атрибут `value` тега компонента использует выражение значения EL. Чтобы связать объект компонента со свойством бина, атрибут `binding` тега компонента использует выражение значения.

Когда объект компонента связан со свойством Managed-бина, свойство содержит локальное значение компонента. И наоборот, когда значение компонента связано со свойством Managed-бина, свойство содержит значение, хранящееся в Managed-бине. Это значение обновляется локальным значением в фазе обновления значений модели жизненного цикла. У обоих этих методов есть свои преимущества.

Привязка объекта компонента к свойству компонента имеет следующие преимущества.

- Managed-бин может программно изменять атрибуты компонента.
- Managed-бин может создавать компоненты, а не позволять автору страницы делать это.

Привязка значения компонента к свойству компонента имеет следующие преимущества.

- Автор страницы имеет больший контроль над атрибутами компонента.
- Managed-бин не имеет зависимостей от API Jakarta Faces (например, классов компонентов), что позволяет лучше отделить уровень представления от уровня модели.
- Jakarta Faces может выполнять конвертацию данных на основе типа свойства бина, при этом разработчику не требуется применять конвертер.

В большинстве случаев со свойством связывается именно значение компонента, а не весь компонент целиком. Связывание компонента следует использовать только тогда, когда требуется динамически изменить один из его атрибутов. Например, если приложение отображает компонент только при определённых условиях, оно может соответствующим образом установить свойство `rendered` компонента, получив свойство, с которым связан компонент.

При ссылке на свойство с помощью атрибута `value` тега компонента необходимо использовать правильный синтаксис. Для примера предположим, что Managed-бин с именем `MyBean` имеет это свойство типа `int`:

```java
protected int currentOption = null;
public int getCurrentOption(){...}
public void setCurrentOption(int option){...}
```

Атрибут `value`, который ссылается на это свойство, должен иметь выражение привязки значения:

```java
#{myBean.currentOption}
```

Помимо связывания значения компонента со свойством бина атрибут `value` может указывать литерал или отображать данные компонента в любой примитив (например, `int`), структуру (например, массив) или коллекцию (например, список), независимо от компонента JavaBeans. Таблица 15-3 перечисляет некоторые примеры выражений привязки значений, которые могут использоваться с атрибутом `value`.

Таблица 15-3 Примеры выражений привязки значений

Значение	**Выражение**
Логическое	`cart.numberOfItems > 0`
Свойство, инициализированное контекстным параметром	`initParam.quantity`
Свойство бина	`cashierBean.name`
Значение в массиве	`books[3]`
Значение в коллекции	`books["fiction"]`
Свойство объекта в массиве объектов	`books[3].price`

В следующих двух разделах объясняется, как использовать атрибут `value` для связывания значения компонента со свойством компонента или другим объектом данных и как использовать атрибут `binding` для связывания объекта компонента со свойством бина.

Связывание значения компонента со свойством

Чтобы связать значение компонента со свойством Managed-бина, укажите имя компонента и свойство с помощью атрибута `value`.

Это означает, что первая часть выражения значения EL должна соответствовать имени Managed-бина до первого периода (`.`), а часть выражения значения после периода должна соответствовать свойству объекта Managed-бина.

Например, в примере Duke's Bookstore тег `h:dataTable` в `bookcatalog.xhtml` устанавливает значение компонента равным значению `books` свойства вспомогательного бина `BookstoreBean` с именем `store`:

XML

```
<h:dataTable id="books"
             value="#{store.books}"
             var="book"
             headerClass="list-header"
             styleClass="list-background"
             rowClasses="list-row-even, list-row-odd"
             border="1"
             summary="#{bundle.BookCatalog}">
```

Это значение получается путём вызова метода `getBooks` вспомогательного бина, который в свою очередь вызывает метод `getBooks` сессионного бина `BookRequestBean`.

Если вы используете файл конфигурации приложения для настройки Managed-бинов вместо определения их в классах Managed-бинов, имя бина в выражении `value` должно соответствовать элементу `managed-bean-name` объявления Managed-бина до первого появления `.` в выражении. Аналогично, часть выражения значения после точки должна соответствовать имени, указанному в соответствующем элементе `property-name` в файле конфигурации приложения.

Например, рассмотрим конфигурацию Managed-бина, которая настраивает компонент `ImageArea`, соответствующий верхней левой книге в карте изображения на странице `index.html` в примере Duke's Bookstore:

XML

```
<managed-bean eager="true">
    ...
    <managed-bean-name>Book201</managed-bean-name>
    <managed-bean-class>dukesbookstore.model.ImageArea</managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
    <managed-property>
        ...
        <property-name>shape</property-name>
        <value>rect</value>
    </managed-property>
    <managed-property>
        ...
        <property-name>alt</property-name>
        <value>Duke</value>
    </managed-property>
    ...
```

В этом примере настраивается компонент с именем `Book201`, который имеет несколько свойств, одно из которых называется `shape`.

Хотя теги `bookstore:area` на странице `index.xhtml` не связываются со свойством `ImageArea` (а связываются с самим компонентом), вы можете ссылаться на свойство, используя выражение значения из атрибута `value` тега компонента:

```xml
<h:outputText value="#{Book201.shape}" />
```

Смотрите Настройка Managed-бинов для получения информации о том, как настроить бины в файле конфигурации приложения.

Связывание значения компонента с неявным объектом

Неявный объект — это внешний источник данных, на который можно сослаться из атрибута `value`.

Страница `bookreceipt.xhtml` примера Duke's Bookstore содержит ссылку на неявный объект:

```xml
<h:outputFormat title="thanks"
                value="#{bundle.ThankYouParam}">
    <f:param value="#{sessionScope.name}"/>
</h:outputFormat>
```

Этот тег получает имя клиента от области видимости сессии и вставляет его в параметризованное сообщение с ключом `ThankYouParam` из bundle-ресурса. Например, если имя клиента Gwen Canigetit, этот тег будет выводить:

```
Thank you, Gwen Canigetit, for purchasing your books from us.
```

Получение значений из других неявных объектов выполняется аналогично примеру, показанному в этом разделе. Таблица 15-4 перечисляет неявные объекты, на которые можно сослаться из атрибута value. Все неявные объекты, за исключением объектов области видимости, доступны только для чтения и поэтому не должны использоваться в качестве значений для компонента `UIInput`.

Таблица 15-4 Неявные объекты

Неявный объект	**Пояснение**
`applicationScope`	`Map` значений атрибутов области приложения, ключи которого — имена атрибутов
`cookie`	`Map` значений файлов cookie для текущего запроса с указанием имени файла cookie
`facesContext`	Объект `FacesContext` для текущего запроса
`header`	`Map` значений заголовка HTTP для текущего запроса, ключи которого — имена заголовков
`HeaderValues`	Массивы `Map` из `String`, содержащие все значения заголовков для заголовков HTTP в текущем запросе с указанием имени заголовка
`initParam`	`Map` параметров инициализации контекста для этого веб-приложения

Неявный объект	Пояснение
`param`	`Map` параметров запроса, ключи которого — имена параметров
`paramValues`	Массивы `Map` из `String`, содержащие все значения параметров для параметров запроса в текущем запросе, с указанием имени параметра
`requestScope`	`Map` атрибутов запроса, ключи которого — имена атрибутов
`sessionScope`	`Map` атрибутов сессии, ключи которого — имена атрибутов
`view`	Корень `UIComponent` в текущем дереве компонентов, хранящийся в `FacesRequest` для этого запроса

Связывание объекта компонента со свойством бина

Объект компонента может быть связан со свойством бина с помощью выражения значения в атрибуте `binding` тега компонента. Если требуется, чтобы компонент динамически изменял атрибуты, то обычно связывается объект компонента целиком, а не его значение.

Вот два тега со страницы `bookcashier.xhtml`, которые связывают компоненты со свойствами бина:

XML

```
<h:selectBooleanCheckbox id="fanClub"
                         rendered="false"
                         binding="#{cashierBean.specialOffer}" />
<h:outputLabel for="fanClub"
               rendered="false"
               binding="#{cashierBean.specialOfferText}"
               value="#{bundle.DukeFanClub}"/>
</h:outputLabel>
```

Тег `h:selectBooleanCheckbox` отображает флажок и связывает компонент `fanClub` типа `UISelectBoolean` со свойством `specialOffer` бина `cashier`. Тег `h:outputLabel` связывает компонент, представляющий собой метку флажка, со свойством `specialOfferText` бина `cashier`. Если для приложения установлена английская локаль, тег `h:outputLabel` выводит

```
I'd like to join the Duke Fan Club, free with my purchase of over $100
```

Атрибуты `rendered` обоих тегов установлены в `false`, чтобы не отображать флажок и его метку. Если клиент делает большой заказ и кликает кнопку «Отправить», метод `submit` для `CashierBean` устанавливает для свойств `rendered` обоих компонентов значение `true`, вызывая отображение переключателя и его метки.

В этих тегах используются связи с компонентами, а не значениями, поскольку Managed-бин должен динамически устанавливать значения свойств `rendered` компонентов.

Если бы теги использовали связывание значений вместо связывания компонентов, Managed-бин не имел бы прямого доступа к компонентам и, следовательно, потребовался бы дополнительный код для доступа к компонентам из объекта `FacesContext` для изменения свойств `rendered` у них.

Запись свойств, связанных с объектами компонентов объясняет, как записать свойства бинов, связанных с компонентами примера.

Связывание конвертеров, слушателей и валидаторов со свойствами Managed-бинов

Как описано в Добавление компонентов на страницу с помощью библиотеки тегов HTML, автор страницы может связать реализации конвертера, слушателя и валидатора со свойствами Managed-бинов, используя атрибуты `binding` тегов, которые используются для регистрации реализаций компонентов.

Этот метод имеет те же преимущества, что и связывание объектов компонентов со свойствами Managed-бинов, как описано в Связывание значений и объектов компонента со свойствами Managed-бинов. В частности, связывание реализации конвертера, слушателя или валидатора со свойством Managed-бина даёт следующие преимущества.

- Managed-бин может создавать объект реализации, не делая этого на странице.
- Managed-бин может программно изменять атрибуты реализации. В случае кастомной реализации единственным способом изменения атрибутов вне класса реализации будет создание для него кастомного тега и требование от автора страницы установить значения атрибутов со страницы.

Независимо от того, что именно связывается со свойством Managed-бина: конвертер, слушатель или валидатор, — этот процесс одинаков во всех случаях.

- Вложите тег конвертера, слушателя или валидатора в соответствующий тег компонента.
- Убедитесь, что Managed-бин имеет свойство, которое принимает и возвращает класс реализации конвертера, слушателя или валидатора, который вы хотите связать со свойством.
- Ссылайтесь на свойство Managed-бина с помощью выражения значения из атрибута `binding` тега конвертера, слушателя или валидатора.

Для примера предположим, что вы хотите связать стандартный конвертер `DateTime` со свойством Managed-бина, чтобы установить шаблон форматирования ввода пользователя в Managed-бин, а не на страницу Facelets. Сначала страница регистрирует конвертер в компоненте, вкладывая тег `f:convertDateTime` в тег компонента. Затем страница ссылается на свойство атрибутом `binding` тега `f:convertDateTime`:

XML

```
<h:inputText value="#{loginBean.birthDate}">
    <f:convertDateTime binding="#{loginBean.convertDate}" />
</h:inputText>
```

Свойство `convertDate` будет выглядеть примерно так:

JAVA

```
private DateTimeConverter convertDate;
public DateTimeConverter getConvertDate() {
    ...
    return convertDate;
}
public void setConvertDate(DateTimeConverter convertDate) {
    convertDate.setPattern("EEEEEEEE, MMM dd, yyyy");
    this.convertDate = convertDate;
}
```

Смотрите Запись свойств объектов, связанных с конвертерами, слушателями или валидаторами для получения дополнительной информации о написании свойств Managed-бина для реализаций конвертера, слушателя и валидатора.

Глава 16. Настройка приложений Jakarta Faces

В этой главе описываются дополнительные задачи настройки, необходимые при создании больших и сложных приложений.

Введение в настройку приложений Jakarta Faces

Процесс создания и развёртывания простых приложений Jakarta Faces описан в предыдущих главах этого учебника, включая главу 6 *Начало работы с веб-приложениями*, главу 8 *Введение в Facelets*, главу 13 *Использование Ajax с Jakarta Faces* и главу 14 *Составные компоненты: расширенные темы и пример*. Однако при создании больших и сложных приложений требуются различные дополнительные задачи настройки. Эти задачи включают в себя следующее:

- Регистрация Managed-бинов в приложении, чтобы все части приложения имели к ним доступ
- Конфигурирование Managed-бинов и бинов модели, чтобы их объекты имели правильные значения, когда страница ссылается на них
- Определение правил навигации для каждой страницы в приложении, чтобы приложение имело поток (flow) страниц, если необходима навигация не по умолчанию
- Упаковка приложения для включения всех страниц, ресурсов и других файлов, чтобы приложение можно было развернуть в любом JavaEE-совместимом контейнере

Использование аннотаций для настройки Managed-бинов

> В Jakarta Faces 2.3 аннотации Managed-бинов устарели. Предпочтительным подходом является использование CDI.

Поддержка Jakarta Faces для аннотирования компонентов представлена в разделе 1.8.3 «Технология Jakarta Faces». Аннотации бинов могут использоваться для настройки приложений Jakarta Faces.

Аннотация `@Named` (`Jakarta.Inject.Named`) при использовании вместе с классом автоматически регистрирует этот класс как ресурс с реализацией Jakarta Faces. Компонент, использующий эти аннотации, является Managed-бином CDI.

Ниже показано использование аннотаций `@Named` и `@SessionScoped` в классе:

JAVA

```
@Named("cart")
@SessionScoped
public class ShoppingCart ... { ... }
```

Приведённый выше фрагмент кода показывает бин, управляемый Jakarta Faces и доступный в течение жизни сессии.

Вы можете аннотировать компоненты для использования в одной из областей видимости, перечисленных в нижеследующем разделе Использование областей видимости Managed-бинов.

Все классы будут проверяться на наличие аннотаций при запуске, если только элемент `faces-config` в файле `faces-config.xml` не имеет атрибута `metadata-complete`, установленного в `true`.

Аннотации также доступны для других артефактов, таких как компоненты, конвертеры, валидаторы и отрисовщики, которые будут использоваться вместо записей файла конфигурации приложения. Они обсуждаются, наряду с регистрацией кастомных слушателей, кастомных валидаторов и кастомный конвертеров, в главе 15 *Создание кастомных компонентов пользовательского интерфейса и других кастомных объектов*.

Использование областей видимости Managed-бинов

Вы можете использовать аннотации для определения области видимости, в которой будет храниться бин. Вы можете указать одну из следующих областей видимости для класса бина.

- Приложение (`jakarta.enterprise.context.ApplicationScoped`): область видимости приложения сохраняется при всех взаимодействиях пользователей с веб-приложением.
- Сессия (`jakarta.enterprise.context.SessionScoped`): область видимости сессии сохраняется в течение нескольких HTTP-запросов к веб-приложению.
- Поток (Flow) (`jakarta.faces.flows.FlowScoped`): область видимости потока сохраняется в течение взаимодействия пользователя с определённым потоком веб-приложения. См. Использование Faces Flows для получения дополнительной информации.
- Запрос (`jakarta.enterprise.context.RequestScoped`): область видимости запроса сохраняется в течение одного HTTP-запроса к веб-приложению.
- Зависимый (`jakarta.enterprise.context.Dependent`): указывает, что компонент зависит от другого компонента.

Возможно, вы захотите использовать `@Dependent` , когда Managed-бин ссылается на другой Managed-бин. Второй компонент не должен находиться в области видимости (`@Dependent`), даже если предполагается, что он создаётся только тогда, когда на него ссылаются. Если вы определяете бин как `@Dependent` , бин создаётся заново каждый раз, когда на него ссылаются, поэтому он не сохраняется ни в какой области видимости.

Если на ваш Managed-бин ссылается атрибут `binding` тега компонента, вы должны определить бин с областью видимости запроса. Если вместо этого вы поместили компонент в область видимости сессии или приложения, компонент должен будет принять меры предосторожности для обеспечения потокобезопасности, потому что каждый объект `jakarta.faces.component.UIComponent` зависит от работы внутри одного потока.

Если вы конфигурируете компонент, позволяющий связывать атрибуты с представлением, можете использовать область видимости представления. Атрибуты сохраняются до тех пор, пока пользователь не перейдёт к следующему представлению.

Файл конфигурации приложения

Jakarta Faces обеспечивает переносимый формат конфигурации (в виде документа XML) для настройки ресурсов приложения. Один или несколько документов XML, называемых файлами конфигурации приложения, могут использовать этот формат для регистрации и настройки объектов и ресурсов, а также для определения правил навигации для приложений. Файл конфигурации приложения обычно называется `faces-config.xml` .

Файл конфигурации приложения необходим в следующих случаях:

- Чтобы указать элементы конфигурации для вашего приложения, которые недоступны через аннотации Managed-бина, такие как локализованные сообщения и правила навигации
- Чтобы переопределить аннотации Managed-бина при развёртывании приложения

Файл ресурсов конфигурации приложения должен быть валидным по XML-схеме, расположенной по ссылке `https://jakarta.ee/xml/ns/jakartaee/web-facesconfig_3_0.xsd`.

Кроме того, каждый файл должен содержать следующую информацию в следующем порядке:

- Номер версии XML, обычно с атрибутом `encoding`:

```xml
<?xml version="1.0" encoding="UTF-8"?>
```

- Тег `faces-config`, содержащий все другие объявления:

```xml
<faces-config version="3.0" xmlns="https://jakarta.ee/xml/ns/jakartaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
              https://jakarta.ee/xml/ns/jakartaee/web-facesconfig_3_0.xsd">
    ...
</faces-config>
```

Вы можете иметь более одного файла конфигурации для приложения. Jakarta Faces находит файл/файлы конфигурации следующим образом.

- Ресурс с именем `/META-INF/faces-config.xml` в любом из файлов JAR в каталоге `/WEB-INF/lib/` веб-приложения и в загрузчиках родительских классов , Если ресурс с таким именем существует, он загружается как ресурс конфигурации. Этот метод полезен для упакованной библиотеки, содержащей некоторые компоненты и отрисовщики. Кроме того, любой файл с именем, оканчивающимся на `faces-config.xml`, также считается ресурсом конфигурации и загружается как таковой.
- Контекстный параметр инициализации `jakarta.faces.application.CONFIG_FILES` в файле дескриптора развёртывания указывает один или несколько (разделенных запятыми) путей к нескольким файлам конфигурации для веб-приложения. Этот метод чаще всего используется для приложений масштаба предприятия, которые делегируют отдельным группам ответственность за ведение файла для каждой части большого приложения.
- Ресурс с именем `faces-config.xml` в каталоге `/WEB-INF/` вашего приложения. Таким образом простые веб-приложения делают свои файлы конфигурации доступными.

Для доступа к ресурсам, зарегистрированным в приложении, разработчик приложения может использовать объект `jakarta.faces.application.Application`, который автоматически создаётся для каждого приложения. Объект `Application` действует как централизованная фабрика для ресурсов, определённых в файле XML.

Когда приложение запускается, Jakarta Faces создаёт один объект класса `Application` и настраивает его в соответствии с информацией из файла конфигурации приложения.

Ранняя инициализация Managed-бинов с областью видимости приложения

Объекты Managed-бинов Jakarta Faces (указанных в файле `faces-config.xml` или аннотированных `jakarta.faces.bean.ManagedBean`) создаются отложенно. То есть, они создаются при выполнении запроса из приложения.

Чтобы объект компонента в области видимости приложения создавался и помещался в область приложения сразу после запуска приложения и до выполнения любого запроса, атрибут `eager` Managed-бина должен быть установлен в `true`, как показано в следующих примерах.

Объявление файла `faces-config.xml` выглядит следующим образом:

```xml
<managed-bean eager="true">
```

Аннотация выглядит следующим образом:

```xml
@ManagedBean(eager=true)
@ApplicationScoped
```

Упорядочение файлов конфигурации приложения

Поскольку Jakarta Faces позволяет иметь несколько файлов конфигурации приложения, хранящихся в разных местах, в определённых ситуациях может быть важен порядок их загрузки (например, при использовании объектов уровня приложения). Этот порядок может быть определён через элемент `ordering` и его подэлементы в самом файле конфигурации приложения. Порядок файлов конфигурации приложения может быть абсолютным или относительным.

Абсолютный порядок определяется элементом `absolute-ordering` в файле. При абсолютном упорядочении пользователь указывает порядок, в котором будут загружаться файлы конфигурации приложения. В следующем примере показана запись для абсолютного упорядочения.

Файл `my-faces-config.xml` содержит следующие элементы:

```xml
<faces-config>
    <name>myJSF</name>
    <absolute-ordering>
        <name>A</name>
        <name>B</name>
        <name>C</name>
    </absolute-ordering>
</faces-config>
```

В этом примере A, B и C являются различными файлами конфигурации приложения и должны быть загружены в указанном порядке.

Если в файле есть элемент `absolute-ordering`, обрабатываются только файлы, перечисленные подэлементом `name`. Для обработки любых других файлов конфигурации приложения требуется подэлемент `others`. В отсутствие подэлемента `others` все неперечисленные файлы будут игнорироваться во время загрузки.

Относительный порядок определяется элементом `ordering` и его подэлементами `before` и `after`. При относительном упорядочении порядок загрузки файлов конфигурации приложения вычисляется с учётом порядка записей из разных файлов. В следующем примере показаны некоторые из этих соображений. В следующем примере `config-A`, `config-B` и `config-C` — это разные файлы конфигурации приложения.

Файл `config-A` содержит следующие элементы:

```xml
<faces-config>
    <name>config-A</name>
    <ordering>
        <before>
            <name>config-B</name>
        </before>
    </ordering>
</faces-config>
```

Файл `config-B` (здесь не показан) не содержит элементов `ordering`.

Файл `config-C` содержит следующие элементы:

```xml
<faces-config>
    <name>config-C</name>
    <ordering>
        <after>
            <name>config-B</name>
        </after>
    </ordering>
</faces-config>
```

Благодаря записи подэлемента `before` файл `config-A` будет загружен перед файлом `config-B`. Благодаря записи подэлемента `after` файл `config-C` будет загружен после файла `config-B`.

Кроме того, подэлемент `others` также может быть вложен в подэлементы `before` и `after`. Если присутствует элемент `others`, указанный файл может получить наивысшее или наименьшее предпочтение как среди перечисленных, так и незарегистрированных файлов конфигурации.

Если элемент `ordering` отсутствует в файле конфигурации приложения, этот файл будет загружен после всех файлов, содержащих элементы `ordering`.

Использование Faces Flows

Функция Faces Flows Jakarta Faces позволяет создавать набор страниц с областью видимости `FlowScoped`, которая больше области видимости запроса, но меньше области видимости сессии. Например, вы можете создать серию страниц для оформления заказа в онлайн-магазине. Вы можете создать набор автономных страниц, которые могут быть перенесены из одного магазина в другой при необходимости.

Faces Flow в некоторой степени аналогичны процедурам процедурного программирования.

- Как и процедура, Flow имеет чётко определённую точку входа, список параметров и возвращаемое значение. Однако, в отличие от процедуры, Flow может возвращать несколько значений.
- Как и процедура, Flow имеет область видимости, позволяющую получать информацию только во время вызова Flow. Такая информация недоступна вне области видимости Flow и не потребляет никаких ресурсов после возврата из Flow.
- Как и процедура, Flow может вызвать другие Flow перед возвратом. Вызов Flow поддерживается в стеке вызовов: новый Flow помещается в стек вызова, а после возврата извлекается из него.

Приложение может иметь любое количество Flow. Каждый Flow включает в себя набор страниц и, как правило, один или несколько Managed-бинов, ограниченных областью видимости этого Flow. Каждый Flow имеет начальную точку, называемую начальным узлом, и точку выхода, называемую возвратным узлом.

Данные во Flow ограничиваются только этим Flow, но вы можете передавать данные из одного Flow в другой, задав параметры и вызвав другой Flow.

Flow могут быть вложенными, так что если вы вызываете один Flow из другого и затем выходите из второго Flow, то возвращаетесь к вызывающему Flow, а не к возвратному узлу второго Flow.

Вы можете настроить Flow программно, создав аннотированный класс `@FlowDefinition`, или вы можете настроить Flow с помощью файла конфигурации. Файл конфигурации может содержать один Flow, или вы можете использовать файл `faces-config.xml`, чтобы поместить все Flow в одно место, если в вашем

приложении много Flow. Программная конфигурация размещает код ближе к остальной части кода Flow и позволяет организовывать Flow в модули.

На рисунке 16-1 показаны два Flow и показано, как они взаимодействуют

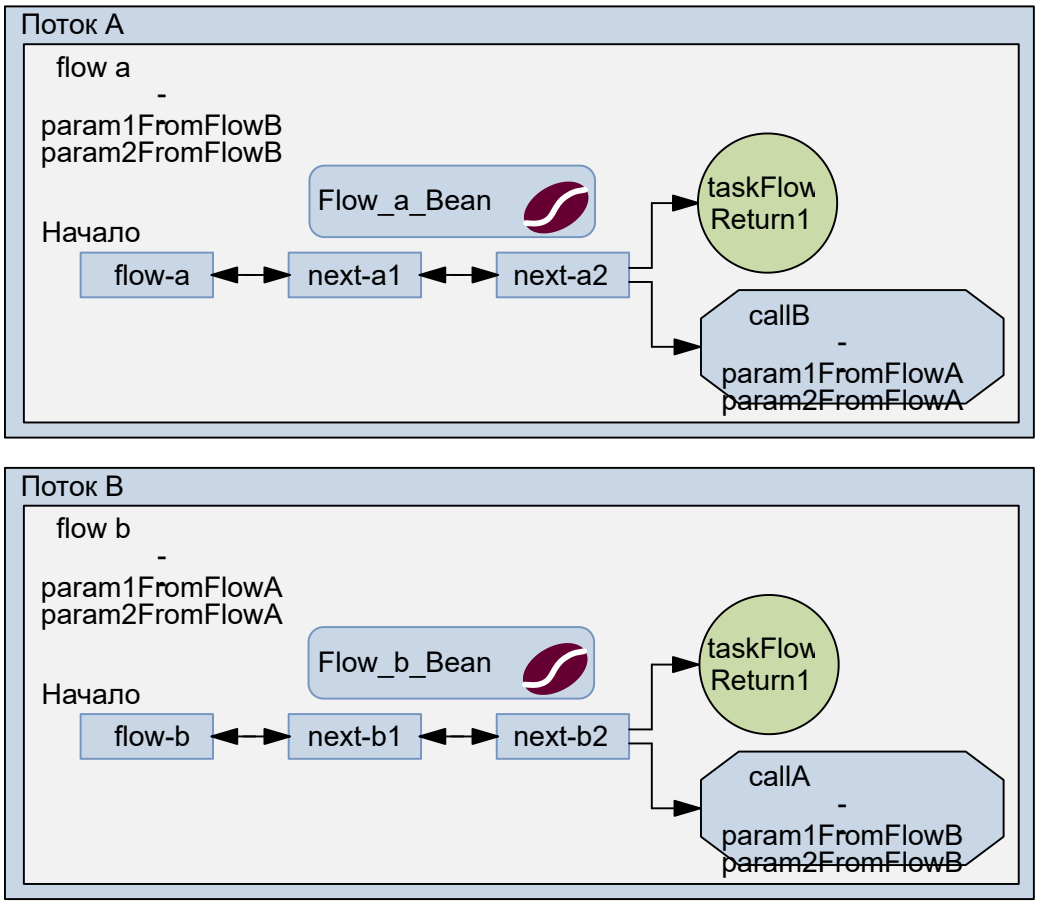

Рис. 16-1. Два Faces Flow и их взаимодействие

На этом рисунке Flow A имеет начальный узел с именем `flow-a` и две дополнительные страницы, `next_a1` и `next_a2`. Из `next_a2` пользователь может либо выйти из Flow, используя узел возврата `taskFlowReturn1`, либо вызвать Flow B, передав два параметра. Flow A также определяет два входящих параметра, которые он может принимать из Flow B. Flow B идентичен Flow A во всём, за исключением имён и файлов. Каждый Flow также имеет связанный Managed-бин. Бинами являются `Flow_a_Bean` и `Flow_b_Bean`.

Упаковка Flows в приложение

Как правило, Flow упаковывается в веб-приложении с использованием структуры каталогов, когда Flow отделены друг от друга. Например, в каталоге `src/main/webapp` проекта Maven вы поместите файлы Facelets, которые находятся вне Flow, на верхний уровень, как обычно. Тогда файлы `webapp` каждого Flow будут находиться в отдельных каталах, а файлы Java будут находиться в `src/main/java`. Например, файлы для приложения, показанные на Рис. 16-1, могут выглядеть следующим образом:

```
src/main/webapp/
    index.xhtml
    return.xhtml
    WEB_INF/
        beans.xml
        web.xml
    flow-a/
        flow-a.xhtml
        next_a1.xhtml
        next_a2.xhtml
    flow-b/
        flow-b-flow.xml
        next_b1.xhtml
        next_b2.xhtml
src/main/java/ee/jakarta/tutorial/flowexample
            FlowA.java
            Flow_a_Bean.java
            Flow_b_Bean.java
```

В этом примере `flow-a` определяется программно в `FlowA.java`, а `flow-b` определяется файлом конфигурации `flow-b-flow.xml`.

Простейший Flow: пример simple-flow

Приложение `simple-flow` демонстрирует самые базовые строительные блоки приложения Faces Flows и иллюстрирует некоторые соглашения, облегчающие начало работы с итеративной разработкой с использованием Flow. Вы можете начать с простого примера, подобного этому, и продолжить его.

В этом примере приводится неявное определение Flow путём включения пустого файла конфигурации. Файл конфигурации с содержимым или аннотированный класс `@FlowDefinition` предоставляет явное определение Flow.

Исходный код для этого приложения находится в каталоге *`tut-install`*`/examples/web/jsf/simple-flow/`.

Расположение файла в примере `simple-flow` выглядит следующим образом:

```
src/main/webapp
    index.xhtml
    simple-flow-return.xhtml
    WEB_INF/
        web.xml
    simple-flow
        simple-flow-flow.xml
        simple-flow.xhtml
        simple-flow-page2.xhtml
```

В примере `simple-flow` есть пустой файл конфигурации, который по соглашению называется flow-name-`flow.xml`. Flow не требует какой-либо настройки по следующим причинам.

- Flow не вызывает другой Flow и не передаёт параметры другому Flow.
- Поток (flow) использует имена по умолчанию для первой страницы потока, *`flow-name`*`.xhtml`, и страницы возврата *`flow-name`*`-return.xhtml`.

В этом примере всего четыре страницы Facelets.

- `index.xhtml` — стартовая страница, которая почти ничего не содержит, кроме кнопки, перемещающей на первую страницу Flow:

```xml
<p><h:commandButton value="Enter Flow" action="simple-flow"/></p>
```

- `simple-flow.xhtml` и `simple-flow-page2.xhtml` — две страницы самого Flow. При отсутствии явного определения Flow страница, имя которой совпадает с именем Flow, считается начальным узлом Flow. В этом случае Flow называется `simple-flow`, поэтому предполагается, что начальным узлом Flow является страница `simple-flow.xhtml`. Начальный узел — это узел, к которому переходят при входе в Flow. Его можно рассматривать как домашнюю страницу Flow.

  Страница `simple-flow.xhtml` попросит вас ввести значение в области Flow и предоставит кнопку для перехода на следующую страницу Flow:

  ```xml
  <p>Value: <h:inputText id="input" value="#{flowScope.value}" /></p>

  <p><h:commandButton value="Next" action="simple-flow-page2" /></p>
  ```

  Вторая страница, которая может иметь любое имя, отображает значение в области видимости Flow и предоставляет кнопку для перехода на страницу возврата:

  ```xml
  <p>Value: #{flowScope.value}</p>

  <p><h:commandButton value="Return" action="simple-flow-return" /></p>
  ```

- `simple-flow-return.xhtml` — страница возврата. Страница возврата, которая по соглашению имеет имя *`flow-name`*`-return.xhtml`, должны быть расположены за пределами потока. На этой странице отображается значение в рамках потока, чтобы показать, что оно не имеет значения вне потока, и предоставляется ссылка, которая ведёт на страницу `index.xhtml`:

  ```xml
  <p>Value (should be empty):
      "<h:outputText id="output" value="#{flowScope.value}" />"</p>

  <p><h:link outcome="index" value="Back to Start" /></p>
  ```

Страницы Facelets используют только данные области видимости Flow, поэтому в примере не требуется Managed-бин.

Сборка, упаковка и развёртывание simple-flow в IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/web/jsf
   ```

4. Выберите каталог `simple-flow`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `simple-flow` и выберите **Сборка**.

   Эта команда собирает и упаковывает приложение в WAR-файл, `simple-flow.war`, который находится в каталоге `target`. Затем приложение развёртывается на сервере.

Сборка, упаковка и развёртывание simple-flow с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).

2. В окне терминала перейдите в:

```
tut-install/examples/web/jsf/simple-flow/
```

3. Введите следующую команду:

```
mvn install
```
SHELL

Эта команда собирает и упаковывает приложение в WAR-файл, `simple-flow.war`, который находится в каталоге `target`. Затем приложение развёртывается на сервере.

Запуск simple-flow

1. Введите следующий URL в браузере:

```
http://localhost:8080/simple-flow
```

2. На странице `index.xhtml` нажмите Enter Flow.
3. На первой странице Flow введите любую строку в поле «Значение», затем нажмите «Далее».
4. На второй странице Flow вы можете увидеть введённое вами значение. Нажмите Return.
5. На странице возврата пустая пара кавычек заключает в себе недоступное значение. Нажмите кнопку «Назад в начало», чтобы вернуться на страницу `index.xhtml`.

Приложение checkout-module

Приложение `checkout-module` значительно сложнее, чем `simple-flow`. Он показывает, как вы можете использовать функцию Faces Flows для реализации модуля оформления заказа для интернет-магазина.

Как и гипотетический пример на рис. 16-1, пример приложения содержит два Flow, каждый из которых может вызывать другой. Оба Flow имеют явные определения. Один Flow, `checkoutFlow`, определяется программно. Другой Flow, `joinFlow`, указан в файле конфигурации.

Исходный код для этого приложения находится в каталоге *`tut-install`*`/examples/web/jsf/checkout-module/`.

Для приложения `checkout-module` структура каталогов следующая (есть также каталог `src/main/webapp/resources` с таблицей стилей и изображением):

```
src/main/webapp/
    index.xhtml
    exithome.xhtml
    WEB_INF/
        beans.xml
        web.xml
    checkoutFlow/
        checkoutFlow.xhtml
        checkoutFlow2.xhtml
        checkoutFlow3.xhtml
        checkoutFlow4.xhtml
    joinFlow/
        joinFlow-flow.xml
        joinFlow.xhtml
        joinFlow2.xhtml
src/main/java/ee/jakarta/tutorial/checkoutmodule
    CheckoutBean.java
    CheckoutFlow.java
    CheckoutFlowBean.java
    JoinFlowBean.java
```

Например, `index.xhtml` является начальной страницей для приложения, а также узлом возврата для Flow оформления заказа. Страница `exithome.xhtml` является узлом возврата для Flow объединения.

Файл конфигурации `joinFlow-flow.xml` определяет Flow объединения, а исходный файл `CheckoutFlow.java` определяет Flow оформления.

Flow оформления содержит четыре страницы Facelets, а Flow объединения — две.

Managed-бины, охватываемые каждым Flow: `CheckoutFlowBean.java` и `JoinFlowBean.java`, тогда как `CheckoutBean.java` является вспомогательным бином для страницы `index.html`.

Страницы Facelets для checkout-module

Начальная страница для примера `index.xhtml` отображает содержимое гипотетической корзины покупок. Это позволяет пользователю нажать любую из двух кнопок, чтобы войти в один из двух Flow:

XML

```
<p><h:commandButton value="Check Out" action="checkoutFlow"/></p>
...
<p><h:commandButton value="Join" action="joinFlow"/></p>
```

Эта страница также является возвратным узлом для Flow оформления заказа.

Страница Facelets `exithome.xhtml` является узлом возврата для Flow объединения. На этой странице есть кнопка, позволяющая вернуться на страницу `index.xhtml`.

Четыре страницы Facelets во Flow оформления заказа, начиная с `checkoutFlow.xhtml` и заканчивая `checkoutFlow4.xhtml`, позволяют перейти на следующую страницу или, в некоторых случаях, вернуться из Flow. Страница `checkoutFlow.xhtml` позволяет получить доступ к параметрам, переданным из Flow объединения через область видимости Flow. Они отображаются в виде пустых кавычек, если вы не вызвали Flow оформления из Flow объединения.

XML

```
<p>If you called this flow from the Join flow, you can see these parameters:
    "<h:outputText value="#{flowScope.param1Value}"/>" and
    "<h:outputText value="#{flowScope.param2Value}"/>"
</p>
```

Только `checkoutFlow2.xhtml` имеет кнопку для возврата на предыдущую страницу, но перемещение между страницами обычно разрешено во Flow. Вот кнопки для `checkoutFlow2.xhtml`:

```xml
<p><h:commandButton value="Continue" action="checkoutFlow3"/></p>
<p><h:commandButton value="Go Back" action="checkoutFlow"/></p>
<p><h:commandButton value="Exit Flow" action="returnFromCheckoutFlow"/></p>
```

Действие `returnFromCheckoutFlow` определено в файле исходного кода `CheckoutFlow.java`.

Последняя страница Flow оформления `checkoutFlow4.xhtml` содержит кнопку, которая вызывает Flow объединения:

```xml
<p><h:commandButton value="Join" action="calljoin"/></p>
<p><h:commandButton value="Exit Flow" action="returnFromCheckoutFlow"/></p>
```

Действие `calljoin` также определено в файле исходного кода конфигурации, `CheckoutFlow.java`. Это действие входит во Flow объединения, получая два параметра из Flow оформления.

Две страницы во Flow объединения, `joinFlow.xhtml` и `joinFlow2.xhtml`, аналогичны тем, которые находятся во Flow оформления. На второй странице есть кнопка для вызова Flow оформления заказа и кнопка для возврата из Flow объединения:

```xml
<p><h:commandButton value="Check Out" action="callcheckoutFlow"/></p>
<p><h:commandButton value="Exit Flow" action="returnFromJoinFlow"/></p>
```

Для этого Flow действия `callcheckoutFlow` и `returnFromJoinFlow` определены в файле конфигурации `joinFlow-flow.xml`.

Использование файла конфигурации для настройки Flow

Если для настройки потока используется файл ресурсов конфигурации приложения, он должен иметь имя *`flowName`*`-flow.xml`. В этом примере поток объединения использует файл конфигурации с именем `joinFlow-flow.xml`. Это файл `faces-config`, который определяет элемент `flow-definition`. Этот элемент должен определять имя потока с помощью атрибута `id`. Под элементом `flow-definition` должен быть элемент `flow-return`, который определяет точку возврата для потока. Все входящие параметры указываются с элементами `inbound-parameter`. Если поток вызывает другой поток, элемент `call-flow` должен использовать элемент ссылки на поток для имени вызываемого потока и может использовать элемент `outbound-parameter` для указания любого исходящего параметра.

Файл конфигурации для Flow объединения выглядит следующим образом:

XML

```xml
<faces-config version="3.0" xmlns="https://jakarta.ee/xml/ns/jakartaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
              https://jakarta.ee/xml/ns/jakartaee/web-facesconfig_3_0.xsd">

    <flow-definition id="joinFlow">
        <flow-return id="returnFromJoinFlow">
            <from-outcome>#{joinFlowBean.returnValue}</from-outcome>
        </flow-return>

        <flow-call id="callcheckoutFlow">
            <flow-reference>
                <flow-id>checkoutFlow</flow-id>
            </flow-reference>
            <outbound-parameter>
                <name>param1FromJoinFlow</name>
                <value>#{"param1 joinFlow value"}</value>
            </outbound-parameter>
            <outbound-parameter>
                <name>param2FromJoinFlow</name>
                <value>#{"param2 joinFlow value"}</value>
            </outbound-parameter>
        </flow-call>
        <inbound-parameter>
            <name>param1FromCheckoutFlow</name>
            <value>#{flowScope.param1Value}</value>
        </inbound-parameter>
        <inbound-parameter>
            <name>param2FromCheckoutFlow</name>
            <value>#{flowScope.param2Value}</value>
        </inbound-parameter>
    </flow-definition>
</faces-config>
```

Атрибут `id` элемента `flow-definition` определяет имя Flow как `joinFlow`. Значение атрибута `id` элемента `flow-return` идентифицирует имя возвращаемого узла, а его значение определяется в `from-outcome` элемент как свойство `returnValue` Managed-бина области видимости Flow для Flow объединения `JoinFlowBean`.

Имена и значения входных параметров извлекаются из области видимости Flow в следующем порядке (`flowScope.param1Value`, `flowScope.param2Value`), согласно тому, как это было определено в и конфигурации Flow оформления.

Элемент `flow-call` определяет, как Flow объединения вызывает Flow оформления. Атрибут `id` элемента `callcheckoutFlow` определяет действие вызова Flow. В элементе `flow-call` элемент `flow-reference` определяет фактическое имя вызываемого Flow, `checkoutFlow`. Элементы `outbound-parameter` определяют параметры, которые будут переданы при вызове `checkoutFlow`. Здесь это просто произвольные строки.

Использование Java-класса для настройки Flow

Если вы используете класс Java для настройки Flow, он должен совпадать с Flow по имени. Класс для Flow оформления называется `CheckoutFlow.java`.

JAVA

```java
import java.io.Serializable;
import jakarta.enterprise.inject.Produces;
import jakarta.faces.flow.Flow;
import jakarta.faces.flow.builder.FlowBuilder;
import jakarta.faces.flow.builder.FlowBuilderParameter;
import jakarta.faces.flow.builder.FlowDefinition;

class CheckoutFlow implements Serializable {

    private static final long serialVersionUID = 1L;

    @Produces
    @FlowDefinition
    public Flow defineFlow(@FlowBuilderParameter FlowBuilder flowBuilder) {

        String flowId = "checkoutFlow";
        flowBuilder.id("", flowId);
        flowBuilder.viewNode(flowId,
                "/" + flowId + "/" + flowId + ".xhtml").
                markAsStartNode();

        flowBuilder.returnNode("returnFromCheckoutFlow").
                fromOutcome("#{checkoutFlowBean.returnValue}");

        flowBuilder.inboundParameter("param1FromJoinFlow",
                "#{flowScope.param1Value}");
        flowBuilder.inboundParameter("param2FromJoinFlow",
                "#{flowScope.param2Value}");

        flowBuilder.flowCallNode("calljoin").flowReference("", "joinFlow").
                outboundParameter("param1FromCheckoutFlow",
                    "#{checkoutFlowBean.name}").
                outboundParameter("param2FromCheckoutFlow",
                    "#{checkoutFlowBean.city}");
        return flowBuilder.getFlow();
    }
}
```

Класс выполняет действия, практически идентичные тем, которые выполняются файлом конфигурации `joinFlow-flow.xml`. Он содержит единственный метод `defineFlow`, как метод производителя с квалификатором `@FlowDefinition`, возвращающий класс `jakarta.faces.flow.Flow`. Метод `defineFlow` принимает один параметр — `FlowBuilder` с квалификатором `@FlowBuilderParameter`, который передаётся из Jakarta Faces. Затем метод вызывает методы из класса `jakarta.faces.flow.Builder.FlowBuilder` для настройки потока.

Во-первых, метод определяет `id` Flow как `checkoutFlow`. Затем он явно определяет начальный узел для Flow. По умолчанию это имя Flow с суффиксом `.xhtml`.

Затем метод определяет возвратный узел аналогично тому, как это делает файл конфигурации. Метод `returnNode` устанавливает имя возвращаемого узла как `returnFromCheckoutFlow`, а метод `fromOutcome` указывает его значение как свойство `returnValue` Managed-бина во Flow оформления, `CheckoutFlowBean`.

Метод `inboundParameter` устанавливает имена и значения входных параметров из Flow объединения, которые извлекаются из области видимости Flow в следующем порядке (`flowScope.param1Value`, `flowScope.param2Value`), согласно тому, как определено в файле конфигурации Flow объединения.

Метод `flowCallNode` определяет, как Flow оформления вызывает Flow объединения. Аргумент `calljoin` определяет действие вызова Flow. Связанный метод `flowReference` определяет фактическое имя вызываемого Flow, `joinFlow`, а затем вызывает методы `outboundParameter` для определения параметров, передаваемых при вызове `joinFlow`. Здесь это значения из Managed-бина `CheckoutFlowBean`.

Наконец, метод `defineFlow` вызывает метод `getFlow` и возвращает результат.

Область видимости Flow для Managed-бина

Каждый из двух Flow имеет Managed-бин, который определяет свойства страниц в Flow. Например, `CheckoutFlowBean` определяет свойства, значения которых вводятся пользователем как на странице `checkoutFlow.xhtml`, так и на странице `checkoutFlow3.xhtml`.

Каждый Managed-бин имеет метод `getReturnValue`, который устанавливает значение возвращаемого узла. Для `CheckoutFlowBean` возвратным узлом является страница `index.xhtml`, указанная с помощью неявной навигации:

JAVA

```
public String getReturnValue() {
    return "index";
}
```

Для `JoinFlowBean` возвратным узлом является страница `exithome.xhtml`.

Сборка, упаковка и развёртывание checkout-module в IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/web/jsf
   ```

4. Выберите каталог `checkout-module`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `checkout-module` и выберите **Сборка**.

   Эта команда собирает и упаковывает приложение в WAR-файл, `checkout-module.war`, который находится в каталоге `target`. Затем приложение развёртывается на сервере.

Сборка, упаковка и развёртывание checkout-module с помощью Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/web/jsf/checkout-module/
   ```

3. Введите следующую команду:

   SHELL

   ```
   mvn install
   ```

   Эта команда собирает и упаковывает приложение в WAR-файл, `checkout-module.war`, который находится в каталоге `target`. Затем приложение развёртывается на сервере.

Запуск checkout-module

1. Введите следующий URL в браузере:

   ```
   http://localhost:8080/checkout-module
   ```

2. На странице `index.xhtml` представлены гипотетические результаты похода за покупками. Нажмите «Check Out» или «Join», чтобы войти в один из двух Flow.
3. Следуйте за Flow, предоставляя входные данные по мере необходимости и выбирая, продолжить, вернуться назад или выйти из Flow.

   Во Flow оформления заказа проверяется только одно из полей ввода (поле кредитной карты рассчитано на 16 цифр), поэтому можете ввести любые значения. Flow объединения не требует от вас отмечать какие-либо флажки в меню флажков.
4. На последней странице Flow выберите опцию для ввода другого Flow. Это позволяет просматривать входные параметры из предыдущего Flow.
5. Поскольку Flow являются вложенными, если вы нажмёте Exit Flow из вызываемого Flow, то вернётесь к первой странице вызывающего Flow. (Возможно появится предупреждение, которое можно игнорировать.). Нажмите «Выход из Flow» на этой странице, чтобы перейти к указанному возвратному узлу.

Настройка Managed-бинов

Когда страница обращается к Managed-бину в первый раз, Jakarta Faces инициализирует его либо на основе аннотации `@Named` и области видимости в классе компонента, либо в соответствии с конфигурацией в файле конфигурации приложения. Для получения информации об использовании аннотаций для инициализации бинов см. Использование аннотаций для настройки Managed-бинов.

Вы можете использовать аннотации или файл конфигурации приложения для создания объектов Managed-бинов, которые используются в приложении Jakarta Faces, и для сохранения их в области видимости. Средство создания Managed-бина настраивается в файле конфигурации приложения с использованием XML-элементов `managed-bean` для определения каждого компонента. Этот файл обрабатывается во время запуска приложения. Для получения информации об использовании этого средства см. Использование элемента managed-bean.

Managed-бины, созданные в файле конфигурации приложения, управляются Jakarta Faces, но не CDI.

С помощью средства создания Managed-бина вы можете

- Создавать бины в одном централизованном файле, который доступен всему приложению, а не создавайте условные объекты бинов по всему приложению
- Настроить свойства бина без дополнительного кода
- Настроить значения свойств компонента напрямую из файла конфигурации, чтобы он инициализировался этими значениями при его создании
- Используя элементы `value` , установить свойство одного Managed-бина в качестве результата вычисления другого выражения значения

В этом разделе показано, как инициализировать компоненты с помощью средства создания Managed-бинов. Смотрите Запись свойств бинов и Пишем методы Managed-бинов для получения информации о программировании Managed-бинов.

Использование элемента managed-bean

Managed-бин инициируется в файле конфигурации приложения с помощью элемента `managed-bean` , который представляет объект класса компонента, который должен существовать в приложении. Во время выполнения Jakarta Faces обрабатывает элемент `managed-bean` . Если страница ссылается на компонент, а

объект компонента не существует, Jakarta Faces создаёт объект компонента, как указано в конфигурации элемента.

Вот пример конфигурации Managed-бина из примера Duke's Bookstore:

```xml
<managed-bean eager="true">
    <managed-bean-name>Book201</managed-bean-name>
    <managed-bean-class>dukesbookstore.model.ImageArea</managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
    <managed-property>
        <property-name>shape</property-name>
        <value>rect</value>
    </managed-property>
    <managed-property>
        <property-name>alt</property-name>
        <value>Duke</value>
    </managed-property>
    <managed-property>
        <property-name>coords</property-name>
        <value>67,23,212,268</value>
    </managed-property>
</managed-bean>
```

Элемент `managed-bean-name` определяет ключ, под которым компонент будет храниться в области видимости. Чтобы значение компонента отображалось на этот бин, атрибут `value` тега компонента должен соответствовать `managed-bean-name` до первой точки.

Элемент `managed-bean-class` определяет полное имя класса компонента JavaBeans, используемого для создания объекта бина.

Элемент `managed-bean` может содержать любое количество элементов `managed-property`, каждый из которых соответствует свойству, определённому в классе бина. Эти элементы используются для инициализации значений свойств компонента. Если вы не хотите, чтобы конкретное свойство инициализировалось значением при создании объекта компонента, не включайте определение `managed-property` для него в файл конфигурации приложения.

Если элемент `managed-bean` не содержит других элементов `managed-bean`, он может содержать один элемент `map-entries` или `list-entries`. Элемент `map-entries` настраивает набор бинов, которые являются объектами `Map`. Элемент `list-entries` настраивает набор бинов, которые являются объектами `List`.

В следующем примере Managed-бин `newsletters`, представляющий компонент `UISelectItems`, настроен как `ArrayList`, представляющий набор объектов `SelectItem`. Каждый объект `SelectItem` в свою очередь настроен как Managed-бин со свойствами:

```xml
<managed-bean>
    <managed-bean-name>newsletters</managed-bean-name>
    <managed-bean-class>java.util.ArrayList</managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
    <list-entries>
        <value-class>jakarta.faces.model.SelectItem</value-class>
        <value>#{newsletter0}</value>
        <value>#{newsletter1}</value>
        <value>#{newsletter2}</value>
        <value>#{newsletter3}</value>
    </list-entries>
</managed-bean>
<managed-bean>
    <managed-bean-name>newsletter0</managed-bean-name>
    <managed-bean-class>jakarta.faces.model.SelectItem</managed-bean-class>
    <managed-bean-scope>none</managed-bean-scope>
    <managed-property>
        <property-name>label</property-name>
        <value>Duke's Quarterly</value>
    </managed-property>
    <managed-property>
        <property-name>value</property-name>
        <value>200</value>
    </managed-property>
</managed-bean>
...
```

Этот подход может быть полезен для быстрого создания списков выбора элементов до того, как группа разработчиков успеет создать такие списки из базы данных. Обратите внимание, что каждый из отдельных бинов рассылки имеет параметр `managed-bean-scope` со значением `none`, поэтому бины не будут иметь собственной области видимости.

Смотрите Инициализация свойств-массивов и списков для получения дополнительной информации о настройке коллекций в качестве бинов.

Для сопоставления со свойством, определённым элементом `managed-property`, необходимо убедиться, что часть выражения `value` тега компонента после точки совпадает с дочерним элементом `managed-property` — элементом `property-name`. В следующем разделе, Инициализация свойств с использованием элемента managed-property, более подробно объясняется, как использовать элемент `managed-property`. См. Инициализация свойств Managed-бина для примера инициализации свойства Managed-бина.

Инициализация свойств с использованием элемента managed-property

Элемент `managed-property` должен содержать элемент `property-name`, который должен соответствовать имени соответствующего свойства в бине. Элемент `managed-property` также должен содержать один элемент из набора, который определяет значение свойства. Это значение должно быть того же типа, который определён для свойства в соответствующем компоненте. Какой элемент вы используете для определения значения, зависит от типа свойства, определённого в бине. Таблица 16-1 перечисляет все элементы, которые используются для инициализации значения.

Таблица 16-1. Подэлементы элементов управляемого свойства, которые определяют значения свойств

Элемент	**Какое значение определяет**
`list-entries`	Определяет значения в списке
`map-entries`	Определяет значения отображения (Map)

Элемент	Какое значение определяет
`null-value`	Явно устанавливает для свойства значение `null`
`value`	Определяет одно значение, например выражение `String`, `int` или EL Jakarta Faces

Использование элемента managed-bean включает в себя пример инициализации свойства `int` (примитивного типа) с использованием дочернего элемента `value`. Вы также можете использовать дочерний элемент `value` для инициализации `String` и других объектных типов. В оставшейся части этого раздела описано, как использовать дочерний элемент `value` и другие дочерние элементы для инициализации свойств типов Java `Enum`, `Map`, `array` и `Collection`, а также параметры инициализации.

Ссылка на перечислимый тип Java (Enum)

Свойство Managed-бина также может быть Java `Enum` (см. https://docs.oracle.com/javase/8/docs/api/java/lang/Enum.html). В этом случае элемент `value` элемента `managed-property` должен быть строкой, которая соответствует одной из строковых констант в `Enum`. Другими словами, указанная строка должна быть среди значений, с которыми вызов `valueOf(Class, String)` у `enum` возвращает корректное значение. В этом выражении `Class` — это класс `Enum`, а `String` — содержимое вложенного элемента `value`. Для примера предположим, что свойство Managed-бина является следующим:

```java
public enum Suit { Hearts, Spades, Diamonds, Clubs }
 ...
public Suit getSuit() { ... return Suit.Hearts; }
```

Предполагая, что вы хотите настроить это свойство в файле конфигурации приложения, соответствующий элемент `managed-property` выглядит следующим образом:

```xml
<managed-property>
    <property-name>Suit</property-name>
    <value>Hearts</value>
</managed-property>
```

Когда система сталкивается с этим свойством, она выполняет итерацию по каждому члену `enum` и вызывает `toString()` для каждого члена, пока не найдёт тот, который точно равен значению элемента `value`.

Ссылка на контекстный параметр инициализации

Ещё одна мощная функция средства создания Managed-бина — это возможность ссылаться на неявные объекты из свойства Managed-бина.

Предположим, у вас есть страница, которая принимает данные от клиента, включая адрес клиента. Предположим также, что большинство ваших клиентов живут в городе с определённым кодом. Вы можете заставить компонент кода города отображать этот код, сохраняя его в неявном объекте и ссылаясь на него при отрисовке страницы.

Вы можете сохранить код города в качестве начального значения по умолчанию в контексте неявного объекта `initParam`, добавив контекстный параметр в ваше веб-приложение и установив его значение в дескрипторе развёртывания. Например, чтобы задать контекстному параметру с именем `defaultAreaCode` значение `650`, добавьте элемент `context-param` в дескриптор развёртывания и присвойте параметру имя `defaultAreaCode` и значение `650`.

Затем напишите объявление `managed-bean`, которое настраивает свойство, ссылающееся на параметр:

```xml
<managed-bean>
    <managed-bean-name>customer</managed-bean-name>
        <managed-bean-class>CustomerBean</managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
        <managed-property>
            <property-name>areaCode</property-name>
                <value>#{initParam.defaultAreaCode}</value>
            </managed-property>
            ...
</managed-bean>
```

Чтобы получить доступ к коду города во время отображения страницы, обратитесь к свойству из атрибута `value` тега компонента `area`:

```xml
<h:inputText id=area value="#{customer.areaCode}" />
```

Значения из других неявных объектов извлекаются аналогично.

Инициализация свойств-отображений (Map)

Элемент `map-entries` используется для инициализации значений свойства компонента с типом `Map`, если элемент `map-entries` используется внутри элемента `managed-property`. Элемент `map-entries` содержит необязательный элемент `key-class`, необязательный элемент `value-class` и любое количество `map-entry`.

Каждый из элементов `map-entry` должен содержать элемент `key` и элемент `null-value` или `value`. Вот пример, в котором используется элемент `map-entries`:

```xml
<managed-bean>
    ...
    <managed-property>
        <property-name>prices</property-name>
        <map-entries>
            <map-entry>
                <key>My Early Years: Growing Up on *7</key>
                <value>30.75</value>
            </map-entry>
            <map-entry>
                <key>Web Servers for Fun and Profit</key>
                <value>40.75</value>
            </map-entry>
        </map-entries>
    </managed-property>
</managed-bean>
```

Отображение, созданное этим тегом `map-entries`, содержит две записи. По умолчанию все ключи и значения преобразуются в `String`. Если вы хотите указать другой тип для ключей на карте, вставьте элемент `key-class` прямо внутри элемента `map-entries`:

```xml
<map-entries>
    <key-class>java.math.BigDecimal</key-class>
    ...
</map-entries>
```

Это объявление преобразует все ключи в `java.math.BigDecimal`. Конечно, вы должны убедиться, что ключи могут быть преобразованы в указанный тип. Ключ из примера в этом разделе не может быть преобразован в `BigDecimal`, потому что это `String`.

Если вы хотите указать другой тип для всех значений на карте, включите элемент `value-class` после элемента `key-class`:

```xml
<map-entries>
    <key-class>int</key-class>
    <value-class>java.math.BigDecimal</value-class>
    ...
</map-entries>
```

Обратите внимание, что этот тег устанавливает только тип всех подэлементов `value`.

Каждая `map-entry` в предыдущем примере включает в себя подэлемент `value`. Подэлемент `value` определяет одно значение, которое будет преобразовано в тип, указанный в бине.

Вместо использования элемента `map-entries` также можно назначить всю карту, используя элемент `value`, который задаёт выражение с типом карты.

Инициализация свойств-массивов и списков

Элемент `list-entries` используется для инициализации значений массива или свойства `List`. Каждое отдельное значение массива или `List` инициализируется с использованием элемента `value` или `null-value`. Вот пример:

```xml
<managed-bean>
    ...
    <managed-property>
        <property-name>books</property-name>
        <list-entries>
            <value-class>java.lang.String</value-class>
            <value>Web Servers for Fun and Profit</value>
            <value>#{myBooks.bookId[3]}</value>
            <null-value/>
        </list-entries>
    </managed-property>
</managed-bean>
```

В этом примере инициализируется массив или `List`. Тип соответствующего свойства в компоненте определяет, какая структура данных создаётся. Элемент `list-entries` определяет список значений в массиве или `List`. Элемент `value` указывает одно значение в массиве или `List` и может ссылаться на свойство в другом бине. Элемент `null-value` будет вызывать метод `setBooks` с аргументом `null`. Свойство `null` нельзя указывать для свойства, тип данных которого является примитивом Java, например `int` или `boolean`.

Инициализация свойств Managed-бина

Иногда может понадобиться создать бин, который также ссылается на другие Managed-бины, чтобы можно было построить граф или дерево бинов. Для примера предположим, что вы хотите создать компонент, представляющий информацию о клиенте, включая почтовый адрес и адрес улицы, каждый из которых также является компонентом. Следующие объявления `managed-bean` создают объект `CustomerBean`, который имеет два свойства `AddressBean`: одно представляет почтовый адрес, а другое — адрес улицы. Результатом этого объявления является дерево компонентов с `CustomerBean` в качестве его родителя и двумя объектами `AddressBean` в качестве дочерних.

```xml
<managed-bean>
    <managed-bean-name>customer</managed-bean-name>
    <managed-bean-class>
        com.example.mybeans.CustomerBean
    </managed-bean-class>
    <managed-bean-scope> request </managed-bean-scope>
    <managed-property>
        <property-name>mailingAddress</property-name>
        <value>#{addressBean}</value>
    </managed-property>
    <managed-property>
        <property-name>streetAddress</property-name>
        <value>#{addressBean}</value>
    </managed-property>
    <managed-property>
        <property-name>customerType</property-name>
        <value>New</value>
    </managed-property>
</managed-bean>
<managed-bean>
    <managed-bean-name>addressBean</managed-bean-name>
    <managed-bean-class>
        com.example.mybeans.AddressBean
    </managed-bean-class>
    <managed-bean-scope> none </managed-bean-scope>
    <managed-property>
        <property-name>street</property-name>
        <null-value/>
    <managed-property>
    ...
</managed-bean>
```

Первое объявление `CustomerBean` (с `managed-bean-name` из `customer`) создаёт `CustomerBean` в области видимости запроса. Этот бин имеет два свойства: `mailingAddress` и `streetAddress`. Эти свойства используют элемент `value` для ссылки на компонент с именем `addressBean`.

Второе объявление Managed-бина определяет `AddressBean`, но не создаёт его, потому что его элемент `managed-bean-scope` определяет область видимости `none`. Напомним, что область видимости `none` означает, что компонент создаётся только тогда, когда на него ссылается что-то другое. Поскольку свойства `mailingAddress` и `streetAddress` ссылаются на `addressBean`, используя элемент `value`, два объекта `AddressBean` создаются при создании `CustomerBean`.

Когда вы создаёте объект, который указывает на другие объекты, не пытайтесь указывать на объект с более коротким сроком службы, потому что может быть невозможно восстановить ресурсы этой области видимости, когда объект будет удалён. Например, объект в области видимости сессии не может указывать на объект в области видимости запроса. А объекты с областью видимости `none` не имеют чётко заданной продолжительности жизни, управляемой фреймворком, поэтому они могут указывать только на другие объекты с областью видимости `none`. Таблица 16-2 описывает все разрешённые соединения.

Таблица 16-2. Допустимые соединения между объектами в области видимости

Объект этой области	**Может указывать на объект этой области**
none	none
application	none, application

Объект этой области	Может указывать на объект этой области
`session`	`none`, `application`, `session`
`request`	`none`, `application`, `session`, `request`, `view`
`view`	`none`, `application`, `session`, `view`

Не допускайте циклических ссылок между объектами. Например, ни один из объектов `AddressBean` в предыдущем примере не должен указывать на объект `CustomerBean`, поскольку `CustomerBean` уже указывает на `AddressBean` объекты.

Инициализация отображений и списков

Помимо настройки свойств `Map` и `List`, вы также можете напрямую настроить `Map` и `List`, чтобы ссылаться на них из тега, а не через свойство, которое переносит `Map` или `List`.

Регистрация сообщений приложения

Сообщения приложений могут содержать любые строки, отображаемые пользователю, а также настраиваемые сообщения об ошибках (которые отображаются тегами `message` и `messages`) для ваших кастомных конвертеров или валидаторов. Чтобы сделать сообщения доступными во время запуска приложения, выполните одно из следующих действий:

- Поставить отдельное сообщение в очередь на `jakarta.faces.context.FacesContext` программно, как описано в Использование FacesMessage для создания сообщения
- Зарегистрируйте все сообщения в вашем приложении, используя файл конфигурации приложения.

Вот раздел файла `faces-config.xml`, в котором регистрируются сообщения для программы Duke's Bookstore:

XML

```
<application>
    <resource-bundle>
        <base-name>
            ee.jakarta.tutorial.dukesbookstore.web.messages.Messages
        </base-name>
        <var>bundle</var>
    </resource-bundle>
    <locale-config>
        <default-locale>en</default-locale>
        <supported-locale>es</supported-locale>
        <supported-locale>de</supported-locale>
        <supported-locale>fr</supported-locale>
    </locale-config>
</application>
```

Этот набор элементов вызывает заполнение приложения сообщениями, содержащимися в указанном bundle-ресурсе.

Элемент `resource-bundle` представляет набор локализованных сообщений. Он должен содержать полный путь к bundle-ресурсу, содержащему локализованные сообщения (в этом случае `dukesbookstore.web.messages.Messages`). Элемент `var` определяет имя EL, под которым авторы страниц ссылаются на bundle-ресурс.

Элемент `locale-config` перечисляет локаль по умолчанию и другие поддерживаемые локали. Элемент `locale-config` позволяет системе найти правильную локаль на основе языковых настроек браузера.

`supported-locale` и `default-locale` принимают двухсимвольные коды в нижнем регистре, определённые ISO 639-1 (см. https://www.loc.gov/standards/iso639-2/php/English_list.php). Убедитесь, что bundle-ресурс действительно содержит сообщения для локалей, указанных с помощью этих тегов.

Чтобы получить доступ к локализованному сообщению, разработчик приложения просто ссылается на ключ сообщения из bundle-ресурса.

Можно извлечь локализованный текст в тег `alt` для графического изображения, как в следующем примере:

XML

```
<h:graphicImage id="mapImage"
                name="book_all.jpg"
                library="images"
                alt="#{bundle.ChooseBook}"
                usemap="#bookMap" />
```

Атрибут `alt` может принимать выражения значений. В этом случае атрибут `alt` относится к локализованному тексту, который будет включён в альтернативный текст изображения, представленного этим тегом.

Использование FacesMessage для создания сообщения

Вместо регистрации сообщений в файле конфигурации приложения вы можете получить доступ к `java.util.ResourceBundle` напрямую из кода Managed-бина. Фрагмент кода ниже находит сообщение об ошибке электронной почты:

JAVA

```
String message = "";
...
message = ExampleBean.loadErrorMessage(context,
    ExampleBean.EX_RESOURCE_BUNDLE_NAME,
         "EMailError");
context.addMessage(toValidate.getClientId(context),
    new FacesMessage(message));
```

Эти строки вызывают метод `loadErrorMessage` бина для получения сообщения из `ResoureBundle`. Вот метод `loadErrorMessage`:

JAVA

```
public static String loadErrorMessage(FacesContext context,
     String basename, String key) {
    if ( bundle == null ) {
         try {
            bundle = ResourceBundle.getBundle(basename,
                 context.getViewRoot().getLocale());
        } catch (Exception e) {
            return null;
        }
    }
    return bundle.getString(key);
}
```

Ссылки на сообщения об ошибках

Страница Jakarta Faces использует теги `message` или `messages` для доступа к сообщениям об ошибках, как описано в Отображение сообщений об ошибках тегами h:message и h:messages.

Сообщения об ошибках, к которым имеют доступ эти теги, включают

- Стандартные сообщения об ошибках, которые сопровождают стандартные конвертеры и валидаторы, которые поставляются с API. (см. Раздел 2.5.2.4 (https://jakarta.ee/specifications/faces/3.0/jakarta-faces-3.0.html#a584) спецификации Jakarta Faces для получения полного списка стандартных сообщений об ошибках).
- Кастомные сообщения об ошибках, содержащиеся в bundle-ресурсах, зарегистрированных в приложении архитектором приложения с помощью элемента `resource-bundle` в файле конфигурации

Когда конвертер или валидатор зарегистрирован в компоненте ввода, соответствующее сообщение об ошибке автоматически ставится в очередь на компоненте.

Автор страницы может переопределить сообщения об ошибках, поставленные в очередь в компоненте, используя следующие атрибуты тега компонента:

- `converterMessage` : ссылается на сообщение об ошибке, которое отображается, когда данные на содержащем его компоненте не могут быть преобразованы конвертером, зарегистрированным в этом компоненте.
- `requiredMessage` : ссылается на сообщение об ошибке, которое будет отображаться, если в содержащий его компонент не было введено никакого значения.
- `validatorMessage` : ссылается на сообщение об ошибке, которое отображается, когда данные содержащего его компонента выдают ошибку при проверке валидатором, зарегистрированным в этом компоненте.

Все три атрибута могут принимать реальные значения и выражения значений. Если атрибут использует выражение значения, это выражение ссылается на сообщение об ошибке в bundle-ресурсе. Этот bundle-ресурс можно сделать доступным для приложения одним из следующих способов:

- Разработчиком приложения, использующим элемент `resource-bundle` в файле конфигурации
- Автором страницы с помощью тега `f:loadBundle`

И наоборот, элемент `resource-bundle` должен использоваться для предоставления приложению тех bundle-ресурсов, содержащих кастомные сообщения об ошибках, которые ставятся в очередь в компоненте в результате регистрации в компоненте кастомного конвертера или валидатора.

Следующие теги показывают, как указать атрибут `requiredMessage` , используя выражение значения для ссылки на сообщение об ошибке:

```xml
<h:inputText id="ccno" size="19"
    required="true"
    requiredMessage="#{customMessages.ReqMessage}">
    ...
</h:inputText>
<h:message styleClass="error-message" for="ccno"/>
```

Выражение значения, используемое `requiredMessage` в этом примере, ссылается на сообщение об ошибке с ключом `ReqMessage` в bundle-ресурсе `customMessages` .

Это сообщение заменяет соответствующее сообщение, поставленное в очередь в компоненте, и будет отображаться везде, где на странице размещены теги `message` или `messages` .

Использование валидаторов по умолчанию

В дополнение к валидаторам, которые объявлются для компонентов, также можно указать валидаторы по умолчанию в файле конфигурации приложения. Валидатор по умолчанию применяется ко всем объектам `jakarta.faces.component.UIInput` в представлении или дереве компонентов и добавляется после локально определённых валидаторов. Вот пример валидатора по умолчанию, зарегистрированного в файле конфигурации приложения:

```xml
<faces-config>
    <application>
        <default-validators>
            <validator-id>jakarta.faces.Bean</validator-id>
        </default-validators>
    <application/>
</faces-config>
```

Регистрация кастомного валидатора

Если разработчик приложения предоставляет реализацию `jakarta.faces.validator.Validator` для выполнения валидации, вы должны зарегистрировать этот кастомный валидатор либо с помощью `@FacesValidator`, как описано в разделе Реализация интерфейса Validator, или с помощью XML-элемента `validator` в файле ресурсов конфигурации приложения:

```xml
<validator>
    ...
    <validator-id>FormatValidator</validator-id>
    <validator-class>
        myapplication.validators.FormatValidator
    </validator-class>
    <attribute>
        ...
        <attribute-name>formatPatterns</attribute-name>
        <attribute-class>java.lang.String</attribute-class>
    </attribute>
</validator>
```

Атрибуты, указанные в теге `validator`, переопределяют любые параметры в аннотации `@FacesValidator`.

Элементы `validator-id` и `validator-class` являются обязательными элементами. Элемент `validator-id` представляет идентификатор, под которым должен быть зарегистрирован класс `Validator`. Этот идентификатор используется классом тега, соответствующим кастомному тегу `validator`.

Элемент `validator-class` представляет полное имя класса `Validator`.

Элемент `attribute` идентифицирует атрибут, связанный с реализацией `Validator`. Элементы `attribute-name` и `attribute-class` обязательны. Элемент `attribute-name` ссылается на имя атрибута в том виде, в каком оно отображается в теге `validator`. Элемент `attribute-class` определяет полное имя класса Java, связанного с атрибутом.

Создание и использование кастомного валидатора объясняет, как реализовать интерфейс `Validator`.

Использование кастомного валидатора объясняет, как ссылаться на валидатор со страницы.

Регистрация кастомного конвертера

Как и в случае с кастомным валидатором, если разработчик приложения создаёт собственный конвертер, вы должны зарегистрировать его в приложении, используя аннотацию `@FacesConverter`, как описано в Создание кастомного конвертера или с помощью XML-элемента `converter` в файле конфигурации

приложения. Вот гипотетическая конфигурация `converter` для `CreditCardConverter` из примера Duke's Bookstore:

```xml
<converter>
    <description>
        Converter for credit card numbers that normalizes
        the input to a standard format
    </description>
    <converter-id>CreditCardConverter</converter-id>
    <converter-class>
        dukesbookstore.converters.CreditCardConverter
    </converter-class>
</converter>
```

Атрибуты, указанные в теге `converter` , переопределяют любые параметры в аннотации `@FacesConverter` .

Элемент `converter` представляет реализацию `jakarta.faces.convert.Converter` и содержит обязательные элементы `converter-id` и `converter-class` .

Элемент `converter-id` представляет собой идентификатор, который используется атрибутом `converter` тега компонента пользовательского интерфейса для применения конвертера к данным компонента. Использование кастомного конвертера включает в себя пример ссылки на кастомный конвертер из тега компонента.

Элемент `converter-class` указывает реализацию `Converter` .

Создание и использование кастомного конвертера объясняет, как создать кастомный конвертер.

Настройка правил навигации

Навигация между различными страницами приложения Jakarta Faces, например выбор следующей страницы, которая будет отображаться после клика кнопки или ссылки, определяется набором правил. Правила навигации могут быть неявными, либо они могут быть явно определены в файле конфигурации приложения. Для получения дополнительной информации о неявных правилах навигации см. Модель навигации.

Каждое правило навигации определяет, как перемещаться с одной страницы на другую или набор страниц. Jakarta Faces выбирает подходящее правило навигации в соответствии с тем, какая страница отображается в данный момент.

После того, как правило навигации выбрано, выбор той страницы, к которой следует перейти с текущей страницы, зависит от двух факторов:

- Метод действия, вызываемый при клике компонента
- Результат, на который ссылается тег компонента или который возвращается из метода действия

Результатом может быть всё, что выберет разработчик, но таблица 16-3 перечисляет некоторые результаты, обычно используемые в веб-приложениях.

Таблица 16-3 Строки общего результата

Результат	Что это означает

Результат	Что это означает
`success`	Всё работает как надо. Перейти на следующую страницу.
`failure`	Что-то не так. Перейти на страницу с ошибкой.
`login`	Сначала пользователь должен войти в систему. Перейти на страницу входа.
`no results`	Поиск не дал результатов. Снова перейти на страницу поиска.

Обычно метод действия выполняет некоторую обработку данных формы текущей страницы. Например, метод может проверить, соответствуют ли имя пользователя и пароль, введённые на форме, имени пользователя и паролю в файле. Если они совпадают, метод возвращает результат `success`. В противном случае он возвращает результат `failure`. Как показывает этот пример, и метод, используемый для обработки действия, и возвращаемый результат необходимы для определения подходящей страницы для перехода.

Вот правило навигации, которое можно использовать с только что описанным примером:

XML

```
<navigation-rule>
    <from-view-id>/login.xhtml</from-view-id>
    <navigation-case>
        <from-action>#{LoginForm.login}</from-action>
        <from-outcome>success</from-outcome>
        <to-view-id>/storefront.xhtml</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-action>#{LoginForm.logon}</from-action>
        <from-outcome>failure</from-outcome>
        <to-view-id>/logon.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

Это правило навигации определяет возможные способы навигации из `login.xhtml`. Каждый элемент `navigation-case` определяет один возможный путь навигации из `login.xhtml`. Первый `navigation-case` говорит, что если `LoginForm.login` вернёт результат `success`, то будет выполнен переход на `storefront.xhtml`. Второй `navigation-case` говорит, что `login.xhtml` будет перерисован, если `LoginForm.login` вернёт `failure`.

Конфигурация Flow страниц приложения состоит из набора правил навигации. Каждое правило определяется элементом `navigation-rule` в файле `faces-config.xml`.

Каждый элемент `navigation-rule` соответствует одному идентификатору дерева компонентов, определённому необязательным элементом `from-view-id`. Это означает, что каждое правило определяет все возможные способы перехода с одной конкретной страницы в приложении. Если элемент `from-view-id` отсутствует, правила навигации, определённые в элементе `navigation-rule`, применяются ко всем страницам приложения. Элемент `from-view-id` также позволяет использовать шаблоны подстановочных знаков. Например, этот элемент `from-view-id` говорит, что правило навигации применяется ко всем страницам в каталоге `books`:

```xml
<from-view-id>/books/*</from-view-id>
```

Элемент `navigation-rule` может содержать любое количество элементов `navigation-case`. Элемент `navigation-case` определяет набор критериев соответствия. Когда эти критерии будут выполнены, приложение перейдёт на страницу, определённую элементом `to-view-id`, содержащимся в том же элементе `navigation-case`.

Критерии навигации определяются необязательными элементами `from-result` и `from-action`. Элемент `from-result` определяет строковый результат, такой как `success`. Элемент `from-action` использует выражение метода для ссылки на метод действия, который возвращает `String`, что является результатом. Метод выполняет некоторую логику обработки и возвращает результат.

Элементы `navigation-case` проверяются по результату и выражению метода в следующем порядке.

1. Случаи, указывающие как значение `from-result`, так и значение `from-action`. Оба эти элемента могут использоваться, если метод действия возвращает разные результаты в зависимости от результата обработки, которую он выполняет.
2. Случаи, указывающие только значение `from-result`. Элемент `from-result` должен соответствовать либо результату, определённому атрибутом `action` компонента `jakarta.faces.component.UICommand`, либо возвращаемому результату методом, на который ссылается компонент `UICommand`.
3. Случаи, указывающие только значение `from-action`. Это значение должно соответствовать выражению `action`, указанному в теге компонента.

При совпадении любого из этих случаев для визуализации будет выбрано дерево компонентов, определённое элементом `to-view-id`.

Регистрация кастомного отрисовщика с помощью инструментария отрисовки (Render Kit)

Когда разработчик приложения создаёт кастомный отрисовщик, как описано в Делегирование отрисовки отрисовщику, он должен зарегистрировать его с помощью соответствующего инструментария отрисовки. Поскольку приложение карты изображения реализует карту изображения HTML, классы `AreaRenderer` и `MapRenderer` в примере Duke's Bookstore должны быть зарегистрированы с использованием инструментария отрисовки HTML.

Вы регистрируете отрисовщик либо используя аннотацию `@FacesRenderer`, как описано в Создание класса отрисовщика, либо используя элемент `render-kit` файла конфигурации приложения. Вот гипотетическая конфигурация `AreaRenderer`:

```xml
<render-kit>
    <renderer>
        <component-family>Area</component-family>
        <renderer-type>DemoArea</renderer-type>
        <renderer-class>
            dukesbookstore.renderers.AreaRenderer
        </renderer-class>
        <attribute>
            <attribute-name>onmouseout</attribute-name>
            <attribute-class>java.lang.String</attribute-class>
        </attribute>
        <attribute>
            <attribute-name>onmouseover</attribute-name>
            <attribute-class>java.lang.String</attribute-class>
        </attribute>
        <attribute>
            <attribute-name>styleClass</attribute-name>
            <attribute-class>java.lang.String</attribute-class>
        </attribute>
    </renderer>
    ...
```

Атрибуты, указанные в теге `renderer`, переопределяют любые параметры в аннотации `@FacesRenderer`.

Элемент `render-kit` представляет собой реализацию `jakarta.faces.render.RenderKit`. Если `render-kit-id` не указан, предполагается, что используется инструментарий отрисовки HTML по умолчанию. Элемент `renderer` представляет собой реализацию `jakarta.faces.render.Renderer`. Вложив элемент `renderer` в элемент `render-kit`, вы регистрируете отрисовщик с помощью реализации `RenderKit`, связанной с `render-kit`.

`renderer-class` — это полное имя класса `Renderer`.

Элементы `component-family` и `renderer-type` используются компонентом для поиска отрисовщиков, которые могут его отрисовать. Идентификатор `component-family` должен совпадать с идентификатором, возвращаемым методом `getFamily` класса компонента. Семейство компонентов представляет компонент или набор компонентов, которые может отображать конкретный отрисовщик. `renderer-type` должен совпадать с тем, который возвращается методом `getRendererType` класса обработчика тега.

Используя семейство компонентов и тип отрисовщика для поиска отрисовщиков для компонентов, Jakarta Faces позволяет компоненту отображаться несколькими отрисовщиками и позволяет отрисовщику отображать несколько компонентов.

Каждый из тегов `attribute` определяет зависящий от отрисовки атрибут и его тип. Элемент `attribute` не влияет на работу приложения во время выполнения. Скорее, он предоставляет инструментам разработки информацию об атрибутах, которые поддерживает `Renderer`.

Объект, отвечающий за отрисовщик компонента (будь то сам компонент или отрисовщик, которому компонент делегирует отрисовку), может использовать фасеты для помощи в процессе отрисовки. Эти аспекты позволяют разработчику кастомного компонента управлять некоторыми аспектами отрисовки компонента. Рассмотрите этот пример тега кастомного компонента:

XML

```
<d:dataScroller>
    <f:facet name="header">
        <h:panelGroup>
            <h:outputText value="Account Id"/>
            <h:outputText value="Customer Name"/>
            <h:outputText value="Total Sales"/>
        </h:panelGroup>
    </f:facet>
    <f:facet name="next">
        <h:panelGroup>
            <h:outputText value="Next"/>
            <h:graphicImage url="/images/arrow-right.gif" />
        </h:panelGroup>
    </f:facet>
    ...
</d:dataScroller>
```

Тег компонента `dataScroller` включает в себя компонент, который будет отображать заголовок, и компонент, который будет отображать кнопку «Далее». Если отрисовщик, связанный с этим компонентом, отображает фасеты, вы можете включить следующие элементы `facet` в элемент `renderer`:

XML

```
<facet>
    <description>This facet renders as the header of the table. It should be
         a panelGroup with the same number of columns as the data.
    </description>
    <display-name>header</display-name>
    <facet-name>header</facet-name>
</facet>
<facet>
    <description>This facet renders as the content of the "next" button in
         the scroller. It should be a panelGroup that includes an outputText
         tag that has the text "Next" and a right arrow icon.
    </description>
    <display-name>Next</display-name>
    <facet-name>next</facet-name>
</facet>
```

Если компонент, который поддерживает фасеты, обеспечивает свой собственный отрисовщик, и вы хотите включить элементы `facet` в файл конфигурации приложения, нужно поместить их в конфигурацию компонента, а не в конфигурацию отрисовщика.

Регистрация кастомного компонента

В дополнение к регистрации кастомных отрисовщиков (как описано в предыдущем разделе), вы также должны зарегистрировать кастомные компоненты, которые обычно связаны с кастомными отрисовщиками. Используйте аннотацию `@FacesComponent`, как описано в Создание кастомных классов компонентов, или элемент `component` файла конфигурации приложения.

Вот гипотетический элемент `component` из файла конфигурации приложения, который регистрирует `AreaComponent`:

```xml
<component>
    <component-type>DemoArea</component-type>
    <component-class>
        dukesbookstore.components.AreaComponent
    </component-class>
    <property>
        <property-name>alt</property-name>
        <property-class>java.lang.String</property-class>
    </property>
    <property>
        <property-name>coords</property-name>
        <property-class>java.lang.String</property-class>
    </property>
    <property>
        <property-name>shape</property-name>
        <property-class>java.lang.String</property-class>
    </property>
</component>
```

Атрибуты, указанные в теге `component`, переопределяют любые параметры в аннотации `@FacesComponent`.

Элемент `component-type` указывает имя, под которым должен быть зарегистрирован компонент. Другие объекты ссылаются на этот компонент по этому имени. Например, элемент `component-type` в конфигурации для `AreaComponent` определяет значение `DemoArea`, которое соответствует значению, возвращённому методом `getComponentType` класса `AreaTag`.

Элемент `component-class` указывает полное имя класса компонента. Элементы `property` определяют свойства компонента и их типы.

Если кастомный компонент может включать фасеты, вы можете настроить фасеты в конфигурации компонента, используя элементы `facet`, которые разрешены после элементов `component-class`. Смотрите Регистрация кастомного отрисовщика с помощью инструментария отрисовки для получения дополнительной информации о настройке фасетов.

Основные требования к приложениям Jakarta Faces

В дополнение к настройке приложения вы должны удовлетворить другие требования к приложениям Jakarta Faces, включая правильную упаковку всех необходимых файлов и предоставление дескриптора развёртывания. В этом разделе описывается, как выполнять эти административные задачи.

Приложения Jakarta Faces могут быть упакованы в WAR-файл, который должен соответствовать определённым требованиям для выполнения в разных контейнерах. Как минимум, WAR-файл для приложения Jakarta Faces должен содержать следующее:

- Дескриптор развёртывания веб-приложения, который называется `web.xml`, для настройки ресурсов, необходимых веб-приложению (обязательно)
- Определённый набор JAR-файлов, содержащих необходимые классы (необязательно)
- Набор классов приложений, страниц Jakarta Faces и других необходимых ресурсов, таких как файлы изображений

WAR-файл также может содержать:

- Файл конфигурации приложения, который настраивает ресурсы приложения
- Набор файлов дескрипторов библиотеки тегов

Например, WAR-файл веб-приложения Jakarta Faces, использующий Facelets, обычно имеет следующую структуру каталогов:

```
$PROJECT_DIR
[Web Pages]
+- /[xhtml or html documents]
+- /resources
+- /WEB-INF
   +- /web.xml
   +- /beans.xml (optional)
   +- /classes (optional)
   +- /lib (optional)
   +- /faces-config.xml (optional)
   +- /*.taglib.xml (optional)
   +- /glassfish-web.xml (optional)
```

Файл `web.xml` (дескриптор развёртывания), набор JAR-файлов и набор файлов приложения должны содержаться в каталоге `WEB-INF` WAR-файла.

Настройка веб-приложения с помощью дескриптора развёртывания

Веб-приложения обычно настраиваются с использованием элементов, содержащихся в дескрипторе развёртывания веб-приложения `web.xml` . Дескриптор развёртывания для приложения Jakarta Faces должен указывать определённые конфигурации, включая следующие:

- Сервлет, используемый для обработки запросов Jakarta Faces
- Отображение сервлета для обработки сервлета
- Путь к файлу конфигурации, если он существует и не находится в расположении по умолчанию

Дескриптор развёртывания может также включать в себя другие, необязательные конфигурации, которые

- Указывают, где сохраняется состояние компонента
- Шифруют состояние в случае сохранения его на клиенте
- Сжимают состояние в случае сохранения его на клиенте
- Ограничивают доступ к страницам, содержащим теги Jakarta Faces
- Включают валидацию XML
- Указывают этап проекта
- Верифицируют кастомные объекты

Этот раздел даёт более подробную информацию об этих настройках. Там, где это уместно, также описывается, как можно выполнить эти настройки в IDE NetBeans.

Идентификация сервлета для обработки жизненного цикла

Требование к приложению Jakarta Faces состоит в том, что все запросы к приложению, которые ссылаются на ранее сохранённые компоненты Jakarta Faces, должны проходить через `jakarta.faces.webapp.FacesServlet` . Объект `FacesServlet` управляет жизненным циклом обработки запросов для веб-приложений и инициализирует ресурсы, необходимые для Jakarta Faces.

Прежде чем приложение Jakarta Faces запустит свою первую веб-страницу, веб-контейнер должен вызвать объект `FacesServlet` для старта процесса жизненного цикла приложения. См. Жизненный цикл приложения Jakarta Faces для получения дополнительной информации.

В следующем примере показана конфигурация по умолчанию для `FacesServlet`:

```xml
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>jakarta.faces.webapp.FacesServlet</servlet-class>
</servlet>
```

Нужно соответствующим образом оформить файл конфигурации, чтобы быть уверенными в вызове объекта `FacesServlet`. Назначение `FacesServlet`-у может происходить по префиксу, например `/faces/`, или по расширению, например `.xhtml`. Сопоставление используется для определения страницы с содержимым Jakarta Faces. По этой причине URL первой страницы приложения должен подходить под назначение шаблона URL.

Следующие элементы определяют назначение по префиксу:

```xml
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
...
<welcome-file-list>
    <welcome-file>faces/greeting.xhtml</welcome-file>
</welcome-file-list>
```

Следующие элементы, используемые в примерах учебника, определяют назначение по расширению:

```xml
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.xhtml</url-pattern>
</servlet-mapping>
...
<welcome-file-list>
    <welcome-file>index.xhtml</welcome-file>
</welcome-file-list>
```

При использовании этого механизма пользователи получают доступ к приложению, как показано в следующем примере:

```
http://localhost:8080/guessNumber
```

В случае назначения по расширению, если на сервер поступает запрос на страницу с расширением `.xhtml`, контейнер отправит запрос объекту `FacesServlet`, ожидающему, что соответствующая страница с тем же именем существует.

Чтобы свести к минимуму беспорядок и разрешить простые URL-ы, вы можете создать URL-ы без расширений, вручную назначив `FacesServlet` существующим префиксам и суффиксам в `web.xml`.

Если вы используете IDE NetBeans, дескриптор развёртывания с конфигурациями по умолчанию создаётся автоматически. Если вы создали своё приложение без IDE, можно создать дескриптор развёртывания вручную.

Указание пути к файлу конфигурации приложения

Как объясняется в Файле конфигурации приложения, приложение может иметь несколько файлов конфигурации приложения. Если эти файлы находятся не в каталогах, в которых реализация ищет их по умолчанию, или файлы называются не `faces-config.xml` , необходимо указать пути к этим файлам.

Чтобы указать эти пути в IDE NetBeans, выполните следующие действия.

1. Разверните узел проекта на вкладке **Проекты**.
2. Разверните узлы **Веб-страницы** и **WEB-INF**, которые находятся под узлом проекта.
3. Выполните двойной клик на `web.xml` .
4. После того как `web.xml` появится в редакторе, нажмите кнопку **Общие** в верхней части окна редактора.
5. Разверните узел **Контекстные параметры**.
6. Кликните **Добавить**.
7. В диалоговом окне «Добавить контекстный параметр»:
    a. Введите `jakarta.faces.CONFIG_FILES` в поле **Имя параметра**.
    b. Введите путь к файлу конфигурации в поле **Значение параметра**.
    c. Кликните **OK**.
    d. Повторите шаги с 1 по 7 для каждого файла конфигурации.

Указание хранилища состояний

Для всех компонентов в веб-приложении можно указать в дескрипторе развёртывания, где вы хотите сохранить состояние: на клиенте или на сервере. Вы делаете это, устанавливая контекстный параметр в вашем дескрипторе развёртывания. По умолчанию состояние сохраняется на сервере, поэтому этот контекстный параметр необходимо указывать только в том случае, если вы хотите сохранить состояние на клиенте. Смотрите Сохранение и восстановление состояния для получения информации о преимуществах и недостатках каждого местоположения.

Чтобы указать место сохранения состояния в IDE NetBeans, выполните следующие действия.

1. Разверните узел проекта на вкладке **Проекты**.
2. Разверните узлы **Веб-страницы** и **WEB-INF** под узлом проекта.
3. Выполните двойной клик на `web.xml` .
4. После того как файл `web.xml` отобразится в окне редактора, кликните **Общие** в верхней части окна редактора.
5. Разверните узел **Контекстные параметры**.
6. Кликните **Добавить**.
7. В диалоговом окне «Добавить контекстный параметр»:
    a. Введите `jakarta.faces.STATE_SAVING_METHOD` в поле **Имя параметра**.
    b. Введите `client` или `server` в поле **Значение параметра**.
    c. Кликните **OK**.

Если состояние сохраняется на клиенте, состояние всего представления отображается в скрытом поле на странице. Jakarta Faces по умолчанию сохраняет состояние на сервере. Duke's Forest сохраняет своё состояние на клиенте.

Настройка этапа проекта

Этап проекта — это контекстный параметр, определяющий состояние приложения Jakarta Faces в жизненном цикле программного обеспечения. Этап приложения может повлиять на поведение приложения. Например, сообщения об ошибках могут отображаться при разработке но подавляться при боевом использовании.

Возможные значения стадии проекта:

- `Development`
- `UnitTest`
- `SystemTest`
- `Production`

Этап проекта настраивается с помощью контекстного параметра в файле дескриптора развёртывания. Вот пример:

JAVA

```
<context-param>
    <param-name>jakarta.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
</context-param>
```

Если этап проекта не определён, по умолчанию берётся `Production`. Вы также можете добавить кастомные этапы в соответствии с вашими требованиями.

Включение классов, страниц и других ресурсов

При упаковке веб-приложений с использованием включённых сценариев сборки вы заметите, что сценарии упаковывают ресурсы следующими способами.

- Все веб-страницы находятся на верхнем уровне файла WAR.
- Файл `faces-config.xml` и файл `web.xml` упакованы в каталог `WEB-INF`.
- Все пакеты хранятся в каталоге `WEB-INF/classes/`.
- Все файлы JAR приложения упакованы в каталог `WEB-INF/lib/`.
- Все файлы ресурсов находятся либо в корне веб-приложения `/resources`, либо в пути к классам веб-приложения, каталоге `META-INF/resources/`*`resourceIdentifier`*. Дополнительные сведения о ресурсах см. в разделе Веб-ресурсы.

При упаковке приложений может использоваться среда IDE NetBeans или файлы XML, например, созданные для Maven. Вы можете изменить файлы XML в соответствии с вашей ситуацией. Однако вы можете продолжать упаковывать WAR-файлы, используя структуру каталогов, описанную в этом разделе, поскольку этот метод соответствует общепринятой практике для упаковки веб-приложений.

Глава 17. Использование веб-сокетов с Jakarta Faces

В этой главе описывается использование веб-сокетов в веб-приложениях Jakarta Faces.

О веб-сокетах в Jakarta Faces

Тег `f:websocket` используется в представлении, чтобы разрешить передачу сообщений на стороне сервера всем объектам сокета, содержащим указанное имя канала. Когда сообщение получено, можно установить обработчик событий JavaScript на стороне клиента `onmessage`, который вызывается всякий раз, когда с сервера поступает push-уведомление.

На серверной стороне веб-сокета есть возможность создавать push-сообщения. Вы можете сделать это с использованием `jakarta.faces.push.PushContext`, который является инъецируемым контекстом, позволяющим серверу выполнить push в указанный канал.

Конфигурирование веб-сокетов

Чтобы сконфигурировать веб-сокеты для использования в веб-приложениях Faces, сначала включите конечную точку веб-сокета с помощью контекстного параметра в файле `web.xml`:

XML

```
<context-param>
    <param-name>jakarta.faces.ENABLE_WEBSOCKET_ENDPOINT</param-name>
    <param-value>true</param-value>
</context-param>
```

Если ваш сервер настроен для запуска контейнера веб-сокетов на TCP-порте, отличном от HTTP-контейнера, вы можете использовать необязательный целочисленный контекстный параметр `jakarta.faces.WEBSOCKET_ENDPOINT_PORT` для явного указания порта:

XML

```
<context-param>
    <param-name>jakarta.faces.WEBSOCKET_ENDPOINT_PORT</param-name>
    <param-value>8000</param-value>
</context-param>
```

Использование веб-сокета на стороне клиента

Объявите тег `f: websocket` в представлении Faces с именем канала и JavaScript-обработчиком `onmessage`.

Следующий пример ссылается на существующую функцию слушателя JavaScript:

JAVASCRIPT

```
<f:websocket channel="someChannel" onmessage="someWebsocketListener" />

function someWebsocketListener(message, channel, event) { console.log(message); }
```

В этом примере объявляется встроенная функция слушателя JavaScript:

XML

```
<f:websocket channel="someChannel" onmessage="function(m){console.log(m);}" />
```

Функция слушателя JavaScript `onmessage` вызывается с тремя аргументами:

- `message`: push-сообщение как объект JSON
- `channel`: название канала

- `event` : объект `MessageEvent`

При успешном подключении веб-сокет открывается по умолчанию до тех пор, пока документ открыт, и он будет автоматически переподключаться с увеличивающимися интервалами, когда соединение закрыто или прервано, в результате таких событий, как ошибка сети или перезагрузка сервера. Он не будет автоматически переподключён, если первая попытка подключения не удалась. Веб-сокет будет неявно закрыт после выгрузки документа.

Использование веб-сокетов на стороне сервера

В коде Java инъецируйте `PushContext` используя аннотацию `@Push` на заданном канале в любом Managed-бине CDI, таком как `@Named` или `@WebServlet` , куда вы хотите отправить push-сообщение. Затем вызовите `PushContext.send(Object)` с любым объектом Java, представляющим push-сообщение.

Например:

JAVA

```
@Inject @Push
private PushContext someChannel;

public void sendMessage(Object message) {
    someChannel.send(message);
}
```

По умолчанию имя канала берётся из имени переменной, в которую происходит инъецирование.

При желании имя канала можно указать с помощью атрибута `channel` . В следующем примере контекст вставки для имени канала `foo` инъецируется в переменную `bar` .

JAVA

```
@Inject
@Push(channel="foo")
private PushContext bar;
```

Объект сообщения будет закодирован как JSON и доставлен в качестве аргумента сообщения функции слушателя JavaScript `onmessage` , связанной с именем канала. Это может быть строка, коллекция, отображение (Map) или JavaBean.

Использование тега f:websocket

Таблица 17-1 описывает атрибуты тега `f:websocket` .

Таблица 17-1 Атрибуты тега `f:websocket`

Название	Тип	Описание
`channel`	String	Обязательный. Название канала веб-сокета. Он не может быть выражением EL и может содержать только буквенно-цифровые символы, дефисы, подчеркивания и точки. Все открытые веб-сокеты с одним и тем же именем канала получат одно и то же push-уведомление от сервера.
`id`	String	Необязательный. Идентификатор компонента UIWebSocket, который будет создан.

Название	Тип	Описание
`scope`	String	Необязательный. Область применения канала веб-сокета. Не может быть выражением EL. Допустимые значения (без учёта регистра): `application`, `session` и `view`. Когда значение равно `application`, все каналы с одинаковым именем в приложении получают одно и то же push-сообщение. Когда значение равно `session`, только каналы с одинаковым именем в текущей сессии пользователя получают одно и то же push-сообщение. Когда значением является `view`, только канал в текущем представлении получает push-сообщение. Область видимости по умолчанию — `application`. Если указан атрибут `user`, то областью по умолчанию является `session`.
`user`	Serializable	Необязательный. Идентификатор пользователя канала веб-сокета, чтобы можно было отправлять push-сообщения, ориентированные на пользователя. Он должен реализовывать Serializable и предпочтительно иметь небольшой объём памяти. **Подсказка**: Используйте `#{request.remoteUser}` или `#{someLoggedInUser.id}`. Все открытые веб-сокеты на одном канале и пользователь получат одно и то же push-сообщение с сервера.
`onopen`	String	Необязательный. Функция обработчика событий JavaScript, которая вызывается при открытии веб-сокета. Функция вызывается с одним аргументом: именем канала.
`onmessage`	String	Необязательный. Функция обработчика событий JavaScript, которая вызывается при получении push-сообщения от сервера. Функция вызывается с тремя аргументами: push-сообщением, именем канала и объектом `MessageEvent`.
`onclose`	String	Необязательный. Функция обработчика событий JavaScript, которая вызывается при закрытии веб-сокета. Функция вызывается с тремя аргументами: кодом причины закрытия, именем канала и `CloseEvent`. Обратите внимание, что функция будет вызываться также и при ошибках. Если произошла ошибка, вы можете проверить код причины закрытия и какой код был задан (например, когда код не 1000).

Название	Тип	Описание
`connected`	Boolean	Необязательный. Указывает, нужно ли автоматически повторно подключать веб-сокет. По умолчанию `true`. Интерпретируется как инструкция JavaScript для открытия или закрытия push-соединения веб-сокета. Этот атрибут неявным образом вычисляется при каждом запросе ajax слушателем `PreRenderViewEvent` в `UIViewRoot`. Вы также можете явно установить для него значение `false`, а затем вручную управлять им в JavaScript, используя `jsf.push.open(clientId)` и `jsf.push.close(clientId)`.
`rendered`	Boolean	Необязательный. Указывает, следует ли отображать сценарии веб-сокета. По умолчанию `true`. Этот атрибут неявным образом вычисляется при каждом запросе ajax слушателем `PreRenderViewEvent` в `UIViewRoot`. Если значение изменится на `false`, когда веб-сокет уже открыт, то веб-сокет будет неявно закрыт.
`binding`	UIComponent	Необязательный. Выражение привязки значения к свойству базового компонента, привязанному к объекту компонента для `UIComponent`, созданного этим тегом.

Области видимости и пользователи веб-сокетов

По умолчанию веб-сокет находится в области видимости приложения. Например, любое представление или сессия в веб-приложении, в котором открыт один и тот же канал веб-сокета, получит одно и то же push-сообщение. Push-сообщение может быть отправлено любым пользователем и приложением. Чтобы ограничить push-сообщения всеми представлениями только в текущей сессии пользователя, установите для необязательного атрибута scope значение `session`. В этом случае push-сообщение может отправлять только пользователь, а не приложение.

```xml
<f:websocket channel="someChannel" scope="session" ... />
```

Чтобы ограничить push-сообщения только текущим представлением, вы можете установить для атрибута scope значение `view`. Push-сообщение не будет отображаться в других представлениях той же сессии, даже если оно имеет тот же URL. Это push-сообщение может отправлять только пользователь, а не приложение.

```xml
<f:websocket channel="someChannel" scope="view" ... />
```

Атрибут scope не может быть выражением EL.

Кроме того, вы можете установить необязательный атрибут `user` для уникального идентификатора вошедшего в систему пользователя, обычно это имя для входа или идентификатор пользователя. Таким образом, push-сообщение может быть предназначено для конкретного пользователя, а также может

отправляться другими пользователями и приложением. Значение атрибута `user` должно реализовывать `Serializable` и иметь небольшой объём памяти, поэтому не рекомендуется использовать весь объект пользователя.

Например, когда вы используете управляемую контейнером аутентификацию, фреймворк или библиотеку:

```xml
<f:websocket channel="someChannel"user="#{request.remoteUser}" ... />
```

Или, когда у вас есть пользователь, доступный через EL, например, `#{someLoggedInUser}`, который имеет свойство `id`, представляющее его идентификатор:

```xml
<f:websocket channel="someChannel" user="#{someLoggedInUser.id}" ... />
```

Когда указан атрибут `user`, область видимости по умолчанию равна `session` и не может быть установлена в `application`.

На стороне сервера push-сообщение может быть предназначено для пользователя, указанного в атрибуте `user` с помощью `PushContext.send(Object, Serializable)`. Push-сообщение может быть отправлено любым пользователем и приложением.

```java
@Inject @Push
private PushContext someChannel;

public void sendMessage(Object message, User recipientUser) {
    Long recipientUserId = recipientUser.getId();
    someChannel.send(message, recipientUserId);
}
```

Получателями могут быть несколько пользователей путём передачи `Collection`, содержащего идентификаторы пользователей, в `PushContext.send(Object, Collection)`.

```java
public void sendMessage(Object message, Group recipientGroup) {
    Collection<Long> recipientUserIds = recipientGroup.getUserIds();
    someChannel.send(message, recipientUserIds);
}
```

Условное подключение веб-сокетов

Вы можете использовать необязательный атрибут connected для управления автоматическим переподключением веб-сокета.

```xml
<f:websocket ... connected="#{bean.pushable}" />
```

По умолчанию для атрибута connected установлено значение `true`, и оно интерпретируется как инструкция JavaScript для открытия или закрытия push-соединения веб-сокета. Если значение является выражением EL и оно установлено в `false` во время запроса ajax, то принудительное соединение будет явно закрыто во время `oncomplete` этого запроса ajax.

Вы также можете явно установить для него значение `false` и вручную открыть push-соединение на стороне клиента, вызвав `jsf.push.open(clientId)`, передав идентификатор клиента компонента.

XML

```xml
<h:commandButton ... onclick="jsf.push.open('foo')">
    <f:ajax ... />
</h:commandButton>
<f:websocket id="foo" channel="bar" scope="view" ... connected="false" />
```

Если намереваетесь использовать однократное push-уведомление и не ожидаете больше сообщений, вы можете явно закрыть push-соединение со стороны клиента, вызвав `jsf.push.close(clientId)`, передав компоненту ID клиента. Например, в функции слушателя JavaScript `onmessage`, как показано ниже:

JAVASCRIPT

```javascript
function someWebsocketListener(message) {
// ... jsf.push.close('foo');
}
```

События веб-сокета: сервер

Когда сессия или представление автоматически закрываются сервером с кодом причины закрытия `1000` (и, следовательно, не закрываются вручную клиентом через `jsf.push.close(clientId)`), это означает, что время жизни сессии или представления истекло.

JAVA

```java
@ApplicationScoped
public class WebsocketObserver {

    public void onOpen(@Observes @Opened WebsocketEvent event) {
        String channel = event.getChannel();
        // Возвращает <f:websocket channel>. Long userId = event.getUser();
        // Возвращает <f:websocket user>, если нужно.
        // ...
    }

    public void onClose(@Observes @Closed WebsocketEvent event) { String channel = event.getChannel();
        // Возвращает <f:websocket channel>. Long userId = event.getUser();
        // Возвращает <f:websocket user>, если нужно. CloseCode code = event.getCloseCode();
        // Возвращает код закрытия.
       // ...
    }
}
```

События веб-сокета: клиенты

Вы можете использовать дополнительную функцию слушателя JavaScript `onopen` для обработки события открытия веб-сокета на стороне клиента. Эта функция вызывается при первой попытке подключения, независимо от того, будет ли она успешной. Он не будет вызываться, когда веб-сокет автоматически переподключит разорванное соединение после первого успешного соединения.

JAVASCRIPT

```javascript
<f:websocket ... onopen="websocketOpenListener" />
function websocketOpenListener(channel) {
// ...
}
```

Функция слушателя JavaScript `onopen` вызывается с одним аргументом: `channel` (имя канала, особенно полезно, если у вас есть глобальный слушатель).

Вы можете использовать дополнительную функцию слушателя JavaScript `onclose` для обработки события завершения (корректного или некорректного) работы веб-сокета. Эта функция вызывается, когда первая попытка подключения не удалась или сервер возвратил код причины закрытия `1000` (нормальное закрытие)

или `1008` (нарушение политики безопасности) или было превышено максимальное кол-во попыток повторное соединения. Он не будет вызываться, если веб-сокет делает попытку автоматического переподключения при разорванном соединении после первого успешного соединения.

JAVASCRIPT

```
<f:websocket ... onclose="websocketCloseListener" />
function websocketCloseListener(code, channel, event) {
    if (code == -1) {
        // веб-сокеты не поддерживаются клиентом.
    } else if (code == 1000) {
        // Корректное завершение (в результате истечения сессии).
    } else {
        // Некорректное завершение (в результате ошибки).
    }
}
```

Функция слушателя JavaScript `onclose` вызывается с тремя аргументами:

- `code` : код причины закрытия в виде целого числа. Если это `-1` , веб-сокет не поддерживается клиентом. Если это `1000` , то он был нормально закрыт. В противном случае, если это не `1000` , то может быть ошибка.
- `channel` : название канала
- `event` : объект `CloseEvent`

Вопросы безопасности в веб-сокетах

Если веб-сокет объявлен на странице, доступ к которой разрешён только зарегистрированным пользователям с определённой ролью, возможно, вы захотите добавить URL push-запроса в число URL-ов с ограниченным доступом.

URL запроса установления push-соединения состоит из префикса URI, `/jakarta.faces.push/` , за которым следует имя канала. В примере управляемой контейнером безопасности, который уже ограничивает доступ к странице `/user/foo.xhtml` для зарегистрированных пользователей с ролью `USER` шаблоном URL `/user/*` в `web.xml` , см. ниже:

XML

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Restrict access to role USER.</web-resource-name>
        <url-pattern>/user/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>USER</role-name>
    </auth-constraint>
</security-constraint>
```

Если страница `/user/foo.xhtml` содержит `<f:websocket channel="foo">` , то вы должны добавить ограничение на шаблон URL-адреса запроса установления push-соединения `/jakarta.faces.push/foo` как показано ниже:

XML

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Restrict access to role USER.</web-resource-name>
        <url-pattern>/user/*</url-pattern>
        <url-pattern>/jakarta.faces.push/foo</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>USER</role-name>
    </auth-constraint>
</security-constraint>
```

В качестве дополнительной защиты, особенно для общедоступных каналов, которые не могут быть защищены ограничениями безопасности, тег `f:websocket` зарегистрирует все ранее объявленные каналы в текущей сессии HTTP, и любой входящий запрос на открытие веб-сокета будет проверяться. соответствует ли он этим каналам в текущей сессии HTTP. Если канал неизвестен (например, случайно угадан, подделан конечными пользователями или повторно подключён вручную после истечения сессии), тогда веб-сокет будет немедленно закрыт с кодом причины закрытия `CloseCodes.VIOLATED_POLICY (1008)`. Кроме того, когда сессия HTTP будет завершена, все каналы области видимости сессии и представления, которые всё ещё открыты, будут явно закрыты со стороны сервера с кодом причины закрытия `CloseCodes.NORMAL_CLOSURE (1000)`. Только сокеты в области видимости приложения остаются открытыми и по-прежнему доступны для сервера, даже если время жизни сессии или представления, связанного со страницей на стороне клиента, исткло.

Использование Ajax с веб-сокетами

При желании выполнять сложные обновления пользовательского интерфейса в зависимости от полученного push-сообщения, вы можете вложить тег `f:ajax` в тег `f:websocket`. Смотрите следующий пример:

XML

```
<h:panelGroup id="foo">
    ... (some complex UI here) ...
</h:panelGroup>
<h:form>
    <f:websocket channel="someChannel" scope="view">
        <f:ajax event="someEvent" listener="#{bean.pushed}" render=":foo" />
    </f:websocket>
</h:form>
```

Здесь push-сообщение просто представляет имя события ajax. Вы можете использовать любое произвольное имя события.

JAVASCRIPT

```
someChannel.send("someEvent");
```

Альтернативой является объединение тега `f:websocket` с тегом `h:commandScript`. `<f:websocket onmessage>` в точности ссылается на `<h:commandScript name>`.

Например:

XML

```
<h:panelGroup id="foo">
    ... (some complex UI here) ...
</h:panelGroup>
<f:websocket channel="someChannel" scope="view" onmessage="pushed" />
<h:form>
    <h:commandScript name="pushed" action="#{bean.pushed}" render=":foo" />
</h:form>
```

Если вы передадите `Map<String, V>` или JavaBean в качестве объекта push-сообщения, то все записи или свойства будут прозрачно доступны в качестве параметров запроса в методе сценария `#{bean.pushed}`.

Глава 18. Технология Jakarta Servlet

Технология Jakarta Servlet обеспечивает динамическое, ориентированное на пользователя содержимое в веб-приложениях с использованием модели запрос-ответ.

Что такое сервлет?

Сервлет — это класс Java, используемый для расширения возможностей сервера, на котором размещаются приложения и доступ к которому осуществляется по модели запрос-ответ. Хотя сервлеты могут отвечать на любые запросы, они обычно используются для расширения приложений, размещаемых на веб-серверах. Для таких приложений технология Jakarta Servlet определяет специфичные для HTTP классы сервлетов.

Пакеты `jakarta.servlet` и `jakarta.servlet.http` предоставляют интерфейсы и классы для написания сервлетов. Все сервлеты должны реализовывать интерфейс `Servlet`, который определяет методы жизненного цикла. Наиболее общий сервис может быть реализован с использованием или расширением класса `GenericServlet`, предоставляемого с API сервлетов Jakarta. Класс `HttpServlet` предоставляет методы, такие как `doGet` и `doPost`, для обработки специфичных для HTTP сервисов.

Жизненный цикл сервлета

Жизненный цикл сервлета контролируется контейнером, в котором был развёрнут сервлет. Когда правила навигации определяют, что запрос должен обрабатываться сервлетом, контейнер выполняет следующие шаги.

1. Если объект сервлета не существует, веб-контейнер:
   a. Загружает класс сервлета
   b. Инстанцирует объект сервлета
   c. Инициализирует объект сервлета, вызывая метод `init` (инициализация описана в Создание и инициализация сервлета)
2. Контейнер вызывает метод `service`, передавая объекты запроса и ответа. Сервисные методы обсуждаются в Написание сервисных методов.

Если контейнер посчитает необходимым удалить сервлет, он завершает сервлет, вызывая метод `destroy` сервлета. Для получения дополнительной информации см. Финализация сервлета.

Обработка событий жизненного цикла сервлета

Вы можете отслеживать события в жизненном цикле сервлета и реагировать на них, определяя объекты слушателей, чьи методы вызываются при возникновении событий жизненного цикла. Чтобы использовать эти объекты слушателя, вы должны определить и указать класс слушателя.

Определение класса слушателя

Вы определяете класс слушателя как реализацию интерфейса слушателя. Таблица 18-1 перечисляет события, которые можно отслеживать, и соответствующий интерфейс, который должен быть реализован. Когда вызывается метод слушателя, ему передаётся событие, которое содержит информацию, соответствующую этому событию. Например, методы в интерфейсе `HttpSessionListener` передаются в `HttpSessionEvent`, который содержит `HttpSession`.

Таблица 18-1 События жизненного цикла сервлета

Object	Событие	Интерфейс слушателя и класс события

Object	Событие	Интерфейс слушателя и класс события
Веб-контекст	Инициализация и уничтожение	`jakarta.servlet.ServletContextListener` и `ServletContextEvent`
Веб-контекст	Атрибут добавлен, удалён или заменён	`jakarta.servlet.ServletContextAttributeListener` и `ServletContextAttributeEvent`
Сессия	Создание, аннулирование, активация, деактивация и тайм-аут	`jakarta.servlet.http.HttpSessionListener`, `jakarta.servlet.http.HttpSessionActivationListener` и `HttpSessionEvent`
Сессия	Атрибут добавлен, удалён или заменён	`jakarta.servlet.http.HttpSessionAttributeListener` и `HttpSessionBindingEvent`
Запрос	Запрос сервлета начал обрабатываться веб-компонентами	`jakarta.servlet.ServletRequestListener` и `ServletRequestEvent`
Запрос	Атрибут добавлен, удалён или заменён	`jakarta.servlet.ServletRequestAttributeListener` и `ServletRequestAttributeEvent`

Используйте аннотацию `@WebListener` чтобы определить слушателя для получения событий для различных операций в определённом контексте веб-приложения. Классы, аннотированные `@WebListener`, должны реализовывать один из следующих интерфейсов:

JAVA

```
jakarta.servlet.ServletContextListener
jakarta.servlet.ServletContextAttributeListener
jakarta.servlet.ServletRequestListener
jakarta.servlet.ServletRequestAttributeListener
jakarta.servlet..http.HttpSessionListener
jakarta.servlet..http.HttpSessionAttributeListener
```

Например, следующий фрагмент кода определяет слушателя, который реализует два из этих интерфейсов:

JAVA

```
import jakarta.servlet.ServletContextAttributeListener;
import jakarta.servlet.ServletContextListener;
import jakarta.servlet.annotation.WebListener;

@WebListener()
public class SimpleServletListener implements ServletContextListener,
        ServletContextAttributeListener {
    ...
}
```

Обработка ошибок сервлета

При выполнении сервлета могут возникать исключения. При возникновении исключения веб-контейнер создаёт страницу по умолчанию, содержащую следующее сообщение:

```
A Servlet Exception Has Occurred
```

Но можно также указать, что контейнер должен возвращать определённую страницу ошибки для данного исключения.

Обмен информацией

Веб-компоненты, как и большинство объектов, обычно работают с другими объектами для выполнения своих задач. Веб-компоненты могут делать это следующим образом.

- Использование приватных вспомогательных объектов (например, компонентов JavaBeans).
- Использование общих объектов, которые являются атрибутами общедоступной области видимости.
- Использование базы данных.
- Вызов других веб-ресурсов. Механизмы технологии Jakarta Servlet, которые позволяют веб-компоненту вызывать другие веб-ресурсы, описаны в Вызов других веб-ресурсов.

Использование объектов областей видимости

Совместно работающие веб-компоненты обмениваются информацией с помощью объектов, которые поддерживаются в качестве атрибутов четырёх объектов областей видимости. Вы получаете доступ к этим атрибутам с помощью методов `getAttribute` и `setAttribute` класса, представляющего область видимости. Таблица 18-2 перечисляет объекты областей видимости.

Таблица 18-2 Объекты областей видимости

Область видимости	**Класс**	**Доступно с**
Веб-контекст	`jakarta.servlet.ServletContext`	Веб-компоненты в веб-контексте. Смотрите Доступ к веб-контексту.
Сессия	`jakarta.servlet.http.HttpSession`	Веб-компоненты, обрабатывающие запрос, принадлежащий сессии. Смотрите Поддержание состояния клиента.
Запрос	Дочерний тип от `jakarta.servlet.ServletRequest`	Веб-компоненты, обрабатывающие запрос.
Страница	`jakarta.servlet.jsp.JspContext`	Страница Jakarta Server Pages, на которой создаётся объект.

Управление одновременным доступом к общим ресурсам

На многопоточном сервере к общим ресурсам можно обращаться одновременно. В дополнение к атрибутам объекта области видимости общие ресурсы включают в себя данные в памяти, такие как переменные объекта или класса, и внешние объекты, такие как файлы, соединения с базой данных и сетевые соединения.

Параллельный доступ может возникнуть в нескольких ситуациях.

- Несколько веб-компонентов, обращающихся к объектам, хранящимся в веб-контексте.
- Несколько веб-компонентов, обращающихся к объектам, хранящимся в сессии.
- Несколько потоков внутри веб-компонента обращаются к переменным объекта.

Веб-контейнер обычно создаёт поток для обработки каждого запроса. Чтобы гарантировать, что объект сервлета обрабатывает только один запрос за один раз, сервлет может реализовать интерфейс `SingleThreadModel`. Если сервлет реализует этот интерфейс, никакие два потока не будут выполняться одновременно в сервисном методе сервлета. Веб-контейнер может гарантировать это, синхронизируя доступ к одному объекту сервлета или поддерживая пул объектов веб-компонента и отправляя каждый новый запрос в свободный объект. Этот интерфейс не предотвращает проблемы синхронизации, возникающие в результате доступа веб-компонентов к общим ресурсам, таким как статические переменные класса или внешние объекты.

Когда к ресурсам можно получить доступ одновременно, они могут использоваться противоречивым образом. Вы предотвращаете это, контролируя доступ с помощью методов синхронизации, описанных в разделе потоков на https://docs.oracle.com/javase/tutorial/essential/concurrency/.

Создание и инициализация сервлета

Используйте аннотацию `@WebServlet` для определения компонента сервлета в веб-приложении. Эта аннотация указана для класса и содержит метаданные об объявленном сервлете. Аннотированный сервлет должен указывать хотя бы один шаблон URL. Это делается с помощью атрибутов `urlPatterns` или `value` в аннотации. Все остальные атрибуты являются необязательными, с настройками по умолчанию. Используйте атрибут `value`, когда единственным атрибутом в аннотации является шаблон URL. В противном случае используйте атрибут `urlPatterns`, когда также используются другие атрибуты.

Классы, аннотированные `@WebServlet`, должны расширять класс `jakarta.servlet.http.HttpServlet`. Например, следующий фрагмент кода определяет сервлет с шаблоном URL `/report`:

JAVA

```
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;

@WebServlet("/report")
public class MoodServlet extends HttpServlet {
    ...
}
```

Веб-контейнер инициализирует сервлет после загрузки и создания объекта класса сервлета и перед получением запросов от клиентов. Чтобы настроить этот процесс и позволить сервлету считывать данные конфигурации, инициализировать ресурсы и выполнять любые другие действия, вы можете переопределить метод `init` интерфейса `Servlet` или укажите атрибут `initParams` аннотации `@WebServlet`. Атрибут `initParams` содержит аннотацию `@WebInitParam`. Если он не может завершить процесс инициализации, сервлет генерирует исключение `UnavailableException`.

Используйте параметр инициализации для предоставления данных, необходимых для конкретного сервлета. Напротив, контекстный параметр предоставляет данные, которые доступны всем компонентам веб-приложения.

Написание сервисных методов

Сервис, предоставляемый сервлетом, реализован в методе `service` класса `GenericServlet`, в методах `do`*`Method`* (где *Method* может принимать значение `Get`, `Delete`, `Options`, `Post`, `Put` или `Trace`) класса `HttpServlet` или в любых других методах, специфичных для протокола, определённых классом, реализующим интерфейс `Servlet`. Термин "сервисный метод" используется для любого метода в классе сервлета, который предоставляет сервисы клиенту.

Общий шаблон для сервисного метода состоит в том, чтобы извлечь информацию из запроса, получить доступ к внешним ресурсам, а затем заполнить ответ на основе этой информации. Для HTTP-сервлетов корректная процедура заполнения ответа заключается в следующем:

1. Получить выходной поток из ответа.
2. Заполните заголовки ответа.
3. Запишите содержимое в выходной поток.

Заголовки ответа всегда должны быть установлены до того, как ответ будет зафиксирован. Веб-контейнер будет игнорировать любые попытки установить или добавить заголовки после фиксации ответа. В следующих двух разделах описывается, как получать информацию из запросов и генерировать ответы.

Получение информации из запросов

Запрос содержит данные, передаваемые между клиентом и сервлетом. Все запросы реализуют интерфейс `ServletRequest`. Этот интерфейс определяет методы для доступа к следующей информации:

- Параметры, которые обычно используются для передачи информации между клиентами и сервлетами
- Атрибуты со значениями-объектами, которые обычно используются для передачи информации между веб-контейнером и сервлетом или между взаимодействующими сервлетами.
- Информация о протоколе, используемом для передачи запроса, а также о клиенте и сервере, участвующих в запросе.
- Информация, относящаяся к локализации

Вы также можете извлечь входной поток из запроса и вручную проанализировать данные. Чтобы прочитать символьные данные, используйте объект `BufferedReader`, возвращаемый методом запроса `getReader`. Чтобы прочитать бинарные данные, используйте `ServletInputStream`, возвращаемый `getInputStream`.

HTTP-сервлетам передаётся объект HTTP-запроса, `HttpServletRequest`, который содержит URL запроса, HTTP-заголовки, строку запроса и т. д. URL HTTP-запроса содержит следующие части:

```
http://[host]:[port][request-path]?[query-string]
```

Путь запроса (request path) дополнительно состоит из следующих элементов.

- Контекстный путь (Context path): конкатенация косой черты (`/`) с корневым контекстом веб-приложения сервлета.
- Путь сервлета (Servlet path): часть пути, соответствующей псевдониму компонента, который активировал этот запрос. Этот путь начинается с косой черты (`/`).
- Информация о пути (Path info): часть пути запроса, которая не является частью контекстного пути или пути сервлета.

Вы можете использовать методы `getContextPath`, `getServletPath` и `getPathInfo` интерфейса `HttpServletRequest` для получения этой информации. За исключением различий в кодировке URL между URI запроса и частями пути, URI запроса всегда состоит из контексного пути плюс путь сервлета и информация о пути.

Строки запроса состоят из набора параметров и их значений. Отдельные параметры извлекаются из запроса с использованием метода `getParameter`. Есть два способа генерации строк запроса.

- Строка запроса может явно отображаться на веб-странице.
- Строка запроса добавляется к URL при отправке формы с HTTP-методом `GET`.

Построение ответов

Ответ содержит данные, передаваемые между сервером и клиентом. Все ответы реализуют интерфейс `ServletResponse`. Этот интерфейс определяет методы, которые позволяют:

- Получить выходной поток, чтобы использовать его для отправки данных клиенту. Чтобы отправить символьные данные, используйте `PrintWriter`, возвращаемый методом ответа `getWriter`. Чтобы отправить бинарные данные в ответе MIME, используйте `ServletOutputStream`, возвращаемый `getOutputStream`. Чтобы сочетать бинарные и текстовые данные, как в многочастном (multipart) ответе, используйте `ServletOutputStream` и управляйте разделами символов вручную.
- Укажите тип содержимого (например, `text/html`), возвращаемый ответом с помощью метода `setContentType(String)`. Этот метод должен быть вызван до фиксации ответа. Реестр имён типов содержимого ведётся Internet Assignent Numbers Authority (IANA) по ссылке https://www.iana.org/assignments/media-types/.
- Укажите, нужно ли буферизовать вывод с помощью метода `setBufferSize(int)`. По умолчанию любой контент, записанный в выходной поток, немедленно отправляется клиенту. Буферизация позволяет записывать содержимое до того, как что-либо отправлено обратно клиенту, что даёт сервлету больше времени для установки соответствующих кодов состояния и заголовков или переадресации на другой веб-ресурс. Метод должен быть вызван до того, как будет написано какое-либо содержимое, или до того, как ответ будет зафиксирован.
- Установите информацию о локализации, такую как локаль и кодировка символов. Подробности см. в главе 22 *Интернационализация и локализация веб-приложений*.

Объекты HTTP-ответа, `jakarta.servlet.http.HttpServletResponse`, имеют поля, представляющие заголовки HTTP, например следующие.

- Коды состояния, которые используются для указания причины, по которой запрос не удовлетворён или что запрос был перенаправлен.
- Файлы cookie, которые используются для хранения информации о приложении на клиенте. Иногда файлы cookie используются для поддержания идентификатора для отслеживания сессии пользователя (см. Отслеживание сессии).

Фильтрация запросов и ответов

Фильтр — это объект, который может преобразовать заголовок и содержимое (или оба) запроса или ответа. Фильтры отличаются от веб-компонентов тем, что сами фильтры обычно не создают ответ. Вместо этого фильтр обеспечивает функциональность, которую можно «прикрепить» к любому веб-ресурсу. Следовательно, фильтр не должен иметь никаких зависимостей от веб-ресурса, для которого он выступает фильтром. Таким образом, он может быть составлен из более чем одного типа веб-ресурса.

Основные задачи, которые может выполнять фильтр, следующие.

- Создавать запрос и действовать соответственно.
- Блокировать пару запрос-ответ от дальнейшей передачи.
- Изменять заголовки и данные запроса. Вы делаете это, предоставляя кастомную версию запроса.
- Изменить заголовки и данные ответа. Вы делаете это, предоставляя кастомную версию ответа.

- Взаимодействовать с внешними ресурсами.

Фильтры применяются для аутентификации, ведения журнала, преобразования изображений, сжатия данных, шифрования, токенизации потоков, преобразования XML и так далее.

Вы можете настроить веб-ресурс для фильтрации по цепочке из нуля, одного или нескольких фильтров в определённом порядке. Эта цепочка указывается при развёртывании веб-приложения, содержащего компонент, и создаётся при загрузке веб-контейнера компонентом.

Программные фильтры

API фильтрации определяется интерфейсами `Filter`, `FilterChain` и `FilterConfig` пакета `jakarta.servlet`. Вы определяете фильтр путём реализации интерфейса `Filter`.

Используйте аннотацию `@WebFilter`, чтобы зарегистрировать фильтр в веб-приложении. Эта аннотация указывается для класса и содержит метаданные об объявленном фильтре. Аннотированный фильтр должен указывать хотя бы один шаблон URL. Это делается с помощью атрибутов `urlPatterns` или `value` в аннотации. Все остальные атрибуты являются необязательными, с настройками по умолчанию. Используйте атрибут `value`, когда единственным атрибутом в аннотации является шаблон URL. Используйте атрибут `urlPatterns`, когда также используются другие атрибуты.

Классы, аннотированные `@WebFilter` должны реализовывать интерфейс `jakarta.servlet.Filter`.

Чтобы добавить данные конфигурации в фильтр, укажите атрибут `initParams` аннотации `@WebFilter`. Атрибут `initParams` содержит аннотацию `@WebInitParam`. Следующий фрагмент кода определяет фильтр, задающий параметр инициализации:

JAVA

```
import jakarta.servlet.Filter;
import jakarta.servlet.annotation.WebFilter;
import jakarta.servlet.annotation.WebInitParam;

@WebFilter(filterName = "TimeOfDayFilter", urlPatterns = {"/*"},
initParams = {@WebInitParam(name = "mood", value = "awake")})
public class TimeOfDayFilter implements Filter {
    ...
}
```

Наиболее важным методом в интерфейсе `Filter` является `doFilter`, который передаёт объекты запроса, ответа и цепочки фильтров. Этот метод может выполнять следующие действия.

- Исследование заголовков запроса.
- Кастомизация объекта запроса, если фильтр хочет изменить заголовки или данные запроса.
- Кастомизация объекта ответа, если фильтр хочет изменить заголовки или данные ответа.
- Вызов следующей сущности в цепочке фильтров. Если текущий фильтр является последним фильтром в цепочке, которая заканчивается целевым веб-компонентом или статическим ресурсом, следующим объектом является ресурс в конце цепочки. В противном случае это следующий фильтр, который был настроен в WAR. Фильтр вызывает следующую сущность, вызывая метод `doFilter` для объекта цепочки, передавая запрос и ответ, с которыми он был вызван, или изменённые версии, которые он создал. В качестве альтернативы, фильтр может заблокировать запрос, не делая вызова следующего объекта. В последнем случае фильтр отвечает за заполнение ответа.
- Исследование заголовков ответа после вызова следующего фильтра в цепочке.
- Генерация исключения для указания ошибки в обработке.

В дополнение к `doFilter` вы должны реализовать методы `init` и `destroy`. Метод `init` вызывается контейнером при создании объекта фильтра. Если вы хотите передать параметры инициализации в фильтр, извлекайте их из объекта `FilterConfig`, переданного в `init`.

Программирование кастомных запросов и ответов

У фильтра есть много способов изменить запрос или ответ. Например, фильтр может добавить атрибут к запросу или вставить данные в ответ.

Фильтр, который изменяет ответ, должен обычно захватывать ответ прежде, чем он будет возвращён клиенту. Для этого вы передаёте замещающий поток сервлету, который генерирует ответ. Замещающий поток не позволяет сервлету закрывать исходный поток ответов после его завершения и позволяет фильтру изменять ответ сервлета.

Чтобы передать этот замещающий поток сервлету, фильтр создаёт обёртку (wrapper) ответа, которая переопределяет метод `getWriter` или `getOutputStream` для возврата этого замещающего потока. Обёртка (wrapper) передаётся методу `doFilter` цепочки фильтров. По умолчанию методы обёртки (wrapper) обращаются к обёртываемому объекту запроса или ответа.

Чтобы переопределить методы запроса, вы помещаете запрос в объект, который расширяет либо `ServletRequestWrapper`, либо `HttpServletRequestWrapper`. Чтобы переопределить методы ответа, вы оборачиваете ответ в объект, который расширяет либо `ServletResponseWrapper`, либо `HttpServletResponseWrapper`.

Указание назначений для фильтров

Чтобы веб-контейнер мог решить, как применять фильтр к веб-ресурсам, необходимо настроить назначение для фильтров. Назначение фильтров соотносит фильтр с веб-компонентом по имени или с веб-ресурсами по шаблону URL. Фильтры вызываются в том порядке, в котором назначение фильтров идёт в списке назначений фильтров WAR. Список назначений фильтров для WAR указывается в его дескрипторе развёртывания, используя IDE NetBeans или редактируя XML вручную.

Если вы хотите регистрировать каждый запрос в веб-приложении, назначьте фильтр счётчика посещений шаблону URL `/*`.

Вы можете назначить один фильтр одному или нескольким веб-ресурсам, а также назначить более одного фильтра каждому веб-ресурсу. Это проиллюстрировано на рисунке 18-1, в котором фильтру F1 назначены сервлеты S1, S2 и S3, фильтру F2 — сервлет S2, фильтру F3 — сервлеты S1 и S2.

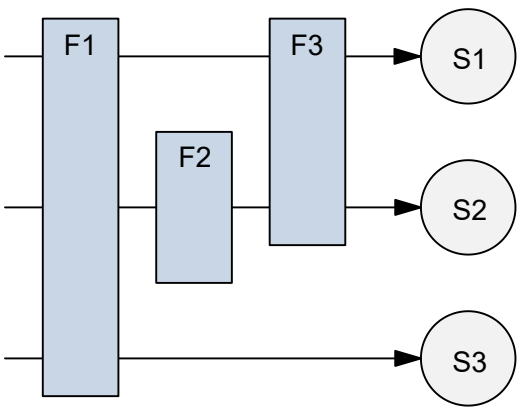

Рис. 18-1. Назначение фильтров сервлетам

Напомним, что цепочка фильтров является одним из объектов, переданных методу `doFilter` фильтра. Эта цепочка формируется косвенно с помощью назначения фильтров. Порядок фильтров в цепочке такой же, как порядок назначения фильтров в дескрипторе развёртывания веб-приложения.

Когда фильтр назначен сервлету S1, веб-контейнер вызывает метод `doFilter` для F1. Метод `doFilter` каждого фильтра в цепочке фильтров S1 вызывается предыдущим фильтром в цепочке с помощью метода `chain.doFilter`. Поскольку цепочка фильтров S1 содержит фильтры F1 и F3, вызов F1 для `chain.doFilter` вызывает метод `doFilter` фильтра F3. Когда метод F3 `doFilter` завершается, управление возвращается к методу F1 `doFilter`.

Указание назначений фильтров в IDE NetBeans

1. Разверните узел проекта приложения во вкладке **Проекты**.
2. Разверните узлы **Веб-страницы** и **WEB-INF** под узлом проекта.
3. Выполните двойной клик на `web.xml`.
4. Кликните **Фильтры** в верхней части окна редактора.
5. Разверните узел **Фильтры сервлетов** в окне редактора.
6. Кликните **Добавить элемент фильтра**, чтобы назначить фильтр веб-ресурсам по имени или по шаблону URL.
7. В диалоговом окне «Добавить фильтр сервлета» введите имя фильтра в поле **Имя фильтра**.
8. Кликните **Обзор**, чтобы найти класс сервлета, к которому применяется фильтр.

   Вы можете включить символы подстановки, чтобы применить фильтр к нескольким сервлетам.
9. Кликните **OK**.
10. Чтобы ограничить применение фильтра к запросам, выполните следующие действия:

    a. Разверните узел **Назначение фильтров**.

    b. Выберите фильтр из списка фильтров.

    c. Кликните **Добавить**.

    d. В диалоговом окне «Добавить назначение фильтра» выберите один из следующих типов диспетчеризации:

REQUEST	Только когда запрос поступает непосредственно от клиента
ASYNC	Только когда от клиента исходит асинхронный запрос
FORWARD	Только в том случае, если запрос был перенаправлен компоненту (см. Передача управления другому веб-компоненту)
INCLUDE	Только когда запрос обрабатывается включённым компонентом (см. Включение других ресурсов в ответ)
ERROR	Только при обработке запроса с помощью механизма страницы ошибок (см. Обработка ошибок сервлета)

Вы можете настроить фильтр для применения к любой комбинации предыдущих ситуаций, выбрав несколько типов диспетчеризации. Если типы не указаны, опция по умолчанию — **REQUEST**.

Вызов других веб-ресурсов

Веб-компоненты могут вызывать другие веб-ресурсы как косвенно, так и напрямую. Веб-компонент косвенно вызывает другой веб-ресурс путём встраивания URL, который указывает на другой веб-компонент в содержимом, возвращаемом клиенту. Во время выполнения веб-компонент напрямую вызывает другой ресурс, включая содержимое другого ресурса или перенаправляя запрос другому ресурсу.

Чтобы вызвать ресурс, доступный на сервере, на котором выполняется веб-компонент, сначала необходимо получить объект `RequestDispatcher` с помощью метода `getRequestDispatcher("URL")`. Вы можете получить объект `RequestDispatcher` из запроса или веб-контекста. Тем не менее, два метода имеют немного различное поведение. Метод принимает путь к запрашиваемому ресурсу в качестве аргумента. Запрос может принимать относительный путь (то есть тот, который не начинается с `/`), но веб-контекст требует указания абсолютного пути. Если ресурс недоступен или если сервер не реализовал объект `RequestDispatcher` для этого типа ресурса, `getRequestDispatcher` вернёт null. Ваш сервлет должен быть готов к такому результату.

Включение других ресурсов в ответ

Часто бывает полезно включить в ответ, возвращаемый веб-компонентом, другой веб-ресурс, например содержимое баннера или информацию об авторских правах. Чтобы включить другой ресурс, вызовите метод `include` объекта `RequestDispatcher`:

```java
include(request, response);
```

Если ресурс статический, метод `include` включает программные включения на стороне сервера. Если ресурс является веб-компонентом, эффект метода состоит в том, чтобы отправить запрос включённому веб-компоненту, выполнить веб-компонент, а затем включить результат выполнения в ответ от содержащегося в нём сервлета. Включённый веб-компонент имеет доступ к объекту запроса, но ограничен в том, что он может делать с объектом ответа.

- Он может записать тело ответа и зафиксировать ответ.
- Он не может устанавливать заголовки или вызывать любой метод, такой как `setCookie`, который влияет на заголовки ответа.

Передача управления другому веб-компоненту

В некоторых приложениях может потребоваться, чтобы один веб-компонент выполнял предварительную обработку запроса, а другой компонент генерировал ответ. Например, вы можете захотеть частично обработать запрос, а затем перенести его в другой компонент, в зависимости от характера запроса.

Чтобы передать управление другому веб-компоненту, нужно вызвать метод `forward` у `RequestDispatcher`. Когда запрос пересылается, URL запроса устанавливается на путь перенаправленной страницы. Исходный URI и его составные части сохраняются как следующие атрибуты запроса:

```java
jakarta.servlet.forward.request_uri
jakarta.servlet.forward.context_path
jakarta.servlet.forward.servlet_path
jakarta.servlet.forward.path_info
jakarta.servlet.forward.query_string
```

Метод `forward` должен использоваться, чтобы передать другому ресурсу ответственность за формирование ответа пользователю. Получив доступ к объекту `ServletOutputStream` или `PrintWriter` внутри сервлета, вы не можете использовать этот метод. При попытке сделать это будет выброшено исключение `IllegalStateException`.

Доступ к веб-контексту

Контекст, в котором выполняются веб-компоненты, является объектом, который реализует интерфейс `ServletContext`. Вы извлекаете веб-контекст с помощью метода `getServletContext`. Веб-контекст предоставляет методы для доступа к:

- параметрам инициализации;
- ресурсам, связанные с веб-контекстом;
- атрибутам значений-объектов;
- возможности ведения журнала.

Методы доступа счётчика синхронизированы для предотвращения несовместимых операций сервлетами, которые работают одновременно. Фильтр получает объект счётчика с помощью метода контекста `getAttribute`. Увеличенное значение счётчика заносится в журнал.

Поддержка состояния клиента

Многие приложения требуют, чтобы ряд запросов от клиента был связан друг с другом. Например, веб-приложение может сохранять состояние корзины покупок пользователя по запросам. Веб-приложения отвечают за поддержание такого состояния, называемого сессией, поскольку HTTP не имеет состояния. Для поддержки приложений, которым необходимо сохранять состояние, технология Jakarta Servlet предоставляет API для управления сессиями и предоставляет несколько механизмов для реализации сессий.

Доступ к сессии

Сессии представлены объектом `HttpSession`. Вы получаете доступ к сессии, вызывая метод `getSession` объекта запроса. Этот метод возвращает текущую сессию, связанную с этим запросом. Или, если в запросе нет сессии, этот метод создаёт её.

Связывание объектов с сессией

Вы можете связать атрибуты значений-объектов с сессией по имени. Такие атрибуты доступны любому веб-компоненту, который принадлежит одному и тому же веб-контексту и обрабатывает запрос, являющийся частью одной и той же сессии.

Напомним, что ваше приложение может уведомлять объекты веб-контекста и слушателя сессии о событиях жизненного цикла сервлета (Обработка событий жизненного цикла сервлета). Вы также можете уведомлять объекты об определённых событиях, связанных с сессией, таких как следующие.

- Когда объект добавляется в сессию или удаляется из неё. Чтобы получить это уведомление, ваш объект должен реализовывать интерфейс `jakarta.servlet.http.HttpSessionBindingListener`.
- Когда сессия, с которой связан объект, активируется или деактивируется. Сессия будет активирована или деактивирована, когда она перемещается между виртуальными машинами или сохраняется и восстанавливается из персистентного хранилища. Чтобы получить это уведомление, объект должен реализовывать интерфейс `jakarta.servlet.http.HttpSessionActivationListener`.

Управление сессиями

Поскольку HTTP-клиент не может сигнализировать о том, что ему больше не нужна сессия, каждая сессия имеет тайм-аут, чтобы её ресурсы могли быть освобождены. Значение тайм-аута можно получить и изменить с помощью методов сессии `getMaxInactiveInterval` и `setMaxInactiveInterval`.

- Чтобы гарантировать, что время активной сессии не истекло, вы должны периодически получать доступ к сессии, используя сервисные методы, поскольку это сбрасывает счётчик времени простоя сессии.
- Когда конкретное взаимодействие с клиентом завершено, вы можете использовать метод `invalidate` сессии, чтобы аннулировать сессию на стороне сервера и удалить все данные сессии.

Установка тайм-аута с IDE NetBeans

Чтобы установить период ожидания в дескрипторе развёртывания с IDE NetBeans, выполните следующие действия.

1. Откройте проект, если вы ещё этого не сделали.
2. Разверните узел проекта на вкладке **Проекты**.
3. Разверните узлы **Веб-страницы** и **WEB-INF**, которые находятся под узлом проекта.
4. Выполните двойной клик на `web.xml`.
5. Кликните **Общие** в верхней части редактора.
6. В поле **Тайм-аут сессии** введите целочисленное значение.

   Целочисленное значение представляет количество минут бездействия, которое должно пройти до истечения тайм-аута сессии.

Отслеживание сессии

Чтобы связать сессию с пользователем, веб-контейнер может использовать несколько методов, каждый из которых включает передачу идентификатора между клиентом и сервером. Идентификатор может храниться на клиенте в виде файла cookie, или веб-компонент может включать идентификатор в каждый URL, который возвращается клиенту.

Если ваше приложение использует объекты сессии, вы должны убедиться, что отслеживание сессии включено, если приложение перезаписывает URL каждый раз, когда клиент отключает использование файлов cookie. Это можно сделать, вызвав метод ответа `encodeURL(URL)` для всех URL, возвращаемых сервлетом. Этот метод включает идентификатор сессии в URL только если cookies отключены. В противном случае метод возвращает URL без изменений.

Финализация сервлета

Веб-контейнер может определить, что сервлет должен быть удалён (например, когда контейнер хочет освободить память или когда он выключен). В таком случае контейнер вызывает метод `destroy` интерфейса `Servlet`. В этом методе вы освобождаете любые ресурсы, которые использует сервлет, и сохраняете любое персистентное состояние. Метод `destroy` освобождает объект базы данных, созданный в методе `init`.

Все сервисные методы сервлета должны быть завершены при удалении сервлета. Сервер пытается убедиться в этом, вызывая метод `destroy` только после того, как все запросы на обслуживание выполнены или по истечении заданного времени (grace period), в зависимости от того, что наступит раньше. Если в вашем сервлете есть операции, которые могут выполняться дольше, чем специальный период сервера (grace period), эти операции могут выполняться при вызове `destroy`. Вы должны убедиться, что все потоки, по-прежнему обрабатывающие клиентские запросы, завершены.

В оставшейся части этого раздела объясняется, как это сделать.

- Отслеживайте, сколько потоков в настоящее время выполняет метод `service`.
- Обеспечьте чистое завершение работы с помощью метода `destroy`, чтобы уведомить долго выполняющиеся потоки о завершении работы и дождаться их завершения.
- Периодически опрашивайте долго выполняющиеся методы для проверки завершения их работы и, при необходимости, их прекращение, очистку и возврат результатов.

Сервис отслеживания запросов

Для отслеживания запросов на обслуживание:

1. Включите в класс сервлета поле, которое подсчитывает количество запущенных сервисных методов.

   Поле должно иметь синхронизированные методы доступа для увеличения, уменьшения и возврата значения:

```java
public class ShutdownExample extends HttpServlet {
    private int serviceCounter = 0;
    ...
    // Методы доступа к serviceCounter
    protected synchronized void enteringServiceMethod() {
        serviceCounter++;
    }
    protected synchronized void leavingServiceMethod() {
        serviceCounter--;
    }
    protected synchronized int numServices() {
        return serviceCounter;
    }
}
```

   Метод `service` должен увеличивать сервисный счётчик каждый раз, когда выполняется вход в метод, и должен уменьшать счётчик каждый раз, когда выполняется выход из метода. Это один из немногих случаев, когда ваш дочерний класс `HttpServlet` должен переопределять метод `service`. Новый метод должен вызывать `super.service` для сохранения функциональности исходного метода `service`:

```java
protected void service(HttpServletRequest req,
                       HttpServletResponse resp)
                       throws ServletException,IOException {
    enteringServiceMethod();
    try {
        super.service(req, resp);
    } finally {
        leavingServiceMethod();
    }
}
```

Уведомление методов о выключении

Чтобы обеспечить полное выключение, ваш метод `destroy` не должен освобождать какие-либо общие ресурсы, пока не будут завершены все запросы на обслуживание:

1. Проверьте сервисный счётчик.
2. Уведомите долго выполняющиеся методы о том, что пора завершать работу.

   Для этого уведомления требуется другое поле. Поле должно иметь обычные методы доступа:

```java
public class ShutdownExample extends HttpServlet {
    private boolean shuttingDown;
    ...
    //Доступ к методам для остановки
    protected synchronized void setShuttingDown(boolean flag) {
        shuttingDown = flag;
    }
    protected synchronized boolean isShuttingDown() {
        return shuttingDown;
    }
}
```

   Вот пример метода `destroy`, использующего эти поля для обеспечения корректного завершения работы:

JAVA

```
    public void destroy() {
        /* Проверка, есть ли ещё выполняющиеся методы /*
        /* и если есть, остановка их. */
        if (numServices() > 0) {
            setShuttingDown(true);
        }

        /* Ожидание остановки метода. */
        while (numServices() > 0) {
            try {
                Thread.sleep(interval);
            } catch (InterruptedException e) {
            }
        }
    }
```

Корректное создание долго выполняющихся методов

Последний шаг в обеспечении чистого завершения работы — заставить все долго выполняющиеся методы вести себя корректно. Долго выполняющиеся методы должны проверять значение поля, которое уведомляет их о выключении, и они должны прервать свою работу, если необходимо:

JAVA

```
public void doPost(...) {
    ...
    for(i = 0; ((i < lotsOfStuffToDo) &&
         !isShuttingDown()); i++) {
        try {
            partOfLongRunningOperation(i);
        } catch (InterruptedException e) {
            ...
        }
    }
}
```

Загрузка файлов с помощью сервлетов Jakarta

Поддержка загрузки файлов является весьма распространённым требованием для многих веб-приложений. В предыдущих версиях спецификации Servlet для загрузки файлов требовалось использование внешних библиотек или сложной обработки ввода. Спецификация сервлета Jakarta теперь обеспечивает жизнеспособное решение проблемы в общем и переносимом виде. Технология Jakarta Servlet теперь поддерживает загрузку файлов "из коробки", поэтому любой веб-контейнер, реализующий спецификацию, может анализировать многочастные (multipart) запросы и делать вложения mime доступными через объект `HttpServletRequest`.

Новая аннотация, `jakarta.servlet.annotation.MultipartConfig`, используется для указания того, что сервлет, на котором она объявлена, ожидает, что запросы будут сделаны с использованием типа MIME `multipart/form-data`. Сервлеты, аннотированные `@MultipartConfig`, могут извлечь объекты `Part` данного запроса `multipart/form-data`, вызвав запрос `request.getPart(String name)` или `request.getParts()`.

Аннотация @MultipartConfig

Аннотация `@MultipartConfig` поддерживает следующие необязательные атрибуты.

- `location`: абсолютный путь к каталогу в файловой системе. Атрибут `location` не поддерживает путь относительно контекста приложения. Это расположение используется для вре́менного хранения файлов во время обработки деталей или когда размер файла превышает заданный параметр `fileSizeThreshold`. Расположение по умолчанию: `""`.

- `fileSizeThreshold` : размер файла в байтах, после которого файл будет временно сохранён на диске. Размер по умолчанию составляет 0 байт.
- `maxFileSize` : максимальный размер загружаемых файлов в байтах. Если размер какого-либо загруженного файла больше этого значения, веб-контейнер выдаст исключение (`IllegalStateException`). По умолчанию размер не ограничен.
- `maxRequestSize` : максимальный размер, разрешённый для запроса `multipart/form-data` , в байтах. Веб-контейнер выдаст исключение, если общий размер всех загруженных файлов превысит этот порог. По умолчанию размер не ограничен.

Например, аннотация `@MultipartConfig` может быть построена следующим образом:

JAVA

```
@MultipartConfig(location="/tmp", fileSizeThreshold=1024*1024,
    maxFileSize=1024*1024*5, maxRequestSize=1024*1024*5*5)
```

Вместо использования аннотации `@MultipartConfig` для жёсткого кодирования (hard-code) этих атрибутов в вашем сервлете загрузки файлов, вы можете добавить следующее как дочерний элемент элемента конфигурации сервлета в файл `web.xml` :

XML

```
<multipart-config>
    <location>/tmp</location>
    <max-file-size>20848820</max-file-size>
    <max-request-size>418018841</max-request-size>
    <file-size-threshold>1048576</file-size-threshold>
</multipart-config>
```

Методы getParts и getPart

Спецификация Servlet поддерживает два дополнительных метода `HttpServletRequest` :

- `Collection<Part> getParts()`
- `Part getPart(String name)`

Метод `request.getParts()` возвращает коллекции всех объектов `Part` . Если у вас на странице есть более одного элемента ввода файла, возвращается несколько объектов `Part` . Поскольку объекты `Part` имеют имена, метод `getPart(String name)` можно использовать для доступа к определённому `Part` . Кроме того, метод `getParts()` , который возвращает `Iterable<Part>` , может использоваться для получения `Iterator` по всем объектам `Part` .

Интерфейс `jakarta.servlet.http.Part` предоставляет методы, которые позволяют работу с каждым объектом `Part` . Методы позволяют делать следующее:

- получить имя, размер и тип содержимого для `Part` ;
- запросить заголовки, отправленные с `Part` ;
- удалить `Part` ;
- записать `Part` на диск.

Например, интерфейс `Part` предоставляет метод `write(String filename)` для записи файла с указанным именем. Затем файл можно сохранить в каталоге, указанном с помощью атрибута `location` аннотации `@MultipartConfig` или, в случае `fileupload` , в месте, указанном в поле «Назначение» формы.

Асинхронная обработка

Веб-контейнеры на серверах приложений обычно используют отдельный поток сервера для каждого запроса клиента. В условиях высокой нагрузки контейнерам требуется большое количество потоков для обслуживания всех клиентских запросов. Ограничения масштабируемости включают нехватку памяти или исчерпание пула потоков контейнера. Для создания масштабируемых веб-приложений вы должны убедиться, что никакие потоки, связанные с запросом, не работают, поэтому контейнер может использовать их для обработки новых запросов.

Существует два распространённых сценария, в которых поток, связанный с запросом, может бездействовать.

- Поток должен дождаться, когда ресурс станет доступным, или обработать данные, прежде чем создавать ответ. Например, приложению может потребоваться запросить базу данных или получить доступ к данным из удалённого веб-сервиса, прежде чем генерировать ответ.
- Поток должен ждать события, прежде чем генерировать ответ. Например, приложение может ожидать сообщения Jakarta Messaging, новой информации от другого клиента или новых данных, доступных в очереди перед генерацией ответа.

Эти сценарии представляют операции блокировки, которые ограничивают масштабируемость веб-приложений. Асинхронная обработка сводится к выполнению этих блокирующих операций в новом потоке и немедленному возврату потока, связанного с запросом, в контейнер.

Асинхронная обработка в сервлетах

Jakarta EE обеспечивает поддержку асинхронной обработки для сервлетов и фильтров. Если сервлет или фильтр достигают потенциально блокирующей операции при обработке запроса, он может назначить операцию асинхронному контексту выполнения и немедленно вернуть поток, связанный с запросом, в контейнер без генерации ответа. Операция блокировки завершается в контексте асинхронного выполнения в другом потоке, который может генерировать ответ или отправлять запрос другому сервлету.

Чтобы включить асинхронную обработку в сервлете, установите для параметра `asyncSupported` значение `true` в аннотации `@WebServlet` следующим образом:

JAVA

```
@WebServlet(urlPatterns={"/asyncservlet"}, asyncSupported=true)
public class AsyncServlet extends HttpServlet { ... }
```

Класс `jakarta.servlet.AsyncContext` предоставляет функциональные возможности, необходимые для выполнения асинхронной обработки внутри методов сервиса. Чтобы получить объект `AsyncContext`, вызовите метод `startAsync()` для объекта запроса вашего сервисного метода. Например:

JAVA

```
public void doGet(HttpServletRequest req, HttpServletResponse resp) {
   ...
   AsyncContext acontext = req.startAsync();
   ...
}
```

Этот вызов переводит запрос в асинхронный режим и гарантирует, что ответ не будет зафиксирован после выхода из сервисного метода. Вы должны сгенерировать ответ в асинхронном контексте после завершения операции блокировки или отправить запрос другому сервлету.

Таблица 18-3 описывает основные функциональные возможности, предоставляемые классом `AsyncContext`.

Таблица 18-3. Функциональность, предоставляемая классом AsyncContext

Сигнатура метода	Описание
`void start(Runnable run)`	Контейнер предоставляет другой поток, в котором может быть обработана операция блокировки. Вы предоставляете код для операции блокировки как класс, который реализует интерфейс `Runnable`. Вы можете предоставить этот класс как внутренний класс при вызове метода `start` или использовать другой механизм для передачи объекта `AsyncContext` в ваш класс.
`ServletRequest getRequest()`	Возвращает запрос, использованный для инициализации этого асинхронного контекста. В приведённом выше примере запрос такой же, как и в сервисном методе. Вы можете использовать этот метод внутри асинхронного контекста для получения параметров из запроса.
`ServletResponse getResponse()`	Возвращает ответ, использованный для инициализации этого асинхронного контекста. В приведённом выше примере ответ такой же, как и в сервисном методе. Вы можете использовать этот метод внутри асинхронного контекста для записи в ответ с результатами операции блокировки.
`void complete()`	Завершает асинхронную операцию и закрывает ответ, связанный с этим асинхронным контекстом. Вызывайте этот метод после записи в объект ответа в асинхронном контексте.
`void dispatch(String path)`	Отправляет объекты запроса и ответа по указанному пути. Этот метод используется для записи другим сервлетом в ответ после завершения операции блокировки.

В ожидании ресурса

В этом разделе показано, как использовать функциональные возможности, предоставляемые классом `AsyncContext` для следующего варианта использования:

1. Сервлет получает параметр из запроса GET.
2. Сервлет использует ресурс, такой как база данных или веб-сервис, для извлечения информации на основе значения параметра. Время от времени ресурс может работать медленно, поэтому это может стать операцией блокировки.
3. Сервлет генерирует ответ, используя результат из ресурса.

Следующий код показывает базовый сервлет, который не использует асинхронную обработку:

JAVA

```
@WebServlet(urlPatterns={"/syncservlet"})
public class SyncServlet extends HttpServlet {
   private MyRemoteResource resource;
   @Override
   public void init(ServletConfig config) {
      resource = MyRemoteResource.create("config1=x,config2=y");
   }

   @Override
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
      response.setContentType("text/html;charset=UTF-8");
      String param = request.getParameter("param");
      String result = resource.process(param);
      /* ... запись ответа ... */
   }
}
```

Следующий код показывает тот же сервлет с использованием асинхронной обработки:

JAVA

```
@WebServlet(urlPatterns={"/asyncservlet"}, asyncSupported=true)
public class AsyncServlet extends HttpServlet {
   /* ... Same variables and init method as in SyncServlet ... */
   @Override
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
      response.setContentType("text/html;charset=UTF-8");
      final AsyncContext acontext = request.startAsync();
      acontext.start(new Runnable() {
         public void run() {
            String param = acontext.getRequest().getParameter("param");
            String result = resource.process(param);
            HttpServletResponse response = acontext.getResponse();
            /* ... print to the response ... */
            acontext.complete();
         }
      });
   }
}
```

`AsyncServlet` добавляет `asyncSupported=true` к аннотации `@WebServlet`. Остальные различия находятся внутри сервисного метода.

- `request.startAsync()` заставляет запрос обрабатываться асинхронно. Ответ не отправляется клиенту в конце сервисного метода.
- `acontext.start(new Runnable() {…})` получает новый поток из контейнера.
- Код внутри метода `run()` внутреннего класса выполняется в новом потоке. Внутренний класс имеет доступ к асинхронному контексту для чтения параметров из запроса и записи в ответ. Вызов метода `complete()` асинхронного контекста фиксирует ответ и отправляет его клиенту.

Служебный метод `AsyncServlet` немедленно возвращается, и запрос обрабатывается в асинхронном контексте.

Неблокирующий ввод/вывод

Веб-контейнеры на серверах приложений обычно используют отдельный поток сервера для каждого запроса клиента. Для разработки масштабируемых веб-приложений вы должны убедиться, что потоки, связанные с клиентскими запросами, никогда не бездействуют, ожидая завершения операции блокировки. Асинхронная

обработка предоставляет механизм для выполнения специфических для приложения операций блокировки в новом потоке, немедленно возвращая поток, связанный с запросом, в контейнер. Даже если вы используете асинхронную обработку для всех специфических для приложения операций блокировки внутри ваших сервисных методов, потоки, связанные с клиентскими запросами, могут мгновение бездействовать в ожидании операций ввода/вывода.

Например, если клиент отправляет большой HTTP запрос POST по медленному сетевому соединению, сервер может прочитать запрос быстрее, чем клиент может его предоставить. При использовании традиционного ввода-вывода поток контейнера, связанный с этим запросом, иногда будет простаивать в ожидании оставшейся части запроса.

Jakarta EE обеспечивает неблокирующую поддержку ввода-вывода для сервлетов и фильтров при обработке запросов в асинхронном режиме. Следующие шаги описывают, как использовать неблокирующий ввод-вывод для обработки запросов и записи ответов внутри сервисных методов.

1. Переведите запрос в асинхронный режим, как описано в Асинхронная обработка.
2. Получите входной поток и/или выходной поток из объектов запроса и ответа в сервисном методе.
3. Назначьте слушатель чтения входному потоку и/или слушатель записи выходному потоку.
4. Обработайте запрос и ответ внутри Callback-методов слушателя.

Втаблице 18-5 описаны методы, доступные во входном и выходном потоках сервлетов для поддержки неблокирующего ввода-вывода. Таблица 18-6 описывает интерфейсы для слушателей чтения и записи.

Таблица 18-4 Поддержка неблокирующего ввода-вывода в jakarta.servlet.ServletInputStream

Метод	**Описание**
`void setReadListener(ReadListener rl)`	Связывает данный входной поток с объектом слушателя, который содержит Callback-методы для асинхронного чтения данных. Вы предоставляете объект слушателя как анонимный класс или используете другой механизм для передачи входного потока в объект слушателя чтения.
`boolean isReady()`	Возвращает true, если данные могут быть прочитаны без блокировки.
`boolean isFinished()`	Возвращает true, когда все данные были прочитаны.

Таблица 18-5 Поддержка неблокирующего ввода-вывода в jakarta.servlet.ServletOutputStream

Метод	**Описание**
`void setWriteListener(WriteListener wl)`	Связывает данный выходной поток с объектом слушателя, который содержит Callback-методы для асинхронной записи данных. Вы предоставляете объект слушателя записи как анонимный класс или используете другой механизм для передачи потока вывода объекту слушателя записи.
`boolean isReady()`	Возвращает true, если данные могут быть записаны без блокировки.

Таблица 18-6 Интерфейсы слушателей для поддержки неблокирующего ввода-вывода

Интерфейс	Методы	Описание

`ReadListener`	`void onDataAvailable()`
	
	`void onAllDataRead()`
	
	`void onError(Throwable t)`
		Объект `ServletInputStream` вызывает эти методы на своём слушателе, когда есть данные, доступные для чтения, когда все данные прочитаны или когда есть ошибка.

`WriteListener`	`void onWritePossible()`
	
	`void onError(Throwable t)`
		Объект `ServletOutputStream` вызывает эти методы на своём слушателе, когда возможно записать данные без блокировки или когда есть ошибка.

Чтение большого HTTP POST-запроса с использованием неблокирующего ввода/вывода

Код в этом разделе показывает, как прочитать большой HTTP POST-запрос внутри сервлета, переведя запрос в асинхронный режим (как описано в Асинхронная обработка) и используя функцию неблокирующего ввода-вывода из таблицы 18-4 и таблицы 18-6.

JAVA

```
@WebServlet(urlPatterns={"/asyncioservlet"}, asyncSupported=true)
public class AsyncIOServlet extends HttpServlet {
   @Override
   public void doPost(HttpServletRequest request,
                      HttpServletResponse response)
                      throws IOException {
      final AsyncContext acontext = request.startAsync();
      final ServletInputStream input = request.getInputStream();

      input.setReadListener(new ReadListener() {
         byte buffer[] = new byte[4*1024];
         StringBuilder sbuilder = new StringBuilder();
         @Override
         public void onDataAvailable() {
            try {
               do {
                  int length = input.read(buffer);
                  sbuilder.append(new String(buffer, 0, length));
               } while(input.isReady());
            } catch (IOException ex) { ... }
         }
         @Override
         public void onAllDataRead() {
            try {
               acontext.getResponse().getWriter()
                                     .write("...the response...");
            } catch (IOException ex) { ... }
            acontext.complete();
         }
         @Override
         public void onError(Throwable t) { ... }
      });
   }
}
```

В этом примере объявляется веб-сервлет с асинхронной поддержкой с использованием параметра аннотации `@WebServlet asyncSupported=true` . Сервисный метод сначала переводит запрос в асинхронный режим, вызывая метод `startAsync()` объекта запроса, который необходим для использования неблокирующего

ввода-вывода. Затем сервисный метод получает входной поток, связанный с запросом, и назначает слушатель чтения, определённый как внутренний класс. Слушатель читает части запроса, когда они становятся доступными, и затем записывает некоторый ответ клиенту, когда он заканчивает чтение запроса.

Серверный Push

Серверный Push — это способность сервера предвидеть, что будет необходимо клиенту, до его запроса. Это позволяет серверу предварительно заполнить кэш браузера до того, как браузер запросит ресурс для помещения в кэш.

Серверный Push — это наиболее заметное из улучшений в HTTP/2, появившееся в API сервлетов. Все новые функции в HTTP/2, включая серверный Push, направлены на повышение производительности работы в Интернете.

Серверный Push вносит свой вклад в повышение производительности браузера из-за того, что серверы знают, какие дополнительные ресурсы (такие как изображения, таблицы стилей и скрипты) соответствуют начальным запросам. Например, серверы могут знать, что всякий раз, когда браузер запрашивает `index.html`, он вскоре после этого запрашивает `header.gif`, `footer.gif` и `style.css`. Серверы могут преимущественно начать отправку байтов этих ресурсов вместе с байтами `index.html`.

Чтобы использовать серверный Push, получите ссылку на `PushBuilder` из `HttpServletRequest`, отредактируйте его по своему желанию, затем вызовите `push()`. См. javadoc (https://jakarta.ee/specifications/platform/9/apidocs/) для класса `jakarta.servlet.http.PushBuilder` и метода `jakarta.servlet.http.HttpServletRequest.newPushBuilder()`.

Обработка обновления протокола

В HTTP/1.1 клиенты могут запросить переключение на другой протокол в текущем соединении, используя поле заголовка `Upgrade`. Если сервер принимает запрос переключения на протокол, указанный клиентом, он генерирует HTTP-ответ со статусом 101 (переключение протоколов). После этого обмена клиент и сервер обмениваются данными по новому протоколу.

Например, клиент может сделать HTTP-запрос для переключения на протокол XYZP следующим образом:

```HTTP
GET /xyzpresource HTTP/1.1
Host: localhost:8080
Accept: text/html
Upgrade: XYZP
Connection: Upgrade
OtherHeaderA: Value
```

Клиент может указать параметры для нового протокола, используя заголовки HTTP. Сервер может принять запрос и сгенерировать ответ следующим образом:

```HTTP
HTTP/1.1 101 Switching Protocols
Upgrade: XYZP
Connection: Upgrade
OtherHeaderB: Value

(XYZP data)
```

Jakarta EE поддерживает функцию обновления протокола HTTP в сервлетах, как описано в табл. 18-7.

Таблица 18-7. Поддержка обновления протокола

Класс или интерфейс	**Метод**
`HttpServletRequest`	`HttpUpgradeHandler upgrade(Class handler)` Метод обновления запускает обработку обновления протокола. Этот метод создаёт объект класса, который реализует интерфейс `HttpUpgradeHandler` и делегирует ему соединение. Вызывайте метод `upgrade` внутри сервисного метода, когда принимаете запрос от клиента на переключение протоколов.
`HttpUpgradeHandler`	`void init(WebConnection wc)` Метод `init` вызывается, когда сервлет принимает запрос переключения протокола. Реализуйте этот метод и получите входные и выходные потоки из объекта `WebConnection` для реализации нового протокола.
`HttpUpgradeHandler`	`void destroy()` Метод `destroy` вызывается, когда клиент выключается. Реализуйте этот метод и освободите все ресурсы, связанные с обработкой нового протокола.
`WebConnection`	`ServletInputStream getInputStream()` Метод `getInputStream` обеспечивает доступ к входному потоку соединения. Вы можете использовать неблокирующий ввод/вывод с возвращённым потоком для реализации нового протокола.
`WebConnection`	`ServletOutputStream getOutputStream()` Метод `getOutputStream` обеспечивает доступ к выходному потоку соединения. Вы можете использовать неблокирующий ввод/вывод с возвращённым потоком для реализации нового протокола.

Следующий код демонстрирует, как принять запрос на обновление протокола HTTP от клиента:

JAVA

```
@WebServlet(urlPatterns={"/xyzpresource"})
public class XYZPUpgradeServlet extends HttpServlet {
   @Override
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
      if ("XYZP".equals(request.getHeader("Upgrade"))) {
         /* Принятие запроса upgrade */
         response.setStatus(101);
         response.setHeader("Upgrade", "XYZP");
         response.setHeader("Connection", "Upgrade");
         response.setHeader("OtherHeaderB", "Value");
         /* Делегирование соединения обработчику upgrade */
         XYZPUpgradeHandler = request.upgrade(XYZPUpgradeHandler.class);
         /* (возврат управление немедленно) */
      } else {
         /* ... запись ошибки в ответ... */
      }
   }
}
```

Класс `XYZPUpgrityHandler` обрабатывает подключение:

JAVA

```
public class XYZPUpgradeHandler implements HttpUpgradeHandler {
   @Override
   public void init(WebConnection wc) {
      ServletInputStream input = wc.getInputStream();
      ServletOutputStream output = wc.getOutputStream();
      /* ... реализация XYZP на stream-ах (зависит от protocol) ... */
   }
   @Override
   public void destroy() { ... }
}
```

Класс, который реализует `HttpUpgradeHandler`, использует потоки из текущего соединения для связи с клиентом, используя новый протокол. См. спецификацию Servlet 5.0 по ссылке https://jakarta.ee/specifications/servlet/5.0 для получения подробной информации о поддержке функции HTTP protocol upgrade.

HTTP Trailer

HTTP-трейлер — это коллекция особого типа HTTP-заголовков, которые идут после тела ответа. Заголовок ответа трейлера позволяет отправителю включать дополнительные поля в конце фрагментированных сообщений, чтобы предоставлять метаданные, которые могут динамически генерироваться при отправке тела сообщения, такие как проверка целостности сообщения, цифровая подпись или статус постобработки.

Если заголовки трейлера готовы к чтению, `isTrailerFieldsReady()` вернёт `true`. Затем сервлет может читать заголовки трейлера HTTP-запроса, используя метод `getTrailerFields` интерфейса `HttpServletRequest`. Если заголовки трейлера не готовы к чтению, `isTrailerFieldsReady()` возвращает `false` и вызовет исключение `IllegalStateException`.

Сервлет может записывать заголовки трейлера в ответ, предоставляя поставщика в метод `setTrailerFields()` интерфейса `HttpServletResponse`. Следующие заголовки и типы заголовков *не* должны быть включены в набор ключей в отображении (Map), передаваемом в `setTrailerFields()`: `Transfer-Encoding`, `Content-Length`, `Host`, элементы управления и условные заголовки, заголовки аутентификации, `Content-Encoding`, `Content-Type`, `Content-Range` и `Trailer`. При отправке трейлеров ответов вы должны

включить обычный заголовок `Trailer`, значение которого представляет собой разделённый запятыми список всех ключей в отображении (Map), которые передаются в метод `setTrailerFields()`. Значение заголовка `Trailer` позволяет клиенту знать, каких трейлеров ожидать.

Поставщик заголовков трейлера можно получить, обратившись к методу `getTrailerFields()` интерфейса `HttpServletResponse`.

Смотрите javadoc (https://jakarta.ee/specifications/platform/9/apidocs/) для `getTrailerFields()` и `isTrailerFieldsReady()` в `HttpServletRequest` и `getTrailerFields()` и `setTrailerFields()` в `HttpServletResponse`.

Приложение mood

Приложение `mood`, расположенное в каталоге *`tut-install`*`/examples/web/servlet/mood/`, является простым примером, который отображает настроения Дюка в разное время в течение дня. В примере показано, как разработать простое приложение с помощью аннотаций `@WebServlet`, `@WebFilter` и `@WebListener` для создания сервлета, слушателя и фильтра.

Компоненты примера mood

Приложение `mood` состоит из трёх компонентов: `mood.web.MoodServlet`, `mood.web.TimeOfDayFilter` и `mood.web.SimpleServletListener`.

`MoodServlet`, уровень представления приложения, отображает настроение Дюка в виде графика в зависимости от времени суток. Аннотация `@WebServlet` указывает шаблон URL:

JAVA

```
@WebServlet("/report")
public class MoodServlet extends HttpServlet {
    ...
}
```

`TimeOfDayFilter` устанавливает параметр инициализации, указывающий, что Дюк бодрствует:

JAVA

```
@WebFilter(filterName = "TimeOfDayFilter",
urlPatterns = {"/*"},
initParams = {
    @WebInitParam(name = "mood", value = "awake")})
public class TimeOfDayFilter implements Filter {
    ...
}
```

Фильтр вызывает метод `doFilter`, который содержит инструкцию `switch`, устанавливающую настроение Дюка в зависимости от текущего времени.

`SimpleServletListener` регистрирует изменения в жизненном цикле сервлета. Записи журнала появляются в журнале сервера.

Запуск примера mood

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска примера `mood`.

Запуск mood с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.

3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/web/servlet
   ```

4. Выберите каталог `mood` .
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `mood` и выберите **Сборка**.
7. В веб-браузере введите следующий URL:

   ```
   http://localhost:8080/mood/report
   ```

   URL указывает корень контекста, за которым следует шаблон URL.

   Появится веб-страница с заголовком «Servlet MoodServlet at /mood», текстовая строка, описывающая настроение Дюка, и иллюстрирующий рисунок.

Запуск mood с помощью Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/web/servlet/mood/
   ```

3. Введите следующую команду для развёртывания приложения:

   ```
   mvn install
   ```
   SHELL

4. В веб-браузере введите следующий URL:

   ```
   http://localhost:8080/mood/report
   ```

   URL указывает корень контекста, за которым следует шаблон URL.

   Появится веб-страница с заголовком «Servlet MoodServlet at /mood», текстовая строка, описывающая настроение Дюка, и иллюстрирующий рисунок.

Приложение fileupload

В примере `fileupload` , расположенном в каталоге *`tut-install`*`/examples/web/servlet/fileupload/` , показано, как реализовать и использовать функцию передачи файлов.

Пример Duke's Forest содержит более сложный пример, который загружает файл изображения и сохраняет его содержимое в базе данных.

За исключением случаев, когда прямо сказано иное, сайт и весь контент, предоставляемый на сайте или через него, предоставляются на условиях «как есть» и «по мере доступности». Oracle однозначно отказывается ото всех гарантий любого рода, явных или подразумеваемых, включая, помимо прочего, подразумеваемые гарантии товарной пригодности, пригодности для конкретной цели и отсутствия нарушений в отношении сайта и всего контента, предоставляемого на сайте или через сайт. Oracle не даёт никаких гарантий, что: (а) сайт или контент будут соответствовать вашим требованиям; (б) сайт будет доступен бесперебойно, своевременно, безопасно или без ошибок; (в) результаты, которые могут быть получены при использовании сайта или любого контента, размещённого на сайте или через сайт, будут точными или надёжными; (г) качество любого контента, приобретенного или полученного вами на сайте или через сайт, будет соответствовать вашим ожиданиям.

Любой контент, к которому можно получить доступ, загрузить или каким-либо иным образом получить или использовать сайт, используется на ваше усмотрение и на ваш риск. Oracle не несет ответственности за любой ущерб вашей компьютерной системе или потерю данных в результате загрузки или использования контента.

Архитектура приложения fileupload

Приложение `fileupload` состоит из одного сервлета и HTML-формы, которая отправляет запрос на загрузку файла в сервлет.

Этот пример включает в себя очень простую HTML-форму с двумя полями, File и Destination. Тип `file` компонента ввода позволяет пользователю просматривать локальную файловую систему и выбирать файл. Когда файл выбран, он отправляется на сервер как часть запроса POST. В ходе этого процесса к форме с типом ввода `file` применяются два обязательных ограничения.

- Атрибут `enctype` должен быть установлен в значение `multipart/form-data`.
- Его метод должен быть POST.

Когда форма объявлена таким образом, весь запрос отправляется на сервер в закодированной форме. Затем сервлет использует свои собственные средства для обработки запроса, чтобы обработать входящие данные файла и извлечь файл из потока. Destination — это путь к каталогу, в котором файл будет сохранён на вашем компьютере. Клик кнопки «Загрузить» в нижней части формы отправляет данные в сервлет, который сохраняет файл в указанном месте.

Форма HTML в `index.html` выглядит следующим образом:

HTML

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>File Upload</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    </head>
    <body>
        <form method="POST" action="upload" enctype="multipart/form-data" >
            File:
            <input type="file" name="file" id="file" /> <br/>
            Destination:
            <input type="text" value="/tmp" name="destination"/>
            <br/>
            <input type="submit" value="Upload" name="upload" id="upload" />
        </form>
    </body>
</html>
```

Метод запроса POST используется, когда клиенту необходимо отправить данные на сервер как часть запроса, например, при загрузке файла или отправке заполненной формы. Метод запроса GET, напротив, отправляет на сервер только URL и заголовки, тогда как запросы POST также включают тело сообщения. Это позволяет отправлять на сервер данные произвольной длины любого типа. Поле заголовка в запросе POST обычно указывает тип MIME тела сообщения.

При отправке формы браузер передаёт содержимое, объединяя все части, причём каждая часть представляет поле формы. Части имеют такие же названия, как и соответствующие им элементы `input` и отделены друг от друга разделителями строк с именем `border`.

Вот как выглядят отправленные данные из формы `fileupload` после выбора файла `sample.txt`, который будет загружен из каталога `tmp` на локальной файловой системе:

HTTP

```
POST /fileupload/upload HTTP/1.1
Host: localhost:8080
Content-Type: multipart/form-data;
boundary=---------------------------263081694432439 Content-Length: 441
-----------------------------263081694432439
Content-Disposition: form-data; name="file"; filename="sample.txt"
Content-Type: text/plain
 Data from sample file
-----------------------------263081694432439
Content-Disposition: form-data; name="destination"
 /tmp
-----------------------------263081694432439
Content-Disposition: form-data; name="upload"
 Upload
-----------------------------263081694432439--
```

Сервлет `FileUploadServlet.java` начинается следующим образом:

JAVA

```
@WebServlet(name = "FileUploadServlet", urlPatterns = {"/upload"})
@MultipartConfig
public class FileUploadServlet extends HttpServlet {
    private final static Logger LOGGER =
            Logger.getLogger(FileUploadServlet.class.getCanonicalName());
}
```

Аннотация `@WebServlet` использует свойство `urlPatterns` для определения отображений сервлета.

Аннотация `@MultipartConfig` указывает, что сервлет ожидает выполнения запросов с использованием типа MIME `multipart/form-data`.

Метод `processRequest` извлекает пункт назначения и файловую часть из запроса, затем вызывает метод `getFileName` для получения имени файла из файловой части. Затем метод создаёт `FileOutputStream` и копирует файл в указанное место. Раздел обработки ошибок метода ловит и обрабатывает некоторые из наиболее распространённых причин, по которым файл не будет найден. Методы `processRequest` и `getFileName` выглядят так:

JAVA

```
protected void processRequest(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");

    // Создание компонентов пути для сохранения в файл
    final String path = request.getParameter("destination");
    final Part filePart = request.getPart("file");
    final String fileName = getFileName(filePart);

    OutputStream out = null;
    InputStream filecontent = null;
    final PrintWriter writer = response.getWriter();

    try {
        out = new FileOutputStream(new File(path + File.separator
                + fileName));
        filecontent = filePart.getInputStream();

        int read = 0;
        final byte[] bytes = new byte[1024];

        while ((read = filecontent.read(bytes)) != -1) {
            out.write(bytes, 0, read);
        }
        writer.println("New file " + fileName + " created at " + path);
        LOGGER.log(Level.INFO, "File{0}being uploaded to {1}",
                new Object[]{fileName, path});
    } catch (FileNotFoundException fne) {
        writer.println("You either did not specify a file to upload or are "
                + "trying to upload a file to a protected or nonexistent "
                + "location.");
        writer.println("<br/> ERROR: " + fne.getMessage());

        LOGGER.log(Level.SEVERE, "Problems during file upload. Error: {0}",
                new Object[]{fne.getMessage()});
    } finally {
        if (out != null) {
            out.close();
        }
        if (filecontent != null) {
            filecontent.close();
        }
        if (writer != null) {
            writer.close();
        }
    }
}

private String getFileName(final Part part) {
    final String partHeader = part.getHeader("content-disposition");
    LOGGER.log(Level.INFO, "Part Header = {0}", partHeader);
    for (String content : part.getHeader("content-disposition").split(";")) {
        if (content.trim().startsWith("filename")) {
            return content.substring(
                    content.indexOf('=') + 1).trim().replace("\"", "");
        }
    }
    return null;
}
```

Запуск примера fileupload

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска примера `fileupload`.

Сборка, упаковка и развёртывание примера fileupload с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/web/servlet
   ```

4. Выберите каталог `fileupload`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `fileupload` и выберите **Сборка**.

Сборка, упаковка и развёртывание примера fileupload с помощью Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/web/servlet/fileupload/
   ```

3. Введите следующую команду для развёртывания приложения:

   ```
   mvn install
   ```
   SHELL

Запуск примера fileupload

1. В веб-браузере введите следующий URL:

   ```
   http://localhost:8080/fileupload/
   ```

2. На странице загрузки файла нажмите «Выбрать файл», чтобы открыть окно браузера файлов.
3. Выберите файл для загрузки и нажмите «Открыть».

   Имя выбранного файла отображается в поле «Файл». Если вы не выберете файл, будет сгенерировано исключение.

4. В поле Destination введите имя каталога.

   Каталог должен быть уже создан и должен быть доступен для записи. Если вы не введёте имя каталога или имя несуществующего или защищённого каталога, возникнет исключение.

5. Нажмите «Загрузить», чтобы загрузить выбранный файл в каталог, указанный в поле «Место назначения».

   Сообщение сообщает, что файл был создан в указанном вами каталоге.

6. Перейдите в каталог, который вы указали в поле Destination, и убедитесь, что загруженный файл присутствует.

Приложение dukeetf

Пример приложения `dukeetf`, расположенный в каталоге *`tut-install`*`/examples/web/dukeetf/`, демонстрирует, как использовать асинхронную обработку в сервлете для предоставления обновлений данных веб-клиентам. Пример напоминает сервис, который предоставляет периодические обновления цены и объёма торгов биржевым инвестиционным фондом (ETF).

Архитектура приложения dukeetf

Приложение `dukeetf` состоит из сервлета, Enterprise-бина и страницы HTML.

- Сервлет помещает запросы в асинхронном режиме, сохраняет их в очереди и записывает ответы, когда становятся доступными новые данные о цене и объёме торгов.
- Enterprise-бин обновляет информацию о цене и объёме каждую секунду.
- HTML-страница использует код JavaScript, чтобы отправлять сервлету запросы на новые данные, анализировать ответ сервлета и обновлять информацию о цене и объёме без перезагрузки страницы.

В примере приложения `dukeetf` используется программная модель, известная как long polling. В традиционной модели HTTP-запросов и ответов пользователь должен сделать явный запрос (например, кликнуть ссылку или отправить форму), чтобы получить любую новую информацию с сервера, и страница должна быть перезагружена. Long polling предоставляет веб-приложениям механизм для отправки обновлений клиентам, использующим HTTP, без явного запроса пользователя. Сервер обрабатывает соединения асинхронно, а клиент использует JavaScript для создания новых соединений. В этой модели клиенты делают новый запрос сразу после получения новых данных, и сервер сохраняет соединение открытым, пока новые данные не станут доступны.

Сервлет

Класс `DukeETFServlet` использует асинхронную обработку:

JAVA

```
@WebServlet(urlPatterns={"/dukeetf"}, asyncSupported=true)
public class DukeETFServlet extends HttpServlet {
...
}
```

В следующем коде метод `init` инициализирует очередь для хранения клиентских запросов и регистрирует сервлет с Enterprise-бином, который предоставляет обновления цены и объёма. Раз в секунду вызывается метод `send` у `PriceVolumeBean` для отправки обновлений и закрытия соединения:

JAVA

```
@Override
public void init(ServletConfig config) {
   /* Очередь для запросов */
   requestQueue = new ConcurrentLinkedQueue<>();
   /* Регистрация с Enterprise-бином, предоставляющим обновление цены и объёма */
   pvbean.registerServlet(this);
}

/* PriceVolumeBean вызываем этот метод каждую секунду для отправки обновлений */
public void send(double price, int volume) {
   /* Отправка обновлений всем подключённым клиентам */
   for (AsyncContext acontext : requestQueue) {
      try {
         String msg = String.format("%.2f / %d", price, volume);
         PrintWriter writer = acontext.getResponse().getWriter();
         writer.write(msg);
         logger.log(Level.INFO, "Sent: {0}", msg);
         /* Закрытие соединения
          * Клиент (JavaScript) создаст новое */
         acontext.complete();
      } catch (IOException ex) {
         logger.log(Level.INFO, ex.toString());
      }
   }
}
```

Сервисный метод переводит клиентские запросы в асинхронный режим и добавляет слушатель к каждому запросу. Слушатель реализован как анонимный класс, который удаляет запрос из очереди, когда сервлет заканчивает писать ответ или когда возникает ошибка. Наконец, метод service добавляет запрос в очередь запросов, созданную в методе `init`. Сервисный метод описан следующим образом:

JAVA

```
@Override
public void doGet(HttpServletRequest request,
                  HttpServletResponse response) {
   response.setContentType("text/html");
   /* Перевод запроса в асинхронный режим */
   final AsyncContext acontext = request.startAsync();
   /* Удаление из очереди после завершения */
   acontext.addListener(new AsyncListener() {
      public void onComplete(AsyncEvent ae) throws IOException {
         requestQueue.remove(acontext);
      }
      public void onTimeout(AsyncEvent ae) throws IOException {
         requestQueue.remove(acontext);
      }
      public void onError(AsyncEvent ae) throws IOException {
         requestQueue.remove(acontext);
      }
      public void onStartAsync(AsyncEvent ae) throws IOException {}
   });
   /* Добавление в очередь */
   requestQueue.add(acontext);
}
```

Enterprise-бин

Класс `PriceVolumeBean` является Enterprise-бином, который использует сервис таймера из контейнера для обновления информации о цене и объёме и вызывает метод `send` сервлета раз в секунду:

JAVA

```
@Startup
@Singleton
public class PriceVolumeBean {
    /* Использование сервиса таймера контейнера */
    @Resource TimerService tservice;
    private DukeETFServlet servlet;
    ...

    @PostConstruct
    public void init() {
        /* Инициализация EJB и создание таймера */
        random = new Random();
        servlet = null;
        tservice.createIntervalTimer(1000, 1000, new TimerConfig());
    }

    public void registerServlet(DukeETFServlet servlet) {
        /* Установление сервлета для отправки обновлений */
        this.servlet = servlet;
    }

    @Timeout
    public void timeout() {
        /* Вычисление цены и объёма и отправка обновления */
        price += 1.0*(random.nextInt(100)-50)/100.0;
        volume += random.nextInt(5000) - 2500;
        if (servlet != null)
            servlet.send(price, volume);
    }
}
```

См. Использование сервиса таймера в главе 37 *Выполнение примеров Enterprise-бинов* для получения дополнительной информации о сервисе таймера.

HTML-страница

HTML-страница состоит из таблицы и кода JavaScript. Таблица содержит два поля, на которые ссылается код JavaScript:

HTML

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>...</head>
<body onload="makeAjaxRequest();">
  ...
  <table>
    ...
    <td id="price">--.--</td>
    ...
    <td id="volume">--</td>
    ...
  </table>
</body>
</html>
```

Код JavaScript использует API `XMLHttpRequest`, который обеспечивает функциональность для передачи данных между клиентом и сервером. Скрипт выполняет асинхронный запрос к сервлету и назначает Callback-метод. Когда сервер предоставляет ответ, Callback-метод обновляет поля в таблице и создаёт новый запрос. Код JavaScript выглядит следующим образом:

JAVASCRIPT

```
var ajaxRequest;
function updatePage() {
   if (ajaxRequest.readyState === 4) {
      var arraypv = ajaxRequest.responseText.split("/");
      document.getElementById("price").innerHTML = arraypv[0];
      document.getElementById("volume").innerHTML = arraypv[1];
      makeAjaxRequest();
   }
}
function makeAjaxRequest() {
   ajaxRequest = new XMLHttpRequest();
   ajaxRequest.onreadystatechange = updatePage;
   ajaxRequest.open("GET", "http://localhost:8080/dukeetf/dukeetf",
                    true);
   ajaxRequest.send(null);
}
```

API `XMLHttpRequest` поддерживается большинством современных браузеров и широко используется при разработке веб-клиента Ajax (асинхронный JavaScript и XML).

См. Пример dukeetf2 в главе 19 *Веб-сокеты Jakarta* для эквивалентной версии этого примера, реализованной с использованием конечной точки веб-сокета.

Запуск приложения dukeetf

В этом разделе описывается, как запустить пример `dukeetf` в IDE NetBeans и из командной строки.

Запуск приложения dukeetf в IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне «Открыть проект» перейдите к:

```
tut-install/examples/web/servlet
```

4. Выберите каталог `dukeetf`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `dukeetf` и выберите **Запуск**.

   Эта команда собирает и упаковывает приложение в WAR-файл (`dukeetf.war`), расположенный в каталоге `target`, развёртывает его на сервере и запускает окно веб-браузера со следующим URL:

   ```
   http://localhost:8080/dukeetf/
   ```

   Откройте этот же URL в другом веб-браузере, чтобы увидеть, как обе страницы одновременно получают обновления цены и объёма.

Запуск приложения dukeetf с помощью Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/web/servlet/dukeetf/
   ```

3. Введите следующую команду для развёртывания приложения:

   ```
   mvn install
   ```
   SHELL

4. Откройте окно веб-браузера и введите следующий URL:

   ```
   http://localhost:8080/dukeetf/
   ```

   Откройте этот же URL в другом веб-браузере, чтобы увидеть, как обе страницы одновременно получают обновления цены и объёма.

Дополнительная информация о сервлета Jakarta

Для получения дополнительной информации о технологии Jakarta Servlet см. спецификацию Jakarta Servlet 5.0 по ссылке https://jakarta.ee/specifications/servlet/5.0/.

Глава 19. Jakarta WebSocket

В этой главе описываются веб-сокеты Jakarta, которые обеспечивает поддержку создания приложений веб-сокетов. Веб-сокеты — это прикладной протокол, который обеспечивает полнодуплексный канал связи между двумя узлами по протоколу TCP.

Введение в веб-сокеты

В традиционной модели запрос-ответ, используемой в HTTP, клиент запрашивает ресурсы, а сервер предоставляет ответы. Обмен всегда инициируется клиентом. Сервер не может отправить какие-либо данные, если клиент их не запрашивал. Эта модель хорошо работала для Интернета, когда клиенты время от времени делали запросы на документы, которые менялись редко. Однако недостатки этого подхода становятся всё более актуальными, поскольку контент быстро меняется, и пользователи ожидают всё более интерактивного взаимодействия. Протокол веб-сокетов устраняет эти недостатки, предоставляя полнодуплексный канал связи между клиентом и сервером. В сочетании с другими клиентскими технологиями, такими как JavaScript и HTML5, веб-сокеты позволяют веб-приложениям предоставлять пользователям больше возможностей.

В приложении веб-сокетов сервер публикует конечную точку веб-сокета, а клиент использует URI конечной точки для подключения к серверу. Протокол веб-сокета является симметричным после установления соединения. Клиент и сервер могут отправлять сообщения друг другу в любое время, когда соединение открыто. Они могут закрыть соединение в любое время. Клиенты обычно подключаются только к одному серверу, а серверы принимают подключения от нескольких клиентов.

Протокол веб-сокетов состоит из двух частей: установление соединения и передача данных. Клиент инициирует соединение, отправляя запрос конечной точке веб-сокета, используя его URI. Установление соединения совместимо с существующей HTTP-инфраструктурой: веб-серверы интерпретируют его как запрос на обновление HTTP-соединения. Пример установления соединения от клиента выглядит следующим образом:

```HTTP
GET /path/to/websocket/endpoint HTTP/1.1
Host: localhost
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: xqBt3ImNzJbYqRINxEFlkg==
Origin: http://localhost
Sec-WebSocket-Version: 13
```

Пример ответа сервера при получении запроса клиента на установление соединения:

```HTTP
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: K7DJLdLooIwIG/MOpvWFB3y3FE8=
```

Сервер применяет известную операцию к значению заголовка `Sec-WebSocket-Key`, чтобы сгенерировать значение заголовка `Sec-WebSocket-Accept`. Клиент применяет ту же операцию к значению заголовка `Sec-WebSocket-Key`, и соединение устанавливается успешно, если результат соответствует значению, полученному с сервера. Клиент и сервер могут отправлять сообщения друг другу после того, как соединение успешно установлено.

Веб-сокеты поддерживают текстовые (в кодировке UTF-8) и бинарные сообщения. Управляющие команды в веб-сокетах — это close, ping и pong (ответ на команду ping). Команды ping и pong также могут содержать данные приложения.

Конечные точки веб-сокетов представлены URI, которые имеют вид:

```
ws://host:port/path?query
wss://host:port/path?query
```

Схема `ws` представляет незашифрованное соединение веб-сокета, а схема `wss` — зашифрованное соединение. Компонент `port` является необязательным. Номер порта по умолчанию — 80 для незашифрованных соединений и 443 для зашифрованных. Компонент `path` указывает местоположение конечной точки на сервере. Компонент `query` является необязательным.

Современные веб-браузеры реализуют протокол веб-сокетов и предоставляют JavaScript API для подключения к конечным точкам, отправки сообщений и назначения Callback-методов для событий веб-сокетов (таких как открытие и закрытие соединения, получение сообщения).

Создание приложений веб-сокетов в платформе Jakarta EE

Платформа Jakarta EE включает в себя веб-сокеты Jakarta, которые позволяет создавать, настраивать и развёртывать конечные точки веб-сокетов в веб-приложениях. Клиентский API веб-сокетов, указанный в Jakarta WebSocket, также позволяет получать доступ к удалённым конечным точкам веб-сокетов из любого приложения Java.

Jakarta WebSocket состоит из следующих пакетов.

- Пакет `jakarta.websocket.server` содержит аннотации, классы и интерфейсы для создания и настройки серверных конечных точек.
- Пакет `jakarta.websocket` содержит аннотации, классы, интерфейсы и исключения, общие для клиентских и серверных конечных точек.

Конечные точки веб-сокета являются объектами класса `jakarta.websocket.Endpoint` . Jakarta WebSocket позволяет создавать два вида конечных точек: программные конечные точки и аннотированные конечные точки. Чтобы создать программную конечную точку, расширьте класс `Endpoint` и переопределите его методы жизненного цикла. Чтобы создать аннотированную конечную точку, вы отмечаете класс Java и некоторые его методы аннотациями, предоставленными пакетами, упомянутыми ранее. После создания конечной точки вы развёртываете её в определённом URI в приложении, чтобы удалённые клиенты могли к ней подключаться.

> В большинстве случаев проще создать и развернуть аннотированную конечную точку, чем программную. Эта глава сфокусирована на аннотированных конечных точках, хотя ниже приведён простой пример программной конечной точки.

Создание и развёртывание конечной точки веб-сокета

Процесс создания и развёртывания конечной точки веб-сокета:

1. Создайте класс конечной точки.
2. Реализуйте методы жизненного цикла конечной точки.
3. Добавьте свою бизнес-логику в конечную точку.

4. Разверните конечную точку внутри веб-приложения.

Процесс немного отличается для программных конечных точек и аннотированных конечных точек, и он подробно рассматривается в следующих разделах.

> В отличие от сервлетов конечные точки веб-сокетов создаются несколько раз. Контейнер создаёт объект конечной точки для каждого соединения с его URI развёртывания. Каждый объект связан с одним и только одним соединением. Это облегчает сохранение состояния пользователя для каждого соединения и облегчает разработку, поскольку в любой момент времени только один поток выполняет код объекта конечной точки.

Программные конечные точки

В следующем примере показано, как создать конечную точку путём расширения класса `Endpoint`:

JAVA

```
public class EchoEndpoint extends Endpoint {
   @Override
   public void onOpen(final Session session, EndpointConfig config) {
      session.addMessageHandler(new MessageHandler.Whole<String>() {
         @Override
         public void onMessage(String msg) {
            try {
               session.getBasicRemote().sendText(msg);
            } catch (IOException e) { ... }
         }
      });
   }
}
```

Эта конечная точка повторяет каждое полученное сообщение. Класс `Endpoint` определяет три метода жизненного цикла: `onOpen`, `onClose` и `onError`. Класс `EchoEndpoint` реализует метод `onOpen`, который является единственным абстрактным методом в классе `Endpoint`.

Параметр `Session` представляет диалог между этой конечной точкой и удалённой конечной точкой. Метод `addMessageHandler` регистрирует обработчики сообщений, а метод `getBasicRemote` возвращает объект, представляющий удалённую конечную точку. Интерфейс `Session` подробно рассматривается в оставшейся части этой главы.

Обработчик сообщений реализован как анонимный внутренний класс. Метод `onMessage` обработчика сообщений вызывается, когда конечная точка получает текстовое сообщение.

Чтобы развернуть эту программную конечную точку, используйте следующий код в приложении Jakarta EE:

JAVA

```
ServerEndpointConfig.Builder.create(EchoEndpoint.class, "/echo").build();
```

При развёртывании приложения конечная точка доступна по ссылке `ws://<хост>:<порт>/<приложение>/echo`. Например, `ws://localhost:8080/echoapp/echo`.

Аннотированные конечные точки

В следующем примере показано, как создать ту же конечную точку, что и в Программных конечных точках, используя вместо этого аннотации:

JAVA

```
@ServerEndpoint("/echo")
public class EchoEndpoint {
   @OnMessage
   public void onMessage(Session session, String msg) {
      try {
         session.getBasicRemote().sendText(msg);
      } catch (IOException e) { ... }
   }
}
```

Аннотированная конечная точка проще, чем эквивалентная программная, и она автоматически развёртывается вместе с приложением по относительному пути, определённому в аннотации `@ServerEndpoint`. Вместо того, чтобы создавать дополнительный класс для обработчика сообщений, в этом примере используется аннотация `@OnMessage` для обозначения метода, вызываемого для обработки сообщений.

Таблица 19-1 перечисляет аннотации, доступные в пакете `jakarta.websocket`, чтобы назначить методы, которые обрабатывают события жизненного цикла. Примеры в таблице показывают наиболее распространённые параметры для этих методов. См. ссылку API для получения подробной информации о том, какие комбинации параметров допускаются в каждом случае.

Таблица 19-1 Аннотации жизненного цикла конечной точки веб-сокета

Аннотация	Событие	Пример

`OnOpen`	Соединение открыто	JAVA
		`@OnOpen`
		`public void open(Session session,`
		`EndpointConfig conf) { }`

`OnMessage`	Сообщение получено	JAVA
		`@OnMessage`
		`public void message(Session session,`
		`String msg) { }`

`OnError`	Ошибка подключения	JAVA
		`@OnError`
		`public void error(Session session,`
		`Throwable error) { }`

`OnClose`	Соединение закрыто	JAVA
		`@OnClose`
		`public void close(Session session,`
		`CloseReason reason) { }`

Отправка и получение сообщений

Конечные точки веб-сокетов могут отправлять и получать текстовые и бинарные сообщения. Кроме того, они также могут отправлять ping-команды и получать pong-команды. В этом разделе описывается, как использовать интерфейсы `Session` и `RemoteEndpoint` для отправки сообщений на подключённый одноранговый узел и как использовать аннотацию `@OnMessage` для получения сообщений от него.

Отправка сообщений

Выполните следующие действия для отправки сообщений конечной точке.

1. Получите объект `Session` из соединения.

Объект `Session` доступен как параметр в аннотированных методах жизненного цикла конечной точки, как в табл. 19-1. Когда ваше сообщение является ответом на сообщение от партнёра, у вас есть объект `Session`, доступный внутри метода, который получил сообщение (метод, аннотированный `@OnMessage`). Если нужно отправлять сообщения, которые не являются ответами, сохраните объект `Session` как переменную объекта класса конечной точки в методе, аннотированном с помощью `@OnOpen`, чтобы можно было получить к нему доступ из других методов.

2. Используйте объект `Session` для получения объекта `RemoteEndpoint`.

   Метод `Session.getBasicRemote` и метод `Session.getAsyncRemote` возвращают объекты `RemoteEndpoint.Basic` и `RemoteEndpoint.Async` соответственно. Интерфейс `RemoteEndpoint.Basic` предоставляет блокирующие методы для отправки сообщений. Интерфейс `RemoteEndpoint.Async` предоставляет неблокирующие методы.

3. Используйте объект `RemoteEndpoint` для отправки сообщений одноранговому узлу.

   В следующем списке показаны некоторые методы, которые можно использовать для отправки сообщений одноранговому узлу.

   - `void RemoteEndpoint.Basic.sendText(String text)`

     Отправка текстового сообщения узлу. Этот метод блокируется, пока не будет передано всё сообщение.

   - `void RemoteEndpoint.Basic.sendBinary(ByteBuffer data)`

     Отправка бинарного сообщения узлу. Этот метод блокируется, пока не будет передано всё сообщение.

   - `void RemoteEndpoint.sendPing(ByteBuffer appData)`

     Отправка ping-команды узлу.

   - `void RemoteEndpoint.sendPong(ByteBuffer appData)`

     Отправка pong-команды узлу.

Пример в Аннотированные конечные точки демонстрирует, как использовать эту процедуру для ответа на каждое входящее текстовое сообщение.

Отправка сообщений всем узлам, подключённым к конечной точке

Каждый объект класса конечной точки связан только с одним соединением и одним узлом. Однако в некоторых случаях объекту конечной точки необходимо отправлять сообщения всем подключённым узлам. Примерами могут служить приложения чатов и онлайн-аукционов. Интерфейс `Session` предоставляет метод `getOpenSessions` для этой цели. В следующем примере показано, как использовать этот метод для пересылки входящих текстовых сообщений всем подключённым узлам:

JAVA

```
@ServerEndpoint("/echoall")
public class EchoAllEndpoint {
   @OnMessage
   public void onMessage(Session session, String msg) {
      try {
         for (Session sess : session.getOpenSessions()) {
            if (sess.isOpen())
               sess.getBasicRemote().sendText(msg);
         }
      } catch (IOException e) { ... }
   }
}
```

Получение сообщений

Методы, аннотированные `OnMessage`, обрабатывают входящие сообщения. Допустимо иметь не более трёх методов, аннотированных `@OnMessage`, в конечной точке, по одному для каждого типа сообщения: текстового, бинарного и pong-команды. В следующем примере показано, как назначить методы для получения всех трёх типов сообщений:

JAVA

```
@ServerEndpoint("/receive")
public class ReceiveEndpoint {
   @OnMessage
   public void textMessage(Session session, String msg) {
      System.out.println("Text message: " + msg);
   }
   @OnMessage
   public void binaryMessage(Session session, ByteBuffer msg) {
      System.out.println("Binary message: " + msg.toString());
   }
   @OnMessage
   public void pongMessage(Session session, PongMessage msg) {
      System.out.println("Pong message: " +
                         msg.getApplicationData().toString());
   }
}
```

Поддержка состояния клиента

Поскольку контейнер создаёт объект класса конечной точки для каждого соединения, вы можете определить и использовать переменные объекта для хранения информации о состоянии клиента. Кроме того, метод `Session.getUserProperties` предоставляет изменяемое отображение (Map) для хранения кастомных свойств. Например, следующая конечная точка отвечает на входящие текстовые сообщения с содержимым предыдущего сообщения от каждого клиента:

JAVA

```
@ServerEndpoint("/delayedecho")
public class DelayedEchoEndpoint {
   @OnOpen
   public void open(Session session) {
      session.getUserProperties().put("previousMsg", " ");
   }
   @OnMessage
   public void message(Session session, String msg) {
      String prev = (String) session.getUserProperties()
                                    .get("previousMsg");
      session.getUserProperties().put("previousMsg", msg);
      try {
         session.getBasicRemote().sendText(prev);
      } catch (IOException e) { ... }
   }
}
```

Для хранения информации, общей для всех подключённых клиентов, могут использоваться статические переменные. Однако в этом случае вы несёте ответственность за потокобезопасность (thread-safe) доступа.

Использование кодировщиков и декодировщиков

Jakarta WebSocket обеспечивает поддержку преобразования между сообщениями веб-сокетов и пользовательскими типами Java с помощью кодировщиков и декодировщиков. Кодировщик берёт объект Java и создаёт представление, которое может быть передано в виде сообщения веб-сокета. Например, кодировщики обычно создают JSON, XML или бинарные представления. Декодировщик выполняет обратную функцию. Он читает сообщение веб-сокета и создаёт объект Java.

Этот механизм упрощает приложения веб-сокетов, поскольку он отделяет бизнес-логику от сериализации и десериализации объектов.

Реализация кодировщика для преобразования объектов Java в сообщения веб-сокетов

Процедура реализации и использования кодировщиков в конечных точках следующая.

1. Реализуйте один из следующих интерфейсов:
   - `Encoder.Text<T>` для текстовых сообщений
   - `Encoder.Binary<T>` для бинарных сообщений

   Эти интерфейсы определяют метод `encode`. Реализуйте класс кодировщика для каждого пользовательского типа Java, который вы хотите отправить в виде сообщения веб-сокета.
2. Добавьте имена ваших реализаций кодировщика в необязательный параметр `encoders` аннотации `@ServerEndpoint`.
3. Используйте `sendObject(Object data)` метод интерфейсов `RemoteEndpoint.Basic` или `RemoteEndpoint.Async` для отправки ваших объектов в виде сообщений. Контейнер ищет кодировщик, соответствующий типу, и использует его для преобразования объекта в сообщение веб-сокета.

Например, если есть два типа Java (`MessageA` и `MessageB`), которые требуется отправить в виде текстовых сообщений, можно реализовать `Encoder.Text<MessageA>` и `Encoder.Text<MessageB>`, взаимодействующие следующим образом:

JAVA

```
public class MessageATextEncoder implements Encoder.Text<MessageA> {
   @Override
   public void init(EndpointConfig ec) { }
   @Override
   public void destroy() { }
   @Override
   public String encode(MessageA msgA) throws EncodeException {
      // Доступ к свойствам msgA и конвертация их в JSON...
      return msgAJsonString;
   }
}
```

`Encoder.Text<MessageB>` реализуется аналогичным образом. Затем следует добавить параметр `encoders` в аннотацию `@ServerEndpoint` таким образом:

JAVA

```
@ServerEndpoint(
   value = "/myendpoint",
   encoders = { MessageATextEncoder.class, MessageBTextEncoder.class }
)
public class EncEndpoint { ... }
```

Теперь вы можете отправлять объекты `MessageA` и `MessageB` как сообщения веб-сокета, используя метод `sendObject` следующим образом:

JAVA

```
MessageA msgA = new MessageA(...);
MessageB msgB = new MessageB(...);
session.getBasicRemote.sendObject(msgA);
session.getBasicRemote.sendObject(msgB);
```

Как и в этом примере, вы можете иметь более одного кодировщика для текстовых сообщений и более одного кодировщика для бинарных сообщений. Как и конечные точки, объекты кодировщика связаны с одним и только одним подключением веб-сокета, поэтому в любой момент времени только один поток выполняет код объекта кодировщика.

Реализация декодировщика для преобразования сообщений веб-сокетов в объекты Java

Процедура для реализации и использования декодировщиков в конечных точках следующая.

1. Реализуйте один из следующих интерфейсов:
   - `Decoder.Text<T>` для текстовых сообщений
   - `Decoder.Binary<T>` для бинарных сообщений

   Эти интерфейсы определяют методы `willDecode` и `decode`.

   

   В отличие от кодировщиков, можно указать не более одного декодировщика для бинарных сообщений и один декодировщик для текстовых сообщений.

2. Добавьте имена ваших реализаций декодировщика в необязательный параметр `decoders` аннотации `@ServerEndpoint`.
3. Используйте аннотацию `@OnMessage` в конечной точке, чтобы назначить метод, который принимает ваш пользовательский тип Java в качестве параметра. Когда конечная точка получает сообщение, которое может быть декодировано одним из указанных декодировщиков, контейнер вызывает метод, помеченный `@OnMessage`, который принимает пользовательский тип Java в качестве параметра, если этот метод существует.

Например, если у вас есть два типа Java (`MessageA` и `MessageB`), которые вы хотите отправлять и получать как текстовые сообщения, определите их так, чтобы они расширяли общий класс (`Message`). Поскольку вы можете определить только один декодировщик для текстовых сообщений, реализуйте декодировщик для класса `Message` следующим образом:

JAVA

```
public class MessageTextDecoder implements Decoder.Text<Message> {
   @Override
   public void init(EndpointConfig ec) { }
   @Override
   public void destroy() { }
   @Override
   public Message decode(String string) throws DecodeException {
      // Чтение сообщения...
      if ( /* сообщение A */ )
         return new MessageA(...);
      else if ( /* сообщение B */ )
         return new MessageB(...);
   }
   @Override
   public boolean willDecode(String string) {
      // Определение, может ли сообщение быть конвертировано к
      // MessageA или MessageB...
      return canDecode;
   }
}
```

Затем добавьте параметр `decoder` в аннотацию `@ServerEndpoint` следующим образом:

JAVA

```
@ServerEndpoint(
   value = "/myendpoint",
   encoders = { MessageATextEncoder.class, MessageBTextEncoder.class },
   decoders = { MessageTextDecoder.class }
)
public class EncDecEndpoint { ... }
```

Теперь определите метод в классе конечных точек, который получает объекты `MessageA` и `MessageB` следующим образом:

JAVA

```
@OnMessage
public void message(Session session, Message msg) {
   if (msg instanceof MessageA) {
      // Получено сообщение MessageA...
   } else if (msg instanceof MessageB) {
      // Получено сообщение MessageB...
   }
}
```

Как и конечные точки, объекты декодировщика связаны с одним и только одним подключением к веб-сокету, поэтому в любой момент времени только один поток выполняет код объекта декодировщика.

Параметры пути

Аннотация `@ServerEndpoint` позволяет использовать шаблоны URI для указания частей URI развёртывания конечной точки в качестве параметров приложения. Например, рассмотрим эту конечную точку:

JAVA

```
@ServerEndpoint("/chatrooms/{room-name}")
public class ChatEndpoint {
   ...
}
```

Если конечная точка развёрнута внутри веб-приложения с именем `chatapp` на локальном сервере Jakarta EE на порту 8080, клиенты могут подключаться к конечной точке с помощью любого из следующих URI:

```
http://localhost:8080/chatapp/chatrooms/currentnews
http://localhost:8080/chatapp/chatrooms/music
http://localhost:8080/chatapp/chatrooms/cars
http://localhost:8080/chatapp/chatrooms/technology
```

Аннотированные конечные точки могут получать параметры пути в качестве аргументов в методах, аннотированных с помощью `@OnOpen`, `@OnMessage` и `@OnClose`. В этом примере конечная точка использует параметр в методе `@OnOpen`, чтобы определить, к какой комнате чата клиент хочет присоединиться:

JAVA

```
@ServerEndpoint("/chatrooms/{room-name}")
public class ChatEndpoint {
   @OnOpen
   public void open(Session session,
                    EndpointConfig c,
                    @PathParam("room-name") String roomName) {
      // Добавление клиента в комнату чата...
   }
}
```

Параметры пути, используемые в качестве аргументов в этих методах, могут быть строками, примитивными типами или соответствующими им типами-обёртками (wrapper).

Обработка ошибок

Чтобы назначить метод, который обрабатывает ошибки в аннотированной конечной точке веб-сокета, пометьте его с помощью `@OnError`:

JAVA

```
@ServerEndpoint("/testendpoint")
public class TestEndpoint {
   ...
   @OnError
   public void error(Session session, Throwable t) {
      t.printStackTrace();
      ...
   }
}
```

Этот метод вызывается при возникновении проблем с подключением, ошибок времени выполнения из обработчиков сообщений или ошибок преобразования при декодировании сообщений.

Указание класса конфигуратора конечной точки

Jakarta WebSocket позволяет настроить процесс создания объектов конечных точек контейнером. Вы можете предоставить кастомную логику конфигурации конечной точки для:

- доступа к деталям начального HTTP-запроса для соединения с веб-сокетом;
- выполнения кастомной проверки заголовка HTTP `Origin`;
- изменения ответ веб-сокета при установлении соединения;
- выбора подпротокола веб-сокетов из тех, которые запрашивает клиент;
- управления созданием и инициализацией объектов конечных точек.

Чтобы обеспечить кастомную логику конфигурации конечной точки, расширьте класс `ServerEndpointConfig.Configurator` и переопределите некоторые из его методов. В классе конечной точки укажите класс конфигуратора с помощью параметра `configurator` аннотации `@ServerEndpoint`.

Например, следующий класс конфигуратора делает объект запроса установления соединения доступным для объектов конечной точки:

JAVA

```
public class CustomConfigurator extends ServerEndpointConfig.Configurator {

    @Override
    public void modifyHandshake(ServerEndpointConfig conf,
                                HandshakeRequest req,
                                HandshakeResponse resp) {

        conf.getUserProperties().put("handshakereq", req);
    }

}
```

Следующий класс конечной точки настраивает объекты конечных точек с помощью кастомного конфигуратора, который позволяет им получать доступ к объекту запроса установления соединения:

JAVA

```java
@ServerEndpoint(
    value = "/myendpoint",
    configurator = CustomConfigurator.class
)
public class MyEndpoint {

    @OnOpen
    public void open(Session s, EndpointConfig conf) {
        HandshakeRequest req = (HandshakeRequest) conf.getUserProperties()
                                                      .get("handshakereq");
        Map<String,List<String>> headers = req.getHeaders();
        ...
    }
}
```

Класс конечной точки может использовать объект запроса установления соединения для доступа к деталям исходного HTTP-запроса, таким как его заголовки или объект `HttpSession`.

Для получения дополнительной информации о конфигурации конечной точки см. ссылку на API для класса `ServerEndpointConfig.Configurator`.

Приложение dukeetf2

Пример приложения `dukeetf2`, расположенный в каталоге *`tut-install`*`/examples/web/websocket/dukeetf2/`, демонстрирует, как использовать конечную точку веб-сокета для предоставления обновлений данных веб-клиентам. Пример напоминает сервис, который предоставляет периодические обновления цены и объёма торгов биржевым инвестиционным фондом (ETF).

Архитектура приложения dukeetf2

Приложение `dukeetf2` состоит из конечной точки веб-сокета, Enterprise-бина и страницы HTML.

- Конечная точка принимает подключения от клиентов и отправляет им обновления, когда становятся доступными новые данные о цене и объёме торгов.
- Enterprise-бин обновляет информацию о цене и объёме каждую секунду.
- HTML-страница использует код JavaScript для подключения к конечной точке веб-сокета, анализа входящих сообщений и обновления информации о цене и объёме без перезагрузки страницы.

Конечная точка

Конечная точка веб-сокета реализована в классе `ETFEndpoint`, который хранит все подключённые сессии в очереди и предоставляет метод, который вызывает Enterprise-бин, когда появляется новая информация, доступная для отправки:

JAVA

```
@ServerEndpoint("/dukeetf")
public class ETFEndpoint {
   private static final Logger logger = Logger.getLogger("ETFEndpoint");
   /* Очередь для всех открытых сессий веб-сокета */
   static Queue<Session> queue = new ConcurrentLinkedQueue<>();

   /* PriceVolumeBean вызывает метода для отправки обновления */
   public static void send(double price, int volume) {
      String msg = String.format("%.2f / %d", price, volume);
      try {
         /* Отправка обновления всем открытым сессиям веб-сокета */
         for (Session session : queue) {
            session.getBasicRemote().sendText(msg);
            logger.log(Level.INFO, "Sent: {0}", msg);
         }
      } catch (IOException e) {
         logger.log(Level.INFO, e.toString());
      }
    }
    ...
}
```

Методы жизненного цикла конечной точки добавляют и удаляют сессии в очередь и из неё:

JAVA

```
@ServerEndpoint("/dukeetf")
public class ETFEndpoint {
   ...
   @OnOpen
   public void openConnection(Session session) {
      /* Регистрация соединения в очереди */
      queue.add(session);
      logger.log(Level.INFO, "Connection opened.");
   }

   @OnClose
   public void closedConnection(Session session) {
      /* Удаление соединения из очереди */
      queue.remove(session);
      logger.log(Level.INFO, "Connection closed.");
   }

   @OnError
   public void error(Session session, Throwable t) {
      /* Удаление соединения из очереди */
      queue.remove(session);
      logger.log(Level.INFO, t.toString());
      logger.log(Level.INFO, "Connection error.");
   }
}
```

Enterprise-бин

Enterprise-бин использует сервис таймера для генерации новой информации о цене и объёме каждую секунду:

JAVA

```java
@Startup
@Singleton
public class PriceVolumeBean {
   /* Использование сервиса таймера контейнера */
   @Resource TimerService tservice;
   private Random random;
   private volatile double price = 100.0;
   private volatile int volume = 300000;
   private static final Logger logger = Logger.getLogger("PriceVolumeBean");

   @PostConstruct
   public void init() {
       /* Инициализация EJB и создание таймера */
       logger.log(Level.INFO, "Initializing EJB.");
       random = new Random();
       tservice.createIntervalTimer(1000, 1000, new TimerConfig());
   }

   @Timeout
   public void timeout() {
       /* Отправка price и volume в обновлении */
       price += 1.0*(random.nextInt(100)-50)/100.0;
       volume += random.nextInt(5000) - 2500;
       ETFEndpoint.send(price, volume);
   }
}
```

Enterprise-бин вызывает метод `send` класса `ETFEndpoint` в методе timeout. См. Использование сервиса таймера в xref: Выполнение примеров Enterprise-бинов для получения дополнительной информации о сервисе таймера.

HTML-страница

HTML-страница состоит из таблицы и кода JavaScript. Таблица содержит два поля, на которые ссылается код JavaScript:

HTML

```html
<!DOCTYPE html>
<html>
<head>...</head>
<body>
  ...
  <table>
    ...
    <td id="price">--.--</td>
    ...
    <td id="volume">--</td>
    ...
  </table>
</body>
</html>
```

Код JavaScript использует API веб-сокетов для подключения к конечной точке сервера и определения Callback-метода для входящих сообщений. Callback-метод обновляет страницу новой информацией.

JAVASCRIPT

```
var wsocket;
function connect() {
   wsocket = new WebSocket("ws://localhost:8080/dukeetf2/dukeetf");
   wsocket.onmessage = onMessage;
}
function onMessage(evt) {
   var arraypv = evt.data.split("/");
   document.getElementById("price").innerHTML = arraypv[0];
   document.getElementById("volume").innerHTML = arraypv[1];
}
window.addEventListener("load", connect, false);
```

API веб-сокетов поддерживается большинством современных браузеров и широко используется при разработке веб-клиентов на HTML5.

Запуск приложения dukeetf2

В этом разделе описывается, как запустить пример `dukeetf2` в IDE NetBeans и из командной строки.

Запуск приложения dukeetf2 с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/web/websocket
   ```

4. Выберите каталог `dukeetf2`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `dukeetf2` и выберите **Запуск**.

   Эта команда собирает и упаковывает приложение в WAR-файл (`dukeetf2.war`), расположенный в каталоге `target/`, развёртывает его на сервере и запускает окно веб-браузера со следующим URL:

   ```
   http://localhost:8080/dukeetf2/
   ```

   Откройте один и тот же URL на другой вкладке или в окне веб-браузера, чтобы увидеть, как обе страницы одновременно получают обновления цены и объёма.

Запуск приложения dukeetf2 с помощью Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/web/websocket/dukeetf2/
   ```

3. Введите следующую команду для развёртывания приложения:

   SHELL

   ```
   mvn install
   ```

4. Откройте окно веб-браузера и введите следующий URL:

   ```
   http://localhost:8080/dukeetf2/
   ```

Откройте один и тот же URL на другой вкладке или в окне веб-браузера, чтобы увидеть, как обе страницы одновременно получают обновления цены и объёма.

Приложение websocketbot

Пример приложения `websocketbot`, расположенный в каталоге *`tut-install`*`/examples/web/websocket/websocketbot/`, демонстрирует, как использовать конечную точку веб-сокета для реализации чата. Пример напоминает чат-комнату, к которой пользователи могут присоединиться и пообщаться. Пользователи могут задавать простые вопросы бот-агенту, который всегда доступен в чате.

Архитектура приложения websocketbot

Приложение `websocketbot` состоит из следующих элементов:

- Компонент CDI — Компонент CDI (`BotBean`), который содержит логику ответов на сообщения для бота
- Конечная точка веб-сокета — Конечная точка веб-сокета (`BotEndpoint`), которая реализует комнату чата
- Сообщения приложения — набор классов (`Message`, `ChatMessage`, `InfoMessage`, `JoinMessage` и `UsersMessage`), которые представляют сообщения приложения
- Классы кодировщика — набор классов (`ChatMessageEncoder`, `InfoMessageEncoder`, `JoinMessageEncoder` и `UsersMessageEncoder`), которые кодируют сообщения приложения в текстовые сообщения вебсокета как данные JSON
- Декодировщик сообщений — класс (`MessageDecoder`) анализирует текстовые сообщения веб-сокета как данные JSON и декодирует их в `JoinMessage` или `ChatMessage` объекты
- HTML-страница — HTML-страница (`index.html`), которая использует код JavaScript для реализации клиента для чата

Компонент CDI

Компонент CDI (`BotBean`) — это класс Java, который содержит метод `response`. Этот метод сравнивает входящее сообщение чата с набором предварительно заданных вопросов и возвращает ответ в чате.

JAVA

```
@Named
public class BotBean {
    public String respond(String msg) { ... }
}
```

Конечная точка веб-сокета

Конечная точка веб-сокета (`BotEndpoint`) является аннотированной конечной точкой, которая выполняет следующие функции:

- Получает сообщения от клиентов
- Пересылает сообщения клиентам
- Поддерживает список подключённых клиентов
- Вызывает функциональность бот-агента

Конечная точка указывает свой URI развёртывания, а также кодировщики и декодировщики сообщений, используя аннотацию `@ServerEndpoint`. Конечная точка получает объект класса `BotBean` и ресурс управляемого ExecutorService используя инъецирование зависимостей:

JAVA

```
@ServerEndpoint(
   value = "/websocketbot",
   decoders = { MessageDecoder.class },
   encoders = { JoinMessageEncoder.class, ChatMessageEncoder.class,
                InfoMessageEncoder.class, UsersMessageEncoder.class }
)
/* Один объект BotEndpoint на соединение */
public class BotEndpoint {
   private static final Logger logger = Logger.getLogger("BotEndpoint");
   /* функциональность Bot */
   @Inject private BotBean botbean;
   /* Executor service для асинхронной обработки */
   @Resource(name="comp/DefaultManagedExecutorService")
   private ManagedExecutorService mes;

   @OnOpen
   public void openConnection(Session session) {
       logger.log(Level.INFO, "Connection opened.");
   }
   ...
}
```

Метод `message` обрабатывает входящие сообщения от клиентов. Декодировщик преобразует входящие текстовые сообщения в объекты `JoinMessage` или `ChatMessage`, которые наследуются от класса `Message`. Метод `message` получает объект `Message` в качестве параметра:

JAVA

```
@OnMessage
public void message(Session session, Message msg) {
   logger.log(Level.INFO, "Received: {0}", msg.toString());

   if (msg instanceof JoinMessage) {
      /* Добавление нового пользователя для уведомления всех */
      JoinMessage jmsg = (JoinMessage) msg;
      session.getUserProperties().put("name", jmsg.getName());
      session.getUserProperties().put("active", true);
      logger.log(Level.INFO, "Received: {0}", jmsg.toString());
      sendAll(session, new InfoMessage(jmsg.getName() +
              " has joined the chat"));
      sendAll(session, new ChatMessage("Duke", jmsg.getName(),
              "Hi there!!"));
      sendAll(session, new UsersMessage(this.getUserList(session)));

   } else if (msg instanceof ChatMessage) {
      /* Пересылка сообщения всем */
      ChatMessage cmsg = (ChatMessage) msg;
      logger.log(Level.INFO, "Received: {0}", cmsg.toString());
      sendAll(session, cmsg);
      if (cmsg.getTarget().compareTo("Duke") == 0) {
         /* ответ бота на сообщение */
         mes.submit(new Runnable() {
            @Override
            public void run() {
               String resp = botbean.respond(cmsg.getMessage());
               sendAll(session, new ChatMessage("Duke",
                       cmsg.getName(), resp));
            }
         });
      }
   }
}
```

Если сообщение является сообщением о присоединении, конечная точка добавляет нового пользователя в список и уведомляет всех подключённых клиентов. Если сообщение является сообщением чата, конечная точка пересылает его всем подключённым клиентам.

Если сообщение чата предназначено для агента бота, конечная точка получает ответ, используя объект `BotBean`, и отправляет его всем подключённым клиентам. Метод `sendAll` аналогичен примеру в Отправка сообщений всем узлам, подключённым к конечной точке.

Асинхронная обработка и параллелизм

Конечная точка веб-сокета вызывает метод `BotBean.respond` для получения ответа от бота. В этом примере это операция блокировки. Пользователь, который отправил соответствующее сообщение, не сможет отправлять или получать другие сообщения чата, пока операция не завершится. Чтобы избежать этой проблемы, конечная точка получает ExecutorService из контейнера и выполняет операцию блокировки в другом потоке, используя метод `ManagedExecutorService.submit` утилит параллелизма в Jakarta EE.

В соответствии со спецификацией Jakarta WebSocket необходимо, чтобы реализации Jakarta EE создавали новый объект конечной точки для каждого соединения. Это облегчает разработку конечных точек веб-сокетов, потому что гарантированно только один поток выполняет код в классе конечных точек веб-сокетов в любой момент времени. Когда вы вводите новый поток в конечную точку, как в этом примере, нужно убедиться, что переменные и методы, к которым обращается более одного потока, являются потокобезопасными. В этом примере код в `BotBean` является потокобезопасным, а метод `BotEndpoint.sendAll` объявлен как `synchronized`.

Обратитесь к главе 60 *Jakarta Concurrency* для получения дополнительной информации о managed executor service и Concurrency Utilities for Jakarta EE.

Сообщения приложения

Классы, представляющие сообщения приложения (`Message`, `ChatMessage`, `InfoMessage`, `JoinMessage` и `UsersMessage`) содержат только свойства и get- и set-методы. Например, класс `ChatMessage` выглядит следующим образом:

```java
public class ChatMessage extends Message {
    private String name;
    private String target;
    private String message;
    /* ... Конструктор, get- и set-методы... */
}
```

Классы кодировщика

Классы кодировщика преобразуют объекты сообщений приложения в текст JSON, используя Java API для обработки JSON. Например, класс `ChatMessageEncoder` реализован следующим образом:

JAVA

```java
/* Кодирование ChatMessage в JSON.
 * Например, (new ChatMessage("Peter","Duke","How are you?"))
 * будет закодировано в:
 * {"type":"chat","target":"Duke","message":"How are you?"}
 */
public class ChatMessageEncoder implements Encoder.Text<ChatMessage> {
   @Override
   public void init(EndpointConfig ec) { }
   @Override
   public void destroy() { }
   @Override
   public String encode(ChatMessage chatMessage) throws EncodeException {
      // Чтение свойств chatMessage и запись их в JSON...
   }
}
```

См. главу 20 *Обработка JSON* для получения дополнительной информации о Jakarta JSON Processing.

Декодировщик сообщений

Класс декодировщика сообщений (`MessageDecoder`) преобразует текстовые сообщения веб-сокета в сообщения приложения путём парсинга JSON. Это реализовано следующим образом:

JAVA

```java
/* Декодирование сообщения JSON в JoinMessage или ChatMessage.
 * Например, входящее сообщение
 * {"type":"chat","name":"Peter","target":"Duke","message":"How are you?"}
 * будет декодировано как (new ChatMessage("Peter", "Duke", "How are you?"))
 */
public class MessageDecoder implements Decoder.Text<Message> {
    /* Сохраняет пары имя-значение из JSON в отображение (Map) */
    private Map<String,String> messageMap;

    @Override
    public void init(EndpointConfig ec) { }
    @Override
    public void destroy() { }

    /* Создаёт новый объект Message object если сообщение может быть декодировано */
    @Override
    public Message decode(String string) throws DecodeException {
       Message msg = null;
       if (willDecode(string)) {
          switch (messageMap.get("type")) {
             case "join":
                msg = new JoinMessage(messageMap.get("name"));
                break;
             case "chat":
                msg = new ChatMessage(messageMap.get("name"),
                                      messageMap.get("target"),
                                      messageMap.get("message"));
          }
       } else {
          throw new DecodeException(string, "[Message] Can't decode.");
       }
       return msg;
    }

    /* Декодирование сообщения JSON в отображение (Map) и проверка его содержимого
     * на соответствие типов полей. */
    @Override
    public boolean willDecode(String string) {
       // Преобразование данных из JSON в отображение ключ-значение...
       // Проверка, все ли неободимые поля для соответствующего типа сообщения присутствуют...
    }
}
```

HTML-страница

Страница HTML (`index.html`) содержит поле для имени пользователя. После того, как пользователь вводит имя и кликает кнопку «Присоединиться», доступны три текстовые области: одна для ввода и отправки сообщений, одна для чата и одна со списком пользователей. Страница также содержит консоль веб-сокета, которая показывает сообщения, отправленные и полученные в виде текста JSON.

Код JavaScript на странице использует API веб-сокетов для подключения к конечной точке, отправки сообщений и назначения Callback-методов. API веб-сокетов поддерживается большинством современных браузеров и широко используется для разработки веб-клиентов на HTML5.

Запуск приложения websocketbot

В этом разделе описывается, как запустить пример `websocketbot` в IDE NetBeans и из командной строки.

Запуск приложения websocketbot с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.

3. В диалоговом окне «Открыть проект» перейдите к:

```
tut-install/examples/web/websocket
```

4. Выберите каталог `websocketbot` .
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `websocketbot` и выберите **Запуск**.

   Эта команда собирает и упаковывает приложение в WAR-файл, `websocketbot.war` , расположенный в каталоге `target/` , развёртывает его на сервере и запускает окно веб-браузера со следующим URL:

```
http://localhost:8080/websocketbot/
```

   См. Тестирование приложения websocketbot для получения дополнительной информации.

Запуск приложения websocketbot с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

```
tut-install/examples/web/websocket/websocketbot/
```

3. Введите следующую команду для развёртывания приложения:

```
mvn install
```
SHELL

4. Откройте окно веб-браузера и введите следующий URL:

```
http://localhost:8080/websocketbot/
```

   См. Тестирование приложения websocketbot для получения дополнительной информации.

Тестирование приложения websocketbot

1. На главной странице введите своё имя в первом текстовом поле и нажмите клавишу Enter.

   Список подключённых пользователей отображается в текстовой области справа. Текстовая область слева — комната чата.

2. Введите сообщение в текстовой области под кнопкой входа в систему. Например, введите сообщения, как показано ниже, и нажмите клавишу Enter, чтобы получить ответы, подобные следующим:

```
[--Peter has joined the chat--]
Duke: @Peter Hi there!!
Peter: @Duke how are you?
Duke: @Peter I'm doing great, thank you!
Peter: @Duke when is your birthday?
Duke: @Peter My birthday is on May 23rd. Thanks for asking!
```

3. Присоединитесь к чату из другого окна браузера, скопировав и вставив URI в адресную строку и присоединившись под другим именем.

   Новое имя пользователя появится в списке пользователей в обоих окнах браузера. Вы можете отправлять сообщения из любого окна и видеть, как они появляются в другом.

4. Нажмите Показать консоль веб-сокета.

   Консоль показывает сообщения, отправленные и полученные в виде текста JSON.

Дополнительная информация о веб-сокетах

Дополнительная информация о веб-сокетах в Jakarta EE приведена в спецификации Jakarta WebSocket:

https://jakarta.ee/specifications/websocket/2.0/

Глава 20. Обработка JSON

В этой главе описывается обработка JSON в Jakarta. JSON — это формат обмена данными, широко используемый в веб-сервисах и других связанных приложениях. Jakarta JSON Processing предоставляет API для анализа, преобразования и выборких данных JSON с помощью объектной или потоковой модели.

Введение в JSON

JSON — это текстовый формат обмена данными, пришедший из JavaScript, который используется в веб-сервисах и других связанных приложениях. В следующих разделах представлены введение в синтаксис, обзор использования и описание наиболее распространённых подходов для генерации и парсинга JSON.

Синтаксис JSON

JSON определяет только две структуры данных: объекты и массивы. Объект — это набор пар имя-значение, а массив — это список значений. JSON определяет семь типов значений: строка, число, объект, массив, true, false и null.

В следующем примере показаны данные JSON для примера объекта, который содержит пары имя-значение. Значение для имени `"phoneNumbers"` является массивом, элементами которого являются два объекта.

JSON

```
{
   "firstName": "Duke",
   "lastName": "Java",
   "age": 18,
   "streetAddress": "100 Internet Dr",
   "city": "JavaTown",
   "state": "JA",
   "postalCode": "12345",
   "phoneNumbers": [
      { "Mobile": "111-111-1111" },
      { "Home": "222-222-2222" }
   ]
}
```

JSON имеет следующий синтаксис.

- Объекты заключены в фигурные скобки (`{}`), их пары имя-значение разделены запятой (`,`), а имя и значение в паре разделены двоеточием (`:`). Имена в объекте являются строками, тогда как значения могут быть любого из семи типов значений, включая другой объект или массив.
- Массивы заключены в квадратные скобки (`[]`), а их значения разделены запятой (`,`). Каждое значение в массиве может быть другого типа, включая другой массив или объект.
- Когда объекты и массивы содержат другие объекты или массивы, данные имеют древовидную структуру.

Использование JSON

JSON часто используется в качестве общего формата для сериализации и десериализации данных в приложениях, которые взаимодействуют друг с другом через Интернет. Эти приложения создаются с использованием разных языков программирования и работают в самых разных средах. JSON подходит для этого сценария, потому что это открытый стандарт, его легко читать и писать, и он более компактен, чем другие представления.

RESTful веб-сервисы широко используют JSON в качестве формата данных внутри запросов и ответов. Заголовок HTTP, используемый для указания того, что содержимое запроса или ответа представляет собой данные JSON:

```HTTP
Content-Type: application/json
```

Представления JSON обычно более компактны, чем представления XML, потому что JSON не имеет закрывающих тегов. В отличие от XML, JSON не имеет общепринятой схемы для определения и проверки структуры данных JSON.

Генерация и парсинг JSON

Для генерации и парсинга данных JSON существуют две программные модели, которые аналогичны тем, что используются для документов XML.

- Объектная модель создаёт дерево, которое представляет данные JSON в памяти. По дереву можно затем перемещаться, его можно анализировать или изменять. Этот подход является наиболее гибким и допускает обработку, которая требует доступа ко всему содержимому дерева. Однако это часто медленнее, чем потоковая модель, и требует больше памяти. Объектная модель генерирует вывод JSON путём одновременного перемещения по всему дереву.
- Модель потоковой передачи использует анализатор на основе событий, который читает данные JSON по одному элементу за раз. Парсер генерирует события и останавливается для обработки, когда объект или массив начинается или заканчивается, когда он находит ключ или значение. Каждый элемент может быть обработан или отброшен кодом приложения, а затем анализатор переходит к следующему событию. Этот подход удобен для локальной обработки, при которой обработка текущего элемента не требует информации об остальных данных. Потоковая модель генерирует вывод JSON для данного потока, вызывая функцию с одним элементом за раз.

Существует множество генераторов и анализаторов JSON для разных языков программирования и сред. Обработка JSON в Jakarta EE описывает функциональные возможности, предоставляемые Jakarta JSON Processing.

Обработка JSON в платформе Jakarta EE

Jakarta EE включает поддержку спецификации Jakarta JSON Processing, которая предоставляет API для анализа, преобразования и выборки данных JSON с использованием объектной или потоковой модели, описанной в Создание и анализ данных JSON. Jakarta JSON Processing содержит следующие пакеты:

- Пакет `jakarta.json` содержит интерфейсы чтения, записи, конструктора моделей для объектной модели, а также служебные классы и типы Java для элементов JSON. Этот пакет также включает несколько классов, которые реализуют другие стандарты, связанные с JSON: JSON Pointer (https://tools.ietf.org/html/rfc6901), JSON Patch (https://tools.ietf.org/html/rfc6902) и JSON Merge Patch (https://tools.ietf.org/html/rfc7396). Эти стандарты используются для извлечения, преобразования или манипулирования значениями в объектной модели. Таблица 20-1 перечисляет основные классы и интерфейсы в этом пакете.
- Пакет `jakarta.json.stream` содержит интерфейсы синтаксического анализатора и генератора для потоковой модели. Таблица 20-2 перечисляет основные классы и интерфейсы в этом пакете.
- Пакет `jakarta.json.spi` содержит интерфейс поставщика услуг (SPI) для подключения реализаций для объектов обработки JSON. Этот пакет включает класс `JsonProvider` , который содержит методы, реализующие поставщик услуг.

Таблица 20-1 Основные классы и интерфейсы в jakarta.json

Класс или интерфейс	Описание
`Json`	Содержит статические методы для создания объектов парсеров, конструкторов (builder) и генераторов JSON. Этот класс также содержит методы для создания объектов парсера, сборщика и генератора.
`JsonReader`	Считывает данные JSON из потока и создаёт объектную модель в памяти.
`JsonObjectBuilder`, `JsonArrayBuilder`	Создаёт объектную модель или модель массива в памяти, добавив элементы из кода приложения.
`JsonWriter`	Записывает объектную модель из памяти в поток.
`JsonValue`	Представляет элемент (например, объект, массив или значение) данных в JSON.
`JsonStructure`	Представляет объект или массив данных в JSON. Этот интерфейс является подтипом `JsonValue`.
`JsonObject`, `JsonArray`	Представляет объект или массив данных в JSON. Эти два интерфейса являются подтипами `JsonStructure`.
`JsonPointer`	Содержит методы для работы с конкретными целями в документах JSON. Цели могут быть объектами `JsonValue`, `JsonObject` или `JsonArray`.
`JsonPatch`	Интерфейс для поддержки последовательности операций, которые будут применены к целевому ресурсу JSON. Операции определены в документе исправления JSON.
`JsonMergePatch`	Интерфейс для поддержки обновлений для целевых ресурсов JSON. Документ исправления JSON сравнивается с целевым ресурсом, чтобы определить конкретный набор операций изменения, которые будут применены.
`JsonString`, `JsonNumber`	Представляет типы данных для элементов данных в JSON. Эти два интерфейса являются подтипами `JsonValue`.
`JsonException`	Указывает, что во время обработки JSON возникла проблема.

Таблица 20-2 Основные классы и интерфейсы в jakarta.json.stream

Класс или интерфейс	Описание
`JsonParser`	Представляет парсер на основе событий, который может читать данные JSON из потока или из объектной модели.
`JsonGenerator`	Записывает данные JSON в поток по одному элементу за раз.

Использование API объектной модели

В этом разделе описываются четыре варианта использования API объектной модели: создание объектной модели из данных JSON, создание объектной модели из кода приложения, навигация по объектной модели и запись объектной модели в поток.

Создание объектной модели из данных JSON

Следующий код демонстрирует, как создать объектную модель из данных JSON в текстовом файле:

```java
import java.io.FileReader;
import jakarta.json.Json;
import jakarta.json.JsonReader;
import jakarta.json.JsonStructure;
...
JsonReader reader = Json.createReader(new FileReader("jsondata.txt"));
JsonStructure jsonst = reader.read();
```

Ссылка на объект `jsonst` может иметь тип `JsonObject` или тип `JsonArray`, в зависимости от содержимого файла. `JsonObject` и `JsonArray` являются подтипами `JsonStructure`. Эта ссылка представляет вершину дерева и может использоваться для навигации по дереву или для записи его в поток в виде данных JSON.

Создание объектной модели в коде приложения

Следующий код демонстрирует, как создать объектную модель из кода приложения:

```java
import jakarta.json.Json;
import jakarta.json.JsonObject;
...
JsonObject model = Json.createObjectBuilder()
   .add("firstName", "Duke")
   .add("lastName", "Java")
   .add("age", 18)
   .add("streetAddress", "100 Internet Dr")
   .add("city", "JavaTown")
   .add("state", "JA")
   .add("postalCode", "12345")
   .add("phoneNumbers", Json.createArrayBuilder()
      .add(Json.createObjectBuilder()
         .add("type", "mobile")
         .add("number", "111-111-1111"))
      .add(Json.createObjectBuilder()
         .add("type", "home")
         .add("number", "222-222-2222")))
   .build();
```

Ссылка на объект `model` представляет вершину дерева, которая создаётся путём вложенных вызовов методов `add` и затем вызова метода `build`. Класс `JsonObjectBuilder` содержит следующие методы `add`:

```java
JsonObjectBuilder add(String name, BigDecimal value)
JsonObjectBuilder add(String name, BigInteger value)
JsonObjectBuilder add(String name, boolean value)
JsonObjectBuilder add(String name, double value)
JsonObjectBuilder add(String name, int value)
JsonObjectBuilder add(String name, JsonArrayBuilder builder)
JsonObjectBuilder add(String name, JsonObjectBuilder builder)
JsonObjectBuilder add(String name, JsonValue value)
JsonObjectBuilder add(String name, long value)
JsonObjectBuilder add(String name, String value)
JsonObjectBuilder addNull(String name)
```

Класс `JsonArrayBuilder` содержит похожие методы `add`, которые не имеют параметра name (key). Вы можете вложить массивы и объекты, передав новый объект `JsonArrayBuilder` или новый объект `JsonObjectBuilder` соответствующему методу `add`, как показано в этом примере.

Результирующее дерево представляет данные JSON из Синтаксиса JSON.

Навигация по объектной модели

Следующий код демонстрирует простой подход к навигации по объектной модели:

```java
import jakarta.json.JsonValue;
import jakarta.json.JsonObject;
import jakarta.json.JsonArray;
import jakarta.json.JsonNumber;
import jakarta.json.JsonString;
...
public static void navigateTree(JsonValue tree, String key) {
   if (key != null)
      System.out.print("Key " + key + ": ");
   switch(tree.getValueType()) {
      case OBJECT:
         System.out.println("OBJECT");
         JsonObject object = (JsonObject) tree;
         for (String name : object.keySet())
            navigateTree(object.get(name), name);
         break;
      case ARRAY:
         System.out.println("ARRAY");
         JsonArray array = (JsonArray) tree;
         for (JsonValue val : array)
            navigateTree(val, null);
         break;
      case STRING:
         JsonString st = (JsonString) tree;
         System.out.println("STRING " + st.getString());
         break;
      case NUMBER:
         JsonNumber num = (JsonNumber) tree;
         System.out.println("NUMBER " + num.toString());
         break;
      case TRUE:
      case FALSE:
      case NULL:
         System.out.println(tree.getValueType().toString());
         break;
   }
}
```

Метод `navigateTree` может использоваться с моделями, построенными в Создание объектной модели из данных JSON и Создание объектной модели в коде приложения следующим образом:

```java
navigateTree(model, null);
```

Метод `navigateTree` принимает два аргумента: элемент JSON и ключ. Ключ используется только для распечатки пар ключ-значение внутри объектов. Элементы в дереве представлены типом `JsonValue`. Если элемент является объектом или массивом, новый вызов этого метода выполняется для каждого элемента, содержащегося в объекте или массиве. Если элемент является значением, он выводится в стандартный вывод.

Метод `JsonValue.getValueType` идентифицирует элемент как объект, массив или значение. Для объектов метод `JsonObject.keySet` возвращает набор строк, содержащий ключи в объекте, а метод `JsonObject.get(String name)` возвращает значение элемент с ключом `name`. Для массивов `JsonArray` реализует интерфейс `List<JsonValue>`. Вы можете использовать расширенные циклы `for` с объектом `Set<String>`, возвращаемым `JsonObject.keySet` и с объектами `JsonArray`, как показано в этом примере.

Метод `navigateTree` для модели, встроенной в Создание объектной модели из кода приложения, производит следующий вывод:

```
OBJECT
   Key firstName: STRING Duke
   Key lastName: STRING Java
   Key age: NUMBER 18
   Key streetAddress: STRING 100 Internet Dr
   Key city: STRING JavaTown
   Key state: STRING JA
   Key postalCode: STRING 12345
   Key phoneNumbers: ARRAY
      OBJECT
         Key type: STRING mobile
         Key number: STRING 111-111-1111
      OBJECT
         Key type: STRING home
         Key number: STRING 222-222-2222
```

Запись объектной модели в поток (Stream)

Объектные модели, созданные в Создание объектной модели из данных JSON и Создание объектной модели из кода приложения, могут быть записаны в поток с использованием класса `JsonWriter` следующим образом:

JAVA

```
import java.io.StringWriter;
import jakarta.json.JsonWriter;
...
StringWriter stWriter = new StringWriter();
JsonWriter jsonWriter = Json.createWriter(stWriter);
jsonWriter.writeObject(model);
jsonWriter.close();

String jsonData = stWriter.toString();
System.out.println(jsonData);
```

Метод `Json.createWriter` принимает выходной поток в качестве параметра. Метод `JsonWriter.writeObject` записывает объект в поток. Метод `JsonWriter.close` закрывает выходной поток.

В следующем примере `try-with-resources` используется для автоматического закрытия записи JSON:

JAVA

```
StringWriter stWriter = new StringWriter();
try (JsonWriter jsonWriter = Json.createWriter(stWriter)) {
   jsonWriter.writeObject(model);
}

String jsonData = stWriter.toString();
System.out.println(jsonData);
```

Использование потокового API

В этом разделе описываются два варианта использования потокового API.

Чтение данных JSON с использованием парсера

Потоковый API является наиболее эффективным подходом для парсинга текста JSON. Следующий код демонстрирует, как создать объект `JsonParser` и как парсить данные JSON с помощью событий:

JAVA

```java
import jakarta.json.Json;
import jakarta.json.stream.JsonParser;
...
JsonParser parser = Json.createParser(new StringReader(jsonData));
while (parser.hasNext()) {
   JsonParser.Event event = parser.next();
   switch(event) {
      case START_ARRAY:
      case END_ARRAY:
      case START_OBJECT:
      case END_OBJECT:
      case VALUE_FALSE:
      case VALUE_NULL:
      case VALUE_TRUE:
         System.out.println(event.toString());
         break;
      case KEY_NAME:
         System.out.print(event.toString() + " " +
                          parser.getString() + " - ");
         break;
      case VALUE_STRING:
      case VALUE_NUMBER:
         System.out.println(event.toString() + " " +
                            parser.getString());
         break;
   }
}
```

Этот пример состоит из трёх шагов.

1. Получите объект парсера, вызвав статический метод `Json.createParser`.
2. Выполните итерацию событий парсинга с помощью методов `JsonParser.hasNext` и `JsonParser.next`.
3. Выполните локальную обработку для каждого элемента.

В примере показаны десять возможных типов событий из парсера. Метод `next` синтаксического анализатора продвигает его до следующего события. Для типов событий `KEY_NAME`, `VALUE_STRING` и `VALUE_NUMBER` вы можете получить содержимое элемента, вызвав метод `JsonParser.getString`. Для событий `VALUE_NUMBER` вы также можете использовать следующие методы:

- `JsonParser.isIntegralNumber`
- `JsonParser.getInt`
- `JsonParser.getLong`
- `JsonParser.getBigDecimal`

Дополнительная информация приведена в описании API Jakarta EE интерфейса `jakarta.json.stream.JsonParser`.

Выходные данные этого примера следующие:

```
START_OBJECT
KEY_NAME firstName - VALUE_STRING Duke
KEY_NAME lastName - VALUE_STRING Java
KEY_NAME age - VALUE_NUMBER 18
KEY_NAME streetAddress - VALUE_STRING 100 Internet Dr
KEY_NAME city - VALUE_STRING JavaTown
KEY_NAME state - VALUE_STRING JA
KEY_NAME postalCode - VALUE_STRING 12345
KEY_NAME phoneNumbers - START_ARRAY
START_OBJECT
KEY_NAME type - VALUE_STRING mobile
KEY_NAME number - VALUE_STRING 111-111-1111
END_OBJECT
START_OBJECT
KEY_NAME type - VALUE_STRING home
KEY_NAME number - VALUE_STRING 222-222-2222
END_OBJECT
END_ARRAY
END_OBJECT
```

Запись данных JSON с помощью генератора

Следующий код демонстрирует, как записать данные JSON в файл с помощью потокового API:

JAVA

```
FileWriter writer = new FileWriter("test.txt");
JsonGenerator gen = Json.createGenerator(writer);
gen.writeStartObject()
   .write("firstName", "Duke")
   .write("lastName", "Java")
   .write("age", 18)
   .write("streetAddress", "100 Internet Dr")
   .write("city", "JavaTown")
   .write("state", "JA")
   .write("postalCode", "12345")
   .writeStartArray("phoneNumbers")
      .writeStartObject()
         .write("type", "mobile")
         .write("number", "111-111-1111")
      .writeEnd()
      .writeStartObject()
         .write("type", "home")
         .write("number", "222-222-2222")
      .writeEnd()
   .writeEnd()
.writeEnd();
gen.close();
```

В этом примере получен генератор JSON путём вызова статического метода `Json.createGenerator`, который принимает в качестве параметра писателя или выходной поток. В этом примере данные JSON записываются в файл `test.txt` путём вложения вызовов в `write`, `writeStartArray`, `writeStartObject` и `writeEnd`. Метод `JsonGenerator.close` закрывает писателя или выходной поток.

JSON в Jakarta EE RESTful веб-сервисах

В этом разделе объясняется, как Jakarta JSON Processing связана с другими пакетами Jakarta EE, которые обеспечивают поддержку JSON для RESTful веб-сервисов. См. главу 32 *Создание RESTful веб-сервисов с помощью Jakarta REST* для получения дополнительной информации о RESTful веб-сервисах.

Jersey, реализация Jakarta RESTful Web Services, включённая в состав GlassFish Server, обеспечивает поддержку связывания данных JSON с объектамм Java с использованием Jakarta XML Binding, как описано в Использование Jakarta REST с Jakarta XML Binding в главе 34 *Jakarta REST: дополнительные темы и пример*.

Однако поддержка JSON не является частью Jakarta RESTful Web Services или Jakarta XML Binding, поэтому процедура может не работать в реализациях Jakarta EE, отличных от GlassFish Server.

Вы по-прежнему можете использовать обработку JSON Jakarta с методами ресурсов RESTful веб-сервисов Jakarta. Дополнительные сведения см. в примере кода для обработки JSON, включённом в примеры учебника Jakarta EE.

Приложение jsonpmodel

В этом разделе описывается, как создать и запустить приложение `jsonpmodel`. Этот пример представляет собой веб-приложение, которое демонстрирует, как создать объектную модель из данных формы, как парсить данные JSON и как записывать данные JSON с использованием API объектной модели.

Пример приложения `jsonpmodel` находится в каталоге *`tut-install`*`/examples/web/jsonp/jsonpmodel`.

Компоненты jsonpmodel

Приложение `jsonpmodel` содержит следующие файлы.

- Три страницы Jakarta Faces.
  - Страница `index.xhtml` содержит форму для сбора информации.
  - Страница `modelcreated.xhtml` содержит текстовую область, в которой отображаются данные JSON.
  - Страница `parsejson.xhtml` содержит таблицу, в которой показаны элементы объектной модели.
- Managed-бин `ObjectModelBean.java`, который является сессионным бином, хранит данные формы и управляет навигацией между страницами Facelets. Этот файл также содержит код, который использует API объектной модели JSON.

Код, используемый в `ObjectModelBean.java` для создания объектной модели из данных формы, аналогичен коду примера в Создание объектной модели в коде приложения. Код для записи вывода JSON из модели аналогичен примеру из Запись объектной модели в поток. Код для навигации по дереву объектной модели похож на пример в Навигация по объектной модели.

Запуск приложения jsonpmodel

В этом разделе описывается, как запустить приложение `jsonpmodel` в IDE NetBeans и из командной строки.

Запуск приложения jsonpmodel с использованием IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/web/jsonp
   ```

4. Выберите каталог `jsonpmodel`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `jsonpmodel` и выберите **Запуск**.

   Эта команда собирает и упаковывает приложение в WAR-файл (`jsonpmodel.war`), расположенный в каталоге `target`, развёртывает его на сервере и открывает окно веб-браузера со следующим URL:

   ```
   http://localhost:8080/jsonpmodel/
   ```

7. Отредактируйте данные на странице и нажмите «Создать объект JSON», чтобы отправить форму. На следующей странице показан объект JSON, который содержит данные формы.
8. Нажмите Parse JSON. На следующей странице содержится таблица, в которой перечислены узлы дерева объектной модели.

Запуск приложения jsonpmodel с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

```
tut-install/examples/web/jsonp/jsonpmodel
```

3. Введите следующую команду для развёртывания приложения:

```
mvn install
```
SHELL

4. Откройте окно веб-браузера и введите следующий адрес:

```
http://localhost:8080/jsonpmodel/
```

5. Отредактируйте данные на странице и нажмите «Создать объект JSON», чтобы отправить форму. На следующей странице показан объект JSON, который содержит данные формы.
6. Нажмите Parse JSON. На следующей странице содержится таблица, в которой перечислены узлы дерева объектной модели.

Приложение jsonpstreaming

В этом разделе описывается, как создать и запустить приложение `jsonpstreaming`. Этот пример представляет собой веб-приложение, которое демонстрирует, как создавать данные JSON из данных формы, как парсить данные JSON и как записывать выходные данные JSON с помощью потокового API.

Пример приложения `jsonpstreaming` находится в каталоге *`tut-install`*`/examples/web/jsonp/jsonpstreaming`.

Компоненты приложения jsonpstreaming

Приложение `jsonpstreaming` содержит следующие файлы.

- Три страницы Jakarta Faces.
  - Страница `index.xhtml` содержит форму для сбора информации.
  - Страница `filewritten.xhtml` содержит текстовую область, в которой отображаются данные JSON.
  - Страница `parsed.xhtml` содержит таблицу, в которой перечислены события парсера.
- Managed-бин `StreamingBean.java` — сессионный бин, который хранит данные формы и управляет навигацией между страницами Facelets. Этот файл также содержит код, который использует потоковый API JSON.

Код, используемый в `StreamingBean.java` для записи данных JSON в файл, аналогичен коду примера Запись данных JSON с помощью генератора. Код для анализа данных JSON из файла аналогичен примеру в Чтении данных JSON с использованием парсера.

Запуск приложения jsonpstreaming

В этом разделе описывается, как запустить приложение `jsonpstreaming` в IDE NetBeans и из командной строки.

Запуск приложения jsonpstreaming с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/web/jsonp
   ```

4. Выберите каталог `jsonpstreaming` .
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `jsonpstreaming` и выберите **Запуск**.

   Эта команда собирает и упаковывает приложение в WAR-файл (`jsonpstreaming.war`), расположенный в каталоге `target` , развёртывает его на сервере и открывает окно веб-браузера со следующим URL:

   ```
   http://localhost:8080/jsonpstreaming/
   ```

7. Отредактируйте данные на странице и нажмите «Записать объект JSON в файл», чтобы отправить форму и записать объект JSON в текстовый файл. На следующей странице показано содержимое текстового файла.
8. Нажмите Парсить JSON из файла. На следующей странице содержится таблица, в которой перечислены события парсера для данных JSON в текстовом файле.

Запуск приложения jsonpstreaming с помощью Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/web/jsonp/jsonpstreaming/
   ```

3. Введите следующую команду для развёртывания приложения:

   ```
   mvn install
   ```
   SHELL

4. Откройте окно веб-браузера и введите следующий URL:

   ```
   http://localhost:8080/jsonpstreaming/
   ```

5. Отредактируйте данные на странице и нажмите «Записать объект JSON в файл», чтобы отправить форму и записать объект JSON в текстовый файл. На следующей странице показано содержимое текстового файла.
6. Нажмите Парсить JSON из файла. На следующей странице содержится таблица, в которой перечислены события парсера для данных JSON в текстовом файле.

Дополнительная информация о Jakarta JSON Processing

Дополнительные сведения об обработке JSON в Jakarta EE см. в спецификации Jakarta JSON Processing:

https://jakarta.ee/specifications/jsonp/2.0/

Глава 21. Связывание с JSON

В этой главе описывается связывание JSON Jakarta. JSON — это формат обмена данными, широко используемый в веб-сервисах и других связанных приложениях. Краткий обзор JSON см. в разделе Введение в JSON.

Спецификация Jakarta JSON Binding (https://jakarta.ee/specifications/jsonb/) обеспечивает стандартный уровень связывания (метаданные и среда выполнения) между классами Java и документами JSON. Одним из эталонных решений Jakarta JSON Binding является Yasson, который разработан в Eclipse.org и входит в состав GlassFish Server.

Вы можете узнать больше о Yasson на сайте https://projects.eclipse.org/projects/ee4j.yasson.

Связывание с JSON в платформе Jakarta EE

Jakarta EE включает поддержку спецификации Jakarta JSON Binding, которая предоставляет API, который позволяет сериализовать объекты Java в документы JSON и десериализовать документы JSON в объекты Java. Jakarta JSON Binding содержит следующие пакеты:

- Пакет `jakarta.json.bind` содержит интерфейсы связывания, конструктора и класса конфигурации. Таблица 21-1 перечисляет основные классы и интерфейсы в этом пакете.
- Пакет `jakarta.json.bind.adapter` содержит интерфейс `JsonbAdapter`, который предоставляет методы для связывания пользовательских типов Java путём их преобразования в известные типы.
- Пакет `jakarta.json.bind.annotation` определяет аннотации, которые можно использовать для кастомизации поведения связывания по умолчанию. Аннотации могут использоваться для поля, свойства JavaBean, типа или элементов пакета.
- Интерфейсы и классы пакета `jakarta.json.bind.config` для кастомизации поведения связывания по умолчанию. Таблица 21-2 перечисляет основные классы и интерфейсы в этом пакете.
- Пакет `jakarta.json.bind.serializer` содержит интерфейсы, которые используются для создания процедур сериализации и десериализации для пользовательских типов, которые не могут быть сопоставлены стандартными методами `JSONBAdapter`. Таблица 21-3 перечисляет основные интерфейсы в этом пакете.
- Пакет `jakarta.json.bind.spi` содержит интерфейс поставщика услуг (SPI) для создания реализаций связывания JSON. Этот пакет содержит класс `JsonbProvider`, который содержит методы, реализующие поставщик услуг.

Таблица 21-1 Основные классы и интерфейсы в jakarta.json.bind

Класс или интерфейс	Описание
`Jsonb`	Содержит методы преобразования для сериализации объектов Java в JSON и десериализации JSON в объекты Java.
`JsonBuilder`	Используется клиентами для создания объектов `Jsonb`.
`JsonbConfig`	Используется для установки свойств конфигурации в объектах `Jsonb`. Свойства включают стратегии связывания и свойства для настройки кастомных сериализаторов и десериализаторов.
`JsonbException`	Сообщает о проблемах, возникших в процессе преобразования JSON.

Таблица 21-2 Основные классы и интерфейсы в jakarta.json.bind.config

Класс или интерфейс	Описание
`PropertyNamingStrategy`	Используется для установки способа перевода имён свойств.
`PropertyVisibilityStrategy`	Используется для определения, следует ли рассматривать поля и методы как свойства, переопределяющие область видимости по умолчанию и поведение доступа к полям.
`BinaryDataStrategy`	Используется для установки бинарного кодирования.
`PropertyOrderStrategy`	Используется для установки способа упорядочения свойств при сериализации.

Таблица 21-3 Основные классы и интерфейсы в jakarta.json.bind.serializer

Класс или интерфейс	Описание
`JsonbDeserializer`	Используется для создания процедуры десериализации для пользовательского типа.
`JsonbSerializer`	Используется для создания процедуры сериализации для пользовательского типа.

Обзор JSON Binding API

В этом разделе приведены основные инструкции по использованию клиентского API Jakarta JSON Binding. Инструкции служат основой для понимания Запуска примера jsonbasics. Документацию по API и более подробное руководство пользователя см. на странице проекта Jakarta JSON Binding (http://json-b.net/index.html).

Создание объекта jasonb

Объект `jsonb` предоставляет доступ к методам преобразования объектов в JSON. Один объект `jsonb` требуется для большинства приложений. Объект `jsonb` создаётся с использованием интерфейса `JsonbBuilder`, который является точкой входа клиента в JSON Binding API. Например:

```
Jsonb jsonb = JsonbBuilder.create();
```
JAVA

Использование маппинга по умолчанию

Jakarta JSON Binding предоставляет сопоставления по умолчанию для сериализации и десериализации базовых типов Java и Java SE, а также классов даты и времени Java. Чтобы использовать сопоставления по умолчанию, создайте объект jsonb и используйте метод `toJson` для сериализации в JSON, а метод `fromJson` для десериализации обратно в объект. В следующем примере связывание осуществляется для простого объекта `Person`, который содержит одно поле `name`.

```java
Jsonb jsonb = JsonbBuilder.create();

Person person = new Person();
person.name = "Fred";

Jsonb jsonb = JsonbBuilder.create();

// сериализация в JSON
String result = jsonb.toJson(person);

// десериализация из JSON
person = jsonb.fromJson("{name:\"joe\"}", Person.class);
```

Использование настроек

Jakarta JSON Binding поддерживает множество способов настройки поведения сопоставления по умолчанию. Для настроек во время выполнения при создании `jsonbinstance` используется объект конфигурации `JsonbConfig`. Класс `JsonbConfig` поддерживает множество параметров конфигурации и включает расширенные параметры для работы с пользовательскими типами данных. Дополнительные параметры см. в интерфейсах `JsonbAdapter` и интерфейсах `JsonbSerializer` и `JsonbDeserializer`.

В следующем примере создаётся объект конфигурации, который задаёт свойство `FORMATTING`, чтобы указать, будут ли сериализованные данные JSON форматироваться с помощью перевода строки и отступа.

```java
JsonbConfig config = new JsonbConfig()
    .withFormatting(true);

Jsonb jsonb = JsonbBuilder.create(config);
```

Использование аннотаций

Jakarta JSON Binding включает в себя множество аннотаций, которые можно использовать во время компиляции для настройки поведения сопоставления по умолчанию. В следующем примере аннотация `@JsonbProperty` используется для изменения поля `name` на `person-name`, когда объект сериализуется в JSON.

```java
public class Person {
    @JsonbProperty("person-name")
    private String name;
}
```

Результирующий документ JSON записывается как:

```json
{
    "person-name": "Fred",
}
```

Запуск приложения jsonbbasics

В этом разделе описывается, как создать и запустить приложение `jsonbbasics`. Этот пример представляет собой веб-приложение, которое демонстрирует, как сериализовать объект в JSON и как десериализовать JSON в объект.

Приложение `jsonbbasics` находится в каталоге *`tut-install`*`/examples/web/jsonb/jsonbbasics`.

Компоненты приложения jsonbbasics

Приложение `jsonbbasics` содержит следующие файлы.

- Две страницы Jakarta Faces.
  - Страница `index.xhtml` содержит форму для сбора данных, которые используются для создания объекта `Person`.
  - Страница `jsongenerated.xhtml` содержит текстовую область, которая отображает данные в формате JSON.
- В `JsonbBean.java`, который представляет собой Managed-бин с областью видимости сессии, хранит данные из формы и управляет навигацией между страницами Facelets. Этот файл содержит код, который использует JSON Binding API.

Запуск приложения jsonbbasics

В этом разделе описывается, как запустить приложение `jsonbbasics` из командной строки с помощью Maven.

Чтобы запустить пример jsonbbasics с помощью Maven:

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/web/jsonb/jsonbbasics
   ```

3. Введите следующую команду для развёртывания приложения:

   ```
   mvn install
   ```
   SHELL

4. Откройте окно веб-браузера и введите следующий адрес:

   ```
   http://localhost:8080/jsonbbasics/
   ```

5. Введите данные в форму и нажмите Сериализация в JSON, чтобы отправить форму. На следующей странице показаны данные объекта в формате JSON.
6. Нажмите Десериализовать JSON. Страница индекса отображает и содержит поля, заполненные данными объекта.

Дополнительная информация о Jakarta JSON Binding

Для получения дополнительных сведений о Jakarta JSON Binding см.:

- Спецификация Jakarta JSON Binding:
  https://jakarta.ee/specifications/jsonb/
- Проект спецификации:
  https://github.com/eclipse-ee4j/jsonb-api
- Yasson (реализация):
  https://projects.eclipse.org/projects/ee4j.yasson

Глава 22. Интернационализация и локализация веб-приложений

Процесс подготовки приложения для поддержки более чем одного языка и формата данных называется интернационализация. Локализация — это процесс адаптации интернационализированного приложения для поддержки определённого региона или локали. Примеры информации, зависящей от локали, включают сообщения и метки пользовательского интерфейса, наборы символов и кодировку, а также форматы даты и валюты. Хотя все клиентские пользовательские интерфейсы должны быть интернационализированы и локализованы, это особенно важно для веб-приложений из-за глобальной природы сети.

Классы локализации платформы Java

В платформе Java `java.util.Locale` (https://docs.oracle.com/javase/8/docs/api/java/util/Locale.html) представляет конкретный географический, политический или культурный регион. Строковое представление локали состоит из международной стандартной двухсимвольной аббревиатуры для языка и страны и необязательного варианта, разделённых символами подчёркивания (`_`). Примеры строк локали: `fr` (французский), `DE_CH` (швейцарский немецкий) и `en_US_POSIX` (английский язык на POSIX-совместимой платформе).

Данные, зависящие от локали, хранятся в `java.util.ResourceBundle` (https://docs.oracle.com/javase/8/docs/api/java/util/ResourceBundle.html). Bundle-ресурс содержит пары ключ-значение, где ключи однозначно определяют локализованный объект. Bundle-ресурс может быть текстовым файлом (properties-файл) или классом, содержащим пары. Bundle-ресурс создаётся добавлением строкового представления локали к базовому имени.

Приложение Duke's Bookstore (см. глава 61 *Пример Duke's Bookstore*) содержит bundle-ресурсы с базовым именем `messages.properties` для локалей `de` (немецкий), `es` (испанский) и `fr` (французский). Локаль по умолчанию, `en` (английский), которая указана в файле `faces-config.xml` , использует bundle-ресурс с базовым именем `messages.properties` .

Дополнительные сведения об интернационализации и локализации в платформе Java см. https://docs.oracle.com/javase/tutorial/i18n/index.html.

Предоставление локализованных сообщений и меток

Сообщения и метки должны быть адаптированы в соответствии с языком и регионом пользователя. Существует два подхода к предоставлению локализованных сообщений и меток в веб-приложении.

- Укажите версию веб-страницы в каждой из целевых локалей и попросите сервлет контроллера отправить запрос на соответствующую страницу в зависимости от запрашиваемой локали. Этот подход полезен, если необходимо интернационализировать большие объёмы данных на странице или во всём веб-приложении.
- Изоляция любых чувствительных к локали данных на странице в bundle-ресурсы и обращение к ним, чтобы соответствующее переведённое сообщение автоматически выбиралось и вставлялось на страницу. Таким образом, вместо создания строк непосредственно в вашем коде, вы создаёте пакет ресурсов, который содержит переводы, и читаете переводы из этого пакета, используя соответствующий ключ.

Приложение Duke's Bookstore следует второму подходу. Вот несколько строк из bundle-ресурса по умолчанию `messages.properties` :

```
TitleShoppingCart=Shopping Cart
TitleReceipt=Receipt
TitleBookCatalog=Book Catalog
TitleCashier=Cashier
TitleBookDescription=Book Description
Visitor=You are visitor number
What=What We\'re Reading
```

Создание Локали

Чтобы получить подходящие для данного пользователя строки, веб-приложение либо извлекает локаль (заданную в языковых настройках браузера) из запроса, используя метод `getLocale`, либо позволяет пользователю явно выбрать локаль.

Компонент может явно установить локаль с помощью тега `fmt:setLocale`.

Элемент `locale-config` в файле конфигурации регистрирует локаль по умолчанию, а также регистрирует другие поддерживаемые локали. Этот элемент в Duke's Bookstore регистрирует английский язык как язык по умолчанию и указывает, что поддерживаются также немецкий, французский и испанский языки.

XML

```xml
<locale-config>
    <default-locale>en</default-locale>
    <supported-locale>es</supported-locale>
    <supported-locale>de</supported-locale>
    <supported-locale>fr</supported-locale>
</locale-config>
```

`LocaleBean` в приложении Duke's Bookstore использует метод `getLocale` для получения локали.

JAVA

```java
public class LocaleBean {

    ...
    private FacesContext ctx = FacesContext.getCurrentInstance();
    private Locale locale = ctx.getViewRoot().getLocale();;

    ...
}
```

Настройка bundle-ресурса

Bundle-ресурс устанавливается с помощью элемента `resource-bundle` в файле конфигурации. Настройка для Duke's Bookstore выглядит следующим образом:

XML

```xml
<resource-bundle>
    <base-name>
        ee.jakarta.tutorial.dukesbookstore.web.messages.Messages
    </base-name>
    <var>bundle</var>
</resource-bundle>
```

После установки локали контроллер веб-приложения может получить пакет ресурсов для этой локали и сохранить его как атрибут сессии (см. Связывание объектов с сессией) для использования другими компонентами или просто используется для возврата текстовой строки, соответствующей выбранной локали:

JAVA

```
public String toString(Locale locale) {
    ResourceBundle res =
        ResourceBundle.getBundle(
            "ee.jakarta.tutorial.dukesbookstore.web.messages.Messages", locale);
    return res.getString(name() + ".string");
}
```

Кроме того, приложение может использовать тег `f:loadBundle` для установки bundle-ресурса. Этот тег загружает корректныц bundle-ресурс в соответствии с локалью, хранящейся в `FacesContext` .

XML

```
<f:loadBundle basename="ee.jakarta.tutorial.dukesbookstore.web.messages.Messages"
    var="bundle"/>
```

Bundle-ресурсы, содержащие сообщения, на которые явно ссылаются из атрибута тега Jakarta Faces с использованием выражения значения, должны быть зарегистрированы с использованием элемента `resource-bundle` в файле конфигурации.

Для получения дополнительной информации об использовании этого элемента см. Регистрация сообщений приложения.

Получение локализованных сообщений

Веб-компонент, написанный на Java, получает bundle-ресурс из сессии:

JAVA

```
ResourceBundle messages = (ResourceBundle)session.getAttribute("messages");
```

Затем он ищет строку, связанную с ключом `person.lastName` , следующим образом:

JAVA

```
messages.getString("person.lastName");
```

Вы можете использовать тег `message` или `messages` только для отображения сообщений, помещённых в очередь в компоненте в результате регистрации конвертера или валидатора в компоненте. В следующем примере показан тег `message` , который отображает сообщение об ошибке, поставленное в очередь во входном компоненте `userNo` , если зарегистрированный в компоненте валидатор не может проверить значение, введённое пользователем в компонент.

XML

```
<h:inputText id="userNo" value="#{userNumberBean.userNumber}">
    <f:validateLongRange minimum="0" maximum="10" />
</h:inputText>
...
<h:message style="color: red; text-decoration: overline"
           id="errors1" for="userNo"/>
```

Для получения дополнительной информации об использовании тегов `message` и `messages` см. Отображение сообщений об ошибках с тегами h:message и h:messages.

На сообщения, которые не поставлены в очередь компонента и поэтому не загружаются автоматически, ссылаются с помощью выражения значения. Вы можете ссылаться на локализованное сообщение практически из любого атрибута тега Jakarta Faces.

Выражение значения, которое ссылается на сообщение, имеет одинаковую нотацию, независимо от того, загрузили ли вы bundle-ресурс тегом `loadBundle` или зарегистрировали его с помощью элемента `resource-bundle` в файле конфигурации.

Нотацией выражения значения является `var.message`, в которой `var` соответствует атрибуту `var` тега `loadBundle` или элемент `var` определён в элементе `resource-bundle` файла конфигурации, а `message` соответствует ключу сообщения, содержащемуся в пакете ресурсов, который упоминается с помощью атрибута `var`.

Вот пример из `bookcashier.xhtml` в Duke's Bookstore:

```xml
<h:outputLabel for="name" value="#{bundle.Name}" />
```

Обратите внимание, что `bundle` соответствует элементу `var` из файла конфигурации, а `Name` соответствует ключу в bundle-ресурсе.

Форматирование дат и чисел

Java-программы используют метод `DateFormat.getDateInstance(int, locale)` для синтаксического анализа и форматирования дат с учетом локали. Java-программы используют метод `NumberFormat.get`*XXX*`Instance(locale)`, где *XXX* может быть `Currency`, `Number`, или `Percent`, чтобы распарсить и форматировать числовые значения зависимым от локали образом.

Приложение может использовать конвертеры даты/времени и чисел для форматирования дат и чисел с учётом языка. Например, дату отгрузки можно преобразовать следующим образом:

```xml
<h:outputText value="#{cashier.shipDate}">
    <f:convertDateTime dateStyle="full"/>
</h:outputText>
```

Для получения информации о конвертерах Jakarta Faces см. Использование стандартных конвертеров.

Наборы и кодировки символов

В следующих разделах описываются наборы и кодировки символов.

Наборы символов

Набор символов — это набор текстовых и графических символов, каждый из которых сопоставлен с набором неотрицательных целых чисел.

Первый набор символов, использованный в вычислениях, был US-ASCII. Он ограничен тем, что может представлять только американский английский. US-ASCII содержит прописные и строчные латинские алфавиты, цифры, знаки препинания, набор контрольных кодов и некоторые другие символы.

Unicode определяет стандартизированный универсальный набор символов, который можно расширять добавлением дополнений. Если кодировка исходного файла программы Java не поддерживает Unicode, вы можете представить символы Unicode как escape-последовательности, используя обозначение `\\u` *XXXX*, где *XXXX* — 16-битное представление символа в шестнадцатеричном формате. Например, испанская версия файла сообщения может использовать Unicode для символов, отличных от ASCII, следующим образом:

```
admin.nav.main=P\u00e1gina principal de administraci\u00f3n
```

Кодировки символов

Кодировка символов отображает набор символов в единицы определённой ширины и определяет сериализацию байтов и правила упорядочения. Многие наборы символов имеют более одной кодировки. Например, Java-программы могут представлять японские наборы символов, используя, среди прочего, кодировки `EUC-JP` или `Shift-JIS`. Каждая кодировка имеет правила для представления и сериализации набора символов.

Серия ISO 8859 определяет 13 кодировок символов, которые могут представлять тексты на десятках языков. Каждая кодировка ISO 8859 может содержать до 256 символов. ISO-8859-1 (Latin-1) содержит набор символов ASCII, символы с диакритическими знаками (акценты, диерезисы, седилы, циркумфлексы и т. д.) и дополнительные символы.

UTF-8 (формат преобразования Unicode, 8-битная форма) представляет собой кодировку символов переменной ширины, которая кодирует 16-битные символы Unicode в виде от одного до четырёх байтов. Байт в UTF-8 эквивалентен 7-битному ASCII, если его старший бит равен нулю. В противном случае символ содержит переменное число байтов.

UTF-8 совместим с большинством существующего в веб контента и обеспечивает доступ к набору символов Unicode. Текущие версии браузеров и почтовых клиентов поддерживают UTF-8. Кроме того, многие веб-стандарты указывают кодировку UTF-8. Например, UTF-8 является одним из двух обязательных кодировок для документов XML (другой — UTF-16).

Веб-компоненты обычно используют `PrintWriter` для получения ответов. `PrintWriter` по умолчанию кодирует с использованием ISO-8859-1. Сервлеты также могут выводить бинарные данные, используя классы `OutputStream`, которые не выполняют кодирование. Приложение, использующее набор символов, который не может использовать кодировку по умолчанию, должно явно установить другую кодировку.

Часть IV: Bean Validation

Часть IV исследует Jakarta Bean Validation.

Глава 23. Введение в Jakarta Bean Validation

В этой главе описывается Jakarta Bean Validation, доступная как часть платформы Jakarta EE, и средство валидации объектов, членов объектов, методов и конструкторов.

Обзор Jakarta Bean Validation

Валидация вводимых пользователем данных необходима для поддержания целостности данных и является важной частью логики приложения. Валидация данных может проходить на разных уровнях даже в самых простых приложениях, как показано в Разработка простого приложения Facelets: пример guessnumber-jsf. Приложение `guessnumber-jsf` проверяет пользовательский ввод (в теге `h:inputText`) числовых данных на уровне представления и попадание в допустимый диапазон чисел на уровне бизнес-логики.

Jakarta Bean Validation предоставляет возможность для валидации объектов, членов объектов, методов и конструкторов. В средах Jakarta EE Jakarta Bean Validation интегрируется с контейнерами и сервисами Jakarta EE, что позволяет разработчикам легко задать и применить ограничения валидации. Jakarta Bean Validation доступна как часть платформы Jakarta EE.

Использование ограничений Jakarta Bean Validation

Модель Jakarta Bean Validation поддерживается ограничениями в виде аннотаций, размещаемых на поле, методе или классе компонента JavaBeans, такого как Managed-бин.

Ограничения бывают стандартными или могут быть заданы пользователем. Ограничения, заданные пользователем, называются пользовательскими ограничениями. В пакете `jakarta.validation.constraints` доступно несколько встроенных ограничений. Таблица 23-1 перечисляет все предустановленные ограничения. Смотрите Создание пользовательских ограничений для получения информации о создании пользовательских ограничений.

Таблица 23-1 Встроенные ограничения Jakarta Bean Validation

Ограничение	Описание	Пример

`@AssertFalse`	Значение поля или свойства должно быть `false`.	JAVA
		`@AssertFalse`
		`boolean isUnsupported;`

`@AssertTrue`	Значение поля или свойства должно быть `true`.	JAVA
		`@AssertTrue`
		`boolean isActive;`

`@DecimalMax`	Значение поля или свойства должно быть десятичным числом, меньшим или равным числу в элементе value.	JAVA
		`@DecimalMax("30.00")`
		`BigDecimal discount;`

`@DecimalMin`	Значением поля или свойства должно быть десятичным числом, большим или равным числу в элементе value.	JAVA
		`@DecimalMin("5.00")`
		`BigDecimal discount;`

Ограничение	Описание	Пример

`@Digits`	Значение поля или свойства должно быть числом в указанном диапазоне. Элемент `integer` определяет максимальное количество разрядов целой части числа, а элемент `fraction` — дробной части.	JAVA
		`@Digits(integer=6, fraction=2)`
		`BigDecimal price;`

`@Email`	Значение поля или свойства должно быть валидным адресом электронной почты.	JAVA
		`@Email`
		`String emailaddress;`

`@Future`	Значение поля или свойства должно быть датой в будущем.	JAVA
		`@Future`
		`Date eventDate;`

`@FutureOrPresent`	Значение поля или свойства должно быть датой или временем в настоящем или будущем.	JAVA
		`@FutureOrPresent`
		`Time travelTime;`

`@Max`	Значение поля или свойства должно быть целым числом, меньшим или равным числу в элементе value.	JAVA
		`@Max(10)`
		`int quantity;`

`@Min`	Значение поля или свойства должно быть целым числом, большим или равным числу в элементе value.	JAVA
		`@Min(5)`
		`int quantity;`

`@Negative`	Значение поля или свойства должно быть отрицательным числом.	JAVA
		`@Negative`
		`int basementFloor;`

`@NegativeOrZero`	Значение поля или свойства должно быть отрицательным числом или нулём.	JAVA
		`@NegativeOrZero`
		`int debtValue;`

`@NotBlank`	Значение поля или свойства должно содержать как минимум один символ, не являющийся пробелом.	JAVA
		`@NotBlank`
		`String message;`

`@NotEmpty`	Значение поля или свойства не должно быть пустым. Вычисляется длина символов или массива, а также размер коллекции или отображения (Map).	JAVA
		`@NotEmpty`
		`String message;;`

Ограничение	Описание	Пример

`@NotNull`	Значение поля или свойства не должно быть null.	JAVA
		`@NotNull`
		`String username;`

`@Null`	Значение поля или свойства должно быть null.	JAVA
		`@Null`
		`String unusedString;`

`@Past`	Значение поля или свойства должно быть датой в прошлом.	JAVA
		`@Past`
		`Date birthday;`

`@PastOrPresent`	Значением поля или свойства должна быть дата или время в прошлом или настоящем.	JAVA
		`@PastOrPresent`
		`Date travelDate;`

`@Pattern`	Значение поля или свойства должно соответствовать регулярному выражению, заданному в элементе `regexp`.	JAVA
		`@Pattern(regexp="\\(\\d{3}\\)\\d{3}-\\d{4}")`
		`String phoneNumber;`

`@Positive`	Значение поля или свойства должно быть положительным числом.	JAVA
		`@Positive`
		`BigDecimal area;`

`@PositiveOrZero`	Значение поля или свойства должно быть положительным числом или нулём.	JAVA
		`@PositiveOrZero`
		`int totalGoals;`

`@Size`	Размер поля или свойства вычисляется и должен соответствовать указанным границам. Если поле или свойство представляет собой `String`, вычисляется размер строки. Если поле или свойство представляет собой `Collection`, вычисляется размер `Collection`. Если поле или свойство является `Map`, вычисляется размер `Map`. Если поле или свойство является массивом, вычисляется размер массива. Используйте один из необязательных элементов `max` или `min`, чтобы указать границы.	JAVA
		`@Size(min=2, max=240)`
		`String briefMessage;`

В следующем примере на поле накладывается ограничение с использованием предустановленного ограничения `@NotNull`:

JAVA

```java
public class Name {
    @NotNull
    private String firstname;

    @NotNull
    private String lastname;
    ...
}
```

В объекте компонента JavaBeans можно разместить более одного ограничения. Например, можно дополнительно ограничить размеры полей `firstname` и `lastname`:

JAVA

```java
public class Name {
    @NotNull
    @Size(min=1, max=16)
    private String firstname;

    @NotNull
    @Size(min=1, max=16)
    private String lastname;
    ...
}
```

В следующем примере показан метод с пользовательским ограничением, которое проверяет предварительно заданный шаблон телефонного номера, например номер телефона конкретной страны:

JAVA

```java
@USPhoneNumber
public String getPhone() {
    return phone;
}
```

Для предустановленного ограничения доступна реализация по умолчанию. Определяемое пользователем ограничение требует реализации. В предыдущем примере пользовательскому ограничению `@USPhoneNumber` требуется класс реализации.

Повторяющиеся аннотации

Начиная с Bean Validation 2.0, вы можете указать одно и то же ограничение несколько раз для цели проверки, используя повторяющуюся аннотацию:

JAVA

```java
public class Account {

    @Max (value = 2000, groups = Default.class, message = "max.value")
    @Max (value = 5000, groups = GoldCustomer.class, message = "max.value")
    private long withdrawalAmount;
}
```

Все встроенные ограничения из пакета `jakarta.validation.constraints` поддерживают повторяющиеся аннотации. Аналогичным образом, пользовательские ограничения могут использовать аннотацию `@Repeatable`. В следующем примере, в зависимости от группы `PeakHour` или `NonPeakHour`, проверяется, является ли объект автомобиля трёх- или четырёхместным автомобилем и затем указывается подходящая полоса для движения:

JAVA

```
/**
 * Validate whether a car is eligible for car pool lane
 */
@Documented
@Constraint(validatedBy = CarPoolValidator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
public @interface CarPool {

    String message() default "{CarPool.message}";

    Class[] groups() default {};

    int value();

    Classextends Payload>[] payload() default {};

    /**
     * Defines several @CarPool annotations on the same element
     * @see (@link CarPool}
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        CarPool[] value();
    }
}
public class Car{

  private String registrationNumber;

  @CarPool(value = 2, group = NonPeakHour.class)
  @CarPool(value = 3, group = {Default.class, PeakHour.class})
  private int totalPassengers;
}
```

Любые ошибки валидации корректно обрабатываются и могут отображаться тегом `h:messages`.

Любой Managed-бин, содержащий аннотации Bean Validation, автоматически приобретает ограничения, объявленные для полей на веб-страницах Jakarta Faces.

Для получения дополнительной информации об использовании ограничений проверки см. следующее:

- Глава 24 *Bean Validation: дополнительные темы*
- Валидация данных ресурса с Bean Validation
- Валидация персистентных полей и свойств

Валидация строк на null и пустоту

Язык программирования Java различает null и пустые строки. Пустая строка — это строковый объект нулевой длины, тогда как строка null вообще не имеет значения.

Пустая строка представляется как `""`. Это последовательность из нулевых символов. Строка null представлена как `null`. Это можно описать как отсутствие объекта строки.

Элементы Managed-бина, представленные в виде текстового компонента Jakarta Faces, такого как `inputText`, инициализируются Jakarta Faces значением пустой строки. Проверка этих строк может быть проблемой, когда пользовательский ввод для таких полей не требуется. Рассмотрим следующий пример, где строка `testString`

является переменной компонента, которая будет установлена с использованием пользовательского ввода. В этом случае пользовательский ввод для поля не требуется.

```java
if (testString==null) {
    doSomething();
} else {
    doAnotherThing();
}
```

По умолчанию метод `doAnotherThing` вызывается даже тогда, когда пользователь не вводит данные, поскольку элемент `testString` был инициализирован со значением пустой строки.

Чтобы модель Bean Validation работала должным образом, вы должны установить для контекстного параметра `jakarta.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL` значение `true` в файле дескриптора веб-развёртывания `web .xml` :

```xml
<context-param>
    <param-name>jakarta.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL</param-name>
    <param-value>true</param-value>
</context-param>
```

Этот параметр указывает Jakarta Faces обрабатывать пустые строки как null.

Предположим, с другой стороны, что у вас есть ограничение `@NotNull` для элемента, что означает, что требуется ввод. В этом случае пустая строка пройдет это ограничение проверки. Однако, если вы установите для контекстного параметра `jakarta.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL` значение `true` , значение атрибута Managed-бина передаётся в среду выполнения Jakarta Bean Validation как null, вызывая сбой ограничения `@NotNull` .

Валидация конструкторов и методов

Ограничения Jakarta Bean Validation могут быть применены к параметрам нестатических методов и конструкторов, а также на возвращаемые значения нестатических методов. Статические методы и конструкторы не будут проверяться.

```java
public class Employee {
...
  public Employee (@NotNull String name) { ... }

  public void setSalary(
      @NotNull
      @Digits(integer=6, fraction=2) BigDecimal salary,
      @NotNull
      @ValidCurrency
      String currencyType) {
    ...
  }
...
}
```

В этом примере класс `Employee` имеет ограничение конструктора, требующее имени, и имеет два набора ограничений параметров метода. Размер заработной платы работника не должен быть null, не может быть больше шести цифр слева от десятичной запятой и не может содержать более двух цифр справа от десятичной запятой. Тип валюты не должен быть null и валидируется с использованием пользовательского ограничения.

Если вы добавляете ограничения методов к классам в иерархии объектов, необходимо соблюдать особую осторожность, чтобы избежать непреднамеренного изменения поведения дочерних типов. См. Использование ограничений методов в иерархиях типов для получения дополнительной информации.

Ограничения группы параметров

Ограничения, которые применяются к нескольким параметрам, называются ограничениями группы параметров и могут применяться на уровне метода или конструктора.

JAVA

```
@ConsistentPhoneParameters
@NotNull
public Employee (String name, String officePhone, String mobilePhone) {
  ...
}
```

В этом примере пользовательское ограничение группы параметров `@ConsistentPhoneParameters` проверяет соответствие формата телефонных номеров, переданных в конструктор. Ограничение `@NotNull` применяется ко всем параметрам в конструкторе.

> Аннотации ограничений группы параметров применяются непосредственно к методу или конструктору. Ограничения возвращаемого значения также применяются непосредственно к методу или конструктору. Чтобы избежать путаницы относительно того, где применяется ограничение — в параметре или возвращаемом значении — выберите имена для всех пользовательских ограничений, которые определяют, где применяется каждое ограничение. Например, предыдущий пример применяет пользовательское ограничение `@ConsistentPhoneParameters`, которое указывает, что оно применяется к параметрам метода или конструктора.
>
> Когда вы создаёте пользовательское ограничение, которое применяется как к параметрам метода, так и к возвращаемым значениям, элемент `validationAppliesTo` аннотации ограничения может быть установлен в `ConstraintTarget.RETURN_VALUE` или `ConstraintTarget.PARAMETERS` для явной установки цели ограничения валидации.

Валидация типов аргументов параметризованных типов

Начиная с Bean Validation 2.0, вы можете применять ограничения к аргументам типа параметризованных типов. Например: `List<@NotNull Long> numbers;` Ограничения могут применяться к элементам типов контейнеров, таким как `List`, `Map`, `Optional` и другие.

JAVA

```
List<@Email String> emails;
public Map<@NotNull String, @USPhoneNumber String> getAddressesByType() { }
```

В этом примере `@Email` является предустановленным ограничением, поддерживаемым валидацией бинов, а `@USPhoneNumber` является пользовательским ограничением. Смотрите Использование предустановленных ограничений для создания нового ограничения.

`@USPhoneNumber` имеет `ElementType.TYPE_USE` одним из своих `@Target`, и поэтому можно использовать ограничение `@USPhoneNumber` для валидации типов аргументов параметризованных типов.

Выявление нарушений ограничений параметров

Если во время вызова метода возникает `ConstraintViolationException`, среда выполнения Bean Validation возвращает индекс параметра, чтобы определить, какой параметр вызвал это нарушение. Индекс параметра имеет вид `arg`*`PARAMETER_INDEX`*, где *PARAMETER_INDEX* — это целое число, которое начинается с 0 для первого параметра метода или конструктора.

Добавление ограничений на возвращаемые значения при вызове метода

Чтобы проверить возвращаемое значение для метода, вы можете применить ограничения непосредственно к объявлению метода или конструктора.

JAVA

```java
@NotNull
public Employee getEmployee() { ... }
```

Ограничения группы параметров также применяются на уровне методов. Пользовательские ограничения, которые могут применяться как к возвращаемому значению, так и к параметрам метода, имеют неоднозначную цель ограничения. Чтобы избежать этой неоднозначности, добавьте элемент `validationAppliesTo` в определение аннотации ограничения со значением по умолчанию, равным `ConstraintTarget.RETURN_VALUE` или `ConstraintTarget.PARAMETERS`, чтобы явно установить цель ограничения валидации.

JAVA

```java
@Manager(validationAppliesTo=ConstraintTarget.RETURN_VALUE)
public Employee getManager(Employee employee) { ... }
```

См. Устранение неоднозначности в целях ограничений для получения дополнительной информации.

Дополнительная информация о Jakarta Bean Validation

Дополнительная информация о Jakarta Bean Validation приведена в разделе

- Спецификация Jakarta Bean Validation 3.0:
  https://jakarta.ee/specifications/bean-validation/3.0/
- Сайт спецификации Bean Validation:
  https://beanvalidation.org/

Глава 24. Валидация бинов: дополнительные темы

В этой главе описывается, как создавать кастомные ограничения, кастомные сообщения валидатора и группы ограничений с помощью Jakarta Bean Validation (Bean Validation).

Создание пользовательских ограничений

Jakarta Bean Validation определяет аннотации, интерфейсы и классы, чтобы позволить разработчикам создавать пользовательские ограничения.

Использование предустановленных ограничений для создания нового ограничения

Jakarta Bean Validation включает несколько встроенных ограничений, которые можно комбинировать для создания новых многократно используемых ограничений. Это может упростить определение ограничения, позволяя разработчикам определять пользовательское ограничение, состоящее из нескольких предустановленных ограничений, которые затем могут быть применены к атрибутам компонента одной аннотацией.

JAVA

```
@Pattern.List({
  /* Номер в формате “+1-NNN-NNN-NNNN” */
  @Pattern(regexp = “\\+1-\\d{3}-\\d{3}-\\d{4}”)
})
@Constraint(validatedBy = {})
@Documented
@Target({ElementType.METHOD,
    ElementType.FIELD,
    ElementType.ANNOTATION_TYPE,
    ElementType.CONSTRUCTOR,
    ElementType.PARAMETER
    ElementType.Type_Use})
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(List.class)
public @interface USPhoneNumber {

  String message() default "Not a valid US Phone Number";

  Class[] groups() default {};

  Classextends Payload>[] payload() default {};

  @Target({ElementType.METHOD,
     ElementType.FIELD,
     ElementType.ANNOTATION_TYPE,
     ElementType.CONSTRUCTOR,
     ElementType.PARAMETER
     ElementType.Type_Use })
  @Retention(RetentionPolicy.RUNTIME)
  @Documented
  @interface List {
  USPhoneNumber[] value();

  }
}
```

Вы также можете реализовать `Constraint Validator` для валидации ограничения `@USPhoneNumber` . Для получения дополнительной информации об использовании `Constraint Validator` см. `jakarta.validation.ConstraintValidator` .

JAVA

```
@USPhoneNumber
protected String phone;
```

Устранение неоднозначности в целях ограничений

Для пользовательских ограничений, которые можно применять как к возвращаемым значениям, так и к параметрам метода, требуется элемент `validationAppliesTo`, чтобы определить цель ограничения.

JAVA

```
@Constraint(validatedBy=MyConstraintValidator.class)
@Target({ METHOD, FIELD, TYPE, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
public @interface MyConstraint {
  String message() default "{com.example.constraint.MyConstraint.message}";
  Class[] groups() default {};
  ConstraintTarget validationAppliesTo() default ConstraintTarget.PARAMETERS;
...
}
```

Это ограничение по умолчанию устанавливает цель `validationAppliesTo` для параметров метода.

JAVA

```
@MyConstraint(validationAppliesTo=ConstraintTarget.RETURN_TYPE)
public String doSomething(String param1, String param2) { ... }
```

В предыдущем примере целью является возвращаемое значение метода.

Реализация временны́х ограничений с помощью ClockProvider

Начиная с Bean Validation 2.0 и далее, объект `Clock` доступен для реализаций валидатора для валидации любых временны́х ограничений на основе даты или времени.

JAVA

```
ValidatorFactory validatorFactory = Validation.buildDefaultValidatorFactory();
ClockProvider clockProvider = validatorFactory.getClockProvider();
java.time.Clock Clock = clockProvider.getClock();
```

Вы также можете зарегистрировать пользовательский `ClockProvider` с помощью `ValidatorFactory`:

JAVA

```
//Регистрация кастомной реализации clock provider с фабрикой валидации
ValidatorFactory factory = Validation
       .byDefaultProvider().configure()
          .clockProvider( new CustomClockProvider() )
          .buildValidatorFactory();

//Получение и использование кастомных Clock Provider и Clock в реализации валидатора
public class CustomConstraintValidator implements ConstraintValidator<CustomConstraint, Object> {

    public boolean isValid(Object value, ConstraintValidatorContext context){
        java.time.Clock clock = context.getClockProvider().getClock();
        ...
        ...

    }
}
```

См. ClockProvider в https://jakarta.ee/specifications/platform/9/apidocs/.

Пользовательские ограничения

Рассмотрим сотрудника в фирме, расположенной в США. Когда вы регистрируете номер телефона сотрудника или изменяете номер телефона, его необходимо проверить, чтобы убедиться, что номер телефона соответствует шаблону номера телефона в США.

JAVA

```
public class Employee extends Person {

  @USPhoneNumber
  protected String phone;

  public Employee(String name, String phone, int age){
    super(name, age);
    this.phone = phone;
  }

  public String getPhone() {
    return phone;
  }

  public void setPhone(String phone) {
    this.phone = phone;
  }
}
```

Определение ограничения `@USPhoneNumber` определено в примере, приведённом в разделе Использование предустановленных ограничений для создания нового ограничения. В этом примере другое ограничение `@Pattern` используется для проверки номера телефона.

Использование предустановленных экстракторов значений в кастомных контейнерах

Каскадная валидация

Bean Validation поддерживает каскадную валидацию для различных объектов. Вы можете указать `@Valid` для члена объекта, который подлежит валидации, чтобы гарантировать, что член также валидируется каскадным способом. Вы можете проверить типы аргументов, например, параметризованные типы и их члены, если члены имеют указанную аннотацию `@Valid` .

JAVA

```
public class Department {
    private List<@Valid Employee> employeesList;
}
```

Указывая `@Valid` для параметризованного типа, когда проверяется объект `Department` , все элементы, такие как `Employee` в `employeeList` также проверены. В этом примере «phone» каждого сотрудника проверяется на соответствие ограничению `@USPhoneNumber` .

Дополнительные сведения см. https://jakarta.ee/specifications/platform/9/apidocs/

Value Extractor

При проверке объекта или графа объекта может потребоваться проверка ограничений в параметризованных типах контейнера. Чтобы проверить элементы контейнера, валидатор должен извлечь значения этих элементов в контейнере. Например, чтобы проверить значения элементов `List` на соблюдение одного или нескольких ограничений, таких как `List<@NotOnVacation Employee>` , или применить каскадную валидацию к `List<@Valid Employee>` , нужен экстрактор значений для контейнера `List` .

Jakarta Bean Validation предоставляет встроенные экстракторы значений для наиболее часто используемых типов контейнеров, таких как List, Iterable и других. Однако также возможно создать и зарегистрировать реализации извлечения значений для кастомных типов контейнеров или переопределить предустановленные реализации извлечения значений.

Рассмотрим калькулятор статистики для группы сущностей `Person` и `Employee`, который является одним из дочерних типов сущности `Person`.

JAVA

```
public class StatsCalculator<T extends Person> {

  /* Каскадная валидация с ограничением @NotNull */
  private List<@NotNull @Valid T> members = new ArrayList<T>();


  public void addMember(T member) {
    members.add(member);
  }

  public boolean removeMember(T member) {
    return members.remove(member);
  }

  public int getAverageAge() {

    if (members.size() == 0)
      return 0;

    short sum = 0;
    for (T member : members) {
      if(member != null) {
        sum += member.getAge();
      }
    }
    return sum / members.size();
  }

  public int getOldest() {
    int oldest = -1;

    for (T member : members) {
      if(member != null) {
        if (member.getAge() > oldest) {
          oldest = member.getAge();
        }
      }
    }
    return oldest;
  }
}
```

Когда `StatsCalculator` проверяется, поле «members» также проверяется. Предустановленный экстрактор значений для `List` используется для извлечения значений `List` для валидации элементов в `List`. В случае списка на основе сотрудника, каждый элемент "Employee" валидируется. Например, «phone» сотрудника проверяется с использованием ограничения `@USPhoneNumber`.

В следующем примере рассмотрим `StatisticsPrinter`, который печатает и отображает статистику на экране.

JAVA

```
public class StatisticsPrinter {
  private StatsCalculator<@Valid Employee> calculator;

  public StatisticsPrinter(StatsCalculator<Employee> statsCalculator){
    this.calculator = statsCalculator;
  }

  public void displayStatistics(){
    //Использование StatsCalculator, получение статистики, форматирование и отображение.
  }

  public void printStatistics(){
    //Использование StatsCalculator, получение статистики, форматирование и отображение.
  }
}
```

Контейнер `StatisticsPrinter` использует `StatisticsCalculator`. Когда `StatisticsPrinter` валидируется, `StatisticsCalculator` также валидируется с помощью каскадной валидации, такой как аннотация `@Valid`. Однако для получения значений типа контейнера `StatsCalculator` требуется экстрактор значений. Реализация `ValueExtractor` для `StatsCalculator` выглядит следующим образом:

JAVA

```
public class ExtractorForStatsCalculator implements ValueExtractor<StatsCalculator<@ExtractedValue ?>> {

  @Override
  public void extractValues(StatsCalculator<@ExtractedValue ?> statsCalculator,
      ValueReceiver valueReceiver) {
      /* Получение значения.
          Если необходимо, выполнение действий с ним.*/
    valueReceiver.value("<extracted value>", statsCalculator);
  }
}
```

Существует несколько механизмов регистрации `ValueExtractor` с Jakarta Bean Validation. См. «Регистрация ValueExtractor» в спецификации Jakarta Bean Validation https://jakarta.ee/specifications/bean-validation/3.0/. Один из механизмов — зарегистрировать экстрактор значений в контексте Jakarta Bean Validation.

JAVA

```
ValidatorFactory validatorFactory = Validation
        .buildDefaultValidatorFactory();

    ValidatorContext context = validatorFactory.
        usingContext()
        .addValueExtractor(new ExtractorForStatsCalculator());


    Validator validator = context.getValidator();
```

Используя этот валидатор, `StatisticsPrinter` проверяется в следующей последовательности операций:

1. `StatisticsPrinter` валидный.
   a. Валидируются члены `StatisticsPrinter`, которым требуется каскадная валидация.
   b. Для типов контейнеров определяется экстрактор значений. В случае `StatsCalculator`, `ExtractorForStatsCalculator` найден, а затем получены значения для валидации.
   c. Валидируются `StatsCalculator` и его члены, такие как `List`.

d. Предустановленный `ValueExtractor` для `java.util.List` используется для получения значений элементов списка и их валидации. В этом случае валидируются Сотрудник и поле «phone», аннотированные ограничением `@USPhoneNumber` .

Настройка сообщений валидатора

Jakarta Bean Validation включает в себя пакет ресурсов сообщений по умолчанию для встроенных ограничений. Эти сообщения могут быть настроены и локализованы для локалей, отличных от английской.

Bundle-ресурс ValidationMessages

Bundle-ресурс `ValidationMessages` и его варианты для локалей содержат строки, которые переопределяют сообщения валидации по умолчанию. Bundle-ресурс `ValidationMessages` обычно представляет собой файл свойств `ValidationMessages.properties` в пакете приложения по умолчанию.

Локализация сообщений валидации

Варианты локалей `ValidationMessages.properties` формируются путём добавления подчеркивания и префикса локали к базовому имени файла. Например, bundle-ресурс варианта испанской локали будет `ValidationMessages_es.properties` .

Группировка ограничений

Ограничения могут быть добавлены в одну или несколько групп. Группы ограничений используются для создания подмножеств ограничений, так что для определённого объекта будут проверяться только определённые ограничения. По умолчанию все ограничения включены в группу ограничений `Default` .

Группы ограничений представлены интерфейсами.

```java
public interface Employee {}

public interface Contractor {}
```

Группы ограничений могут наследоваться от других групп.

```java
public interface Manager extends Employee {}
```

Когда к элементу добавляется ограничение, оно объявляет группы, к которым принадлежит это ограничение, путём указания имени класса для имени интерфейса группы в элементе `groups` ограничения.

```java
@NotNull(groups=Employee.class)
Phone workPhone;
```

Несколько групп можно объявить, заключив их в фигурные скобки (`{` и `}`) и разделив имена классов групп запятыми.

```java
@NotNull(groups={ Employee.class, Contractor.class })
Phone workPhone;
```

Если группа наследует от другой группы, проверка этой группы приводит к проверке всех ограничений, объявленных как часть супергруппы. Например, проверка группы `Manager` приводит к проверке поля `workPhone` , поскольку `Employee` является родительским интерфейсом для `Manager` .

Настройка порядка валидации группы

По умолчанию группы ограничений проверяются в произвольном порядке. Есть случаи, когда некоторые группы должны быть проверены раньше других. Например, в конкретном классе базовые данные должны быть проверены раньше более сложных данных.

Чтобы установить порядок проверки для группы, добавьте аннотацию `jakarta.validation.GroupSequence` к определению интерфейса, указав порядок, в котором должна происходить валидация.

JAVA

```
@GroupSequence({Default.class, ExpensiveValidationGroup.class})
public interface FullValidationGroup {}
```

При проверке `FullValidationGroup` сначала проверяется группа `Default`. Если все данные успешно проходят валидацию, проверяется группа `ExpensiveValidationGroup`. Если ограничение является частью групп `Default` и `ExpensiveValidationGroup`, ограничение проверяется как часть группы `Default` и не будет проверяться на последующем проходе `ExpensiveValidationGroup`.

Использование ограничений методов в иерархиях типов

Если вы добавляете ограничения валидации к объектам в иерархии наследования, будьте особенно внимательны, чтобы избежать непреднамеренных ошибок при использовании дочерних типов.

Для данного типа дочерние типы должны иметь возможность замены без возникновения ошибок. Например, если у вас есть класс `Person` и расширяющий его дочерний класс `Employee`, объекты `Employee` могут использоваться повсюду, где используются объекты `Person`. Если `Employee` переопределяет метод в `Person` путём добавления ограничений параметров метода, код, который корректно работает с объектами `Person`, может генерировать исключения валидации с объектами `Employee`.

Следующий код показывает некорректное использование ограничений параметров метода в иерархии классов:

JAVA

```
public class Person {
...
  public void setPhone(String phone) { ... }
}

public class Employee extends Person {
...
  @Override
  public void setPhone(@Verified String phone) { ... }
}
```

При добавлении ограничения `@Verified` в `Employee.setPhone` параметры, которые были валидными для `Person.setPhone`, не будут таковыми для `Employee.setPhone`. Это называется усилением предварительных условий (или параметров метода) метода дочернего типа. Вы не можете усиливать предварительные условия вызовов методов дочернего типа.

Аналогично, возвращаемые значения из вызовов методов не должны быть ослаблены в дочерних типах. В следующем коде показано некорректное использование ограничений на возвращаемые значения метода в иерархии классов:

JAVA

```
public class Person {
...
  @Verified
  public USPhoneNumber getPhone() { ... }
}

public class Employee extends Person {
...
  @Override
  public USPhoneNumber getPhone() { ... }
}
```

В этом примере метод `Employee.getPhone` удаляет ограничение `@Verified` для возвращаемого значения. Возвращаемые значения, которые не прошли бы проверку при вызове `Person.getEmail`, разрешены при вызове `Employee.getPhone`. Это называется ослаблением постусловий (то есть возвращаемых значений) дочернего типа. Вы не можете ослабить постусловия вызова метода дочернего типа.

Если ваша иерархия типов усиливает предварительные условия или ослабляет постусловия вызовов методов дочернего типа, средой выполнения Jakarta Bean Validation будет выброшен `jakarta.validation.ConstraintDeclarationException`.

Классы, реализующие несколько интерфейсов, в каждом из которых имеется метод с совпадающей сигнатурой, должны знать об ограничениях, применяемых к реализуемым интерфейсам, чтобы избежать усиления предварительных условий. Например:

JAVA

```
public interface PaymentService {
  void processOrder(Order order, double amount);
...
}

public interface CreditCardPaymentService {
  void processOrder(@NotNull Order order, @NotNull double amount);
...
}

public class MyPaymentService implements PaymentService,
        CreditCardPaymentService {

  @Override
  public void processOrder(Order order, double amount) { ... }
...
}
```

В этом случае `MyPaymentService` имеет ограничения из метода `processOrder` в `CreditCardPaymentService`, но код клиента, который вызывает `PaymentService.processOrder` не ожидает этих ограничений. Это ещё один пример усиления предварительных условий подтипа, который приведёт к `ConstraintDeclarationException`.

Правила использования ограничений методов в иерархиях типов

Следующие правила определяют, как ограничения метода должны использоваться в иерархиях типов.

- Не добавляйте ограничения параметров метода для переопределённых или реализованных методов в дочернем типе.
- Не добавляйте ограничения параметров метода для переопределённых или реализованных методов в дочернем типе, которые были объявлены с совпадающей сигнатурой в нескольких родительских типах.

- Вы можете добавить ограничения возвращаемого значения в переопределённый или реализованный метод в дочернем типе.

Часть V: Контексты и инъецирование зависимостей Jakarta EE

Часть V исследует Инъецирование контекстов и зависимостей Jakarta EE (CDI).

Глава 25. Введение в Jakarta CDI

В этой главе описывается Jakarta Contexts and Dependency Injection (CDI), которая является одной из нескольких функций Jakarta EE, помогающих собрать воедино веб-уровень и транзакционный уровень платформы Jakarta EE.

Начало работы

Контексты и инъецирование зависимостей (CDI) позволяют вашим объектам автоматически предоставлять свои зависимости вместо того, чтобы создавать их или получать в качестве параметров. CDI также управляет жизненным циклом этих зависимостей.

Например, рассмотрим следующий сервлет:

JAVA

```
@WebServlet("/cdiservlet")
public class NewServlet extends HttpServlet {
    private Message message;

    @Override
    public void init() {
        message = new MessageB();
    }

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
                throws IOException {
        response.getWriter().write(message.get());
    }
}
```

Этот сервлет нуждается в объекте, который реализует интерфейс `Message` :

JAVA

```
public interface Message {
    public String get();
}
```

Сервлет инстанцирует себе объект следующего класса:

JAVA

```
public class MessageB implements Message {
    public MessageB() { }

    @Override
    public String get() {
        return "message B";
    }
}
```

Используя CDI, этот сервлет может объявить свою зависимость от объекта `Message` и автоматически инъецировать его во время выполнения CDI. Новый код сервлета следующий:

JAVA

```
@WebServlet("/cdiservlet")
public class NewServlet extends HttpServlet {
    @Inject private Message message;

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
                  throws IOException {
        response.getWriter().write(message.get());
    }
}
```

Среда выполнения CDI ищет классы, которые реализуют интерфейс `Message`, находит класс `MessageB`, инстанцирует его объект и инъецирует его в сервлет во время выполнения. Для управления жизненным циклом нового объекта среде выполнения CDI необходимо знать, какой должна быть область видимости объекта. В этом примере сервлету нужен только объект для обработки HTTP-запроса. Затем объект может быть собран сборщиком мусора. Это задаётся аннотацией `jakarta.enterprise.context.RequestScoped`:

JAVA

```
@RequestScoped
public class MessageB implements Message { ... }
```

Для получения дополнительной информации об областях видимости смотрите Использование областей видимости.

Класс `MessageB` является бином CDI. Бины CDI — это классы, которые CDI может создавать, управлять и автоматически инъецировать для удовлетворения зависимостей других объектов. Почти любой класс Java может управляться и инъецироваться с помощью CDI. Для получения дополнительной информации о бинах см. О бинах. Файл JAR или WAR, содержащий бин CDI, является архивом бина. Дополнительные сведения об упаковке архивов компонентов см. в разделе Настройка приложения CDI в этой главе и Упаковка приложений CDI в главе 27 *Инъецирование контекстов и зависимостей Jakarta: дополнительные темы*.

В этом примере `MessageB` — единственный класс, который реализует интерфейс `Message`. Если приложение имеет несколько реализаций интерфейса, CDI предоставляет механизмы, которые могут использоваться при выборе реализации для инъецирования. Для получения дополнительной информации см. Использование квалификаторов в этой главе и Использование альтернатив в приложениях CDI в главе 27 *Jakarta Contexts and Dependency Injection: дополнительные темы*.

Обзор CDI

CDI — это набор сервисов, которые, используемые вместе, облегчают разработчикам использование Enterprise-бинов вместе с Jakarta Faces в веб-приложениях. Разработанный для использования с объектами с сохранением состояния, CDI также имеет множество более широких применений, предоставляя разработчикам большую гибкость для интеграции различных видов компонентов слабосвязным, но типобезопасным способом.

CDI 3.0 указан в спецификации Jakarta EE. Связанные спецификации, которые использует CDI:

- Инъекция зависимостей Jakarta
- Спецификация Managed Beans — ответвление спецификации платформы Jakarta EE

CDI предоставляет следующие основные сервисы.

- Контексты. Этот сервис позволяет связать жизненный цикл и взаимодействия компонентов, сохраняющих состояние, с чётко определёнными, но расширяемыми контекстами жизненного цикла.

- Инъецирование зависимостей. Этот сервис позволяет инъецировать компоненты в приложение типобезопасным (typesafe) способом и выбирать во время развёртывания, какую реализацию конкретного интерфейса инъецировать.

Помимо этих, CDI предоставляет следующие сервисы:

- Интеграция с языком выражений (EL), который позволяет использовать любой компонент непосредственно на странице Jakarta Faces или Jakarta Server Pages.
- Способность декорировать инъецированные компоненты
- Способность типобезопасно (typesafe) связывать Interceptor-ы с компонентами
- Модель события-уведомления
- Область видимости веб-диалога (web conversation) в дополнение к трём стандартным областям видимости (запроса, сессии и приложения), определённым в спецификации Jakarta Servlet
- Полный интерфейс Service Provider Interface (SPI), который позволяет сторонним фреймворкам интегрироваться со средой Jakarta EE

Основная тема CDI — слабосвязность. CDI делает следующее:

- Разъединяет сервер и клиент с помощью чётко определённых типов и квалификаторов, так что реализация сервера может отличаться
- Разъединяет жизненные циклы взаимодействующих компонентов
  - Создание компонентов контекстно, с автоматическим управлением жизненным циклом
  - Предоставление компонентам, сохраняющим состояние, возможности взаимодействовать как сервисы исключительно путём передачи сообщений
- Полностью отделяет производителей сообщений от потребителей с помощью событий
- Разделяет ортогональные операции с помощью Interceptor-ов Jakarta EE

Наряду со слабосвязностью, CDI обеспечивает сильную типизацию

- Устранение поиска по строковым именам, что даёт возможность компилятору обнаружить ошибки ввода
- Позволяет использовать декларативные аннотации Java для определения чего угодно, в значительной степени устраняя необходимость в дескрипторах развёртывания XML и упрощая использование инструментов, анализирующих код и разбирающих структуру зависимостей в процессе разработки

О бинах

CDI переопределяет понятие компонента, выходя за рамки его использования другими технологиями Java, таких как JavaBeans и Jakarta Enterprise Beans. В CDI бин является источником контекстных объектов, которые определяют состояние или логику приложения. Компонент Jakarta EE является бином, если контейнер может управлять жизненным циклом его объектов в соответствии с контекстной моделью жизненного цикла, определённой в спецификации CDI.

Более конкретно, бин имеет следующие атрибуты:

- (Непустой) набор типов компонентов
- (Непустой) набор квалификаторов (см. Использование квалификаторов)
- Область видимости (см. Использование областей видимости)

- Необязательно, имя EL бина (см. Присвоение бинам имён EL)
- Набор привязок Interceptor-ов
- Реализация бина

Тип бина будет виден клиентам. В качестве типа для бина может быть выбран почти любой тип Java.

- Тип бина может быть интерфейсом, конкретным или абстрактным классом и может быть объявлен как final или иметь final-методы.
- Тип бина может быть параметризованным типом с типизированными параметрами и переменными.
- Бин может быть массивом. Два бина-массива считаются идентичными тогда и только тогда, когда типы их элементов идентичны.
- Бин может быть примитивного типа. Считается, что примитивные типы идентичны соответствующим им типам-обёрткам из `java.lang`.
- Тип бинов может быть rawtype.

О Managed-бинах CDI

Managed-бин реализуется Java-классом. Java-класс верхнего уровня является Managed-бином, если он определён как Managed-бин любой другой спецификацией Jakarta EE, например, спецификацией Jakarta Faces, или если он соответствует всем следующим условиям.

- Не является нестатическим внутренним классом.
- Это конкретный класс или аннотированный `@Decorator`.
- Он не аннотируется как EJB-компонент и не объявлен как класс EJB-компонента в `ejb-jar.xml`.
- У него есть подходящий конструктор. То есть один из следующих случаев.
  - У класса есть конструктор без параметров.
  - Класс объявляет конструктор с аннотацией `@Inject`.

Никакого специального объявления, такого как аннотация, не требуется для определения Managed-бина.

Бины как инъецируемые объекты

Концепция инъецирования некоторое время была частью технологии Java. С момента появления платформы Java EE 5 аннотации позволили инъецировать ресурсы и некоторые другие типы объектов в Managed-бины. CDI позволяет инъецировать больше видов объектов и инъецировать их в объекты, которые не управляются контейнером (в объекты, не являющиеся Managed-бинами).

Следующие виды объектов могут быть инъецированы:

- Почти любой класс Java
- Сессионные бины
- Ресурсы Jakarta EE: источники данных, темы сообщений, очереди, фабрики подключений и т.п.
- Контексты персистентности (объекты `EntityManager` Jakarta Persistence)
- Поля-производители
- Объекты, возвращаемые методами-производителями
- Ссылки на веб-сервисы

- Ссылки на удалённые (remote) Enterprise-бины

Для примера предположим, что создаётся простой класс Java с методом, возвращающим строку:

JAVA

```
package greetings;

public class Greeting {
    public String greet(String name) {
        return "Hello, " + name + ".";
    }
}
```

Этот класс становится бином, который затем можно инъецировать в другой класс. В таком виде этот бин недоступен для EL. Присвоение бинам имён EL объясняет, как сделать бин доступным для EL.

Использование квалификаторов

Вы можете использовать квалификаторы для предоставления различных реализаций определённого типа бина. Квалификатор — это аннотация, которая применяется к бину. Тип квалификатора — это аннотация Java, определённая как `@Target({METHOD, FIELD, PARAMETER, TYPE})` и `@Retention(RUNTIME)`.

Например, вы можете объявить тип квалификатора `@Informal` и применить его к другому классу, который расширяет класс `Greeting`. Чтобы объявить этот тип квалификатора, используйте следующий код:

JAVA

```
package greetings;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Target;

import jakarta.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Informal {}
```

Затем вы можете определить класс компонента, который расширяет класс `Greeting` и использует этот квалификатор:

JAVA

```
package greetings;

@Informal
public class InformalGreeting extends Greeting {
    public String greet(String name) {
        return "Hi, " + name + "!";
    }
}
```

Обе реализации компонента могут теперь использоваться в приложении.

Если вы определяете бин без квалификатора, то он автоматически получает квалификатор `@Default`. Объявление класса `Greeting` без аннотации равносильно следующему объявлению:

JAVA

```
package greetings;

import jakarta.enterprise.inject.Default;

@Default
public class Greeting {
    public String greet(String name) {
        return "Hello, " + name + ".";
    }
}
```

Инъецирование бинов

Чтобы использовать созданные компоненты, инъецируйте их в ещё один компонент, который затем может использоваться приложением, таким как приложение Jakarta Faces. Например, можно создать бин с именем `Printer`, в который будет инъецироваться один из бинов `Greeting`:

JAVA

```
import jakarta.inject.Inject;

public class Printer {

    @Inject Greeting greeting;
    ...
}
```

Этот код инъецирует реализацию `@Default Greeting` в компонент. Следующий код инъецирует реализацию `@Informal`:

JAVA

```
import jakarta.inject.Inject;

public class Printer {

    @Inject @Informal Greeting greeting;
    ...
}
```

Однако этой информации недостаточно для описания полной картины по бину. Требуется чётко понимать область видимости, которую он должен иметь. Кроме того, приложению Jakarta Faces может потребоваться, чтобы компонент был доступен через EL.

Теперь, когда вы можете определить цель инъецирования, важно понять, что можно инъецировать и в каком контексте. Начиная с версии 2.3 Faces предоставляет производителей, которые позволяют инъецировать наиболее важные артефакты Faces. Для получения подробной информации см. javadoc (https://jakarta.ee/specifications/faces/3.0/apidocs/) для пакета `jakarta.faces.annotation`.

Использование областей видимости

Чтобы веб-приложение использовало компонент, который инъецирует другой компонент, компонент должен иметь возможность сохранять состояние в течение всего времени взаимодействия пользователя с приложением. Способ определить это состояние состоит в том, чтобы назначить бину область видимости. Объекту можно назначить любую из областей видимости, описанных в табл. 25-1, в зависимости от того, как он используется.

Таблица 25-1 Области видимости

Область видимости	Аннотация	Продолжительность
Запрос	`@RequestScoped`	Взаимодействие пользователя с веб-приложением в одном HTTP-запросе.
Сессия	`@SessionScoped`	Взаимодействие пользователя с веб-приложением в течение нескольких HTTP-запросов.
Приложение	`@ApplicationScoped`	Общедоступные данные для всех пользователей с веб-приложением.
Зависимый	`@Dependent`	Область видимости по умолчанию, если не указана явно. Это означает, что объект существует для обслуживания ровно одного клиента (бина) и имеет тот же жизненный цикл, что и этот клиент (бин).
Диалог	`@ConversationScoped`	Взаимодействие пользователя с сервлетом, включая приложения Jakarta Faces. Область видимости диалога существует в границах, контролируемых разработчиком, которые распространяются на несколько запросов для длительных диалогов. Все длительные диалоги ограничиваются определённой сессией сервлета HTTP и не могут выходить за пределы этой сессии.

Первые три области видимости определяются как Jakarta CDI, так и Jakarta Faces. Последние два определяются Jakarta CDI.

Все предопределённые области видимости, кроме `@Dependent` , являются контекстными областями. CDI помещает компоненты в соответствующий их области видимости контекст, жизненный цикл которого определяется спецификациями Jakarta EE. Например, контекст сессии и её компоненты доступны только во время жизни HTTP-сессии. Ссылки на бины инъецируются с учётом контекста. Ссылки всегда применяются к компоненту, связанному с контекстом для потока, который создаёт ссылку. Контейнер CDI гарантирует, что объекты создаются и инъецируются в правильное время, определяемое областью, указанной для этих объектов.

Вы также можете определять и реализовывать кастомные области видимости, но это отдельная и сложная тема. Кастомные области видимости могут использоваться теми, кто реализует и расширяет спецификацию CDI.

Область видимости даёт объекту чётко определённый контекст жизненного цикла. Объект области видимости создаётся автоматически, когда он необходим, и автоматически уничтожается, когда заканчивается создавший его контекст. Более того, его состояние автоматически передаётся любым клиентам, которые выполняются в том же контексте.

Компоненты Jakarta EE, такие как сервлеты и EJB-компоненты, а также компоненты JavaBeans по определению не имеют чётко определённой области видимости. Эти компоненты являются одним из следующих:

- Enterprise-бины-синглтоны имеют одно состояние для всех клиентов
- Объекты без состояния, такие как сервлеты и сессионные компоненты без состояния, не содержат доступного клиенту состояния
- Объекты, которые должны быть явно созданы и уничтожены их клиентом, такие как компоненты JavaBeans и сессионные компоненты с состоянием, состояние которых совместно используется явной передачей ссылок между клиентами

Однако, если создаётся компонент Jakarta EE, являющийся Managed-бином, он становится объектом области видимости, существующим в чётко определённом контексте жизненного цикла.

В веб-приложении для бина `Printer` будет использоваться простой механизм запросов и ответов, поэтому Managed-бин можно аннотировать следующим образом:

JAVA

```
import jakarta.enterprise.context.RequestScoped;
import jakarta.inject.Inject;

@RequestScoped
public class Printer {

    @Inject @Informal Greeting greeting;
    ...
}
```

Бины, использующие область видимости сессии, приложения или диалога, должны быть сериализуемыми. К бинам области видимости запроса такого требования нет.

Присвоение бинам имён EL

Чтобы сделать компонент доступным через EL, используйте предустановленный квалификатор `@Named`:

JAVA

```
import jakarta.enterprise.context.RequestScoped;
import jakarta.inject.Inject;
import jakarta.inject.Named;

@Named
@RequestScoped
public class Printer {

    @Inject @Informal Greeting greeting;
    ...
}
```

Квалификатор `@Named` позволяет получить доступ к компоненту, используя имя компонента, с первой буквой в нижнем регистре. Например, страница Facelets будет ссылаться на компонент как `printer`.

Вы можете указать аргумент для квалификатора `@Named`, чтобы использовать имя не по умолчанию:

JAVA

```
@Named("MyPrinter")
```

С этой аннотацией страница Facelets будет ссылаться на компонент как `MyPrinter`.

Добавление set- и get- методов

Чтобы сделать состояние Managed-бина доступным, добавьте set- и get- методы для этого состояния. Метод `createSalutation` вызывает метод бина `greet`, а метод `getSalutation` получает результат.

После добавления set- и get- методов бин можно считать завершённым. Окончательный код выглядит так:

JAVA
```java
package greetings;

import jakarta.enterprise.context.RequestScoped;
import jakarta.inject.Inject;
import jakarta.inject.Named;

@Named
@RequestScoped
public class Printer {

    @Inject @Informal Greeting greeting;

    private String name;
    private String salutation;

    public void createSalutation() {
        this.salutation = greeting.greet(name);
    }

    public String getSalutation() {
        return salutation;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Использование Managed-бина на странице Facelets

Чтобы использовать Managed-бин на странице Facelets, создайте форму, которая использует элементы пользовательского интерфейса для вызова его методов и отображения их результатов. В следующем примере представлена кнопка, которая просит пользователя ввести имя, получить приветствие и затем отобразить текст в абзаце под кнопкой:

XML
```xml
<h:form id="greetme">
   <p><h:outputLabel value="Enter your name: " for="name"/>
      <h:inputText id="name" value="#{printer.name}"/></p>
   <p><h:commandButton value="Say Hello"
                       action="#{printer.createSalutation}"/></p>
   <p><h:outputText value="#{printer.salutation}"/></p>
</h:form>
```

Инъецирование объектов с использованием методов-производителей

Методы-производители предоставляют способ инъецирования объектов, не являющихся бинами, объектов, значения которых могут изменяться во время выполнения, и объектов, требующими пользовательской инициализации. Например, при желании инициализировать числовое значение, определённое квалификатором с именем `@MaxNumber` вы можете определить значение в Managed-бине и затем определить метод-производитель `getMaxNumber` для этого:

JAVA

```
private int maxNumber = 100;
...
@Produces @MaxNumber int getMaxNumber() {
    return maxNumber;
}
```

Когда вы инъецируете объект в другой Managed-бин, контейнер автоматически вызывает метод производителя, инициализируя переменную значением 100:

JAVA

```
@Inject @MaxNumber private int maxNumber;
```

Если значение может меняться во время выполнения, то процесс немного отличается. Например, следующий код определяет метод-производитель, который генерирует случайное число, определённое квалификатором с именем `@Random`:

JAVA

```
private java.util.Random random =
    new java.util.Random( System.currentTimeMillis() );

java.util.Random getRandom() {
        return random;
}

@Produces @Random int next() {
    return getRandom().nextInt(maxNumber);
}
```

Инъецируя этот объект в другой Managed-бин, вы тем самым объявляете контекстное инстанцирование объекта:

JAVA

```
@Inject @Random Instance<Integer> randomInt;
```

Затем вы вызываете `get`-метод для `Instance`:

JAVA

```
this.number = randomInt.get();
```

Настройка приложения CDI

Когда бины аннотированы типом области видимости, сервер распознаёт приложение как архив бинов без всякой дополнительной настройки. Возможные типы областей видимости для компонентов CDI перечислены в Использование областей видимости.

CDI использует необязательный дескриптор развёртывания с именем `beans.xml`. Как и другие дескрипторы развёртывания Jakarta EE, параметры конфигурации в `beans.xml` используются в дополнение к настройкам аннотаций в классах CDI. Настройки в `beans.xml` переопределяют настройки аннотаций в случае конфликта. Архив должен содержать дескриптор развёртывания `beans.xml` только в определённых ограниченных ситуациях, описанных в главе 27 *Jakarta Contexts and Dependency Injection: дополнительные темы*.

Для веб-приложения дескриптор развёртывания `beans.xml`, если он есть, должен находиться в каталоге `WEB-INF`. Для модулей EJB или JAR-файлов дескриптор развёртывания `beans.xml`, если он присутствует, должен находиться в каталоге `META-INF`.

Использование аннотаций @PostConstruct и @PreDestroy с классами Managed-бинов CDI

Классы Managed-бинов CDI и их родительские классы поддерживают аннотации для инициализации и подготовки к уничтожению компонента. Эти аннотации определены в Jakarta Annotations (https://jakarta.ee/specifications/annotations/2.0/).

Инициализация Managed-бина с помощью аннотации @PostConstruct

Инициализация Managed-бина возможна при использовании Callback-метода жизненного цикла, который вызывается фреймворком CDI после инъецирования зависимостей, но до использования инстанцированного объекта.

1. В классе Managed-бина или любом из его родительских классов определите метод, который выполняет требуемую инициализацию.
2. Аннотируйте объявление метода аннотацией `jakarta.annotation.PostConstruct`.

Когда Managed-бин инъецируется в компонент, CDI вызывает метод после того, как все инъецирования были завершены и все инициализаторы были вызваны.

> Как указано в Jakarta Annotations, если аннотированный метод объявлен в суперклассе, метод вызывается, если подкласс объявляющего класса не переопределяет метод.

Managed-бин `UserNumberBean` в примере CDI guessnumber-cdi использует `@PostConstruct` для аннотирования метода, который сбрасывает все поля бина:

JAVA

```
@PostConstruct
public void reset () {
    this.minimum = 0;
    this.userNumber = 0;
    this.remainingGuesses = 0;
    this.maximum = maxNumber;
    this.number = randomInt.get();
}
```

Подготовка к уничтожению Managed-бина с помощью аннотации @PreDestroy

Подготовка Managed-бина к уничтожению состоит в указании Callback-метода жизненного цикла, вызов которого предшествует уничтожению компонента контейнером.

1. В классе Managed-бина или в любом из его родительских классов определите метод, предшествующий уничтожению Managed-бина.

   В этом методе выполняйте любую очистку, которая необходима для уничтожения компонента, как например, освобождение ресурсов, содержащихся в бине.
2. Аннотируйте объявление метода аннотацией `jakarta.annotation.PreDestroy`.

CDI вызывает этот метод непосредственно перед уничтожением компонента.

Дополнительная информация о CDI

Для получения дополнительной информации о CDI смотрите

- Спецификация Jakarta Contexts and Dependency Injection: https://jakarta.ee/specifications/cdi/3.0/

- Weld — реализация CDI:
  https://docs.jboss.org/weld/reference/latest/en-US/html/
- Спецификация Jakarta Dependency Injection:
  https://jakarta.ee/specifications/dependency-injection/2.0/
- Спецификация Jakarta Managed beans, которая является частью спецификации Jakarta EE Platform:
  https://jakarta.ee/specifications/managedbeans/2.0/

Глава 26. Базовые примеры инъецирования контекстов и зависимостей

В этой главе подробно описывается, как создавать и запускать простые примеры, использующие CDI.

Сборка и запуск примеров CDI

Примеры находятся в каталоге *tut-install*`/examples/cdi/`.

Для сборки и запуска примеров нужно сделать следующее:

1. Используйте IDE NetBeans или Maven, чтобы скомпилировать и упаковать пример.
2. Используйте IDE NetBeans или Maven для развёртывания примера.
3. Запустите пример в веб-браузере.

См. главу 2 *Использование примеров учебника*для получения базовой информации по установке, сборке и запуску примеров.

Пример simplegreeting на CDI

Пример `simplegreeting` иллюстрирует некоторые из основных функций CDI: области видимости, квалификаторы, инъецирование бина и доступ к Managed-бину в приложении Jakarta Faces. При запуске примера кликните кнопку, которая представляет формальное или неформальное приветствие, в зависимости от того, как вы редактировали один из классов. Пример включает четыре исходных файла, страницу и шаблон Facelets и файлы конфигурации.

Исходный код simplegreeting

Четыре исходных файла для примера `simplegreeting`:

- Класс `Greeting` по умолчанию, показанный в Бины как инъецируемые объекты
- Определение интерфейса квалификатора `@Informal` и реализующего его класса `InformalGreeting`, показанных в Использование квалификаторов
- Класс Managed-бина `Printer`, который инъецирует один из двух интерфейсов, показанных в Добавление set- и get- методов

Исходные файлы находятся в каталоге *tut-install*`/examples/cdi/simplegreeting/src/main/java/ee/jakarta/tutorial/simplegreeting`.

Шаблон и страница Facelets

Чтобы использовать Managed-бин в простом приложении Facelets:

1. Используйте очень простой файл шаблона и страницу `index.xhtml`.

   Страница шаблона `template.xhtml` выглядит следующим образом:

XML

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
    <h:head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
        <h:outputStylesheet library="css" name="default.css"/>
        <title><ui:insert name="title">Default Title</ui:insert></title>
    </h:head>

    <body>
        <div id="container">
            <div id="header">
                <h2><ui:insert name="head">Head</ui:insert></h2>
            </div>

            <div id="space">
                <p></p>
            </div>

            <div id="content">
                <ui:insert name="content"/>
            </div>
        </div>
    </body>
</html>
```

2. Чтобы создать страницу Facelets, переопределите title и head, затем добавьте небольшую форму к содержимому:

XML

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <ui:composition template="/template.xhtml">

        <ui:define name="title">Simple Greeting</ui:define>
        <ui:define name="head">Simple Greeting</ui:define>
        <ui:define name="content">
            <h:form id="greetme">
               <p><h:outputLabel value="Enter your name: " for="name"/>
                  <h:inputText id="name" value="#{printer.name}"/></p>
               <p><h:commandButton value="Say Hello"
                                   action="#{printer.createSalutation}"/></p>
               <p><h:outputText value="#{printer.salutation}"/> </p>
            </h:form>
        </ui:define>

    </ui:composition>
</html>
```

Форма просит пользователя ввести имя. Кнопка помечена как **Say Hello**, и определённое для неё действие — вызвать метод `createSalutation` Managed-бина `Printer`. Этот метод, в свою очередь, вызывает метод `greet` определённого класса `Greeting`.

Выходной текст для формы — это значение приветствия, возвращаемое set-методом. В зависимости от того, инъецирована ли версия приветствия по умолчанию или `@Informal`, это одно из следующих значений, где `name` — имя, введённое пользователем:

```
Hello, name.

Hi, name!
```

Страница и шаблон Facelets находятся в каталоге *`tut-install`*`/examples/cdi/simplegreeting/src/main/webapp/`.

Простой файл CSS, используемый страницей Facelets, находится в следующем месте:

```
tut-install/examples/cdi/simplegreeting/src/main/webapp/resources/css/default.css
```

Запуск примера simplegreeting

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска приложения `simplegreeting`.

Сборка, упаковка и запуск simplegreeting с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/cdi
   ```

4. Выберите каталог `simplegreeting`.
5. Нажмите **Открыть проект**.
6. Чтобы изменить файл `Printer.java`, выполните следующие действия:
   a. Разверните узел **Исходные пакеты**.
   b. Разверните узел `greetings`.
   c. Выполните двойной клик на файле `Printer.java`.
   d. В редакторе закомментируйте аннотацию `@Informal`:

   JAVA
   ```
   @Inject
   //@Informal
   Greeting greeting;
   ```

   e. Сохраните файл.
7. На вкладке **Проекты** кликните правой кнопкой мыши проект `simplegreeting` и выберите Сборка.

   Эта команда собирает и упаковывает приложение в WAR-файл `simplegreeting.war`, расположенный в каталоге `target`, а затем развёртывает его в GlassFish Server.

Сборка, упаковка и развёртывание simplegreeting с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

```
tut-install/examples/cdi/simplegreeting/
```

3. Введите следующую команду для развёртывания приложения:

```
mvn install
```
SHELL

Эта команда собирает и упаковывает приложение в WAR-файл `simplegreeting.war`, расположенный в каталоге `target`, а затем развёртывает его в GlassFish Server.

Запуск simplegreeting

1. В веб-браузере введите следующий URL:

```
http://localhost:8080/simplegreeting
```

Откроется страница **Простое приветствие**.

2. Введите имя в поле.

Для примера предположим, что вы вводите `Duke`.

3. Нажмите **Say Hello**.

Если вы не изменили файл `Printer.java`, то под кнопкой появится следующая текстовая строка:

```
Hi, Duke!
```

Если вы закомментировали аннотацию `@Informal` в файле `Printer.java`, то под кнопкой появится следующая текстовая строка:

```
Hello, Duke.
```

Пример CDI gunessnumber-cdi

Пример `guessnumber-cdi` сложнее примера `simplegreeting` и иллюстрирует использование методов-производителей, а также областей видимости сессии и приложения. Примером является игра, в которой вы пытаетесь угадать число менее чем за десять попыток. Он похож на пример `guessnumber-jsf`, описанный в главе 8 *Введение в Facelets*, с тем исключением, что вы можете продолжать угадывать, пока не получите правильный ответ или пока не используете свои десять попыток.

Пример включает четыре исходных файла, страницу и шаблон Facelets и файлы конфигурации. Файлы конфигурации и шаблон такие же, как и в примере `simplegreeting`.

Исходный код guessnumber-cdi

Четыре исходных файла для примера `guessnumber-cdi`:

- Интерфейс квалификатора `@MaxNumber`
- Интерфейс квалификатора `@Random`
- Managed-бин `Generator`, который определяет методы-производители
- Managed-бин `UserNumberBean`

Исходные файлы находятся в каталоге *`tut-install`*`/examples/cdi/guessnumber-cdi/src/main/java/ee/jakarta/tutorial/guessnumber`.

Интерфейсы квалификаторов @MaxNumber и @Random

Интерфейс квалификатора `@MaxNumber` определяется следующим образом:

JAVA

```
package guessnumber;

import java.lang.annotation.Documented;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import java.lang.annotation.Retention;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Target;
import jakarta.inject.Qualifier;

@Target({TYPE, METHOD, PARAMETER, FIELD})
@Retention(RUNTIME)
@Documented
@Qualifier
public @interface MaxNumber {
}
```

Интерфейс квалификатора `@Random` определяется следующим образом:

JAVA

```
package guessnumber;

import java.lang.annotation.Documented;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import java.lang.annotation.Retention;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Target;
import jakarta.inject.Qualifier;

@Target({TYPE, METHOD, PARAMETER, FIELD})
@Retention(RUNTIME)
@Documented
@Qualifier
public @interface Random {
}
```

Managed-бин Generator

Managed-бин `Generator` содержит два метода-производителя для приложения. Бин имеет аннотацию `@ApplicationScoped`, чтобы указать, что его контекст распространяется на время взаимодействия пользователя с приложением:

JAVA

```java
package guessnumber;

import java.io.Serializable;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.enterprise.inject.Produces;

@ApplicationScoped
public class Generator implements Serializable {

    private static final long serialVersionUID = -7213673465118041882L;

    private final java.util.Random random =
        new java.util.Random( System.currentTimeMillis() );

    private final int maxNumber = 100;

    java.util.Random getRandom() {
        return random;
    }

    @Produces @Random int next() {
        return getRandom().nextInt(maxNumber + 1);
    }

    @Produces @MaxNumber int getMaxNumber() {
        return maxNumber;
    }

}
```

Managed-бин UserNumberBean

Managed-бин `UserNumberBean`, Managed-бин для приложения Jakarta Faces, обеспечивает логику игры. Этот бин выполняет следующие действия:

- Реализует set- и get- методы для полей компонента
- Инъецирует объекты двух квалификаторов
- Предоставляет метод `reset`, который позволяет начать новую игру.
- Предоставляет метод `check`, который определяет, угадал ли пользователь число
- Предоставляет метод `validateNumberRange`, который определяет валидность ввода пользователя

Бин определяется следующим образом:

```java
package guessnumber;

import java.io.Serializable;
import jakarta.annotation.PostConstruct;
import jakarta.enterprise.context.SessionScoped;
import jakarta.enterprise.inject.Instance;
import jakarta.faces.application.FacesMessage;
import jakarta.faces.component.UIComponent;
import jakarta.faces.component.UIInput;
import jakarta.faces.context.FacesContext;
import jakarta.inject.Inject;
import jakarta.inject.Named;

@Named
@SessionScoped
public class UserNumberBean implements Serializable {

    private static final long serialVersionUID = -7698506329160109476L;

    private int number;
    private Integer userNumber;
    private int minimum;
    private int remainingGuesses;

    @MaxNumber
    @Inject
    private int maxNumber;

    private int maximum;

    @Random
    @Inject
    Instance<Integer> randomInt;

    public UserNumberBean() {
    }

    public int getNumber() {
        return number;
    }

    public void setUserNumber(Integer user_number) {
        userNumber = user_number;
    }

    public Integer getUserNumber() {
        return userNumber;
    }

    public int getMaximum() {
        return (this.maximum);
    }

    public void setMaximum(int maximum) {
        this.maximum = maximum;
    }

    public int getMinimum() {
        return (this.minimum);
    }

    public void setMinimum(int minimum) {
        this.minimum = minimum;
    }

    public int getRemainingGuesses() {
        return remainingGuesses;
```

```
    }

    public String check() throws InterruptedException {
        if (userNumber > number) {
            maximum = userNumber - 1;
        }
        if (userNumber < number) {
            minimum = userNumber + 1;
        }
        if (userNumber == number) {
            FacesContext.getCurrentInstance().addMessage(null,
                new FacesMessage("Correct!"));
        }
        remainingGuesses--;
        return null;
    }

    @PostConstruct
    public void reset() {
        this.minimum = 0;
        this.userNumber = 0;
        this.remainingGuesses = 10;
        this.maximum = maxNumber;
        this.number = randomInt.get();
    }

    public void validateNumberRange(FacesContext context,
                                    UIComponent toValidate,
                                    Object value) {
        int input = (Integer) value;

        if (input < minimum || input > maximum) {
            ((UIInput) toValidate).setValid(false);

            FacesMessage message = new FacesMessage("Invalid guess");
            context.addMessage(toValidate.getClientId(context), message);
        }
    }
}
```

Страница Facelets

В этом примере используется тот же шаблон, что и в примере `simplegreeting`. Файл `index.xhtml`, однако, устроен более сложно.

XML

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <ui:composition template="/template.xhtml">

        <ui:define name="title">Guess My Number</ui:define>
        <ui:define name="head">Guess My Number</ui:define>
        <ui:define name="content">
            <h:form id="GuessMain">
                <div style="color: black; font-size: 24px;">
                    <p>I'm thinking of a number from
                    <span style="color: blue">#{userNumberBean.minimum}</span>
                    to
                    <span style="color: blue">#{userNumberBean.maximum}</span>.
                    You have
                    <span style="color: blue">
                        #{userNumberBean.remainingGuesses}
                    </span>
                    guesses.</p>
                </div>
                <h:panelGrid border="0" columns="5" style="font-size: 18px;">
                    <h:outputLabel for="inputGuess">Number:</h:outputLabel>
                    <h:inputText id="inputGuess"
                                 value="#{userNumberBean.userNumber}"
                                 required="true" size="3"
disabled="#{userNumberBean.number eq userNumberBean.userNumber or userNumberBean.remainingGuesses le 0}"
                                 validator="#{userNumberBean.validateNumberRange}">
                    </h:inputText>
                    <h:commandButton id="GuessButton" value="Guess"
                                     action="#{userNumberBean.check}"
disabled="#{userNumberBean.number eq userNumberBean.userNumber or userNumberBean.remainingGuesses le 0}"/>
                    <h:commandButton id="RestartButton" value="Reset"
                                     action="#{userNumberBean.reset}"
                                     immediate="true" />
                    <h:outputText id="Higher" value="Higher!"
rendered="#{userNumberBean.number gt userNumberBean.userNumber and userNumberBean.userNumber ne 0}"
                                  style="color: #d20005"/>
                    <h:outputText id="Lower" value="Lower!"
rendered="#{userNumberBean.number lt userNumberBean.userNumber and userNumberBean.userNumber ne 0}"
                                  style="color: #d20005"/>
                </h:panelGrid>
                <div style="color: #d20005; font-size: 14px;">
                    <h:messages id="messages" globalOnly="false"/>
                </div>
            </h:form>
        </ui:define>

    </ui:composition>
</html>
```

Страница Facelets представляет пользователю минимальные и максимальные значения и количество оставшихся попыток. Взаимодействие пользователя с игрой происходит в таблице `panelGrid`, содержащей поле ввода, кнопки **Guess** и **Reset** и поле, появляющееся если предположение выше или ниже правильного числа. Каждый раз, когда пользователь кликает **Guess**, вызывается метод `userNumberBean.check` для сброса максимального или минимального значения или, если предположение верно, для генерации `FacesMessage`. Метод `userNumberBean.validateNumberRange` определяет валидность каждого предположения.

Выполнение примера guessnumber-cdi

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска приложения `guessnumber-cdi`.

Сборка, упаковка и развёртывание guessnumber-cdi с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне **Открыть проект** перейдите к:

```
tut-install/examples/cdi
```
JAVA

4. Выберите каталог `guessnumber-cdi`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `guessnumber-cdi` и выберите Сборка.

   Эта команда собирает и упаковывает приложение в WAR-файл `guessnumber-cdi.war`, расположенный в каталоге `target`, а затем развёртывает его в GlassFish Server.

Сборка, упаковка и развёртывание guessnumber-cdi с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в следующий каталог:

```
tut-install/examples/cdi/guessnumber-cdi/
```
JAVA

3. Введите следующую команду для развёртывания приложения:

```
mvn install
```
JAVA

   Эта команда собирает и упаковывает приложение в WAR-файл `guessnumber-cdi.war`, расположенный в каталоге `target`, а затем развёртывает его в GlassFish Server.

Запуск guessnumber

1. В веб-браузере введите следующий URL:

```
http://localhost:8080/guessnumber-cdi
```

   Откроется страница **Guess My Number**.
2. На странице **Guess My Number** введите число в поле **Number** и нажмите **Guess**.

   Минимальные и максимальные значения изменяются вместе с оставшимся количеством попыток.
3. Продолжайте угадывать числа, пока не получите правильный ответ или не исчерпаете все попытки.

   Если вы дали правильный ответ или у вас закончились попытки, поле ввода и кнопка **Guess** отображаются серым цветом.
4. Нажмите **Reset**, чтобы снова сыграть в игру с новым случайным числом.

Глава 27. Jakarta CDI: дополнительные темы

В этой главе возможности инъецирования контекстов и зависимостей Jakarta описаны более подробно. В частности, он охватывает дополнительные функции, предоставляемые CDI для обеспечения слабой связи компонентов со строгой типизацией, в дополнение к описанным в Обзоре CDI.

Упаковка CDI-приложений

При развёртывании приложения Jakarta EE CDI ищет компоненты внутри архивов бинов. Архив бинов — это любой модуль, содержащий бины, которые могут управляться и инъецироваться средой выполнения CDI. Существует два вида архивов бинов: явные и неявные.

Явный архив компонента — это архив, содержащий `beans.xml`, который может быть пустым файлом, не содержать номера версии или содержать номер версии 3.0 с атрибутом `bean-discovery-mode`, установленным в `all`. Например:

XML

```
<beans xmlns="https://jakarta.ee/xml/ns/jakartaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
                           https://jakarta.ee/xml/ns/jakartaee/beans_3_0.xsd"
       version="3.0" bean-discovery-mode="all">
    ...
</beans>
```

Любые бины явного архива могут управляться и инъецироваться CDI, кроме отмеченных `@Vetoed`.

Неявный архив бинов — это архив, содержащий некоторые бины, аннотированные типом области видимости, не содержащий дескриптор развёртывания `beans.xml` или содержащий дескриптор развёртывания `beans.xml` с атрибутом `bean-discovery-mode`, установленным в `annotated`.

В неявном архиве только бины, аннотированные типом области видимости, могут управляться и инъецироваться CDI.

Для веб-приложения дескриптор развёртывания `beans.xml`, если он есть, должен находиться в каталоге `WEB-INF`. Для модулей EJB-компонентов или файлов JAR дескриптор развёртывания `beans.xml`, если он присутствует, должен находиться в каталоге `META-INF`.

Использование альтернатив в приложениях CDI

Если у вас есть более одной версии бина, который вы используете для разных целей, то можете выбирать между ними на этапе разработки, вводя тот или иной квалификатор, как показано в Примере simplegreeting на CDI.

Однако вместо изменения исходного кода вашего приложения вы можете сделать выбор во время развёртывания, используя альтернативные варианты.

Альтернативы обычно используются для следующих целей:

- Для обработки клиентской бизнес-логики, которая определяется во время выполнения
- Чтобы указать компоненты, валидные для конкретного сценария развёртывания (например, когда законодательство о налоге с продаж для конкретной страны требует специальной бизнес-логики)

- Чтобы создать фиктивные версии бинов для тестирования.

Чтобы сделать компонент доступным для поиска, инъецирования или вычисления механизмом EL, аннотируйте его `jakarta.enterprise.inject.Alternative`, а затем используйте элемент `alternatives`, чтобы указать его в файле `beans.xml`.

Например, вы можете создать полную версию компонента, а также более простую версию, которую будете использовать только для определённых видов тестирования. Пример, описанный в Пример encoder: использование альтернатив, содержит два таких бина: `CoderImpl` и `TestCoderImpl`. Тестовый компонент помечается следующим образом:

```java
@Alternative
public class TestCoderImpl implements Coder { ... }
```

Полная версия не аннотирована:

```java
public class CoderImpl implements Coder { ... }
```

Managed-бин инъецирует объект интерфейса `Coder`:

```java
@Inject
Coder coder;
```

Альтернативная версия компонента используется приложением, только если эта версия объявлена в файле `beans.xml` следующим образом:

```xml
<beans ...>
    <alternatives>
        <class>ee.jakarta.tutorial.encoder.TestCoderImpl</class>
    </alternatives>
</beans>
```

Если элемент `alternatives` закомментирован в файле `beans.xml`, используется класс `CoderImpl`.

Вы также можете иметь несколько бинов, аннотированных `@Alternative` и реализующих один и тот же интерфейс. В этом случае нужно указать в файле `beans.xml`, какой из этих альтернативных компонентов должен использоваться. Если бы `CoderImpl` также был аннотирован `@Alternative`, один из двух бинов всегда должен быть указан в файле `beans.xml`.

Альтернативы, указанные в файле `beans.xml`, применяются только к классам в одном и том же архиве. Используйте аннотацию `@Priority`, чтобы глобально указать альтернативы для приложения, состоящего из нескольких модулей, как в следующем примере:

```java
@Alternative
@Priority(Interceptor.Priority.APPLICATION+10)
public class TestCoderImpl implements Coder { ... }
```

Если существует несколько альтернативных бинов, реализующих один и тот же интерфейс, помеченных `@Priority`, то выбирается альтернатива с более высоким значением приоритета. Когда используется аннотация `@Priority`, указывать альтернативу в файле `beans.xml` не обязательно.

Использование специализации

Специализация выполняет ту же функцию, что и альтернатива, то есть позволяет заменять один компонент на другой. Вам может потребоваться сделать так, чтобы один бин переопределял другой во всех случаях. Предположим, вы определили следующие два компонента:

JAVA

```
@Default @Asynchronous
public class AsynchronousService implements Service { ... }

@Alternative
public class MockAsynchronousService extends AsynchronousService { ... }
```

Если затем вы объявили `MockAsynchronousService` в качестве альтернативы в вашем файле `beans.xml`, следующая точка инъецирования будет преобразована в `MockAsynchronousService`:

JAVA

```
@Inject Service service;
```

Следующее, однако, разрешит `AsynchronousService`, а не `MockAsynchronousService`, потому что `MockAsynchronousService` не имеет квалификатора `@Asynchronous`:

JAVA

```
@Inject @Asynchronous Service service;
```

Чтобы убедиться, что `MockAsynchronousService` всегда инъецировался, необходимо реализовать все типы бинов и квалификаторы бина `AsynchronousService`. Однако, если `AsynchronousService` объявил метод-производитель или метод-наблюдатель, даже этот громоздкий механизм не гарантирует, что другой компонент никогда не будет вызван. Специализация обеспечивает более простой механизм.

Специализация происходит как во время разработки, так и во время выполнения. Если вы объявляете, что один компонент специализирует другой, он расширяет класс другого компонента, и во время выполнения специализированный компонент полностью заменяет этот другой компонент. Если первый компонент создаётся методом-производителем, вы также должны переопределить метод-производитель.

Вы специализируете бин, аннотируя его `jakarta.enterprise.inject.Specializes`. Например, вы можете объявить бин следующим образом:

JAVA

```
@Specializes
public class MockAsynchronousService extends AsynchronousService { ... }
```

В этом случае класс `MockAsynchronousService` всегда будет вызываться вместо класса `AsynchronousService`.

Обычно бин, помеченный аннотацией `@Specializes`, также является альтернативой и объявляется как альтернатива в файле `beans.xml`. Такой компонент предназначен для замены реализации по умолчанию, а альтернативная реализация автоматически наследует все квалификаторы реализации по умолчанию, а также его имя EL, если оно есть.

Использование методов-производителей, полей-производителей и методов закрытия в приложениях CDI

Метод-производитель генерирует объект, который затем может быть инъецирован. Как правило, методы-производители используются в следующих ситуациях:

- Когда вы хотите инъецировать объект, который не является бином

- Когда конкретный тип объекта для инъецирования может изменяться во время выполнения
- Когда объект требует некоторой пользовательской инициализации, которую не выполняет конструктор бина

Для получения дополнительной информации о методах-производителях см. Инъецирование объектов с использованием методов-производителей.

Поле-производитель является более простой альтернативой методу-производителю. Это поле бина, которое генерирует объект. Его можно использовать вместо простого get-метода. Поля производителя особенно удобны для объявления ресурсов Jakarta EE, таких как источники данных, ресурсы JMS и ссылки на веб-сервисы.

Метод или поле производителя аннотируются `jakarta.enterprise.inject.Produces`.

Использование методов-производителей

Метод-производитель позволяет выбрать реализацию компонента во время выполнения, а не во время разработки или развёртывания. Например, в примере, описанном в Пример producermethods: использование метода producer-а для выбора реализации компонента, Managed-бин определяет следующий метод источника:

JAVA

```
@Produces
@Chosen
@RequestScoped
public Coder getCoder() {

    switch (coderType) {
        case TEST:
            return new TestCoderImpl();
        case SHIFT:
            return new CoderImpl();
        default:
            return null;
    }
}
```

Здесь `getCoder` фактически становится get-методом, а когда свойство `coder` инъецируется с тем же квалификатором и аннотациями, что и метод, используется выбранная версия интерфейса.

JAVA

```
@Inject
@Chosen
@RequestScoped
Coder coder;
```

Важно указать квалификатор: он сообщает CDI, какой `Coder` инъецировать. Без этого реализация CDI не сможет выбирать между `CoderImpl`, `TestCoderImpl` и вернувшейся `getCoder` и отменит развёртывание, сообщив пользователю о неоднозначности зависимости.

Использование полей-производителей для генерации ресурсов

Обычное использование поля производителя заключается в создании объекта, такого как JDBC `DataSource` или Jakarta Persistence `EntityManager` (см. главу 40 *Введение в Jakarta Persistence* для получения дополнительной информации). Затем объект может управляться контейнером. Например, вы можете создать `@UserDatabase`, а затем объявить поле производителя для entity manager следующим образом:

JAVA

```
@Produces
@UserDatabase
@PersistenceContext
private EntityManager em;
```

Квалификатор `@UserDatabase` можно использовать, когда вы инъецируете объект в другой компонент, `RequestBean`, в другом месте приложения:

JAVA

```
    @Inject
    @UserDatabase
    EntityManager em;
    ...
```

Пример producermethods: использование полей-производителей для создания ресурсов показывает, как использовать поля-производители для генерации EntityManager-а. Вы можете использовать аналогичный механизм для инъецирования объектов `@Resource`, `@EJB` или `@WebServiceRef`.

Чтобы свести к минимуму зависимость от инъецирования ресурса, укажите поле-производитель для ресурса в одном месте приложения, а затем инъецируйте объект в нужное место приложения.

Использование метода закрытия ресурса

Вы можете использовать метод-производитель или поле-производитель, чтобы сгенерировать объект, который необходимо удалить после завершения его работы. Если это так, требуется соответствующий метод закрытия, аннотированный `@Disposes`. Например, вы можете закрыть EntityManager следующим образом:

JAVA

```
public void close(@Disposes @UserDatabase EntityManager em) {
    em.close();
}
```

Метод закрытия вызывается автоматически по окончании контекста (в данном случае в конце диалога, поскольку `RequestBean` имеет область видимости диалога), а параметр в методе `close` получает объект, созданный полем-производителем.

Использование предопределённых бинов в приложениях CDI

Jakarta EE предоставляет предопределённые компоненты, которые реализуют следующие интерфейсы.

- `jakarta.transaction.UserTransaction`: транзакция пользователя Jakarta Transactions.
- `java.security.Principal`: абстрактное понятие принципала, который представляет любую сущность, например физическое лицо, корпорацию или идентификатор входа. При каждом обращении к инъецированному принципалу он всегда представляет идентификатор текущего вызывающего субъекта. Например, принципал инъецируется в поле при инициализации. Позже, метод, использующий инъецированного принципала, вызывается для объекта, в который этот принципал был инъецирован. В этой ситуации инъецированный принципал представляет идентификатор текущего вызывающего субъекта при запуске метода.
- `jakarta.validation.Validator`: валидатор для объектов EJB. Компонент, реализующий этот интерфейс, позволяет инъецировать объект `Validator` для объекта проверки компонента по умолчанию `ValidatorFactory`.
- `jakarta.validation.ValidatorFactory`: фабричный класс для получения инициализированных объектов `Validator`. Компонент, реализующий этот интерфейс, позволяет инъецировать объект `ValidatorFactory` проверки компонента по умолчанию.

- `jakarta.servlet.http.HttpServletRequest` : HTTP-запрос от клиента. Компонент, реализующий этот интерфейс, позволяет сервлету получать все детали запроса.
- `jakarta.servlet.http.HttpSession` : HTTP-сессия между клиентом и сервером. Компонент, реализующий этот интерфейс, позволяет сервлету получать доступ к информации о сессии и помещать объекты в сессию.
- `jakarta.servlet.ServletContext` : объект контекста, который сервлеты могут использовать для взаимодействия с контейнером сервлетов.

Чтобы инъецировать предопределённый компонент, создайте точку инъецирования для получения объекта компонента аннотацией `jakarta.annotation.Resource` для ресурсов или аннотацией `jakarta.inject.Inject` для бинов CDI. Для типа компонента укажите имя класса интерфейса, который реализует компонент.

Таблица 27-1 Инъецирование предопределённых бинов

Предопределённый бин	**Ресурс или бин CDI**	**Пример инъецирования**
`UserTransaction`	Ресурс	`@Resource UserTransaction transaction;`
`Principal`	Ресурс	`@Resource Principal principal;`
`Validator`	Ресурс	`@Resource Validator validator;`
`ValidatorFactory`	Ресурс	`@Resource ValidatorFactory factory;`
`HttpServletRequest`	Бин CDI	`@Inject HttpServletRequest req;`
`HttpSession`	Бин CDI	`@Inject HttpSession session;`
`ServletContext`	Бин CDI	`@Inject ServletContext context;`

Предопределённые бины инъецируются с областью видимости `@Dependent` и предопределённым квалификатором по умолчанию `@Default` .

Для получения дополнительной информации об инъецировании ресурсов см. Инъецирование ресурсов.

В следующем фрагменте кода показано, как использовать аннотации `@Resource` и `@Inject` для инъецирования предопределённых бинов. Этот фрагмент кода инъецирует пользовательскую транзакцию и объект контекста в класс сервлета `TransactionServlet` . Пользовательская транзакция — это объект предопределённого компонента, реализующего интерфейс `jakarta.transaction.UserTransaction` . Объект контекста — это объект предопределённого компонента, реализующего интерфейс `jakarta.servlet.ServletContext` .

JAVA

```
import jakarta.annotation.Resource;
import jakarta.inject.Inject;
import jakarta.servlet.http.HttpServlet;
import jakarta.transaction.UserTransaction;
...
public class TransactionServlet extends HttpServlet {
    @Resource UserTransaction transaction;
    @Inject ServletContext context;
    ...
}
```

Использование событий в приложениях CDI

События позволяют бинам взаимодействовать без какой-либо взаимозависимости объектов во время компиляции. Один компонент может определять событие, другой компонент может вызывать событие, и ещё один компонент может обрабатывать событие. Кроме того, события могут быть запущены асинхронно. Бины могут быть в разных пакетах и даже в разных слоях приложения.

Определение событий

Событие состоит из следующего:

- Объект события
- Любое количество типов квалификаторов

Например, в примере `billpayment`, описанном в Пример billpayment: использование событий и Interceptor-ов, бин `PaymentEvent` определяет событие, используя три свойства, для которых есть set- и get- методы:

JAVA

```
public String paymentType;
public BigDecimal value;
public Date datetime;

public PaymentEvent() {
}
```

В этом примере также определяются квалификаторы, которые различают два вида `PaymentEvent`. Каждое событие также имеет квалификатор по умолчанию `@Any`.

Использование методов-наблюдателей для обработки событий

Обработчик событий использует метод-наблюдатель для получения событий.

Каждый метод-наблюдатель принимает в качестве параметра событие определённого типа, аннотированное `@Observes` и любыми квалификаторами для этого типа события. Метод-наблюдатель уведомляется о событии, если объект события соответствует типу события и если все квалификаторы события соответствуют квалификаторам события метода-наблюдателя.

Метод-наблюдатель может принимать другие параметры в дополнение к параметру события. Дополнительные параметры являются точками инъецирования и могут объявлять квалификаторы.

Обработчик события `PaymentHandler` для примера `billpayment` определяет два метода-наблюдателя, по одному для каждого типа `PaymentEvent`:

JAVA

```
public void creditPayment(@Observes @Credit PaymentEvent event) {
    ...
}

public void debitPayment(@Observes @Debit PaymentEvent event) {
    ...
}
```

Условные и транзакционные методы-наблюдатели

Методы-наблюдатели также могут быть условными или транзакционными:

- Условный метод-наблюдатель уведомляется о событии, только если объект компонента, который определяет метод-наблюдатель, уже существует в текущем контексте. Чтобы объявить условный метод-наблюдатель, укажите `notifyObserver=IF_EXISTS` в качестве аргумента для `@Observes`:

  JAVA

  ```
  @Observes(notifyObserver=IF_EXISTS)
  ```

  Чтобы получить безусловное поведение по умолчанию, можно указать `@Observes(notifyObserver=ALWAYS)`.

- Транзакционный метод-наблюдатель уведомляется о событии до или после завершения транзакции, в которой произошло событие. Можно также указать, что уведомление должно появляться только после успешного или неудачного завершения транзакции. Чтобы указать метод-наблюдатель транзакции, используйте любой из следующих аргументов для `@Observes`:

  JAVA

  ```
  @Observes(during=BEFORE_COMPLETION)

  @Observes(during=AFTER_COMPLETION)

  @Observes(during=AFTER_SUCCESS)

  @Observes(during=AFTER_FAILURE)
  ```

  Чтобы получить стандартное нетранзакционное поведение по умолчанию, укажите `@Observes(during=IN_PROGRESS)`.

  Метод-наблюдатель, который вызывается до завершения транзакции, может вызвать метод `setRollbackOnly` в объекте транзакции, чтобы вызвать откат транзакции.

Методы-наблюдатели могут генерировать исключения. Если транзакционный метод-наблюдатель выдает исключение, это исключение перехватывается контейнером. Если метод-наблюдатель не является транзакционным, исключение завершает обработку события, и никакие другие методы-наблюдатели для события не вызываются.

Порядок действий метода-наблюдателя

Прежде чем генерируется определённое уведомление наблюдателя о событии, контейнер определяет порядок вызова методов-наблюдателей для этого события. Порядок методов-наблюдателей устанавливается посредством объявления аннотации `@Priority` для параметра события метода-наблюдателя, как в следующем примере:

JAVA

```
void afterLogin(@Observes @Priority(jakarta.interceptor.Interceptor.Priority.APPLICATION) LoggedInEvent event)
{ ... }
```

Обратите внимание на следующее:

- Если параметр `@Priority` не указан, значением по умолчанию является `jakarta.interceptor.Interceptor.Priority.APPLICATION + 500`.
- Если двум или более методам-наблюдателям назначен одинаковый приоритет, порядок их вызова не определён и поэтому непредсказуем.

Генерация событий

События запуска компонентов реализуются с помощью объекта интерфейса `jakarta.enterprise.event.Event`. События могут быть сгенерированы синхронно или асинхронно.

Синхронная генерация событий

Чтобы активировать событие синхронно, вызовите метод `jakarta.enterprise.event.Event.fire`. Этот метод генерирует событие и уведомляет любые методы-наблюдатели.

В примере `billpayment` Managed-бин `PaymentBean` запускает соответствующее событие, используя информацию, полученную из пользовательского интерфейса. На самом деле существует четыре компонента событий, два для объекта события и два для выполнения содержательной части. Managed-бин инъецирует два бина события. Метод `pay` использует оператор `switch` для выбора события, которое нужно запустить, используя `new` для выполнения содержательной части.

JAVA

```
@Inject
@Credit
Event<PaymentEvent> creditEvent;

@Inject
@Debit
Event<PaymentEvent> debitEvent;

private static final int DEBIT = 1;
private static final int CREDIT = 2;
private int paymentOption = DEBIT;
...

@Logged
public String pay() {
    ...
    switch (paymentOption) {
        case DEBIT:
            PaymentEvent debitPayload = new PaymentEvent();
            // обработка платежа...
            debitEvent.fire(debitPayload);
            break;
        case CREDIT:
            PaymentEvent creditPayload = new PaymentEvent();
            // обработка платежа...
            creditEvent.fire(creditPayload);
            break;
        default:
            logger.severe("Invalid payment option!");
    }
    ...
}
```

Аргументом метода `fire` является `PaymentEvent`, который включает содержательная часть. Инициированное событие затем используется методами-наблюдателями.

Генерация асинхронных событий

Чтобы активировать событие асинхронно, вызовите метод `jakarta.enterprise.event.Event.fireAsync`. Этот метод вызывает все разрешённые асинхронные наблюдатели в одном или нескольких разных потоках.

JAVA

```
@Inject Event<LoggedInEvent> loggedInEvent;

public void login() {
    ...
    loggedInEvent.fireAsync( new LoggedInEvent(user) );
}
```

Вызов метода `fireAsync()` возвращается немедленно.

Когда события запускаются асинхронно, методы-наблюдатели уведомляются асинхронно. Следовательно, соблюдение порядка методов-наблюдателей не гарантировано, потому что вызов метода-наблюдателя и запуск асинхронных событий происходят в разных потоках.

Использование Interceptor-ов в приложениях CDI

Interceptor — это класс, используемый для вставки в вызовы методов или события жизненного цикла, которые происходят в связанном целевом классе. Interceptor выполняет такие задачи, как ведение журнала или аудит, которые отделены от бизнес-логики приложения и часто повторяются в приложении. Такие задачи часто называют сквозными задачами. Interceptor-ы позволяют указывать код этих задач в одном месте для удобства сопровождения. Когда Interceptor-ы были впервые представлены в Jakarta EE, они были характерны для Enterprise-бинов. В платформе Jakarta EE их можно использовать с управляемыми объектами Jakarta EE всех видов, включая Managed-бины.

Для получения информации об Interceptor-ах Jakarta EE см. главу 58 *Использование Interceptor-ов в Jakarta EE*.

Класс Interceptor-а часто содержит аннотированный метод `@AroundInvoke`, который определяет задачи, которые Interceptor будет выполнять при вызове перехваченных методов. CackOн также может содержать метод, аннотированный `@PostConstruct`, `@PreDestroy`, `@PrePassivate` или `@PostActivate` для указания Interceptor-ов Callback-методов жизненного цикла и метод, аннотированный `@AroundTimeout`, для указания Interceptor-ов тайм-аута EJB-компонента. Класс Interceptor-а может содержать более одного метода Interceptor-а, но он должен иметь не более одного метода каждого типа.

Наряду с Interceptor-ом приложение определяет один или несколько типов привязки Interceptor-а, которые представляют собой аннотации, которые связывают Interceptor с целевыми компонентами или методами. Например, пример `billpayment` содержит Interceptor, аннотированный `@Logged`, и Interceptor с именем `LoggedInterceptor`. Объявление типа привязки Interceptor-а выглядит как объявление квалификатора, но аннотировано `jakarta.interceptor.InterceptorBinding`:

JAVA

```
@Inherited
@InterceptorBinding
@Retention(RUNTIME)
@Target({METHOD, TYPE})
public @interface Logged {
}
```

Привязка Interceptor-а также имеет аннотацию `java.lang.annotation.Inherited`, чтобы указать, что аннотация может быть унаследована от родительских классов. Аннотация `@Inherited` также применяется к пользовательским областям видимости (не обсуждаемым в этом руководстве), но не относится к квалификаторам.

Тип привязки Interceptor-а может объявлять другие привязки Interceptor-а.

Класс Interceptor-а аннотируется привязкой Interceptor-а, а также аннотацией `@Interceptor`. Пример см. в разделе Класс Interceptor-а LoggedInterceptor.

Каждый `@AroundInvoke` принимает метод `jakarta.interceptor.InvocationContext`, возвращает аргумент `java.lang.Object`, и выбрасывает `Exception`. Он может вызывать методы `InvocationContext`. Метод `@AroundInvoke` должен вызывать метод `continue`, который вызывает метод целевого класса.

После того как Interceptor и тип привязки определены, вы можете аннотировать бины и отдельные методы с помощью типа привязки, чтобы указать, что Interceptor должен вызываться либо для всех методов компонента, либо для конкретных методов. Например, в примере `billpayment` бин `PaymentHandler` аннотируется `@Logged`, что означает, что любой вызов его бизнес-методов вызовет метод `@AroundInvoke` Interceptor-а:

JAVA

```
@Logged
@SessionScoped
public class PaymentHandler implements Serializable {...}
```

Однако в компоненте `PaymentBean` только методы `pay` и `reset` имеют аннотацию `@Logged`, поэтому Interceptor вызывается только тогда, когда эти методы вызываются:

JAVA

```
@Logged
public String pay() {...}

@Logged
public void reset() {...}
```

Чтобы Interceptor вызывался в приложении CDI, он, как альтернатива, должен быть указан в файле `beans.xml`. Например, класс `LoggedInterceptor` указан следующим образом:

XML

```
<interceptors>
    <class>ee.jakarta.tutorial.billpayment.interceptors.LoggedInterceptor</class>
</interceptors>
```

Если приложение использует более одного Interceptor-а, они вызываются в порядке, указанном в файле `beans.xml`.

Interceptor-ы, указанные в файле `beans.xml`, применяются только к классам в одном и том же архиве. Используйте аннотацию `@Priority`, чтобы глобально указать Interceptor-ы для приложения, состоящего из нескольких модулей, как в следующем примере:

JAVA

```
@Logged
@Interceptor
@Priority(Interceptor.Priority.APPLICATION)
public class LoggedInterceptor implements Serializable { ... }
```

Interceptor-ы с более низкими значениями приоритета вызываются первыми. Не обязательно указывать Interceptor в файле `beans.xml`, когда используется аннотация `@Priority`.

Использование декораторов в приложениях CDI

Декоратор — это класс Java, который аннотирован `jakarta.decorator.Decorator` и который имеет соответствующий элемент `decorators` в файле `beans.xml`.

Класс компонента-декоратора также должен иметь точку инъецирования делегата, которая аннотирована `jakarta.decorator.Delegate`. Эта точка инъецирования может быть полем, параметром конструктора или параметром метода инициализатора класса декоратора.

Декораторы внешне похожи на Interceptor-ы. Тем не менее, они дополняют задачи, выполняемые Interceptor-ами. Interceptor-ы выполняют сквозные задачи, связанные с вызовом метода и жизненными циклами компонентов, но не могут выполнять какую-либо бизнес-логику. Декораторы, с другой стороны, выполняют бизнес-логику, перехватывая бизнес-методы бинов. Это означает, что вместо повторного использования для различных типов приложений, как и Interceptor-ы, их логика специфична для конкретного приложения.

Например, вместо использования альтернативного класса `TestCoderImpl` для примера `encoder`, вы можете создать декоратор следующим образом:

JAVA

```
@Decorator
public abstract class CoderDecorator implements Coder {

    @Inject
    @Delegate
    @Any
    Coder coder;

    public String codeString(String s, int tval) {
        int len = s.length();

        return "\"" + s + "\" becomes " + "\"" + coder.codeString(s, tval)
                + "\", " + len + " characters in length";
    }
}
```

Смотрите Пример decorators: декорирование бина для примера, который использует этот декоратор.

Этот простой декоратор возвращает более подробный вывод, чем закодированная строка, возвращённая методом `CoderImpl.codeString`. Более сложный декоратор может хранить информацию в базе данных или выполнять другую бизнес-логику.

Декоратор может быть объявлен как абстрактный класс, так что ему не нужно реализовывать все бизнес-методы интерфейса.

Чтобы декоратор вызывался в приложении CDI, он должен, как и Interceptor, или альтернатива, быть указан в файле `beans.xml`. Например, класс `CoderDecorator` указан следующим образом:

XML

```
<decorators>
    <class>ee.jakarta.tutorial.decorators.CoderDecorator</class>
</decorators>
```

Если приложение использует более одного декоратора, декораторы вызываются в том порядке, в котором они указаны в файле `beans.xml`.

Если в приложении есть как Interceptor-ы, так и декораторы, Interceptor-ы вызываются первыми. По сути, это означает, что вы не можете перехватить декоратор.

Декораторы, указанные в файле `beans.xml`, применяются только к классам в одном и том же архиве. Используйте аннотацию `@Priority`, чтобы глобально указывать декораторы для приложения, состоящего из нескольких модулей, как в следующем примере:

JAVA

```
@Decorator
@Priority(Interceptor.Priority.APPLICATION)
public abstract class CoderDecorator implements Coder { ... }
```

Декораторы с более низкими значениями приоритета вызываются первыми. Не обязательно указывать декоратор в `beans.xml`, когда используется аннотация `@Priority`.

Использование стереотипов в приложениях CDI

Стереотип — это своего рода аннотация, применяемая к бину, которая включает в себя другие аннотации. Стереотипы особенно полезны в больших приложениях, в которых есть несколько компонентов, выполняющих однотипные функции. Стереотип — это своего рода аннотация, в которой указано следующее:

- Область видимости по умолчанию
- Любое количество Interceptor-ов
- По желанию, аннотация `@Named`, гарантирующая именование EL по умолчанию
- Необязательно, аннотация `@Alternative`, указывающая, что все компоненты с этим стереотипом являются альтернативами

Бин, аннотированный определённым стереотипом, всегда будет использовать указанные аннотации, так что не нужно применять одни и те же аннотации ко многим бинам.

Например, можно создать стереотип с именем `Action` используя аннотацию `jakarta.enterprise.inject.Stereotype`:

JAVA

```
@RequestScoped
@Secure
@Transactional
@Named
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action {}
```

Все аннотированные бины `@Action` будут иметь область видимости запроса, использовать именование EL по умолчанию и иметь привязки Interceptor-ов `@Transactional` и `@Secure`.

Вы также можете создать стереотип с именем `Mock`:

JAVA

```
@Alternative
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface Mock {}
```

Все бины с этой аннотацией являются альтернативами.

Можно применить несколько стереотипов к одному и тому же компоненту, поэтому компонент может быть аннотирован следующим образом:

JAVA

```
@Action
@Mock
public class MockLoginAction extends LoginAction { ... }
```

Также возможно переопределить область видимости, указанную стереотипом, просто указав другую область видимости для компонента. В следующем объявлении вместо области видимости запроса указывается область видимости сессии компонента `MockLoginAction`:

JAVA

```
@SessionScoped
@Action
@Mock
public class MockLoginAction extends LoginAction { ... }
```

CDI предоставляет предустановленный стереотип `Model`, предназначенный для использования с бинами, определяющими уровень модели в прикладной архитектуре модель-представление-контроллер. Этот стереотип указывает, что бин является `@Named` и `@RequestScoped`:

JAVA

```
@Named
@RequestScoped
@Stereotype
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface Model {}
```

Использование встроенных литералов аннотации

Следующие предустановленные аннотации определяют статический вложенный класс `Literal`, который можно использовать для удобства инстанцирования объектов аннотаций:

- `jakarta.enterprise.inject.Any`
- `jakarta.enterprise.inject.Default`
- `jakarta.enterprise.inject.New`
- `jakarta.enterprise.inject.Specializes`
- `jakarta.enterprise.inject.Vetoed`
- `jakarta.enterprise.util.Nonbinding`
- `jakarta.enterprise.context.Initialized`
- `jakarta.enterprise.context.Destroyed`
- `jakarta.enterprise.context.RequestScoped`
- `jakarta.enterprise.context.SessionScoped`
- `jakarta.enterprise.context.ApplicationScoped`
- `jakarta.enterprise.context.Dependent`
- `jakarta.enterprise.context.ConversationScoped`
- `jakarta.enterprise.inject.Alternative`
- `jakarta.enterprise.inject.Typed`

Например:

```java
Default defaultLiteral = new Default.Literal();

RequestScoped requestScopedLiteral = RequestScoped.Literal.INSTANCE;

Initialized initializedForApplicationScoped = new Initialized.Literal(ApplicationScoped.class);

Initialized initializedForRequestScoped = Initialized.Literal.of(RequestScoped.class);
```

Использование интерфейсов конфигураторов

Спецификация CDI 2.0 определяет следующие интерфейсы конфигураторов, которые используются для динамического определения и изменения объектов CDI:

Интерфейс	**Описание**

SPI`AnnotatedTypeConfigurator`	Помогает создавать и настраивать метаданные следующего типа:
	
	`AnnotatedType`
	
	`AnnotatedField`
	
	`AnnotatedConstructor`
	
	`AnnotatedMethod`
	
	`AnnotatedParameter`

Интерфейс`InjectionPointConfigurator`	Помогает настроить существующий объект `InjectionPoint`
Интерфейс`BeanAttributesConfigurator`	Помогает настроить новый объект `BeanAttributes`
Интерфейс`BeanConfigurator`	Помогает настроить новый объект `Bean`
Интерфейс`ObserverMethodConfigurator`	Помогает настроить объект `ObserverMethod`
Интерфейс`ProducerConfigurator`	Помогает настроить объект `Producer`

Глава 28. Начальная загрузка контейнера CDI в Java SE

В этой главе объясняется, как использовать API для начальной загрузки контейнера CDI в Java SE. Эта возможность позволяет запускать приложения CDI на Java SE и получать доступ к компонентам независимо от сервера приложений и API-интерфейсов Jakarta EE.

Для получения дополнительной информации о начальной загрузке контейнера CDI в Java SE см. *Справочное руководство по Weld* по ссылке https://weld.cdi-spec.org/documentation/.

Bootstrap API

API для начальной загрузки контейнера CDI в Java SE состоит из следующих объектов:

- `jakarta.enterprise.inject.se.SeContainerInitializer` class — позволяет настраивать и загружать контейнер CDI. Этот класс включает следующие ключевые методы:
  - `newInstance()` получает объект `SeContainerInitializer`, который позволяет настроить контейнер перед его начальной загрузкой.
  - `initialize()` загружает контейнер.
- Интерфейс `jakarta.enterprise.inject.se.SeContainer` — предоставляет доступ к `BeanManager` для программного поиска, как определено в свойстве `SeContainer`, описанного в https://jakarta.ee/specifications/cdi/3.0/jakarta-cdi-spec-3.0.html#se_container.

Конфигурирование контейнера CDI

Конфигурация объекта `SeContainerInitializer` позволяет явно добавлять элементы во внутренний **синтетический архив бинов**. Синтетический архив бинов представляет собой набор компонентов, которые загружаются при инициализации контейнера. Содержимое синтетического архива бинов зависит от того, включено ли обнаружение:

- Если обнаружение включено, синтетический архив компонентов создаётся с использованием стандартных правил обнаружения компонента и содержит расширенный набор всех JAR-файлов в classpath. Архивы, которые не содержат файл `beans.xml`, исключаются.
- Если обнаружение отключено и компоненты добавляются программно, синтетический архив компонентов содержит только компоненты, которые были добавлены программно.

Глава 29. Дополнительные примеры инъецирования контекстов и зависимостей

В этой главе подробно описывается, как создать и запустить несколько расширенных примеров, использующих CDI.

Сборка и запуск дополнительных примеров CDI

Примеры находятся в каталоге *tut-install*`/examples/cdi/`. Для сборки и запуска примеров сделайте следующее.

1. Используйте IDE NetBeans или Maven для компиляции, упаковки и развёртывания примера.
2. Запустите пример в веб-браузере.

См. главу 2 *Использование примеров учебника*для получения базовой информации по установке, сборке и запуску примеров.

Пример encoder: использование альтернатив

В примере `encoder` показано, как использовать альтернативы для выбора между двумя компонентами во время развёртывания, как описано в Использование альтернатив в приложениях CDI. Пример включает в себя интерфейс и две его реализации, Managed-бин, страницу Facelets и файлы конфигурации.

Исходные файлы находятся в каталоге *tut-install*`/examples/cdi/encoder/src/main/java/ee/jakarta/tutorial/encoder/`.

Интерфейс и реализации encoder

Интерфейс `Coder` содержит только один метод, `codeString`, принимающий два аргумента: строку и целочисленное значение, определяющее способ транспонирования букв в строке.

JAVA

```
public interface Coder {

    public String codeString(String s, int tval);
}
```

Интерфейс имеет два класса реализации: `CoderImpl` и `TestCoderImpl`. Реализация `codeString` в `CoderImpl` сдвигает строковый аргумент вперед в алфавите на количество букв, указанное во втором аргументе. Любые символы, которые не являются буквами, остаются без изменений. (Этот простой код смены известен как шифр Цезаря, потому что Юлий Цезарь по сообщениям использовал его для связи со своими генералами.) Реализация в `TestCoderImpl` просто отображает значения аргументов. Реализация `TestCoderImpl` аннотирована `@Alternative`:

JAVA

```
import jakarta.enterprise.inject.Alternative;

@Alternative
public class TestCoderImpl implements Coder {

    @Override
    public String codeString(String s, int tval) {
        return ("input string is " + s + ", shift value is " + tval);
    }
}
```

Файл `beans.xml` примера `encoder` содержит элемент `alternatives` для класса `TestCoderImpl`, но по умолчанию элемент закомментирован:

XML

```
<beans ...>
    <!--<alternatives>
        <class>ee.jakarta.tutorial.encoder.TestCoderImpl</class>
    </alternatives>-->
</beans>
```

Это означает, что по умолчанию класс `TestCoderImpl`, аннотированный `@Alternative`, не будет использоваться. Вместо этого будет использован класс `CoderImpl`.

Страница Facelets и Managed-бин encoder

Простая страница Facelets `index.xhtml` примера `encoder` просит пользователя ввести строковые и целочисленные значения и передаёт их в Managed-бин, `CoderBean` как `coderBean.inputString` и `coderBean.transVal`:

XML

```
<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
        <h:outputStylesheet library="css" name="default.css"/>
        <title>String Encoder</title>
    </h:head>
    <h:body>
        <h2>String Encoder</h2>
        <p>Type a string and an integer, then click Encode.</p>
        <p>Depending on which alternative is enabled, the coder bean
            will either display the argument values or return a string that
            shifts the letters in the original string by the value you
            specify. The value must be between 0 and 26.</p>
        <h:form id="encodeit">
            <p><h:outputLabel value="Enter a string: " for="inputString"/>
                <h:inputText id="inputString"
                             value="#{coderBean.inputString}"/>
                <h:outputLabel value="Enter the number of letters to shift by: "
                               for="transVal"/>
                <h:inputText id="transVal" value="#{coderBean.transVal}"/></p>
            <p><h:commandButton value="Encode"
                                action="#{coderBean.encodeString()}"/></p>
            <p><h:outputLabel value="Result: " for="outputString"/>
                <h:outputText id="outputString"
                              value="#{coderBean.codedString}"
                              style="color:blue"/></p>
            <p><h:commandButton value="Reset"
                                action="#{coderBean.reset}"/></p>
        </h:form>
        ...
    </h:body>
</html>
```

Когда пользователь кликает кнопку «Encode», страница вызывает метод `encodeString` Managed-бина и отображает результат `coderBean.codedString` синим цветом. На странице также есть кнопка сброса, которая очищает поля.

Managed-бин `CoderBean` является компонентом `@RequestScoped`, который объявляет свои свойства ввода и вывода. Свойство `transVal` имеет три ограничения Bean Validation, которые устанавливают ограничения на целочисленное значение, поэтому, если пользователь вводит недопустимое значение, на странице Facelets появляется сообщение об ошибке по умолчанию. Бин также инъецирует объект интерфейса `Coder`:

JAVA

```
@Named
@RequestScoped
public class CoderBean {

    private String inputString;
    private String codedString;
    @Max(26)
    @Min(0)
    @NotNull
    private int transVal;

    @Inject
    Coder coder;
    ...
}
```

В дополнение к простым get- и set-методам для трёх свойств, бин определяет метод действия `encodeString`, вызываемый страницей Facelets. Этот метод устанавливает для свойства `codedString` значение, возвращаемое при вызове метода `codeString` реализации `Coder`:

JAVA

```
public void encodeString() {
    setCodedString(coder.codeString(inputString, transVal));
}
```

Наконец, бин определяет метод `reset` для очистки полей страницы Facelets:

JAVA

```
public void reset() {
    setInputString("");
    setTransVal(0);
}
```

Запуск encoder

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска приложения `encoder`.

Сборка, упаковка и развёртывание encoder с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/cdi
   ```

4. Выберите каталог `encoder`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `encoder` и выберите **Сборка**.

   Эта команда собирает и упаковывает приложение в WAR-файл `encoder.war`, расположенный в каталоге `target`, а затем развёртывает его в GlassFish Server.

Запуск encoder с IDE NetBeans

1. В веб-браузере введите следующий URL:

```
http://localhost:8080/encoder
```

2. На странице String Encoder введите строку и количество букв для сдвига, а затем нажмите Encode.

   Закодированная строка отображается синим цветом в строке результатов. Например, если вы введёте `Java` и `4`, результатом будет `Neze`.

3. Теперь отредактируйте файл `beans.xml`, чтобы включить альтернативную реализацию `Coder`.

   a. На вкладке «Проекты» в проекте `encoder` разверните узел Веб-страницы, затем разверните узел WEB-INF.

   b. Выполните двойной клик на файле `beans.xml`, чтобы открыть его.

   c. Удалите символы комментария, которые окружают элемент `alternatives`, чтобы он выглядел следующим образом:

   XML
   ```
   <alternatives>
       <class>ee.jakarta.tutorial.encoder.TestCoderImpl</class>
   </alternatives>
   ```

   d. Сохраните файл.

4. Кликните правой кнопкой мыши проект `encoder` и выберите команду «Очистить и собрать».

5. В веб-браузере повторно введите URL, чтобы отобразить страницу String Encoder для повторно развёрнутого проекта:

   ```
   http://localhost:8080/encoder/
   ```

6. Введите строку и количество букв для сдвига, а затем нажмите «Encode».

   На этот раз в строке результатов отображаются ваши аргументы. Например, если вы введёте `Java` и `4`, результат будет следующим:

   ```
   Result: input string is Java, shift value is 4
   ```

Сборка, упаковка и развёртывание encoder с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).

2. В окне терминала перейдите в:

   ```
   tut-install/examples/cdi/encoder/
   ```

3. Введите следующую команду для развёртывания приложения:

   SHELL
   ```
   mvn install
   ```

   Эта команда собирает и упаковывает приложение в WAR-файл `encoder.war`, расположенный в каталоге `target`, а затем развёртывает его в GlassFish Server.

Запуск encoder с использованием Maven

1. В веб-браузере введите следующий URL:

   ```
   http://localhost:8080/encoder/
   ```

Откроется страница String Encoder.

2. Введите строку и количество букв для сдвига, а затем нажмите «Encode».

   Закодированная строка отображается синим цветом в строке результатов. Например, если вы введёте `Java` и `4`, результатом будет `Neze`.

3. Теперь отредактируйте файл `beans.xml`, чтобы включить альтернативную реализацию `Coder`.

   a. В текстовом редакторе откройте следующий файл:

   ```
   tut-install/examples/cdi/encoder/src/main/webapp/WEB-INF/beans.xml
   ```

   b. Удалите символы комментария, которые окружают элемент `alternatives`, чтобы он выглядел следующим образом:

   ```xml
   <alternatives>
       <class>ee.jakarta.tutorial.encoder.TestCoderImpl</class>
   </alternatives>
   ```

   c. Сохраните и закройте файл.

4. Введите следующую команду:

   ```shell
   mvn clean install
   ```

5. В веб-браузере повторно введите URL, чтобы отобразить страницу String Encoder для повторно развёрнутого проекта:

   ```
   http://localhost:8080/encoder
   ```

6. Введите строку и количество букв для сдвига, а затем нажмите «Encode».

   На этот раз в строке результатов отображаются ваши аргументы. Например, если вы введёте `Java` и `4`, результат будет следующим:

   ```
   Result: input string is Java, shift value is 4
   ```

Пример producermethods: использование метода-производителя для выбора реализации компонента

В примере `providermethods` показано, как использовать метод-производитель для выбора между двумя бинами во время выполнения, как описано в Использование методов-производителей, полей-производителей и методов закрытия в приложениях CDI. Он очень похож на пример `encoder`, описанный в Пример encoder: использование альтернатив. Пример включает в себя тот же интерфейс и две его реализации, Managed-бин, страницу Facelets и файлы конфигурации. Он также содержит тип квалификатора. При его запуске не требуется редактировать файл `beans.xml` и повторно развёртывать приложение, чтобы изменить его поведение.

Компоненты producermethods

Компоненты `producermethods` очень похожи на компоненты `encoder` с некоторыми существенными отличиями.

Ни одна реализация компонента `Coder` не аннотирована `@Alternative` и файла `beans.xml` нет, поскольку он не нужен.

Страница Facelets и Managed-бин `CoderBean` имеют дополнительное свойство `coderType`, которое позволяет пользователю указать во время выполнения, какую реализацию использовать. Кроме того, Managed-бин имеет метод-производитель, который выбирает реализацию, используя квалификатор `@Chosen`.

Бин объявляет две константы, определяющие, является ли тип кодировщика тестовой реализацией или реализацией, которая фактически сдвигает буквы:

JAVA

```
private final static int TEST = 1;
private final static int SHIFT = 2;
private int coderType = SHIFT; // значение по умолчанию
```

Метод источника, аннотированный `@Produces` и `@Chosen`, а также `@RequestScoped` (так что он длится только в течение одного запроса и ответа) возвращает одну из двух реализаций на основе `coderType`, предоставленного пользователем.

JAVA

```
@Produces
@Chosen
@RequestScoped
public Coder getCoder() {

    switch (coderType) {
        case TEST:
            return new TestCoderImpl();
        case SHIFT:
            return new CoderImpl();
        default:
            return null;
    }
}
```

Наконец, Managed-бин инъецирует выбранную реализацию, указав тот же квалификатор, который был возвращён методом источника для устранения неоднозначности:

JAVA

```
@Inject
@Chosen
@RequestScoped
Coder coder;
```

Страница Facelets содержит изменённые инструкции и группу выбора из двух элементов, выбранное значение которой присваивается свойству `coderBean.coderType`:

XML

```
<h2>String Encoder</h2>
    <p>Select Test or Shift, type a string and an integer, then click
        Encode.</p>
    <p>If you select Test, the TestCoderImpl bean will display the
        argument values.</p>
    <p>If you select Shift, the CoderImpl bean will return a string that
        shifts the letters in the original string by the value you specify.
        The value must be between 0 and 26.</p>
    <h:form id="encodeit">
        <h:selectOneRadio id="coderType"
                          required="true"
                          value="#{coderBean.coderType}">
            <f:selectItem
                itemValue="1"
                itemLabel="Test"/>
            <f:selectItem
                itemValue="2"
                itemLabel="Shift Letters"/>
        </h:selectOneRadio>
        ...
```

Запуск producermethods

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска приложения `providermethods`.

Пример сборки, упаковки и развёртывания producermethods с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/cdi
   ```

4. Выберите каталог `providermethods`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `producermethods` и выберите **Сборка**.

   Эта команда собирает и упаковывает приложение в WAR-файл, `providermethods.war`, расположенный в каталоге `target`, а затем развёртывает его в GlassFish Server.

Сборка, упаковка и развёртывание producermethods с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/cdi/producermethods/
   ```

3. Введите следующую команду для развёртывания приложения:

   SHELL

   ```
   mvn install
   ```

   Эта команда собирает и упаковывает приложение в WAR-файл, `providermethods.war`, расположенный в каталоге `target`, а затем развёртывает его в GlassFish Server.

Запуск producermethods

1. В веб-браузере введите следующий URL:

   ```
   http://localhost:8080/producermethods
   ```

2. На странице String Encoder выберите опцию «Test» или «Shift Letters», введите строку и количество букв для смещения, а затем нажмите «Encode».

   В зависимости от вашего выбора, строка результата отображает либо закодированную строку, либо указанные входные значения.

Пример producerfields: использование полей-производителей для создания ресурсов

Пример `providerfields`, позволяющий создавать список дел, показывает, как использовать поле-производитель для генерации объектов, которыми затем может управлять контейнер. В этом примере создаётся объект `EntityManager`, но ресурсы, такие как соединения JDBC и источники данных, также могут быть сгенерированы таким образом.

Пример `providerfields` — самый простой пример сущности. Он также содержит квалификатор и класс, который генерирует EntityManager. Он также содержит один объект, сессионный компонент с состоянием, страницу Facelets и Managed-бин.

Исходные файлы находятся в каталоге *`tut-install`*`/examples/cdi/producerfields/src/main/java/ee/jakarta/tutorial/producerfields/`.

Пример поля-производителя для producerfields

Наиболее важным компонентом примера `providerfields` является самый маленький класс `db.UserDatabaseEntityManager`, который изолирует генерацию объекта `EntityManager`, чтобы он мог легко использоваться другими компонентами в приложении. Класс использует поле источника, чтобы инъецировать `EntityManager` с аннотацией `@UserDatabase`, также определённой в пакете `db`:

JAVA

```
@Singleton
public class UserDatabaseEntityManager {

    @Produces
    @PersistenceContext
    @UserDatabase
    private EntityManager em;
    ...
}
```

Класс не создаёт поле юнит персистентности явным образом, но в приложении есть файл `persistence.xml`, в котором описан юнит персистентности. Класс аннотирован `jakarta.inject.Singleton` для указания, он должен быть создан только один раз.

Класс `db.UserDatabaseEntityManager` также содержит закомментированный код, который использует методы `create` и `close` для генерации и удаления поля источника:

JAVA

```
/*
@PersistenceContext
private EntityManager em;

@Produces
@UserDatabase
public EntityManager create() {
    return em;
}
*/

public void close(@Disposes @UserDatabase EntityManager em) {
    em.close();
}
```

Вы можете удалить индикаторы комментариев из этого кода и разместить их вокруг объявления поля, чтобы проверить, как работают методы. Поведение приложения одинаково в обоих случаях.

Преимущество создания `EntityManager` в отдельном классе, а не просто встраивание его в Enterprise-бин, состоит в том, что пользовать объект повторно можно легко и типобезопасно (typesafe). Кроме того, более сложное приложение может создавать несколько EntityManager-ов с использованием нескольких юнитов персистентности, и этот механизм изолирует этот код для простоты обслуживания, как в следующем примере:

JAVA

```
@Singleton
public class JPAResourceProducer {
    @Produces
    @PersistenceUnit(unitName="pu3")
    @TestDatabase
    EntityManagerFactory customerDatabasePersistenceUnit;

    @Produces
    @PersistenceContext(unitName="pu3")
    @TestDatabase
    EntityManager customerDatabasePersistenceContext;

    @Produces
    @PersistenceUnit(unitName="pu4")
    @Documents
    EntityManagerFactory customerDatabasePersistenceUnit;

    @Produces
    @PersistenceContext(unitName="pu4")
    @Documents
    EntityManager docDatabaseEntityManager;
}
```

Объявления `EntityManagerFactory` также позволяют использовать EntityManager, управляемый приложением.

Сущность и сессионный бин примера producerfields

Пример `providerfields` содержит простой класс сущности `entity.ToDo` и сессионный компонент с состоянием `ejb.RequestBean`, который его использует.

Класс сущности содержит три поля: автогенерируемое поле `id`, строку, задающую задачу, и метку времени. Поле метки времени `timeCreated` помечено аннотацией `@Temporal`, которая необходима для полей `Date`.

JAVA

```
@Entity
public class ToDo implements Serializable {

    ...
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    protected String taskText;
    @Temporal(TIMESTAMP)
    protected Date timeCreated;

    public ToDo() {
    }

    public ToDo(Long id, String taskText, Date timeCreated) {
        this.id = id;
        this.taskText = taskText;
        this.timeCreated = timeCreated;
    }
    ...
}
```

Остальная часть класса `ToDo` содержит обычные методы получения, установки и другие методы сущностей.

Класс `RequestBean` инъецирует `EntityManager`, созданный методом-производителем, с аннотацией `@UserDatabase`:

JAVA

```
@ConversationScoped
@Stateful
public class RequestBean {

    @Inject
    @UserDatabase
    EntityManager em;
}
```

Затем он определяет два метода, один из которых создаёт и сохраняет один элемент списка `ToDo`, а другой — получает все элементы `ToDo`, созданные на данный момент, путём создания запроса:

JAVA

```
public ToDo createToDo(String inputString) {
    ToDo toDo = null;
    Date currentTime = Calendar.getInstance().getTime();

    try {
        toDo = new ToDo();
        toDo.setTaskText(inputString);
        toDo.setTimeCreated(currentTime);
        em.persist(toDo);
        return toDo;
    } catch (Exception e) {
        throw new EJBException(e.getMessage());
    }
}

public List<ToDo> getToDos() {
    try {
            List<ToDo> toDos =
                (List<ToDo>) em.createQuery(
                "SELECT t FROM ToDo t ORDER BY t.timeCreated")
                .getResultList();
        return toDos;
    } catch (Exception e) {
        throw new EJBException(e.getMessage());
    }
}
```

producerfields — страницы Facelets и Managed-бин

В примере `providerfields` есть две страницы Facelets, `index.xhtml` и `todolist.xhtml`. Простая форма на странице `index.xhtml` запрашивает у пользователя только задачу. Когда пользователь кликает кнопку «Submit», вызывается метод `listBean.createTask`. Когда пользователь кликает кнопку «Show Items», действие указывает, что должен отображаться файл `todolist.xhtml`:

XML

```
    <h:body>
        <h2>To Do List</h2>
        <p>Enter a task to be completed.</p>
        <h:form id="todolist">
            <p><h:outputLabel value="Enter a string: " for="inputString"/>
                <h:inputText id="inputString"
                             value="#{listBean.inputString}"/></p>
            <p><h:commandButton value="Submit"
                                action="#{listBean.createTask()}"/></p>
            <p><h:commandButton value="Show Items"
                                action="todolist"/></p>
        </h:form>
        ...
    </h:body>
```

Managed-бин `web.ListBean` инъецирует сессионный бин `ejb.RequestBean`. Он объявляет сущность `entity.ToDo` и список сущностей вместе со входной строкой, которую он передаёт в сессионный компонент. `inputString` аннотируется ограничением `@NotNull` Bean Validation, поэтому попытка отправить пустую строку приведёт к ошибке.

JAVA

```java
@Named
@ConversationScoped
public class ListBean implements Serializable {

    ...
    @EJB
    private RequestBean request;
    @NotNull
    private String inputString;
    private ToDo toDo;
    private List<ToDo> toDos;
    ...
}
```

Метод `createTask`, обработчик нажатия кнопки Submit, вызывает метод `createToDo` `RequestBean`:

JAVA

```java
public void createTask() {
    this.toDo = request.createToDo(inputString);
}
```

Метод `getToDos`, который вызывается страницей `todolist.xhtml`, в свою очередь вызывает метод `getToDos` `RequestBean`:

JAVA

```java
public List<ToDo> getToDos() {
    return request.getToDos();
}
```

Чтобы заставить страницу Facelets распознавать пустую строку как null и возвращать ошибку, в файле `web.xml` устанавливается контекстный параметра `jakarta.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL` в значение `true`:

XML

```xml
<context-param>
  <param-name>jakarta.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL</param-name>
  <param-value>true</param-value>
</context-param>
```

Страница `todolist.xhtml` немного сложнее, чем страница `index.html`. Она содержит элемент `dataTable`, который отображает содержимое списка `ToDo`. Тело страницы выглядит так:

XML

```
<body>
    <h2>To Do List</h2>
    <h:form id="showlist">
        <h:dataTable var="toDo"
                     value="#{listBean.toDos}"
                     rules="all"
                     border="1"
                     cellpadding="5">
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Time Stamp" />
                </f:facet>
                <h:outputText value="#{toDo.timeCreated}" />
            </h:column>
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Task" />
                </f:facet>
                <h:outputText value="#{toDo.taskText}" />
            </h:column>
        </h:dataTable>
        <p><h:commandButton id="back" value="Back" action="index" /></p>
    </h:form>
</body>
```

Значением `dataTable` является `listBean.toDos` — список, возвращаемый методом `getToDos` Managed-бина, который в свою очередь вызывает метод сессионного компонента `getToDos`. В каждой строке таблицы отображаются поля `timeCreated` и `taskText` отдельной задачи. Наконец, кнопка «Back» возвращает пользователя на страницу `index.xhtml`.

Запуск producerfields

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска приложения `providerfields`.

Сборка, упаковка и развёртывание producerfields с использованием IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. Если сервер базы данных ещё не запущен, запустите его, следуя инструкциям в Запуск и остановка Apache Derby.
3. В меню **Файл** выберите **Открыть проект**.
4. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/cdi
   ```

5. Выберите каталог `providerfields`.
6. Нажмите **Открыть проект**.
7. На вкладке **Проекты** кликните правой кнопкой мыши проект `producerfields` и выберите **Сборка**.

   Эта команда собирает и упаковывает приложение в WAR-файл `providerfields.war`, расположенный в каталоге `target`, а затем развёртывает его в GlassFish Server.

Сборка, упаковка и развёртывание producerfields с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. Если сервер базы данных ещё не запущен, запустите его, следуя инструкциям в Запуск и остановка Apache Derby.

3. В окне терминала перейдите в:

```
tut-install/examples/cdi/producerfields/
```

4. Введите следующую команду для развёртывания приложения:

```
mvn install
```
SHELL

Эта команда собирает и упаковывает приложение в WAR-файл `providerfields.war`, расположенный в каталоге `target`, а затем развёртывает его в GlassFish Server.

Запуск producerfields

1. В веб-браузере введите следующий URL:

```
http://localhost:8080/producerfields
```

2. На странице «Создать список дел» введите строку в поле и нажмите «Отправить».

   Вы можете ввести дополнительные строки и нажать «Отправить», чтобы создать список задач с несколькими элементами.

3. Нажмите Показать элементы.

   Откроется страница «Список дел», показывающая метку времени и текст для каждого созданного вами элемента.

4. Нажмите «Назад», чтобы вернуться на страницу «Создать список дел».

   На этой странице можно ввести дополнительные элементы в список.

Пример billpayment: использование событий и Interceptor-ов

В примере `billpayment` показано, как использовать события и Interceptor-ы.

Исходные файлы находятся в каталоге *`tut-install`*`/examples/cdi/billpayment/src/main/java/ee/jakarta/tutorial/billpayment/`.

Обзор примера billpayment

Пример имитирует оплату счёта с помощью дебетовой или кредитной карты. Когда пользователь выбирает способ оплаты, Managed-бин создаёт соответствующее событие, заполняет его данными и выбрасывает (fires) его. Простой слушатель события обрабатывает его, используя методы-наблюдатели.

В примере также определяется Interceptor, установленный в одном случае целиком на класс и в другом случае на два метода в другом классе.

Класс события PaymentEvent

Класс события, `event.PaymentEvent`, является простым классом компонента, который содержит конструктор без аргументов. Он также имеет метод `toString` и get- и set- методы для данных: `String` для типа платежа, `BigDecimal` для суммы платежа и `Date` для отметки времени.

JAVA

```
public class PaymentEvent implements Serializable {

    ...
    public String paymentType;
    public BigDecimal value;
    public Date datetime;

    public PaymentEvent() {
    }

    @Override
    public String toString() {
        return this.paymentType
                + " = $" + this.value.toString()
                + " at " + this.datetime.toString();
    }
    ...
}
```

Класс события — это простой компонент, объект которого инстанцируется Managed-бином с помощью `new`, а затем заполняется. По этой причине контейнер CDI не может перехватить создание компонента, и, следовательно, он не может разрешить перехват его get- и set-методов.

Слушатель событий PaymentHandler

Слушатель событий `listener.PaymentHandler` содержит два метода-наблюдателя, по одному для каждого из двух типов событий:

JAVA

```
@Logged
@SessionScoped
public class PaymentHandler implements Serializable {

    ...
    public void creditPayment(@Observes @Credit PaymentEvent event) {
        logger.log(Level.INFO, "PaymentHandler - Credit Handler: {0}",
                event.toString());

        // вызов специфичного обработчика класса Credit...
    }

    public void debitPayment(@Observes @Debit PaymentEvent event) {
        logger.log(Level.INFO, "PaymentHandler - Debit Handler: {0}",
                event.toString());

        // вызов специфичного обработчика класса Debit...
    }
}
```

Каждый метод-наблюдатель принимает в качестве аргумента событие, аннотированное `@Observes` и квалификатором типа платежа. В реальном приложении методы-наблюдатели передают информацию о событии другому компоненту, который выполняет бизнес-логику платежа.

Квалификаторы определены в пакете `payment`, описанном в Страницы Facelets и Managed-бин для billpayment.

Компонент `PaymentHandler` аннотирован `@Logged`, так что все его методы могут быть перехвачены.

Страницы Facelets и Managed-бин для billpayment

Пример `billpayment` содержит две страницы Facelets, `index.xhtml` и `response.xhtml`. Тело `index.xhtml` выглядит так:

XML

```
    <h:body>
        <h3>Bill Payment Options</h3>
        <p>Enter an amount, select Debit Card or Credit Card,
            then click Pay.</p>
        <h:form>
            <p>
            <h:outputLabel value="Amount: $" for="amt"/>
            <h:inputText id="amt" value="#{paymentBean.value}"
                         required="true"
                         requiredMessage="An amount is required."
                         maxlength="15" />
            </p>
            <h:outputLabel value="Options:" for="opt"/>
            <h:selectOneRadio id="opt" value="#{paymentBean.paymentOption}">
                <f:selectItem id="debit" itemLabel="Debit Card"
                              itemValue="1"/>
                <f:selectItem id="credit" itemLabel="Credit Card"
                              itemValue="2" />
            </h:selectOneRadio>
            <p><h:commandButton id="submit" value="Pay"
                                action="#{paymentBean.pay}" /></p>
            <p><h:commandButton value="Reset"
                                action="#{paymentBean.reset}" /></p>
        </h:form>
        ...
    </h:body>
```

Поле ввода принимает сумму платежа, переданную в `paymentBean.value`. Из представленных двух вариантов пользователь может выбрать платёж дебетовой или кредитной картой, передав целочисленное значение в `paymentBean.paymentOption`. Наконец, действием кнопки «Pay» назначен метод `paymentBean.pay`, а кнопки «Reset» — метод `paymentBean.reset`.

Managed-бин `payment.PaymentBean` использует квалификаторы чтобы различать два вида событий оплаты:

JAVA

```
@Named
@SessionScoped
public class PaymentBean implements Serializable {

   ...
    @Inject
    @Credit
    Event<PaymentEvent> creditEvent;

    @Inject
    @Debit
    Event<PaymentEvent> debitEvent;
    ...
}
```

Квалификаторы `@Credit` и `@Debit` определяются в пакете `payment` вместе с `PaymentBean`.

Затем `PaymentBean` определяет свойства, которые он получает со страницы Facelets, и передаёт событие:

JAVA

```
public static final int DEBIT = 1;
public static final int CREDIT = 2;
private int paymentOption = DEBIT;

@Digits(integer = 10, fraction = 2, message = "Invalid value")
private BigDecimal value;

private Date datetime;
```

Значение `paymentOption` является целым числом, переданным из компонента option. Значение по умолчанию — `DEBIT.value` — это `BigDecimal` с ограничением Bean Validation, которое проверяет максимальное количество цифр для значения валюты. Отметка времени для события `datetime` является объектом `Date`, инициализированным при вызове метода `pay`.

Метод `pay` компонента сначала устанавливает метку времени для события оплаты. Затем он создаёт и заполняет данные события, используя конструктор для `PaymentEvent` и вызывая методы установки события, используя свойства бина в качестве аргументов. Затем он запускает событие.

JAVA

```
@Logged
public String pay() {
    this.setDatetime(Calendar.getInstance().getTime());
    switch (paymentOption) {
        case DEBIT:
            PaymentEvent debitPayload = new PaymentEvent();
            debitPayload.setPaymentType("Debit");
            debitPayload.setValue(value);
            debitPayload.setDatetime(datetime);
            debitEvent.fire(debitPayload);
            break;
        case CREDIT:
            PaymentEvent creditPayload = new PaymentEvent();
            creditPayload.setPaymentType("Credit");
            creditPayload.setValue(value);
            creditPayload.setDatetime(datetime);
            creditEvent.fire(creditPayload);
            break;
        default:
            logger.severe("Invalid payment option!");
    }
    return "response";
}
```

Метод `pay` возвращает страницу `response.xhtml`, на которую перенаправлено действие.

Класс `PaymentBean` также содержит метод `reset`, который очищает поле значения на странице `index.xhtml` и устанавливает для параметра оплаты значение по умолчанию:

JAVA

```
@Logged
public void reset() {
    setPaymentOption(DEBIT);
    setValue(BigDecimal.ZERO);
}
```

В этом компоненте перехватываются только методы `pay` и `reset`.

На странице `response.xhtml` отображается уплаченная сумма. Для отображения способа оплаты используется выражение `render`:

XML

```
    <h:body>
        <h:form>
            <h2>Bill Payment: Result</h2>
            <h3>Amount Paid with
                <h:outputText id="debit" value="Debit Card: "
                              rendered="#{paymentBean.paymentOption eq 1}" />
                <h:outputText id="credit" value="Credit Card: "
                              rendered="#{paymentBean.paymentOption eq 2}" />
                <h:outputText id="result" value="#{paymentBean.value}">
                    <f:convertNumber type="currency"/>
                </h:outputText>
            </h3>
            <p><h:commandButton id="back" value="Back" action="index" /></p>
        </h:form>
    </h:body>
```

Класс Interceptor-а LoggedInterceptor

Класс Interceptor-а `LoggedInterceptor` и его привязка Interceptor-а `Logged` определены в пакете `interceptor`. Привязка Interceptor-а `Logged` определяется следующим образом:

JAVA

```
@Inherited
@InterceptorBinding
@Retention(RUNTIME)
@Target({METHOD, TYPE})
public @interface Logged {
}
```

Класс `LoggedInterceptor` выглядит следующим образом:

JAVA

```
@Logged
@Interceptor
public class LoggedInterceptor implements Serializable {

    ...

    public LoggedInterceptor() {
    }

    @AroundInvoke
    public Object logMethodEntry(InvocationContext invocationContext)
            throws Exception {
        System.out.println("Entering method: "
                + invocationContext.getMethod().getName() + " in class "
                + invocationContext.getMethod().getDeclaringClass().getName());

        return invocationContext.proceed();
    }
}
```

Класс аннотируется `@Logged` и `@Interceptor`. Метод `@AroundInvoke`, `logMethodEntry`, принимает обязательный аргумент `InvocationContext` и вызывает требуемый метод `continue`. Когда метод перехватывается, `logMethodEntry` отображает имя вызываемого метода, а также его класс.

Чтобы включить Interceptor, файл `beans.xml` определяет его следующим образом:

XML

```
<interceptors>
    <class>ee.jakarta.tutorial.billpayment.interceptor.LoggedInterceptor</class>
</interceptors>
```

В этом приложении классы `PaymentEvent` и `PaymentHandler` аннотируются `@Logged`, поэтому все их методы перехватываются. В `PaymentBean` только методы `pay` и `reset` аннотируются `@Logged`, поэтому только эти методы перехватываются.

Запуск billpayment

Вы можете использовать IDE NetBeans или Maven для создания, упаковки, развёртывания и запуска приложения `billpayment`.

Сборка, упаковка и развёртывание billpayment с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/cdi
   ```

4. Выберите каталог `billpayment`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `billpayment` и выберите **Сборка**.

   Эта команда собирает и упаковывает приложение в WAR-файл `billpayment.war`, расположенный в каталоге `target`, а затем развёртывает его в GlassFish Server.

Сборка, упаковка и развёртывание billpayment с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/cdi/billpayment/
   ```

3. Введите следующую команду для развёртывания приложения:

   ```
   mvn install
   ```
   SHELL

   Эта команда собирает и упаковывает приложение в WAR-файл `billpayment.war`, расположенный в каталоге `target`, а затем развёртывает его в GlassFish Server.

Запуск billpayment

1. В веб-браузере введите следующий URL:

   ```
   http://localhost:8080/billpayment
   ```

2. На странице Варианты оплаты счетов введите значение в поле Сумма.

   Сумма может содержать до 10 цифр и включать до двух десятичных знаков. Например:

   ```
   9876.54
   ```

3. Выберите «Дебетовая карта» или «Кредитная карта» и нажмите «Оплатить».

   Откроется страница «Оплата счёта: результат», отображающая уплаченную сумму и способ оплаты:

```
Amount Paid with Credit Card: $9,876.34
```

4. Нажмите «Назад», чтобы вернуться на страницу «Оплата счетов».

   Вы также можете нажать Сброс, чтобы вернуться к начальным значениям страницы.

5. Проверьте вывод журнала сервера.

   В IDE NetBeans выходные данные отображаются на вкладке «Вывод GlassFish Server». В противном случае просмотрите файл *domain-dir*`/logs/server.log`.

   Выходные данные каждого Interceptor-а появляются в журнале, за которым следуют дополнительные выходные данные регистратора, определённые конструктором и методами:

```
INFO: Entering method: pay in class billpayment.payment.PaymentBean
INFO: PaymentHandler created.
INFO: Entering method: debitPayment in class billpayment.listener.PaymentHandler
INFO: PaymentHandler - Debit Handler: Debit = $1234.56 at Tue Dec 14 14:50:28 EST 2010
```

Пример decorators: декорирование бинов

Пример `decorators` является ещё одним вариантом примера `encoder` и показывает, как использовать декоратор для реализации дополнительной бизнес-логики для компонента.

Исходные файлы находятся в каталоге *tut-install*`/examples/cdi/decorators/src/main/java/ee/jakarta/tutorial/decorators/`.

Обзор decorators

Вместо того, чтобы пользователь выбирал между двумя альтернативными реализациями интерфейса во время развёртывания или выполнения, декоратор добавляет некоторую дополнительную логику в одну реализацию интерфейса.

Пример включает в себя интерфейс, его реализацию, декоратор, Interceptor, Managed-бин, страницу Facelets и файлы конфигурации.

Компоненты decorators

Пример `decorators` очень похож на пример `encoder`, описанный в Пример encoder: использование альтернатив. Однако вместо предоставления двух реализаций интерфейса `Coder` этот пример предоставляет только класс `CoderImpl`. Класс декоратора `CoderDecorator` вместо простого возврата кодированной строки отображает значения и длину входных и выходных строк.

Класс `CoderDecorator`, например `CoderImpl`, реализует бизнес-метод интерфейса `Coder`, `codeString`:

JAVA

```
@Decorator
public abstract class CoderDecorator implements Coder {

    @Inject
    @Delegate
    @Any
    Coder coder;

    public String codeString(String s, int tval) {
        int len = s.length();

        return "\"" + s + "\" becomes " + "\"" + coder.codeString(s, tval)
                + "\", " + len + " characters in length";
    }
}
```

Метод `codeString` декоратора вызывает метод `codeString` объекта делегата для выполнения фактического кодирования.

Пример `decorators` включает в себя привязку Interceptor-а `Logged` и класс `LoggedInterceptor` из примера `billpayment`. В этом примере Interceptor устанавливается для метода `CoderBean.encodeString` и метода `CoderImpl.codeString`. Код Interceptor-а не изменяется. Interceptor-ы, как правило, повторно используемы для различных приложений.

За исключением аннотаций Interceptor-ов, классы `CoderBean` и `CoderImpl` идентичны версиям в примере `encoder`.

Файл `beans.xml` определяет как декоратор, так и Interceptor:

XML

```
<decorators>
    <class>ee.jakarta.tutorial.decorators.CoderDecorator</class>
</decorators>
<interceptors>
    <class>ee.jakarta.tutorial.decorators.LoggedInterceptor</class>
</interceptors>
```

Запуск decorators

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска приложения `decorators`.

Пример сборки, упаковки и развёртывания decorators с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/cdi
   ```

4. Выберите каталог `decorators`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `decorators` и выберите **Сборка**.

   Эта команда собирает и упаковывает приложение в WAR-файл `decorators.war`, расположенный в каталоге `target`, а затем развёртывает его в GlassFish Server.

Пример сборки, упаковки и развёртывания decorators с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/cdi/decorators/
   ```

3. Введите следующую команду для развёртывания приложения:

   ```
   mvn install
   ```
   SHELL

   Эта команда собирает и упаковывает приложение в WAR-файл `decorators.war`, расположенный в каталоге `target`, а затем развёртывает его в GlassFish Server.

Запуск decorators

1. В веб-браузере введите следующий URL:

   ```
   http://localhost:8080/decorators
   ```

2. На странице Decorated String Encoder введите строку и количество букв для сдвига, а затем нажмите Encode.

   Вывод метода декоратора отображается синим цветом в строке результатов. Например, если вы ввели `Java` и `4`, то увидите следующее:

   ```
   "Java" becomes "Neze", 4 characters in length
   ```

3. Проверьте вывод журнала сервера.

   В IDE NetBeans выходные данные отображаются на вкладке «Вывод GlassFish Server». В противном случае просмотрите файл *`domain-dir`*`/logs/server.log`.

   Вывод от Interceptor-ов появляется:

   ```
   INFO: Entering method: encodeString in class ee.jakarta.tutorial.decorators.CoderBean
   INFO: Entering method: codeString in class ee.jakarta.tutorial.decorators.CoderImpl
   ```

Часть VI: Веб-сервисы

В части VI исследуются веб-сервисы.

Глава 30. Введение в веб-сервисы

В этой части учебника обсуждаются технологии веб-сервисов Jakarta EE. Эти технологии включают Jakarta XML Web Services и Jakarta RESTful Web Services.

Что такое веб-сервисы?

Веб-сервисы — это клиентские и серверные приложения, которые обмениваются данными по протоколу HTTP в сети Интернет. Как описывает Консорциум World Wide Web (W3C), веб-сервисы предоставляют стандартные средства взаимодействия между приложениями, работающими на различных платформах. Благодаря использованию XML веб-сервисы характеризуются высокой функциональной совместимостью и расширяемостью, а также их машиночитаемыми описаниями. Веб-сервисы могут быть объединены в слабосвязной форме для выполнения сложных операций. Программы, предоставляющие простые сервисы, могут взаимодействовать друг с другом для предоставления более сложных сервисов.

Типы веб-сервисов

На концептуальном уровне сервис представляет собой программный компонент, предоставляемый через доступную по сети конечную точку. Потребитель и поставщик услуг используют для обмена в запросах и ответах сообщения в виде документов, не зависящих от технических особенностей принимающей стороны.

Технически веб-сервисы могут быть реализованы различными способами. Два типа веб-сервисов, обсуждаемые в этом разделе, можно различить как «большие» веб-сервисы и «RESTful» веб-сервисы.

«Большие» веб-сервисы

Jakarta XML Web Services предоставляют функциональные возможности для «больших» веб-сервисов, которые описаны в главе 31 *Создание веб-сервисов с Jakarta XML Web Services*. Большие веб-сервисы используют сообщения XML, оформленные по стандарту SOAP — языку XML, определяющему архитектуру и форматы сообщений. Такие системы часто содержат машиночитаемое описание предлагаемых сервисом операций на языке описания веб-сервисов (WSDL) — языке XML для синтаксического описания интерфейсов.

> Исторически Java API for XML Web Services (JAX-WS) был перемещён в Java SE 8, но в Java SE 11 удалён. Jakarta XML Web Services теперь включены в платформу Jakarta EE в качестве дополнительной технологии в пространстве имён `jakarta`.

Формат сообщений SOAP и язык определения интерфейсов WSDL получили широкое распространение. Многие инструменты разработки, такие как IDE NetBeans, могут снизить сложность разработки приложений веб-сервисов.

Проект SOAP должен включать следующие элементы.

- Должен быть установлен официальный договор для описания интерфейса, предлагаемого веб-сервисом. WSDL может использоваться для описания деталей контракта, которые могут включать в себя сообщения, операции, привязки и местоположение веб-сервиса. Для обработки сообщений SOAP в сервисе JAX-WS публикация WSDL не обязательна.
- Архитектура должна удовлетворять сложным нефункциональным требованиям. Многие спецификации веб-сервисов отвечают этим требованиям и устанавливают для них общий словарь. Примерами могут служить транзакции, безопасность, адресация, доверие, координирование и так далее.

- Архитектура должна уметь обрабатывать асинхронную обработку и вызов. В таких случаях инфраструктура, обеспечиваемая стандартами, такими как Web Services Reliable Messaging Protocol (WSRM), и API, такими как JAX-WS, с поддержкой асинхронного вызова на стороне клиента, могут быть использованы сразу после установки.

RESTful веб-сервисы

В Jakarta EE Jakarta RESTful Web Services предоставляют функциональные возможности для веб-сервисов REST. REST хорошо подходит для базовых сценариев интеграции ad hoc. RESTful веб-сервисы зачастую лучше интегрированы с HTTP, чем SOAP сервисы, не требуют сообщений XML или файлов WSDL.

Проект Jersey — это готовая реализация спецификации RESTful веб-сервисов Jakarta. В Jersey реализована поддержка аннотаций, опредёленных спецификацией Jakarta RESTful Web Services, что упрощает разработчикам создание RESTful веб-сервисов с Java и JVM.

Поскольку RESTful веб-сервисы используют существующие хорошо известные стандарты W3C и Internet Engineering Task Force (IETF) (HTTP, XML, URI, MIME) и имеют легковесную инфраструктуру, позволяющую создавать сервисы с минимальным набором инструментов, разработка RESTful веб-сервисов стоит недорого и, таким образом, имеет очень низкий барьер для внедрения. При использовании IDE (таких как IDE NetBeans) разработка RESTful веб-сервисов становится ещё проще.

Дизайн RESTful может быть выбран при следующих требованиях.

- Веб-сервисы не сохраняют состояния. Хорошим тестом является проверка того, может ли взаимодействие клиента и сервера пережить перезапуск сервера.
- Инфраструктура кэширования может быть использована для повышения производительности. Если данные, возвращаемые веб-сервисом, не генерируются динамически и могут быть кэшированы, инфраструктура кэширования, предоставляемая веб-серверами и другими посредниками, может быть использована для повышения производительности. Однако разработчик должен позаботиться о том, чтобы такие кэши были ограничены HTTP `GET` -методом для большинства серверов.
- Производитель и потребитель сервисов имеют общее понимание относительно контекста и передаваемого содержимого. Поскольку не существует формального способа описания интерфейсов веб-сервисов, обе стороны должны прийти к соглашению о схемах, описывающих данные, которыми они обмениваются, и об их корректной обработке. В реальном мире большинство коммерческих приложений, публикующих свои сервисы в виде RESTful веб-сервисов, распространяют также и инструкции, описывающие интерфейсы для разработчиков на популярных языках программирования.
- Пропускная способность особенно важна и должна быть ограничена. REST особенно полезен для устройств с ограниченными возможностями, такими как КПК и мобильные телефоны, для которых нужно минимизировать накладные расходы на обработку заголовка и элементов SOAP в содержательной части XML.
- С использованием подхода RESTful можно легко подключить функциональность веб-сервисов на существующие веб-сайты. Разработчики могут использовать технологии JAX-RS, Ajax и им подобные и фреймворки типа Google Web Toolkit (GWT), Vaadin, Angular для использования сервисов в своих веб-приложениях. Вместо того, чтобы начинать с нуля, сервисы могут быть представлены с помощью XML и использоваться HTML-страницами без значительных изменений в существующей архитектуре веб-сайта. Продуктивность разработки растёт, потому что используется уже имеющийся у разработчиков опыт вместо того, чтобы начинать с нуля с новой технологией.

RESTful веб-сервисы обсуждаются в главе 32 *Создание RESTful веб-сервисов с Jakarta REST*. В этой главе содержится информация о создании каркаса RESTful веб-сервиса с IDE NetBeans и инструмента управления проектами Maven.

Выбор типа веб-сервиса для использования

Допустим, вы хотите использовать RESTful веб-сервисы для интеграции через Интернет и большие веб-сервисы в корпоративных сценариях для интеграции приложений, которые имеют повышенные требования к качеству обслуживания (QoS).

Jakarta XML Web Services

Отвечает расширенным требованиям QoS, которые обычно возникают в корпоративных вычислениях. По сравнению с Jakarta RESTful Web Services, XML веб-сервисы упрощают поддержку набора протоколов WS- *, которые, помимо прочего, обеспечивают стандарты безопасности и надёжности и взаимодействуют с другими клиентами и серверами, соответствующими WS- *.

Jakarta RESTful Web Service

Упрощает создание веб-приложений, которые используют стиль REST, чтобы вызвать желаемые свойства в приложении, такие как слабосвязность (проще развитие сервера без нарушения существующих клиентов), масштабируемость (начать с малого и расти) и архитектурная простота (использование готовых компонентов, такие как прокси или HTTP-маршрутизаторы). Вы можете решить использовать Jakarta RESTful Web Services в веб-приложении, потому что многим типам клиентов проще использовать RESTful веб-сервисы, что оставляет серверной стороне развиваться и масштабироваться независимо. Клиенты могут выбрать использование лишь некоторых из аспектов сервиса и объединения её с другими веб-сервисами.

Глава 31. Создание веб-сервисов с Jakarta XML Web Services

В этой главе описывается Jakarta XML Web Services, технология построения веб-сервисов и клиентов, которые взаимодействуют с помощью XML. XML веб-сервисы позволяют разработчикам создавать веб-сервисы, ориентированные как на сообщения, так и на вызовы удалённых процедур (RPC).

Обзор Jakarta XML Web Services

В Jakarta XML Web Services вызов операции веб-сервиса представлен протоколом на основе XML, таким как SOAP. Спецификация SOAP определяет структуру сообщения, правила кодирования и соглашения для представления вызовов и ответов веб-сервиса. Эти вызовы и ответы передаются как сообщения SOAP (файлы XML) по HTTP.

Хотя сообщения SOAP сложны, API XML веб-сервисов скрывает эту сложность от разработчика приложений. На стороне сервера разработчик специфицирует операции веб-сервиса, определяя методы в интерфейсе, написанном на Java. Разработчик также кодирует классы с реализацией этих методов. Клиентские программы также легко кодируются. Клиент создаёт прокси (локальный объект, представляющий сервис), а затем просто вызывает методы на прокси. С помощью XML веб-сервисов разработчик не создаёт и не анализирует сообщения SOAP. Это система времени выполнения XML веб-сервисов, которая преобразует вызовы API и ответы в сообщения SOAP и из них.

С XML веб-сервисами клиенты и веб-сервисы имеют преимущество в кроссплатформенности языка Java. Кроме того, XML веб-сервисы не являются ограничительными: клиент XML веб-сервиса может получить доступ к веб-сервису, который работает на платформе, отличной от Java, и наоборот. Такая гибкость возможна, поскольку XML веб-сервисы используют технологии, определённые W3C: HTTP, SOAP и WSDL. WSDL определяет формат XML для описания сервиса как набора конечных точек, работающих с сообщениями.

Несколько файлов в примерах XML веб-сервисах зависят от порта, указываемого при установке GlassFish Server. Эти учебные примеры предполагают, что сервер работает на порте по умолчанию, 8080. Они не работают с настройками порта не по умолчанию.

Создание простого веб-сервиса и клиентов с Jakarta XML Web Services

В этом разделе показано, как собрать и развернуть простой веб-сервис и два клиента: консольный клиент и веб-клиент. Исходный код сервиса находится в каталоге *tut-install*`/examples/jaxws/helloservice-war/`, а клиенты — в каталогах *tut-install*`/examples/jaxws/hello-appclient/` и *tut-install*`/examples/jaxws/hello-webclient/`.

На рис. 31-1 показано, как технология XML веб-сервисов Services управляет связыванием между веб-сервисом и клиентом.

Рисунок 31-1 Связь между XML веб-сервисом и клиентом

Отправной точкой для разработки XML веб-сервиса является класс Java, аннотированный `jakarta.jws.WebService`. Аннотация `@WebService` определяет класс как конечную точку веб-сервиса.

Интерфейс конечной точки сервиса или реализация конечной точки сервиса (SEI) — это соответственно интерфейс или класс Java, объявляющий методы, которые клиент может вызывать у сервиса. Интерфейс не требуется при построении конечной точки XML веб-сервиса. Класс реализации веб-сервиса неявно определяет SEI.

Можно указать явный интерфейс, добавив элемент `endpointInterface` в аннотацию `@WebService` в классе реализации. Затем нужно предоставить интерфейс, который определяет публичные методы, доступные в классе реализации конечной точки.

Основные шаги для создания веб-сервиса и клиента

Основные шаги для создания веб-сервиса и клиента следующие.

1. Закодируйте класс реализации.
2. Скомпилируйте класс реализации.
3. Упакуйте файлы в WAR-файл.
4. Разверните WAR-файл. Артефакты веб-сервиса, используемые для связи с клиентами, создаются GlassFish Server во время развёртывания.
5. Закодируйте клиентский класс.
6. Используйте цель Maven `wsimport` для генерации и компиляции артефактов веб-сервиса, необходимых для подключения к сервису.
7. Скомпилируйте клиентский класс.
8. Запустите клиент.

Если вы используете IDE NetBeans для создания сервиса и клиента, среда разработки выполняет задачу `wsimport`.

В следующих разделах эти шаги рассматриваются более подробно.

Требования к конечной точке XML веб-сервиса

Конечные точки XML веб-сервиса должны соответствовать этим требованиям.

- Реализующий класс должен быть аннотирован либо `jakarta.jws.WebService`, либо `jakarta.jws.WebServiceProvider`.
- Реализующий класс может явно ссылаться на SEI через элемент `endpointInterface` аннотации `@WebService`, но не обязательно. Если `endpointInterface` не указан в `@WebService`, SEI неявно определяется для реализующего класса.
- Бизнес-методы реализующего класса должны быть публичными и не должны объявляться `static` или `final`.
- Бизнес-методы, которые доступны для клиентов веб-сервисов, должны быть аннотированы `jakarta.jws.WebMethod`.
- Бизнес-методы, предоставляемые клиентам веб-сервиса, должны иметь параметры и возвращаемые типы, совместимые с Jakarta XML Binding. См. список связывания типов данных Jakarta XML Binding по умолчанию в Типы, поддерживаемые XML веб-сервисами.
- Реализующий класс не должен быть объявлен `final` и не должен быть `abstract`.
- Реализующий класс должен иметь публичный конструктор по умолчанию.

- Реализующий класс не должен определять метод `finalize`.
- Класс реализации может использовать аннотации `jakarta.annotation.PostConstruct` или `jakarta.annotation.PreDestroy` на своих методах для Callback-вызовов событий жизненного цикла.

  Метод `@PostConstruct` вызывается контейнером до того, как реализующий класс начинает отвечать клиентам веб-сервиса.

  Метод `@PreDestroy` вызывается контейнером перед удалением конечной точки.

Кодирование класса реализации конечной точки сервиса

В этом примере класс реализации `Hello` аннотируется как конечная точка веб-сервиса с помощью аннотации `@WebService`. `Hello` объявляет один метод с именем `sayHello`, аннотированный `@WebMethod`, который предоставляет аннотированный метод клиентам веб-сервиса. Метод `sayHello` возвращает приветствие клиенту, используя переданное ему имя для создания приветствия. Класс реализации также должен определять конструктор по умолчанию — публичный, без аргументов.

JAVA

```
package ee.jakarta.tutorial.helloservice;

import jakarta.jws.WebService;
import jakarta.jws.WebMethod;

@WebService
public class Hello {
    private final String message = "Hello, ";

    public Hello() {
    }

    @WebMethod
    public String sayHello(String name) {
        return message + name + ".";
    }
}
```

Сборка, упаковка и развёртывание сервиса

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки и развёртывания приложения `helloservice-war`.

Сборка, упаковка и развёртывание сервиса в IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/jaxws
   ```

4. Выберите каталог `helloservice-war`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `helloservice-war` и выберите **Запуск**.

   Эта команда собирает и упаковывает приложение в WAR-файл, `helloservice-war.war`, расположенный в *`tut-install`*`/examples/jaxws/helloservice-war/target/` и развёртывает этот WAR-файл в GlassFish Server. Он также открывает интерфейс тестирования веб-сервиса по URL, указанному в Тестирование сервиса без клиента.

Вы можете просмотреть WSDL-файл развёрнутого сервиса, запросив URL http://localhost:8080/helloservice-war/HelloService?wsdl в браузере. Теперь вы готовы создать клиента, который обращается к этому сервису.

Сборка, упаковка и развёртывание сервиса с помощью Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/jaxws/helloservice-war/
   ```

3. Введите следующую команду:

   ```
   mvn install
   ```
   SHELL

   Эта команда собирает и упаковывает приложение в WAR-файл `helloservice-war.war`, расположенный в каталоге `target`, а затем развёртывает WAR в GlassFish Server.

Вы можете просмотреть WSDL-файл развёрнутого сервиса, запросив URL http://localhost:8080/helloservice-war/HelloService?wsdl в браузере. Теперь вы готовы создать клиента, который обращается к этому сервису.

Тестирование методов конечной точки веб-сервиса

Сервер GlassFish позволяет тестировать методы конечной точки веб-сервиса.

Тестирование сервиса без клиента

Чтобы проверить метод `sayHello` в `HelloService`, выполните следующие действия.

1. Откройте интерфейс тестирования веб-сервиса, введя следующий URL в веб-браузере:

   ```
   http://localhost:8080/helloservice-war/HelloService?Tester
   ```

2. В разделе Методы введите имя в качестве параметра метода `sayHello`.
3. Нажмите sayHello.

   Вы перейдете на страницу вызова метода `sayHello`.

   В разделе Method returned вы увидите ответ от конечной точки.

Простое клиентское приложение XML веб-сервиса

Класс `HelloAppClient` является консольным клиентом, который обращается к методу `sayHello` в `HelloService`. Этот вызов осуществляется через порт — локальный объект, который действует как прокси для удалённого сервиса. Порт создаётся при разработке с помощью задачи Maven `wsimport`, которая генерирует переносимые артефакты XML веб-сервиса по файлу WSDL.

Кодирование консольного клиента

При вызове удалённых методов порта клиент выполняет эти шаги.

1. Использует сгенерированный класс `helloservice.endpoint.HelloService`, который представляет сервис по URI WSDL-файла развёрнутого сервиса:

```java
import ee.jakarta.tutorial.helloservice.endpoint.HelloService;
import jakarta.xml.ws.WebServiceRef;

public class HelloAppClient {
    @WebServiceRef(wsdlLocation =
      "http://localhost:8080/helloservice-war/HelloService?WSDL")
    private static HelloService service;
    ...
}
```

2. Извлекает прокси для сервиса, также известный как порт, вызывая `getHelloPort` в сервисе:

```java
ee.jakarta.tutorial.helloservice.endpoint.Hello port = service.getHelloPort();
```

   Порт реализует SEI, определённый сервисом.

3. Он вызывает метод `sayHello` порта, передавая строку сервису:

```java
return port.sayHello(arg0);
```

Вот полный исходный код `HelloAppClient.java`, который находится в каталоге *`tut-install`*`/examples/jaxws/hello-appclient/src/main/java/ee/jakarta/tutorial/hello/appclient/`:

```java
package ee.jakarta.tutorial.hello.appclient;

import ee.jakarta.tutorial.helloservice.endpoint.HelloService;
import jakarta.xml.ws.WebServiceRef;

public class HelloAppClient {
    @WebServiceRef(wsdlLocation =
      "http://localhost:8080/helloservice-war/HelloService?WSDL")
    private static HelloService service;

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
       System.out.println(sayHello("world"));
    }

    private static String sayHello(java.lang.String arg0) {
        ee.jakarta.tutorial.helloservice.endpoint.Hello port =
                service.getHelloPort();
        return port.sayHello(arg0);
    }
}
```

Запуск консольного клиента

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска приложения `hello-appclient`. Чтобы собрать клиента, вы должны сначала развернуть `helloservice-war`, как описано в Сборка, упаковка и развёртывание сервиса.

Запуск консольного клиента с IDE NetBeans

1. В меню **Файл** выберите **Открыть проект**.
2. В диалоговом окне **Открыть проект** перейдите к:

```
tut-install/examples/jaxws
```

3. Выберите каталог `hello-appclient` .
4. Нажмите **Открыть проект**.
5. На вкладке **Проекты** кликните правой кнопкой мыши проект `hello-appclient` и выберите **Сборка**.

   Эта команда запускает цель `wsimport` , затем собирает, упаковывает и запускает клиент. Вы увидите выходные данные клиентского приложения на вкладке вывода hello-appclient:

```
--- exec-maven-plugin:1.2.1:exec (run-appclient) @ hello-appclient ---
Hello, world.
```

Запуск консольного клиента с помощью Maven

1. В окне терминала перейдите в:

```
tut-install/examples/jaxws/hello-appclient/
```

2. Введите следующую команду:

```shell
mvn install
```

   Эта команда запускает цель `wsimport` , затем собирает, упаковывает и запускает клиент. Выходные данные клиентского приложения выглядят так:

```
--- exec-maven-plugin:1.2.1:exec (run-appclient) @ hello-appclient ---
Hello, world.
```

Простой веб-клиент XML веб-сервиса

`HelloServlet` — это сервлет, который, подобно Java-клиенту, вызывает метод `sayHello` веб-сервиса. Как и клиентское приложение, он делает этот вызов через порт.

Кодирование сервлета

Чтобы вызвать метод для порта, клиент выполняет эти шаги.

1. Он импортирует конечную точку `HelloService` и аннотацию `@WebServiceRef` :

```java
import ee.jakarta.tutorial.helloservice.endpoint.HelloService;
...
import jakarta.xml.ws.WebServiceRef;
```

2. Он определяет ссылку на веб-сервис, указывая местоположение WSDL:

```java
@WebServiceRef(wsdlLocation =
  "http://localhost:8080/helloservice-war/HelloService?WSDL")
```

3. Он объявляет веб-сервис, а затем определяет приватный метод, который вызывает метод `sayHello` для порта:

```java
private HelloService service;
...
private String sayHello(java.lang.String arg0) {
    ee.jakarta.tutorial.helloservice.endpoint.Hello port =
            service.getHelloPort();
    return port.sayHello(arg0);
}
```

4. В сервлете он вызывает этот приватный метод:

JAVA
```
    out.println("<p>" + sayHello("world") + "</p>");
```

Ниже приведены важные части кода `HelloServlet`. Код находится в каталоге *tut-install*`/examples/jaxws/hello-webclient/src/java/ee/jakarta/tutorial/hello/webclient/`.

JAVA
```
package ee.jakarta.tutorial.hello.webclient;

import ee.jakarta.tutorial.helloservice.endpoint.HelloService;
import java.io.IOException;
import java.io.PrintWriter;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.xml.ws.WebServiceRef;

@WebServlet(name="HelloServlet", urlPatterns={"/HelloServlet"})
public class HelloServlet extends HttpServlet {
    @WebServiceRef(wsdlLocation =
      "http://localhost:8080/helloservice-war/HelloService?WSDL")
    private HelloService service;

    /**
     * Обработка как запросов HTTP GET, так и HTTP POST.
     * @param request запрос сервлета
     * @param response ответ сервлета
     * @throws ServletException в случае специфичной сервлетам ошибки
     * @throws IOException в случае ошибки ввода/вывода
     */
    protected void processRequest(HttpServletRequest request,
            HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {

            out.println("<html lang=\"en\">");
            out.println("<head>");
            out.println("<title>Servlet HelloServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet HelloServlet at " +
                request.getContextPath () + "</h1>");
            out.println("<p>" + sayHello("world") + "</p>");
            out.println("</body>");
            out.println("</html>");
        }
    }

    // методы doGet doPost, которые вызывают processRequest и
    //   getServletInfo

    private String sayHello(java.lang.String arg0) {
        ee.jakarta.tutorial.helloservice.endpoint.Hello port =
                service.getHelloPort();
        return port.sayHello(arg0);
    }
}
```

Запуск веб-клиента

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска приложения `hello-webclient`. Чтобы собрать клиента, вы должны сначала развернуть `helloservice-war`, как описано в Сборка, упаковка и развёртывание сервиса.

Запуск веб-клиента в IDE NetBeans

1. В меню **Файл** выберите **Открыть проект**.
2. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/jaxws
   ```

3. Выберите каталог `hello-webclient`.
4. Нажмите **Открыть проект**.
5. На вкладке **Проекты** кликните правой кнопкой мыши проект `hello-webclient` и выберите **Сборка**.

   Эта задача запускает цель `wsimport`, собирает и упаковывает приложение в WAR-файл, `hello-webclient.war`, расположенный в каталоге `target`, и развёртывает его в GlassFish Server.
6. В веб-браузере введите следующий URL:

   ```
   http://localhost:8080/hello-webclient/HelloServlet
   ```

   Вывод метода `sayHello` появится в окне.

Запуск веб-клиента с помощью Maven

1. В окне терминала перейдите в:

   ```
   tut-install/examples/jaxws/hello-webclient/
   ```

2. Введите следующую команду:

   ```
   mvn install
   ```
   SHELL

   Эта команда запускает цель `wsimport`, затем собирает и упаковывает приложение в WAR-файл, `hello-webclient.war`, расположенный в каталоге `target`, WAR-файл затем развёртывается в GlassFish Server.
3. В веб-браузере введите следующий URL:

   ```
   http://localhost:8080/hello-webclient/HelloServlet
   ```

   Вывод метода `sayHello` появится в окне.

Типы, поддерживаемые Jakarta XML Web Services

XML веб-сервисы делегируют Jakarta XML Binding отображение объектных типов Java в определения XML и обратно. Разработчикам приложений не нужно знать подробности этого отображения, но следует помнить, что не каждый класс на языке Java может использоваться в качестве параметра метода или типа возвращаемого значения в XML веб-сервисах.

В следующих разделах объясняется преобразование типов данных XML в объекты Java и обратно:

- Отображение схемы XML на Java объекты
- Отображение Java объектов на схему XML

Отображение схемы XML на Java объекты

Язык Java предоставляет более богатый набор типов данных, чем схема XML. Таблица 31-1 приводит отображение типов данных XML на объектные типы Java.

Таблица 31-1 Сопоставление типов данных XML с объектными типами Java

Тип схемы XML	Тип данных Java
xsd:string	java.lang.String
xsd:integer	java.math.BigInteger
xsd:int	int
xsd.long	long
xsd:short	short
xsd:decimal	java.math.BigDecimal
xsd:float	float
xsd:double	double
xsd:boolean	boolean
xsd:byte	byte
xsd:QName	javax.xml.namespace.QName
xsd:dateTime	javax.xml.datatype.XMLGregorianCalendar
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]
xsd:unsignedInt	long
xsd:unsignedShort	int
xsd:unsignedByte	short
xsd:time	javax.xml.datatype.XMLGregorianCalendar
xsd:date	javax.xml.datatype.XMLGregorianCalendar
xsd:g	javax.xml.datatype.XMLGregorianCalendar
xsd:anySimpleType	java.lang.Object
xsd:anySimpleType	java.lang.String
xsd:duration	javax.xml.datatype.Duration
xsd:NOTATION	javax.xml.namespace.QName

Отображение Java объектов на схему XML

Таблица 31-2 показывает отображение по умолчанию классов Java на типы схемы XML.

Таблица 31-2 Сопоставление классов Java с типами данных XML

Класс Java	Тип данных XML
`java.lang.String`	`xs:string`
`java.math.BigInteger`	`xs:integer`
`java.math.BigDecimal`	`xs:decimal`
`java.util.Calendar`	`xs:dateTime`
`java.util.Date`	`xs:dateTime`
`javax.xml.namespace.QName`	`xs:QName`
`java.net.URI`	`xs:string`
`javax.xml.datatype.XMLGregorianCalendar`	`xs:anySimpleType`
`javax.xml.datatype.Duration`	`xs:duration`
`java.lang.Object`	`xs:anyType`
`java.awt.Image`	`xs:base64Binary`
`jakarta.activation.DataHandler`	`xs:base64Binary`
`javax.xml.transform.Source`	`xs:base64Binary`
`java.util.UUID`	`xs:string`

Совместимость веб-сервисов и Jakarta XML Web Services

Jakarta XML Web Services поддерживает базовый профиль совместимости веб-сервисов (WS-I) версии 1.1. Базовый профиль WS-I — это документ, в котором разъясняются спецификации SOAP 1.1 и WSDL 1.1 для обеспечения совместимости с SOAP. Ссылки, связанные с WS-I, см. в Дополнительные сведения о Jakarta XML Web Services.

Для поддержки WS-I Basic Profile Version 1.1 среда выполнения JAX-WS поддерживает кодирование doc/literal и rpc/literal для сервисов, статических портов, динамических прокси-серверов и интерфейса динамического вызова (DII).

Дополнительная информация о Jakarta XML Web Services

Дополнительные сведения о Jakarta XML Web Services и связанных технологиях см.

- Спецификация Jakarta XML Web Services 3.0:
  https://jakarta.ee/specifications/xml-web-services/3.0/
- Jakarta XML Web Services:
  https://eclipse-ee4j.github.io/metro-jax-ws/

- Simple Object Access Protocol (SOAP) 1.2 W3C Note:
  https://www.w3.org/TR/soap/
- Web Services Description Language (WSDL) 1.1 W3C Note:
  https://www.w3.org/TR/wsdl
- WS-I Basic Profile 1.2 and 2.0:
  http://www.ws-i.org

Глава 32. Создание RESTful веб-сервисов с Jakarta REST

В этой главе описывается архитектура REST, RESTful веб-сервисы и Jakarta RESTful Web Services.

Jakarta REST позволяет разработчикам легко создавать RESTful веб-сервисы с использованием Java.

Что такое RESTful веб-сервисы?

RESTful веб-сервисы — это слабосвязные, легковесные веб-сервисы, которые особенно хорошо подходят для создания API для клиентов, разбросанных по Интернету. REST - это архитектурный стиль клиент-серверного приложения, основанный на передаче представлений ресурсов посредством запросов и ответов. В архитектурном стиле REST данные и функциональные возможности считаются ресурсами и доступны с использованием универсальных идентификаторов ресурсов (URI), обычно ссылок в Интернете. Ресурсы представлены документами и обрабатываются с помощью набора простых, чётко определённых операций.

Например, ресурсом REST могут быть текущие погодные условия для города. Представлением этого ресурса может быть документ XML, файл изображения или страница HTML. Клиент может получить конкретное представление, изменить ресурс путём обновления своих данных или полностью удалить ресурс.

Архитектурный стиль REST разработан для использования протокола связи без сохранения состояния, обычно HTTP. В стиле архитектуры REST клиенты и серверы обмениваются представлениями ресурсов, используя стандартизированный интерфейс и протокол.

Следующие принципы позволяют RESTful приложениям оставаться простыми, легковесными и быстрыми:

- Идентификация ресурса через URI. RESTful веб-сервис предоставляет набор ресурсов, которые идентифицируют цели взаимодействия со своими клиентами. Ресурсы идентифицируются с помощью URI, которые обеспечивают глобальное адресное пространство для обнаружения ресурсов и служб. Смотрите Аннотация @Path и шаблоны пути URI для получения дополнительной информации.
- Унифицированный интерфейс: управление ресурсами осуществляется с помощью фиксированного набора из четырёх операций создания, чтения, обновления, удаления: PUT, GET, POST и DELETE. PUT создаёт новый ресурс, который затем может быть удалён с помощью DELETE. GET извлекает текущее состояние ресурса в некотором представлении. POST переносит новое состояние на ресурс. Смотрите Ответ на HTTP-методы и запросы для получения дополнительной информации.
- Самодокументируемые сообщения: ресурсы отделены от их представления, чтобы к их содержимому можно было обращаться в различных форматах, таких как HTML, XML, обычный текст, PDF, JPEG, JSON и другие. Метаданные о ресурсе доступны и используются, например, для управления кэшированием, обнаружения ошибок передачи, согласования соответствующего формата представления и выполнения аутентификации или контроля доступа. См. Ответ на HTTP-методы и запросы и Использование провайдеров сущностей для сопоставления HTTP-ответа и запроса тел сущностей для получения дополнительной информации.
- Взаимодействие с состоянием через ссылки: ни одно из взаимодействий с ресурсом не имеет состояния. То есть сообщения запроса являются автономными и не зависят друг от друга. Взаимодействия с состоянием основаны на концепции явной передачи состояния. Существует несколько методов обмена состояниями, таких как перезапись URI, cookies и скрытые поля формы. Состояние может быть встроено в ответные сообщения, чтобы указывать на действительные будущие состояния взаимодействия. Дополнительные сведения см. в разделе Использование провайдеров сущностей для сопоставления HTTP-ответов и запросов и Извлечения параметров запроса в документе Обзор Jakarta REST.

Создание класса корневого ресурса RESTful

Корневые классы ресурсов — это POJO, которые либо аннотированы `@Path`, либо имеют хотя бы один метод, аннотированный `@Path`, или указатель метода запроса, такие как `@GET`, `@PUT`, `@POST` или `@DELETE`. Методы ресурса — это методы класса ресурса, аннотированные указателем метода запроса. В этом разделе объясняется, как использовать Jakarta REST для аннотирования классов Java при создании RESTful веб-сервисов.

Разработка RESTful веб-сервисов с помощью Jakarta REST

Jakarta REST — это API Java, предназначенный для упрощения разработки приложений, использующих архитектуру REST.

Jakarta REST API использует аннотации Java для упрощения разработки веб-сервисов RESTful. Разработчики аннотируют классы Java аннотациями Jakarta REST для обозначения ресурсов и действий, которые могут быть выполнены с этими ресурсами. Аннотации Jakarta REST являются аннотациями времени выполнения, поэтому reflection во время выполнения будет генерировать вспомогательные классы и артефакты для ресурса. В архиве приложений Jakarta EE, содержащем классы ресурсов Jakarta REST, будут настроены ресурсы, сгенерированы вспомогательные классы и артефакты, а ресурсы будут доступны клиентам путём развёртывания архива на сервере Jakarta EE.

Таблица 32-1 перечисляет некоторые аннотации Java, определённые Jakarta REST, с кратким описанием их использования. Дополнительную информацию о Jakarta REST API можно найти по ссылке https://jakarta.ee/specifications/platform/9/apidocs/.

Таблица 32-1 Резюме аннотаций Jakarta REST

Аннотация	**Описание**
`@Path`	Значение аннотации `@Path` представляет собой относительный путь URI, указывающий, где будет размещён класс Java: например, `/helloworld`. Вы также можете встраивать переменные в URI для создания шаблона пути URI. Например, вы можете запросить имя пользователя и передать его приложению в качестве переменной в URI: `/helloworld/{username}`.
`@GET`	Аннотация `@GET` является указателем метода запроса и соответствует одноимённому HTTP-методу. Аннотированный таким образом метод Java будет обрабатывать HTTP GET-запросы. Поведение ресурса определяется HTTP-методом, на который отвечает ресурс.
`@POST`	Аннотация `@POST` является указателем метода запроса и соответствует одноимённому HTTP-методу. Аннотированный таким образом метод Java будет обрабатывать HTTP POST-запросы. Поведение ресурса определяется HTTP-методом, на который отвечает ресурс.
`@PUT`	Аннотация `@PUT` является указателем метода запроса и соответствует одноимённому HTTP-методу. Аннотированный таким образом метод Java будет обрабатывать HTTP PUT-запросы. Поведение ресурса определяется HTTP-методом, на который отвечает ресурс.
`@DELETE`	Аннотация `@DELETE` является указателем метода запроса и соответствует одноимённому HTTP-методу. Аннотированный таким образом метод Java будет обрабатывать HTTP DELETE-запросы. Поведение ресурса определяется HTTP-методом, на который отвечает ресурс.

Аннотация	Описание
`@HEAD`	Аннотация `@HEAD` является указателем метода запроса и соответствует одноимённому HTTP-методу. Аннотированный таким образом метод Java будет обрабатывать HTTP HEAD-запросы. Поведение ресурса определяется HTTP-методом, на который отвечает ресурс.
`@OPTIONS`	Аннотация `@OPTIONS` является указателем метода запроса и соответствует одноимённому HTTP-методу. Аннотированный таким образом метод Java будет обрабатывать HTTP OPTIONS-запросы. Поведение ресурса определяется HTTP-методом, на который отвечает ресурс.
`@PATCH`	Аннотация `@PATCH` является указателем метода запроса и соответствует одноимённому HTTP-методу. Аннотированный таким образом метод Java будет обрабатывать HTTP PATCH-запросы. Поведение ресурса определяется HTTP-методом, на который отвечает ресурс.
`@PathParam`	Аннотация `@PathParam` — это тип параметра, который может быть извлечён для использования в классе ресурсов. Параметры пути URI извлекаются из URI запроса, а имена параметров соответствуют именам переменных шаблона пути URI, указанным в аннотации на уровне класса `@Path`.
`@QueryParam`	Аннотация `@QueryParam` — это тип параметра, который может быть извлечён для использования в классе ресурсов. Параметры запроса извлекаются из URI запроса.
`@Consumes`	Аннотация `@Consumes` используется для указания типов MIME, которые может принимать ресурс и которые были отправлены клиентом.
`@Produces`	Аннотация `@Produces` используется для указания типов MIME, которые ресурс может создавать и отправлять обратно клиенту: например, `«text/plain»`.
`@Provider`	Аннотация `@Provider` используется для всего, что представляет интерес для среды выполнения Jakarta REST, например для `MessageBodyReader` и `MessageBodyWriter`. Для HTTP-запросов `MessageBodyReader` используется для сопоставления тела объекта HTTP-запроса с параметрами метода. Возвращаемое значение сопоставляется с телом объекта HTTP-ответа с помощью `MessageBodyWriter`. Если приложению необходимо предоставить дополнительные метаданные, такие как заголовки HTTP или другой код состояния, метод может вернуть `Response`-обёртку (wrapper) сущности и который может быть создан с использованием `Response.ResponseBuilder`.
`@ApplicationPath`	Аннотация `@ApplicationPath` используется для назначения URL приложению. Путь, указанный в `@ApplicationPath`, является базовым URI для всех URI ресурсов, указанных в аннотациях `@Path` в классе ресурсов. Вы можете применить `@ApplicationPath` только к дочернему классу `jakarta.ws.rs.core.Application`.

Обзор приложения Jakarta REST

Следующий простой пример кода задаёт корневой класс ресурсов, который использует аннотации Jakarta REST:

JAVA

```java
package ee.jakarta.tutorial.hello;

import jakarta.ws.rs.Consumes;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.PUT;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.Context;
import jakarta.ws.rs.core.UriInfo;

/**
 * Root resource (exposed at "helloworld" path)
 */
@Path("helloworld")
public class HelloWorld {
    @Context
    private UriInfo context;

    /** Creates a new instance of HelloWorld */
    public HelloWorld() {
    }

    /**
     * Retrieves representation of an instance of helloWorld.HelloWorld
     * @return an instance of java.lang.String
     */
    @GET
    @Produces("text/html")
    public String getHtml() {
        return "<html lang=\"en\"><body><h1>Hello, World!!</h1></body></html>";
    }
}
```

В следующих разделах описаны аннотации, используемые в этом примере.

- Значение аннотации `@Path` представляет собой относительный путь URI. В предыдущем примере класс Java будет размещён по пути URI `/helloworld`. Это чрезвычайно простое использование аннотации `@Path` со статическим путём URI. Переменные могут быть встроены в URI. Шаблоны пути URI — это URI с переменными, встроенными в синтаксис URI.
- Аннотация `@GET` — это указатель метода запроса, как и `@POST`, `@PUT`, `@DELETE` и `@HEAD`, определённые Jakarta REST и соответствующие одноимённым HTTP-методам. В этом примере аннотированный Java-метод будет обрабатывать HTTP GET-запросы. Поведение ресурса определяется HTTP-методом, на который отвечает ресурс.
- Аннотация `@Produces` используется для указания типов MIME, которые ресурс может создавать и отправлять обратно клиенту. В этом примере метод Java создаёт представления с типом MIME `"text/html"`.
- Аннотация `@Consumes` используется для указания типов MIME, которые может принимать ресурс и которые были отправлены клиентом. Пример можно изменить, чтобы установить сообщение, возвращаемое методом `getHtml`, как показано в следующем примере кода:

JAVA

```java
@POST
@Consumes("text/plain")
public void postHtml(String message) {
    // Сохранение сообщения
}
```

Аннотация @Path и шаблоны пути URI

Аннотация `@Path` идентифицирует шаблон пути URI, на который отвечает ресурс, и указывается на уровне класса или метода ресурса. Значение аннотации `@Path` представляет собой частичный шаблон пути URI относительно базового URI сервера, на котором развёрнут ресурс, корневого контекста приложения и шаблона URL, которому отвечает среда выполнения Jakarta REST.

Шаблоны пути URI — это URI с переменными, встроенными в синтаксис URI. Эти переменные подставляются во время выполнения, чтобы ресурс отвечал на запрос на основе замещённого URI. Переменные обозначаются фигурными скобками (`{` и `}`). Например, посмотрите на следующую аннотацию `@Path` :

JAVA

```
@Path("/users/{username}")
```

В этом примере пользователю предлагается ввести своё имя, а затем отвечает веб-сервис Jakarta REST, настроенный на запросы по шаблону пути URI. Например, если пользователь вводит имя пользователя «Galileo», веб-сервис отвечает на следующий URL:

```
http://example.com/users/Galileo
```

Чтобы получить значение имени пользователя, можно использовать аннотацию `@PathParam` для параметра метода запроса, как показано в следующем примере кода:

JAVA

```
@Path("/users/{username}")
public class UserResource {

    @GET
    @Produces("text/xml")
    public String getUser(@PathParam("username") String userName) {
        ...
    }
}
```

По умолчанию переменная URI должна соответствовать регулярному выражению `"[^/]+?"` . Эту переменную можно настроить, указав другое регулярное выражение после имени переменной. Например, если имя пользователя должно состоять только из строчных и прописных буквенно-цифровых символов, переопределите регулярное выражение по умолчанию в определении переменной:

JAVA

```
@Path("users/{username: [a-zA-Z][a-zA-Z_0-9]*}")
```

В этом примере переменная `username` будет соответствовать только именам пользователей, которые начинаются с одной заглавной или строчной буквы, после которых идёт любое количество буквенно-цифровых символов и символов подчеркивания. Если имя пользователя не соответствует этому шаблону, клиенту будет отправлен ответ 404 (Not Found).

Значение `@Path` не обязательно должно иметь начальную или конечную косую черту (/). Среда выполнения Jakarta REST анализирует шаблоны пути URI независимо от того, содержат ли они начальную или конечную косую черту.

Шаблон пути URI имеет одну или несколько переменных, каждое имя переменной заключено в фигурные скобки: `{` для начала имени переменной и `}` для её завершения. В предыдущем примере `username` является именем переменной. Во время выполнения ресурс, настроенный для ответа на предыдущий шаблон пути URI, попытается обработать данные URI, которые соответствуют расположению `{username}` в URI, в качестве переменных данных для `username` .

Например, если вы хотите развернуть ресурс, отвечающий шаблону пути URI `http://example.com/myContextRoot/resources/{name1}/{name2}/`, сначала необходимо развернуть приложение на сервере Jakarta EE, отвечающее на запросы к URI `http://example.com/myContextRoot` а затем аннотировать ресурс аннотацией `@Path`:

JAVA

```
@Path("/{name1}/{name2}/")
public class SomeResource {
    ...
}
```

В этом примере шаблон URL для вспомогательного сервлета Jakarta REST, указанный в `web.xml`, является шаблоном по умолчанию:

XML

```
<servlet-mapping>
      <servlet-name>jakarta.ws.rs.core.Application</servlet-name>
      <url-pattern>/resources/*</url-pattern>
</servlet-mapping>
```

Имя переменной может использоваться более одного раза в шаблоне пути URI.

Если символ в значении переменной будет конфликтовать с зарезервированными символами URI, конфликтующий символ должен быть URL-кодирован. Например, пробелы в значении переменной должны быть заменены на `%20`.

При определении шаблонов пути URI, будьте осторожны, чтобы результирующий URI после замены был валидным.

Таблица 32-2 перечисляет некоторые примеры переменных шаблона пути URI и того, как URI разрешаются после замены. В примерах используются следующие имена и значения переменных:

- `name1`: `james`
- `name2`: `gatz`
- `name3`:
- `location`: `Main%20Street`
- `question`: `why`

Значение переменной `name3` — пустая строка.

Таблица 32-2 Примеры шаблонов пути URI

Шаблон пути URI	**URI после замены**
http://example.com/{name1}/{name2}/	http://example.com/james/gatz/
http://example.com/{question}/{question}/{question}/	http://example.com/why/why/why/
http://example.com/maps/{location}	http://example.com/maps/Main%20Street
http://example.com/{name3}/home/	http://example.com//home/

Ответ на HTTP-методы и запросы

Поведение ресурса определяется методами HTTP (обычно GET, POST, PUT или DELETE), на которые отвечает ресурс.

Обозначение указателя метода запроса

Аннотации указателя метода запроса — это аннотации среды выполнения, определённые Jakarta REST, соответствующие одноимённым HTTP-методам. В файле класса ресурсов методы HTTP отображаются на методы Java с помощью аннотаций указателя метода запроса. Поведение ресурса определяется тем, на какой метод HTTP он отвечает. Jakarta REST определяет набор обозначений методов запроса для общих методов HTTP GET, POST, PUT, DELETE и HEAD. Вы также можете создать свои собственные пользовательские обозначения методов запроса. Создание кастомных указателей методов запроса выходит за рамки этого документа.

В следующем примере показано использование метода PUT для создания или обновления контейнера хранения:

JAVA

```
@PUT
public Response putContainer() {
    System.out.println("PUT CONTAINER " + container);

    URI uri = uriInfo.getAbsolutePath();
    Container c = new Container(container, uri.toString());

    Response r;
    if (!MemoryStore.MS.hasContainer(c)) {
        r = Response.created(uri).build();
    } else {
        r = Response.noContent().build();
    }

    MemoryStore.MS.createContainer(c);
    return r;
}
```

По умолчанию среда выполнения Jakarta REST будет автоматически поддерживать методы HEAD и OPTIONS, если они не реализованы явно. Для HEAD среда выполнения вызовет реализованный метод GET, если он присутствует, и проигнорирует объект ответа, если он установлен. Для OPTIONS заголовок ответа `Allow` будет установлен на набор HTTP-методов, поддерживаемых ресурсом. Кроме того, среда выполнения Jakarta REST вернёт документ на языке определения веб-приложений (WADL), описывающий ресурс. См. https://www.w3.org/Submission/wadl/ для получения дополнительной информации.

Методы, помеченные указателями методов запроса, должны возвращать `void`, тип Java или объект `jakarta.ws.rs.core.Response`. Несколько параметров могут быть извлечены из URI аннотациями `@PathParam` и `@QueryParam`, как описано в Извлечение параметров запроса. За преобразование между типами Java и телом объекта отвечает провайдер сущностей, такой как `MessageBodyReader` или `MessageBodyWriter`. Методы, которым необходимо предоставить дополнительные метаданные с ответом, должны возвращать объект класса `Response`. Класс `ResponseBuilder` предоставляет удобный способ создания объекта `Response` с использованием конструктора (builder). Методы HTTP PUT и POST ожидают тело HTTP-запроса, поэтому вы должны использовать `MessageBodyReader` для методов, которые отвечают на запросы PUT и POST.

И `@PUT`, и `@POST` можно использовать для создания или обновления ресурса. POST может означать что угодно, поэтому при использовании POST семантика определяется приложением. PUT имеет чётко определённую семантику. При использовании PUT для создания клиент объявляет URI для вновь созданного ресурса.

PUT имеет очень чёткую семантику для создания и обновления ресурса. Представление, отправляемое клиентом, должно быть тем же представлением, которое получено с использованием GET, с тем же типом MIME. PUT не позволяет частично обновлять ресурс, что является распространённой ошибкой при попытке использовать метод PUT. Распространённым шаблоном приложения является использование POST для создания ресурса и возврата ответа `201` с заголовком местоположения, значением которого является URI для вновь созданного ресурса. В этом шаблоне веб-сервис объявляет URI для вновь созданного ресурса.

Использование провайдеров сущностей для сопоставления HTTP-ответа и запроса тел сущностей

Поставщики сущностей предоставляют сервисы отображения между представлениями и связанными с ними типами Java. Есть два типа провайдеров сущностей: `MessageBodyReader` и `MessageBodyWriter`. Для HTTP-запросов `MessageBodyReader` используется для сопоставления тела объекта HTTP-запроса с параметрами метода. Возвращаемое значение сопоставляется с телом объекта HTTP-ответа с помощью `MessageBodyWriter`. Если приложению необходимо предоставить дополнительные метаданные, такие как заголовки HTTP или другой код состояния, метод может вернуть `Response`, оборачивающий сущность и который может быть построен с помощью `Response.ResponseBuilder`.

Таблица 32-3 показывает стандартные типы, которые автоматически поддерживаются для тел объектов HTTP-запросов и ответов. Если не выбирается один из следующих стандартных типов, то требуется написать провайдер сущностей.

Таблица 32-3. Типы, поддерживаемые для объектов запросов и ответов HTTP

Тип Java	**Поддерживаемые типы медиа**
`byte[]`	Все типы MIME (`*/*`)
`java.lang.String`	Все текстовые типы MIME (`text/*`)
`java.io.InputStream`	Все типы MIME (`*/*`)
`java.io.Reader`	Все типы MIME (`*/*`)
`java.io.File`	Все типы MIME (`*/*`)
`jakarta.activation.DataSource`	Все типы MIME (`*/*`)
`javax.xml.transform.Source`	XML типы MIME (`text/xml`, `application/xml` и `application/*+xml`)
`jakarta.xml.bind.JAXBElement` и классы Jakarta XML Binding, предоставляемые приложением	XML типы MIME (`text/xml`, `application/xml` и `application/*+xml`)
`MultivaluedMap<String, String>`	Содержимое формы (`application/x-www-form-urlencoded`)

Тип Java	Поддерживаемые типы медиа
`StreamingOutput`	Все типы MIME (`/`), только `MessageBodyWriter`

В следующем примере показано, как использовать `MessageBodyReader` с аннотациями `@Consumes` и `@Provider` :

JAVA

```
@Consumes("application/x-www-form-urlencoded")
@Provider
public class FormReader implements MessageBodyReader<NameValuePair> { }
```

В следующем примере показано, как использовать `MessageBodyWriter` с аннотациями `@Produces` и `@Provider` :

JAVA

```
@Produces("text/html")
@Provider
public class FormWriter implements
        MessageBodyWriter<Hashtable<String, String>> { }
```

В следующем примере показано, как использовать `ResponseBuilder` :

JAVA

```
@GET
public Response getItem() {
    System.out.println("GET ITEM " + container + " " + item);

    Item i = MemoryStore.MS.getItem(container, item);
    if (i == null)
        throw new NotFoundException("Item not found");
    Date lastModified = i.getLastModified().getTime();
    EntityTag et = new EntityTag(i.getDigest());
    ResponseBuilder rb = request.evaluatePreconditions(lastModified, et);
    if (rb != null)
        return rb.build();

    byte[] b = MemoryStore.MS.getItemData(container, item);
    return Response.ok(b, i.getMimeType()).
            lastModified(lastModified).tag(et).build();
}
```

Использование @Consumes и @Produces для настройки запросов и ответов

Информация, отправляемая ресурсу и затем возвращаемая клиенту, указывается как тип MIME в заголовках HTTP-запроса или ответа. Вы можете указать типы MIME, на которые ресурс может реагировать или которые может создавать, используя следующие аннотации:

- `jakarta.ws.rs.Consumes`
- `jakarta.ws.rs.Produces`

По умолчанию класс ресурсов может отвечать и генерировать все типы MIME, указанные в заголовках HTTP-запроса и ответа.

Аннотация @Produces

Аннотация `@Produces` используется для указания типов MIME, которые ресурс может создавать и отправлять обратно клиенту. Если `@Produces` применяется на уровне класса, все методы в ресурсе могут создавать указанные типы MIME по умолчанию. При использовании на уровне метода аннотации переопределяют любые аннотации `@Produces` уровня класса.

Если никакие методы в ресурсе не могут создать тип MIME в запросе клиента, среда выполнения Jakarta REST отвечает ошибкой HTTP «406 Not Acceptable».

Значением `@Produces` является массив `String` типов MIME или разделённый запятыми список констант `MediaType`. Например:

JAVA

```
@Produces({"image/jpeg,image/png"})
```

В следующем примере показано, как применить `@Produces` как на уровне класса, так и на уровне метода:

JAVA

```
@Path("/myResource")
@Produces("text/plain")
public class SomeResource {
    @GET
    public String doGetAsPlainText() {
        ...
    }

    @GET
    @Produces("text/html")
    public String doGetAsHtml() {
        ...
    }
}
```

Метод `doGetAsPlainText` по умолчанию использует тип MIME аннотации `@Produces` на уровне класса. Аннотация `@Produces` метода `doGetAsHtml` переопределяет параметр `@Produces` уровня класса и указывает, что метод может создавать HTML, а не простой текст.

`@Produces` может также использовать константы, определённые в классе `jakarta.ws.rs.core.MediaType`, чтобы указать тип MIME. Например, указание `MediaType.APPLICATION_XML` эквивалентно указанию `«application/xml»`.

JAVA

```
@Produces(MediaType.APPLICATION_XML)
@GET
public Customer getCustomer() { ... }
```

Если класс ресурсов способен создавать более одного типа MIME, выбранный метод ресурса будет соответствовать наиболее подходящему типу MIME из числа объявленных клиентом. Более конкретно, заголовок `Accept` HTTP-запроса объявляет, что является наиболее приемлемым. Например, если заголовок `Accept` имеет значение `Accept: text/plain`, будет вызван метод `doGetAsPlainText`. В качестве альтернативы, если заголовок `Accept` равен `Accept: text/plain;q=0.9, text/html`, который объявляет, что клиент может принимать типы MIME `text/plain` и `text/html`, но предпочитающие последнее, будет вызван метод `doGetAsHtml`.

В одном объявлении `@Produces` может быть объявлено несколько типов MIME. В следующем примере кода показано, как это сделать:

JAVA

```java
@Produces({"application/xml", "application/json"})
public String doGetAsXmlOrJson() {
    ...
}
```

Метод `doGetAsXmlOrJson` будет вызван, если допустим любой из типов MIME — `application/xml` или `application/json`. Если оба одинаково приемлемы, будет выбран тот, который указан первым. Предыдущие примеры явно указывают на типы MIME для ясности. Можно ссылаться на константы, которые исключают возможность опечатки. Дополнительные сведения см. в документации по API для значений константных полей `jakarta.ws.rs.core.MediaType`.

Аннотация @Consumes

Аннотация `@Consumes` используется для указания того, какие типы MIME ресурс может принимать от клиента. Если `@Consumes` применяется на уровне класса, все методы ответа по умолчанию принимают указанные типы MIME. При использовании на уровне метода `@Consumes` переопределяет любые аннотации `@Consumes` уровня класса.

Если ресурс не может использовать тип MIME запроса клиента, среда выполнения Jakarta REST отвечает ошибкой HTTP 415 («Unsupported Media Type»).

Значением `@Consumes` является массив `String` допустимых типов MIME или разделённый запятыми список констант `MediaType`. Например:

JAVA

```java
@Consumes({"text/plain,text/html"})
```

Что эквивалентно:

JAVA

```java
@Consumes({MediaType.TEXT_PLAIN,MediaType.TEXT_HTML})
```

В следующем примере показано, как применить `@Consumes` как на уровне класса, так и на уровне метода:

JAVA

```java
@Path("/myResource")
@Consumes("multipart/related")
public class SomeResource {
    @POST
    public String doPost(MimeMultipart mimeMultipartData) {
        ...
    }

    @POST
    @Consumes("application/x-www-form-urlencoded")
    public String doPost2(FormURLEncodedProperties formData) {
        ...
    }
}
```

Метод `doPost` по умолчанию использует тип MIME аннотации `@Consumes` на уровне класса. Метод `doPost2` переопределяет аннотацию `@Consumes` уровня класса, указывая, что он может принимать данные формы в кодировке URL.

Если никакие методы ресурсов не могут ответить на запрошенный тип MIME, клиенту возвращается ошибка HTTP 415 («Unsupported Media Type»).

Пример `HelloWorld`, рассмотренный ранее в этом разделе, можно изменить, чтобы установить сообщение с помощью `@Consumes`, как показано в следующем примере кода:

JAVA

```java
@POST
@Consumes("text/html")
public void postHtml(String message) {
    // Сохранение сообщения
}
```

В этом примере метод Java будет использовать представления, идентифицированные типом MIME `text/plain`. Обратите внимание, что метод ресурса возвращает `void`. Это означает, что представление не возвращается и что будет возвращён ответ с кодом состояния HTTP 204 («No Content»).

Извлечение параметров запроса

Параметры метода ресурса могут быть аннотированы аннотациями для извлечения информации из запроса. В предыдущем примере было представлено использование параметра `@PathParam` для извлечения параметра пути из компонента пути URL запроса, который соответствует пути, объявленному в `@Path`.

Могут быть извлечены следующие типы параметров для использования в классе ресурсов:

- Запрос
- Путь URI
- Форма
- Cookie
- Заголовок
- Матрица

Параметры запроса извлекаются из URI и указываются аннотацией `jakarta.ws.rs.QueryParam` в аргументах параметра метода. В следующем примере демонстрируется использование `@QueryParam` для извлечения параметров запроса из компонента `Query` URL запроса:

JAVA

```java
@Path("smooth")
@GET
public Response smooth(
        @DefaultValue("2") @QueryParam("step") int step,
        @DefaultValue("true") @QueryParam("min-m") boolean hasMin,
        @DefaultValue("true") @QueryParam("max-m") boolean hasMax,
        @DefaultValue("true") @QueryParam("last-m") boolean hasLast,
        @DefaultValue("blue") @QueryParam("min-color") ColorParam minColor,
        @DefaultValue("green") @QueryParam("max-color") ColorParam maxColor,
        @DefaultValue("red") @QueryParam("last-color") ColorParam lastColor
        ) { ... }
```

Если параметр запроса `step` существует в URI запроса, значение `step` будет извлечено и проанализировано как 32-разрядное целое число со знаком и присвоено параметру метода `step`. Если `step` не существует, параметру метода `step` будет присвоено значение по умолчанию, равное 2, как объявлено в аннотации `@DefaultValue`. Если значение `step` не может быть приведено к 32-разрядному целое число со знаком, возвращается ответ HTTP 400 («Client Error»).

Пользовательские типы Java могут использоваться в качестве параметров запроса. В следующем примере кода показан класс `ColorParam`, использованный в предыдущем примере параметра запроса:

JAVA

```
public class ColorParam extends Color {
    public ColorParam(String s) {
        super(getRGB(s));
    }

    private static int getRGB(String s) {
        if (s.charAt(0) == '#') {
            try {
                Color c = Color.decode("0x" + s.substring(1));
                return c.getRGB();
            } catch (NumberFormatException e) {
                throw new WebApplicationException(400);
            }
        } else {
            try {
                Field f = Color.class.getField(s);
                return ((Color)f.get(null)).getRGB();
            } catch (Exception e) {
                throw new WebApplicationException(400);
            }
        }
    }
}
```

Конструктор для `ColorParam` принимает один параметр `String`.

И `@QueryParam`, и `@PathParam` могут использоваться только на следующих типах Java.

- Все примитивные типы, кроме `char`.
- Все классы-обёртки примитивных типов, кроме `Character`.
- Любой класс с конструктором, который принимает один аргумент `String`.
- Любой класс со статическим методом с именем `valueOf(String)`, который принимает один аргумент `String`.
- `List<T>`, `Set<T>` или `SortedSet<T>`, где T соответствует уже перечисленным критериям. Иногда параметры могут содержать более одного значения для одного и того же имени. Если это так, эти типы могут быть использованы для получения всех значений.

Если `@DefaultValue` не используется вместе с `@QueryParam`, а параметр запроса отсутствует в запросе, это значение будет пустой коллекцией для `List`, `Set` или `SortedSet`, null для других типов объектов и значением по умолчанию для примитивных типов.

Параметры пути URI извлекаются из URI запроса, а имена параметров соответствуют именам переменных шаблона пути URI, указанным в аннотации на уровне класса `@Path`. Параметры URI задаются с помощью свойства `jakarta.ws.rs.PathParam` в аргументах параметра метода. В следующем примере показано, как использовать переменные `@Path` и аннотацию `@PathParam` в методе:

JAVA

```
@Path("/{username}")
public class MyResourceBean {
    ...
    @GET
    public String printUsername(@PathParam("username") String userId) {
        ...
    }
}
```

В предыдущем фрагменте имя переменной шаблона пути URI `username` указывается в качестве параметра для метода `printUsername`. Аннотация `@PathParam` устанавливается для переменной с именем `username`. Во время выполнения перед вызовом `printUsername` значение `username` извлекается из URI и приводится к `String`. Результирующая `String` затем доступна методу в виде переменной `userId`.

Если переменная шаблона пути URI не может быть приведена к указанному типу, среда выполнения Jakarta REST возвращает клиенту ошибку HTTP 400 («Bad Request»). Если аннотация `@PathParam` не может быть приведена к указанному типу, среда выполнения Jakarta REST отвечает ошибкой HTTP 404 («Not Found»).

Параметр `@PathParam` и другие аннотации параметров (`@MatrixParam`, `@HeaderParam`, `@CookieParam` и `@FormParam`) подчиняются тем же правилам, что и `@QueryParam`.

Параметры Cookie, обозначенные параметрами `jakarta.ws.rs.CookieParam`, извлекают информацию из файлов cookie, объявленных в HTTP-заголовках, связанных с файлами cookie. Параметры заголовка, обозначенные параметрами `jakarta.ws.rs.HeaderParam`, извлекают информацию из заголовков HTTP. Параметры матрицы, обозначенные параметрами `jakarta.ws.rs.MatrixParam`, извлекают информацию из сегментов пути URL.

Параметры формы, указанные параметрами `jakarta.ws.rs.FormParam`, извлекают информацию из представления запроса, имеющего тип MIME `application/x-www-form-urlencoded` и соответствует кодировке, указанной в формах HTML, как описано в https://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.1. Этот параметр очень полезен для извлечения информации, отправляемой POST в HTML-формах.

В следующем примере извлекается параметр формы `name` из данных формы POST:

JAVA

```java
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(@FormParam("name") String name) {
    // Сохранение сообщения
}
```

Чтобы получить общее отображение (Map) имён параметров и их значений из запроса, используйте следующий код:

JAVA

```java
@GET
public String get(@Context UriInfo ui) {
    MultivaluedMap<String, String> queryParams = ui.getQueryParameters();
    MultivaluedMap<String, String> pathParams = ui.getPathParameters();
}
```

Следующий метод извлекает имена и значения параметров заголовка и cookie в отображение (Map):

JAVA

```java
@GET
public String get(@Context HttpHeaders hh) {
    MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();
    Map<String, Cookie> pathParams = hh.getCookies();
}
```

В общем, `@Context` может использоваться для получения контекстных типов Java, связанных с запросом или ответом.

Для параметров формы можно сделать следующее:

JAVA

```java
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(MultivaluedMap<String, String> formParams) {
    // Сохранение сообщения
}
```

Настройка приложений Jakarta REST

Приложение Jakarta REST состоит как минимум из одного класса ресурсов, упакованного в WAR-файл. Базовый URI, из которого ресурсы приложения отвечают на запросы, можно установить одним из двух способов:

- С аннотацией `@ApplicationPath` в дочернем классе `jakarta.ws.rs.core.Application` упакуйте в WAR-файл
- Использование тега `servlet-mapping` в дескрипторе развёртывания WAR `web.xml`

Настройка приложения Jakarta REST с использованием унаследованного от Application класса

Создайте подкласс `jakarta.ws.rs.core.Application`, чтобы вручную настроить среду, в которой выполняются ресурсы REST, определённые в ваших классах ресурсов, включая базовый URI. Добавьте аннотацию `@ApplicationPath` на уровне класса, чтобы установить базовый URI.

JAVA

```java
@ApplicationPath("/webapi")
public class MyApplication extends Application { ... }
```

В предыдущем примере базовый URI установлен в `/webapi`, что означает, что все ресурсы, определённые в приложении, относятся к `/webapi`.

По умолчанию, будут обрабатываться все ресурсы в архиве. Переопределите метод `getClasses`, чтобы вручную зарегистрировать классы ресурсов приложения в среде выполнения Jakarta REST.

JAVA

```java
@Override
public Set<Class> getClasses() {
    final Set<Class> classes = new HashSet<>();
    // регистрация корневого ресурса
    classes.add(MyResource.class);
    return classes;
}
```

Настройка базового URI в web.xml

Базовый URI для приложения Jakarta REST может быть установлен с помощью тега `servlet-mapping` в дескрипторе развёртывания `web.xml`, используя имя класса `Application` в качестве сервлета.

XML

```xml
<servlet-mapping>
    <servlet-name>jakarta.ws.rs.core.Application</servlet-name>
    <url-pattern>/webapi/*</url-pattern>
</servlet-mapping>
```

Этот параметр также переопределяет путь, заданный `@ApplicationPath` при использовании дочернего класса `Application`.

XML

```
<servlet-mapping>
   <servlet-name>com.example.rest.MyApplication</servlet-name>
   <url-pattern>/services/*</url-pattern>
</servlet-mapping>
```

Примеры приложений Jakarta REST

В этом разделе содержится введение в создание, развёртывание и запуск собственных приложений Jakarta REST. В этом разделе демонстрируются шаги, необходимые для создания, сборки, развёртывания и тестирования очень простого веб-приложения, использующего аннотации Jakarta REST.

Создание простого RESTful веб-сервиса

В этом разделе объясняется, как использовать IDE NetBeans для создания RESTful веб-сервиса с использованием архетипа Maven. Архетип создаёт каркас для приложения, и остаётся только реализовать соответствующий метод.

Вы можете найти версию этого приложения в *tut-install*`/examples/jaxrs/hello/` .

Создание RESTful веб-сервиса с IDE NetBeans

1. Убедитесь, что вы установили архетипы, как описано в Установка архетипов учебника.
2. В IDE NetBeans создайте веб-приложение, используя архетип Maven `jaxrs-service-archetype` . Этот архетип создаёт очень простое веб-приложение "Hello, World".
   a. В меню «Файл» выберите «Новый проект».
   b. В "Категориях" выберите Maven. В "Проектах" выберите «Проект из архетипа». Нажмите кнопку Далее.
   c. В поле Поиск введите `jaxrs-service` , выберите `jaxrs-service-archetype` и нажмите кнопку Далее.
   d. В поле «Имя проекта» введите `HelloWorldApplication` , установите «Местоположение проекта», установите для имени пакета значение `ee.jakarta.tutorial.hello` и нажмите «Готово».

   Проект создан.
3. В `HelloWorld.java` найдите метод `getHtml()` . Замените комментарий `//TODO` следующим текстом, чтобы в итоге получилось:

   JAVA

   ```
   @GET
   @Produces("text/html")
   public String getHtml() {
       return "<html lang=\"en\"><body><h1>Hello, World!!</body></h1></html>";
   }
   ```

   

   Поскольку созданный тип MIME — это HTML, вы можете использовать теги HTML в выражении return.
4. Кликните правой кнопкой мыши проект `HelloWorldApplication` на панели «Проекты» и выберите «Выполнить».

   Эта команда собирает и развёртывает приложение в GlassFish Server.
5. В браузере откройте следующий URL:

   ```
   http://localhost:8080/HelloWorldApplication/HelloWorldApplication
   ```

   Откроется окно браузера и отобразит возвращаемое значение `Hello, World!!`

Другие примеры приложений, демонстрирующих развёртывание и запуск приложений Jakarta REST с использованием IDE NetBeans, см. в разделе Пример rsvp и Your First Cup: Введение в платформу Jakarta EE на https://eclipse-ee4j.github.io/jakartaee-firstcup/toc.html. Вы также можете просмотреть учебные материалы на сайте IDE NetBeans, например, под названием «Начало работы с RESTful Web Services» по ссылке https://netbeans.apache.org/kb/docs/websvc/rest.html. В этом руководстве содержится раздел о создании приложения CRUD из базы данных. Создание, чтение, обновление и удаление (CRUD) — четыре основные операции для работы с реляционными базами данных.

Приложение rsvp

Пример приложения `rsvp`, расположенный в каталоге *`tut-install`*`/examples/jaxrs/rsvp/`, позволяет приглашённым на мероприятие указать, что приглашение принято. Мероприятия и приглашённые на них люди, ответы людей на приглашения сохраняются в Apache Derby с помощью Jakarta Persistence. Ресурсы Jakarta REST в `rsvp` отображаются в сессионном компоненте без сохранения состояния.

Компоненты приложения rsvp

В примере приложения `rsvp` присутствуют три Enterprise-бина: `rsvp.ejb.ConfigBean`, `rsvp.ejb.StatusBean` и `rsvp.ejb.ResponseBean`.

`ConfigBean` — это сессионный компонент-синглтон, который инициализирует данные в базе данных.

`StatusBean` предоставляет ресурс Jakarta REST для отображения текущего состояния всех приглашенных к событию. Шаблон пути URI объявляется сначала в классе, а затем в методе `getEvent`:

JAVA

```
@Stateless
@Named
@Path("/status")
public class StatusBean {
    ...
    @GET
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    @Path("{eventId}/")
    public Event getEvent(@PathParam("eventId") Long eventId) {
         ...
    }
}
```

Комбинация двух аннотаций `@Path` приводит к следующему шаблону пути URI:

JAVA

```
@Path("/status/{eventId}/")
```

Переменная пути URI `eventId` — это переменная `@PathParam` в методе `getEvent`, которая отвечает на HTTP GET-запросы и снабжена аннотацией `@GET`. Переменная `eventId` используется для поиска всех текущих ответов в базе данных для конкретного мероприятия.

`ResponseBean` предоставляет ресурс Jakarta REST для настройки ответа приглашённого лица на определённое мероприятие. Шаблон пути URI для `ResponseBean` объявлен следующим образом:

JAVA

```
@Path("/{eventId}/{inviteId}")
```

В шаблоне пути объявлены две переменные пути URI: `eventId` и `inviteId`. Как и в `StatusBean`, `eventId` является уникальным идентификатором для конкретного мероприятия. Каждый приглашённый на это мероприятие имеет уникальный идентификатор для приглашения `inviteId`. Обе эти переменные пути

используются в двух методах Jakarta REST: в `getResponse` и `putResponse` объекта `ResponseBean`. Метод `getResponse` отвечает на HTTP GETзапросы и отображает текущий ответ приглашённого и форму для изменения ответа.

Класс `ee.jakarta.tutorial.rsvp.rest.RSVPApplication` определяет путь корневого приложения для ресурсов, применяя аннотацию `jakarta.ws.rs.ApplicationPath` на уровне класса.

JAVA

```
@ApplicationPath("/webapi")
public class RsvpApplication extends Application {
}
```

Приглашённый, желающий изменить свой ответ, выбирает новый ответ и отправляет данные формы, которые обрабатываются как HTTP POST-запрос методом `putResponse`. Новый ответ извлекается из HTTP POST-запроса и сохраняется в виде строки `userResponse`. Метод `putResponse` использует `userResponse`, `eventId` и `inviteId` для изменения ответа приглашённого в базе данных.

Мероприятия, люди и их ответы в `rsvp` инкапсулированы в сущностях Jakarta Persistence. Объекты `rsvp.entity.Event`, `rsvp.entity.Person` и `rsvp.entity.Response` соответственно представляют мероприятия, приглашённых и их ответы.

Класс `rsvp.util.ResponseEnum` объявляет перечислимый тип, представляющий все возможные состояния ответа, которые может иметь приглашённый.

Веб-приложение также включает два Managed-бина CDI — `StatusManager` и `EventManager`, — которые используют клиентский API Jakarta REST для вызова ресурсов, представленных в `StatusBean` и `ResponseBean`. Для получения информации о том, как клиентский API используется в `rsvp`, см. Клиентский API в примере rsvp.

Запуск приложения rsvp

И IDE NetBeans, и Maven можно использовать для развёртывания и запуска примера приложения `rsvp`.

Запуск приложения rsvp в IDE NetBeans

1. Если сервер базы данных ещё не запущен, запустите его, следуя инструкциям в Запуск и остановка Apache Derby.
2. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
3. В меню **Файл** выберите **Открыть проект**.
4. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/jaxrs
   ```

5. Выберите каталог `rsvp`.
6. Нажмите **Открыть проект**.
7. На вкладке **Проекты** кликните правой кнопкой мыши проект `rsvp` и выберите пункт **Запуск**.

   Проект будет скомпилирован, скомпонован и развёрнут в GlassFish Server. Откроется окно веб-браузера по следующему URL:

   ```
   http://localhost:8080/rsvp/index.xhtml
   ```

8. В окне веб-браузера нажмите на ссылку Статус мероприятия для Duke's Birthday event.

Вы увидите текущих приглашённых и их ответы.

9. Кликните текущий ответ одного из приглашённых в столбце «Статус» таблицы, выберите новый ответ и нажмите «Обновить свой статус».

   Новый статус приглашённого теперь должен отображаться в таблице и их статусах ответов.

Запуск приложения rsvp с помощью Maven

1. Если сервер базы данных ещё не запущен, запустите его, следуя инструкциям в Запуск и остановка Apache Derby.
2. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
3. В окне терминала перейдите в:

   ```
   tut-install/examples/jaxrs/rsvp/
   ```

4. Введите следующую команду:

   ```
   mvn install
   ```
   SHELL

   Эта команда собирает, компонует и развёртывает `rsvp` в GlassFish Server.

5. Откройте окно веб-браузера по следующему URL:

   ```
   http://localhost:8080/rsvp/
   ```

6. В окне веб-браузера нажмите на ссылку Статус мероприятия для Duke’s Birthday event.

   Вы увидите текущих приглашённых и их ответы.

7. Кликните текущий ответ одного из приглашённых в столбце «Статус» таблицы, выберите новый ответ и нажмите «Обновить свой статус».

   Новый статус приглашённого теперь должен отображаться в таблице и их статусах ответов.

Примеры из реального мира

Большинство сайтов блогов используют RESTful веб-сервисы. Эти сайты включают загрузку файлов XML в формате RSS или Atom, которые содержат списки ссылок на другие ресурсы. Другие веб-сайты и веб-приложения, которые используют REST-подобные интерфейсы разработчика для данных, включают Twitter и Amazon S3 (Simple Storage Service). Список корзин и объектов Amazon S3, их создание и извлечение возможно с использованием HTTP-интерфейса в стиле REST или SOAP-интерфейса. Примеры, которые поставляются с Jersey, включают пример службы хранения с RESTful интерфейсом.

Дополнительная информация о Jakarta REST

Дополнительные сведения о RESTful веб-сервисах и Jakarta REST см. в разделе

- "Fielding Dissertation: Chapter 5: Representational State Transfer (REST)": https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- RESTful Web Services Леонарда Ричардсона и Сэма Руби доступны в O'Reilly Media по ссылке https://www.oreilly.com/library/view/restful-web-services/9780596529260/
- Спецификация Jakarta RESTful Web Services 3.0: https://jakarta.ee/specifications/restful-ws/3.0/

- Проект Jersey:
  https://eclipse-ee4j.github.io/jersey/

Глава 33. Доступ к ресурсам REST с помощью клиентского API Jakarta REST

В этой главе описывается клиентское API Jakarta REST и приведены примеры доступа к ресурсам REST с помощью Java.

Jakarta REST предоставляет клиентский API для доступа к ресурсам REST из других приложений Java.

Обзор клиентского API

Клиентское API Jakarta REST предоставляет высокоуровневый API для доступа к любым ресурсам REST, а не только к сервисам Jakarta REST. Клиентский API определён в пакете `jakarta.ws.rs.client`.

Создание базового клиентского запроса с использованием клиентского API

Следующие шаги необходимы для доступа к ресурсу REST с помощью клиентского API.

1. Получите объект интерфейса `jakarta.ws.rs.client.Client`.
2. Настройте объект `Client`, указав цель (target).
3. Создайте запрос для цели.
4. Вызовите запрос.

Клиентский API спроектирован так, чтобы свободно, с объединёнными в цепочку вызовами методов настройки и отправки запроса к ресурсу REST уместиться всего в несколько строк кода.

JAVA

```
Client client = ClientBuilder.newClient();
String name = client.target("http://example.com/webapi/hello")
        .request(MediaType.TEXT_PLAIN)
        .get(String.class);
```

В этом примере объект клиента сначала создаётся путём вызова метода `jakarta.ws.rs.client.ClientBuilder.newClient`. Затем запрос конфигурируется и вызывается цепочкой вызовов методов одной строкой кода. Метод `Client.target` устанавливает цель с указанным URI. Метод `jakarta.ws.rs.client.WebTarget.request` устанавливает тип MIME для возвращаемой сущности. Метод `jakarta.ws.rs.client.Invocation.Builder.get` вызывает сервис с помощью HTTP `GET`, установив тип возвращаемой сущности в `String`.

Получение объекта клиента

Интерфейс `Client` определяет действия и инфраструктуру, необходимые клиенту REST для использования RESTful веб-сервиса. Объекты `Client` получены путём вызова метода `ClientBuilder.newClient`.

JAVA

```
Client client = ClientBuilder.newClient();
```

Используйте метод `close`, чтобы закрыть объекты `Client` после выполнения всех вызовов для цели:

JAVA

```
Client client = ClientBuilder.newClient();
...
client.close();
```

Объекты `Client` являются тяжеловесными объектами. По соображениям производительности ограничьте количество объектов `Client` в приложении, так как инициализация и уничтожение этих объектов может оказаться дорогостоящим для среды выполнения.

Установка цели (target) клиента

Целевой объект клиента, ресурс REST по определённому URI, представлен объектом интерфейса `jakarta.ws.rs.client.WebTarget`. Объект `WebTarget` получается вызовом метода `Client.target` с передачей URI целевого ресурса REST.

JAVA

```
Client client = ClientBuilder.newClient();
WebTarget myResource = client.target("http://example.com/webapi");
```

Для сложных ресурсов REST может быть полезно создать несколько объектов `WebTarget`. В следующем примере базовая цель используется для создания нескольких других целей, которые представляют различные услуги, предоставляемые ресурсом REST.

JAVA

```
Client client = ClientBuilder.newClient();
WebTarget base = client.target("http://example.com/webapi");
// WebTarget at http://example.com/webapi/read
WebTarget read = base.path("read");
// WebTarget at http://example.com/webapi/write
WebTarget write = base.path("write");
```

Метод `WebTarget.path` создаёт новый объект `WebTarget`, добавляя URI текущей цели с указанным путём.

Установка параметров пути в цели

Параметры пути в клиентских запросах могут быть указаны как параметры шаблона URI, аналогичные параметрам шаблона, используемым при определении URI ресурса в сервисе Jakarta REST. Параметры шаблона указываются путём помещения переменной шаблона в фигурные скобки (`{}`). Вызовите метод `resolveTemplate` для замены `{username}`, а затем вызовите метод `queryParam`, чтобы добавить другую переменную для передачи.

JAVA

```
WebTarget myResource = client.target("http://example.com/webapi/read")
        .path("{userName}")
        .resolveTemplate("userName", "janedoe")
        .queryParam("chapter", "1");
// http://example.com/webapi/read/janedoe?chapter=1
Response response = myResource.request(...).get();
```

Вызов запроса

После применения всех параметров конфигурации к объекту цели вызовите один из методов `WebTarget.request`, чтобы начать формирование запроса. Обычно это достигается путём передачи в `WebTarget.request` приемлемого типа MIME для запроса либо в виде строки типа MIME, либо с использованием одной из констант в `jakarta.ws.rs.core.MediaType`. Метод `WebTarget.request` возвращает объект `jakarta.ws.rs.client.Invocation.Builder` — вспомогательного объекта, который предоставляет методы для подготовки клиентского запроса.

JAVA

```
Client client = ClientBuilder.newClient();
WebTarget myResource = client.target("http://example.com/webapi/read");
Invocation.Builder builder = myResource.request(MediaType.TEXT_PLAIN);
```

Использование константы `MediaType` эквивалентно использованию строки, определяющей тип MIME.

```java
Invocation.Builder builder = myResource.request("text/plain");
```

После установки типа MIME вызовите запрос, вызвав один из методов объекта `Invocation.Builder` , который соответствует типу HTTP-запроса, ожидаемого целевым ресурсом REST. Это методы:

- `get()`
- `post()`
- `delete()`
- `put()`
- `head()`
- `options()`

Например, если целевой ресурс REST предназначен для HTTP GET-запроса, вызовите метод `Invocation.Builder.get` . Тип возвращаемого значения должен соответствовать сущности, возвращаемой целевым ресурсом REST.

```java
Client client = ClientBuilder.newClient();
WebTarget myResource = client.target("http://example.com/webapi/read");
String response = myResource.request(MediaType.TEXT_PLAIN)
        .get(String.class);
```

Если целевой ресурс REST ожидает HTTP-запрос POST, вызовите метод `Invocation.Builder.post` .

```java
Client client = ClientBuilder.newClient();
StoreOrder order = new StoreOrder(...);
WebTarget myResource = client.target("http://example.com/webapi/write");
TrackingNumber trackingNumber = myResource.request(MediaType.APPLICATION_XML)
                                  .post(Entity.xml(order), TrackingNumber.class);
```

В предыдущем примере возвращаемый тип является пользовательским классом и извлекается путём установки типа в методе `Invocation.Builder.post(Entity<?> entity, Class<T> responseType)` ,

Если возвращаемый тип является коллекцией, используйте `jakarta.ws.rs.core.GenericType<T>` в качестве параметра типа ответа, где `T` — это тип элементов коллекции:

```java
List<StoreOrder> orders = client.target("http://example.com/webapi/read")
        .path("allOrders")
        .request(MediaType.APPLICATION_XML)
        .get(new GenericType<List<StoreOrder>>() {});
```

Этот пример показывает, как вызывается цепочка методов клиентского API, чтобы упростить настройку и вызов запросов.

Использование клиентского API в примере приложения Jakarta REST

Примеры `rsvp` и `customer` используют клиентский API для вызова сервисов Jakarta REST. В этом разделе описывается, как каждый пример приложения использует клиентский API.

Клиентский API в примере rsvp

Приложение `rsvp` позволяет пользователям отвечать на приглашения на мероприятия, используя ресурсы Jakarta REST, как описано в Пример приложения rsvp. Веб-приложение использует клиентский API во вспомогательных компонентах CDI для взаимодействия с ресурсами сервиса, а веб-интерфейс Facelets отображает результаты.

Вспомогательный бин CDI `StatusManager` извлекает все текущие мероприятия в системе. Объект клиента, используемый во вспомогательном компоненте, создаётся в конструкторе:

JAVA

```
public StatusManager() {
    this.client = ClientBuilder.newClient();
}
```

Метод `StatusManager.getEvents` возвращает коллекцию всех текущих мероприятий в системе, вызывая ресурс по ссылке http://localhost:8080/rsvp/webapi/status/all, который возвращает XML-документ с записями для каждого мероприятия. Клиентский API автоматически демаршализирует XML и создаёт объект `List<Event>`.

JAVA

```
public List<Event> getEvents() {
    List<Event> returnedEvents = null;
    try {
        returnedEvents = client.target(baseUri)
                .path("all")
                .request(MediaType.APPLICATION_XML)
                .get(new GenericType<List<Event>>() {
        });
        if (returnedEvents == null) {
            logger.log(Level.SEVERE, "Returned events null.");
        } else {
            logger.log(Level.INFO, "Events have been returned.");
        }
    } catch (WebApplicationException ex) {
        throw new WebApplicationException(Response.Status.NOT_FOUND);
    }
    ...
    return returnedEvents;
}
```

Метод `StatusManager.changeStatus` используется для изменения ответа участника. Он создаёт HTTP-запрос `POST` к сервису с новым ответом. Тело запроса является XML-документом.

JAVA

```
public String changeStatus(ResponseEnum userResponse,
        Person person, Event event) {
    String navigation;
    try {
        logger.log(Level.INFO,
                "changing status to {0} for {1} {2} for event ID {3}.",
                new Object[]{userResponse,
                    person.getFirstName(),
                    person.getLastName(),
                    event.getId().toString()});
            client.target(baseUri)
                    .path(event.getId().toString())
                    .path(person.getId().toString())
                    .request(MediaType.APPLICATION_XML)
                    .post(Entity.xml(userResponse.getLabel()));
        navigation = "changedStatus";
    } catch (ResponseProcessingException ex) {
        logger.log(Level.WARNING, "couldn''t change status for {0} {1}",
                new Object[]{person.getFirstName(),
                    person.getLastName()});
        logger.log(Level.WARNING, ex.getMessage());
        navigation = "error";
    }
    return navigation;
}
```

Клиентский API в примере customer

Приложение `customer` сохраняет данные клиента в базе данных и предоставляет ресурс в виде XML, как описано в Приложение customer. Ресурс сервиса предоставляет методы, которые создают клиентов и возвращают список всех клиентов. Веб-приложение Facelets выступает в качестве клиента для ресурса сервиса с формой для создания клиентов и отображения списка клиентов в таблице.

Сессионный компонент `CustomerBean` без сохранения состояния использует клиентское API Jakarta REST для взаимодействия с ресурсом сервиса. Метод `CustomerBean.createCustomer` принимает объект сущности `Customer`, созданный формой Facelets, и выполняет вызов POST для URI сервиса.

JAVA

```
public String createCustomer(Customer customer) {
    if (customer == null) {
        logger.log(Level.WARNING, "customer is null.");
        return "customerError";
    }
    String navigation;
    Response response =
            client.target("http://localhost:8080/customer/webapi/Customer")
            .request(MediaType.APPLICATION_XML)
            .post(Entity.entity(customer, MediaType.APPLICATION_XML),
                    Response.class);
    if (response.getStatus() == Status.CREATED.getStatusCode()) {
        navigation = "customerCreated";
    } else {
        logger.log(Level.WARNING,
                "couldn''t create customer with id {0}. Status returned was {1}",
                new Object[]{customer.getId(), response.getStatus()});
        FacesContext context = FacesContext.getCurrentInstance();
        context.addMessage(null,
                new FacesMessage("Could not create customer."));
        navigation = "customerError";
    }
    return navigation;
}
```

Сущность XML-запроса создаётся путём вызова метода `Invocation.Builder.post`, передачи нового объекта `Entity`, содержащего объект `Customer`, и указания типа MIME `MediaType.APPLICATION_XML`.

Метод `CustomerBean.retrieveCustomer` извлекает объект сущности `Customer` из сервиса, добавляя идентификатор клиента в URI сервиса.

JAVA

```
public String retrieveCustomer(String id) {
    String navigation;
    Customer customer =
            client.target("http://localhost:8080/customer/webapi/Customer")
            .path(id)
            .request(MediaType.APPLICATION_XML)
            .get(Customer.class);
    if (customer == null) {
        navigation = "customerError";
    } else {
        navigation = "customerRetrieved";
    }
    return navigation;
}
```

Метод `CustomerBean.retrieveAllCustomers` извлекает коллекцию клиентов в виде объекта `List<Customer>`. Затем этот список отображается в виде таблицы в веб-приложении Facelets.

JAVA

```
public List<Customer> retrieveAllCustomers() {
    List<Customer> customers =
            client.target("http://localhost:8080/customer/webapi/Customer")
            .path("all")
            .request(MediaType.APPLICATION_XML)
            .get(new GenericType<List<Customer>>() {
            });
    return customers;
}
```

Поскольку тип ответа является коллекцией, метод `Invocation.Builder.get` вызывается путём передачи нового объекта `GenericType<List<Customer>>`.

Дополнительные возможности клиентского API

IВ этом разделе описаны некоторые дополнительные возможности клиентского API Jakarta REST.

Настройка запроса клиента

Дополнительные параметры конфигурации могут быть добавлены в запрос клиента после его создания, но до его вызова.

Установка заголовков сообщений в клиентском запросе

Вы можете установить заголовки HTTP для запроса, вызвав метод `Invocation.Builder.header`.

JAVA

```
Client client = ClientBuilder.newClient();
WebTarget myResource = client.target("http://example.com/webapi/read");
String response = myResource.request(MediaType.TEXT_PLAIN)
        .header("myHeader", "The header value")
        .get(String.class);
```

Если нужно установить несколько заголовков для запроса, вызовите метод `Invocation.Builder.headers` и передайте в метод `jakarta.ws.rs.core.MultivaluedMap` с парами имя-значение заголовков HTTP. Вызов метода `headers` заменяет все существующие заголовки на заголовки, предоставленные в объекте `MultivaluedMap`.

JAVA

```
Client client = ClientBuilder.newClient();
WebTarget myResource = client.target("http://example.com/webapi/read");
MultivaluedMap<String, Object> myHeaders =
    new MultivaluedMap<>("myHeader", "The header value");
myHeaders.add(...);
String response = myResource.request(MediaType.TEXT_PLAIN)
        .headers(myHeaders)
        .get(String.class);
```

Интерфейс `MultivaluedMap` позволяет указать несколько значений для данного ключа.

JAVA

```
MultivaluedMap<String, Object> myHeaders =
    new MultivaluedMap<String, Object>();
List<String> values = new ArrayList<>();
values.add(...);
myHeaders.add("myHeader", values);
```

Установка файлов cookie в запросе клиента

Вы можете добавить cookies HTTP в запрос, вызвав метод `Invocation.Builder.cookie`, который принимает пару имя-значение в качестве параметров.

JAVA

```
Client client = ClientBuilder.newClient();
WebTarget myResource = client.target("http://example.com/webapi/read");
String response = myResource.request(MediaType.TEXT_PLAIN)
        .cookie("myCookie", "The cookie value")
        .get(String.class);
```

Класс `jakarta.ws.rs.core.Cookie` инкапсулирует атрибуты cookie HTTP, включая имя, значение, путь, домен и версию спецификации RFC cookie. В следующем примере объект `Cookie` настроен парой имя-значение, путём и доменом.

JAVA

```
Client client = ClientBuilder.newClient();
WebTarget myResource = client.target("http://example.com/webapi/read");
Cookie myCookie = new Cookie("myCookie", "The cookie value",
    "/webapi/read", "example.com");
String response = myResource.request(MediaType.TEXT_PLAIN)
        .cookie(myCookie)
        .get(String.class);
```

Добавление фильтров к клиенту

Пользовательские фильтры можно зарегистрировать с помощью клиентского запроса или ответа, полученного от целевого ресурса. Чтобы зарегистрировать классы фильтров при создании объекта `Client`, вызовите метод `Client.register`.

JAVA

```
Client client = ClientBuilder.newClient().register(MyLoggingFilter.class);
```

В предыдущем примере для всех вызовов, которые используют этот объект `Client`, зарегистрирован фильтр `MyLoggingFilter`.

Вы также можете зарегистрировать классы фильтров на цели, вызвав `WebTarget.register`.

JAVA
```
Client client = ClientBuilder.newClient().register(MyLoggingFilter.class);
WebTarget target = client.target("http://example.com/webapi/secure")
        .register(MyAuthenticationFilter.class);
```

В предыдущем примере фильтры `MyLoggingFilter` и `MyAuthenticationFilter` присоединяются к вызову.

Классы фильтров запросов и ответов реализуют интерфейсы `jakarta.ws.rs.client.ClientRequestFilter` и `jakarta.ws.rs.client.ClientResponseFilter` соответственно. Оба этих интерфейса определяют метод `filter`. Все фильтры должны быть аннотированы `jakarta.ws.rs.ext.Provider`.

Следующий класс представляет собой фильтр журнала для клиентских запросов и ответов.

JAVA
```
@Provider
public class MyLoggingFilter implements ClientRequestFilter,
        ClientResponseFilter {
    static final Logger logger = Logger.getLogger(...);

    // реализация метода ClientRequestFilter.filter
    @Override
    public void filter(ClientRequestContext requestContext)
            throws IOException {
        logger.log(...);
        ...
    }

    // реализация метода ClientResponseFilter.filter
    @Override
    public void filter(ClientRequestContext requestContext,
           ClientResponseContext responseContext) throws IOException {
        logger.log(...);
        ...
    }
}
```

Если требуется остановить вызов во время активности фильтра, вызывается метод `abortWith` контекстного объекта и передаётся объект `jakarta.ws.rs.core.Response` из фильтра.

JAVA
```
@Override
public void filter(ClientRequestContext requestContext) throws IOException {
    ...
    Response response = new Response();
    response.status(500);
    requestContext.abortWith(response);
}
```

Асинхронные вызовы в клиентском API

В сетевых приложениях проблемы сети могут повлиять на воспринимаемую производительность приложения, особенно при длительных или сложных сетевых вызовах. Асинхронная обработка помогает предотвратить блокировку и позволяет лучше использовать ресурсы приложения.

Метод клиентского API Jakarta REST `Invocation.Builder.async` используется при создании клиентского запроса для указания, что вызов сервиса должен выполняться асинхронно. Асинхронный вызов немедленно возвращает управление вызывающей стороне вместе с объектом типа `java.util.concurrent.Future<T>` (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html?is-external=true) (часть API параллелизма Java SE) и

с типом, установленным на возвращаемый тип вызова сервиса. Объекты `Future<T>` имеют методы для проверки завершения асинхронного вызова, получения окончательного результата, отмены вызова и проверки отмены вызова.

В следующем примере показано, как вызвать асинхронный запрос к ресурсу.

JAVA

```
Client client = ClientBuilder.newClient();
WebTarget myResource = client.target("http://example.com/webapi/read");
Future<String> response = myResource.request(MediaType.TEXT_PLAIN)
        .async()
        .get(String.class);
```

Использование кастомных Callback-вызовов в асинхронных вызовах

Интерфейс `InvocationCallback` определяет два метода: `complete` и `failed`, которые вызываются, когда асинхронный вызов завершается соответственно либо успешно, либо неудачей. Объект `InvocationCallback` может быть зарегистрирован при формировании запроса.

В следующем примере показано, как зарегистрировать объект Callback-вызова при асинхронном вызове.

JAVA

```
Client client = ClientBuilder.newClient();
WebTarget myResource = client.target("http://example.com/webapi/read");
Future<Customer> fCustomer = myResource.request(MediaType.TEXT_PLAIN)
        .async()
        .get(new InvocationCallback<Customer>() {
            @Override
            public void completed(Customer customer) {
            // Выполнение действий с объектом
            }
            @Override
             public void failed(Throwable throwable) {
            // обработка ошибки
            }
        });
```

Использование реактивного подхода в асинхронных вызовах

Использование кастомных Callback-вызовов в асинхронных вызовах легко в простых случаях и в случаях множества независимых вызовов. В случае вложенных вызовов использование кастомных Callback-вызовов становится очень трудным для реализации, отладки и обслуживания.

Jakarta REST задаёт новый тип вызывающего объекта, называемый `RxInvoker`, и реализацию этого типа по умолчанию — `CompletionStageRxInvoker`. Новый метод `rx` используется как в следующем примере:

JAVA

```
CompletionStage<String> csf = client.target("forecast/{destination}")
    .resolveTemplate("destination","mars")
    .request().rx().get(String.class);
csf.thenAccept(System.out::println);
```

В этом примере создаётся асинхронная обработка интерфейса `CompletionStage<String>`, которая ожидает его завершения и отображения результата. Возвращённый `CompletionStage` может затем использоваться только для получения результата, как показано в приведённом выше примере, или может быть объединён с другими этапами завершения для упрощения и улучшения обработки асинхронных задач.

Использование серверных событий

Технология серверных событий (Server-sent Events — SSE) используется для асинхронной отправки уведомлений клиенту по стандартным протоколам HTTP или HTTPS. Клиенты могут подписаться на уведомления о событиях, которые отправляются на сервер. Сервер генерирует события и отправляет уведомления об этих событиях клиентам, которые подписаны на получение уведомлений. Соединение с односторонним каналом связи устанавливается клиентом. Как только соединение установлено, сервер отправляет уведомления клиенту всякий раз, когда доступны новые данные.

Канал связи, установленный клиентом, действует до тех пор, пока клиент не закроет соединение. Он же используется сервером для отправки уведомлений о событиях.

Обзор SSE API

API SSE определено в пакете `jakarta.ws.rs.sse`, который включает интерфейсы `SseEventSink`, `SseEvent`, `Sse` и `SseEventSource`. Чтобы принимать подключения и отправлять уведомления одному или нескольким клиентам, вставьте `SseEventSink` в метод ресурса, который создаёт ответы типа MIME `text/event-stream`.

В следующем примере показано, как принимать подключения SSE и отправлять уведомления клиентам:

JAVA

```
@GET
@Path("eventStream")
@Produces(MediaType.SERVER_SENT_EVENTS)
public void eventStream(@Context SseEventSink eventSink, @Context Sse sse) {
    executor.execute(() -> {
        try (SseEventSink sink = eventSink) {
            eventSink.send(sse.newEvent("event1"));
            eventSink.send(sse.newEvent("event2"));
            eventSink.send(sse.newEvent("event3"));
        }
    });
}
```

`SseEventSink` инъецируется в метод ресурса, а основное клиентское соединение остаётся открытым и используется для отправки уведомлений. Соединение сохраняется до тех пор, пока клиент не отключится от сервера. Метод `send` возвращает объект `CompletionStage<T>`, который указывает, что действие асинхронной отправки сообщения клиенту включено.

Уведомления, передающиеся клиентам, могут содержать такие подробности, как `event`, `data`, `id`, `retry` и `comment`.

Широковещательная рассылка с использованием SSE

Широковещательная рассылка — это действие по отправке уведомления одновременно нескольким клиентам. SSE API Jakarta REST предоставляет `SseBroadcaster` для регистрации всех объектов `SseEventSink` и отправки событий на все зарегистрированные выходы событий. Жизненный цикл и область видимости `SseBroadcaster` полностью контролируются приложениями, а не средой выполнения Jakarta REST. В следующем примере показано использование широковещательной рассылки:

JAVA

```java
@Path("/")
@Singleton
public class SseResource {
    @Context
    private Sse sse;

    private volatile SseBroadcaster sseBroadcaster;

    @PostConstruct
    public init() {
        this.sseBroadcaster = sse.newBroadcaster();
    }

    @GET
    @Path("register")
    @Produces(MediaType.SERVER_SENT_EVENTS)
        public void register(@Context SseEventSink eventSink) {
        eventSink.send(sse.newEvent("welcome!"));
        sseBroadcaster.register(eventSink);
    }

    @POST
    @Path("broadcast")
    @Consumes(MediaType.MULTIPART_FORM_DATA)
    public void broadcast(@FormParam("event") String event) {
        sseBroadcaster.broadcast(sse.newEvent(event));
    }
}
```

Аннотация `@Singleton` определена для класса ресурса и делает невозможным создание нескольких объектов класса. Метод `register` используется для добавления нового `SseEventSink`. Метод `broadcast` используется для отправки уведомления SSE всем зарегистрированным клиентам.

Ожидание получение уведомлений о серверных событиях

SSE Jakarta REST предоставляет интерфейс `SseEventSource` для подписки клиента на уведомления. Клиент может получать асинхронные уведомления о входящих событиях, вызывая один из методов `subscribe` в `jakarta.ws.rs.sse.SseEventSource`.

В следующем примере показано, как использовать API `SseEventSource` для открытия соединения SSE и чтения сообщений за период:

JAVA

```java
WebTarget target = client.target("http://...");
try (SseEventSource source = SseEventSource.target(target).build()) {
    source.register(System.out::println);
    source.open();
    Thread.sleep(500); // Проверка событий каждые 500 мс
    source.close();
} catch (InterruptedException e) {
    // обработка ошибки
}
```

Глава 34. Jakarta REST: дополнительные темы и пример

Jakarta RESTful Web Services (Jakarta REST) предназначены для упрощения разработки приложений, использующих архитектуру REST. В этой главе описаны дополнительные возможности Jakarta REST. Если вы новичок в Jakarta REST, прочтите главу 32 *Создание RESTful веб-сервисов с Jakarta REST* прежде чем продолжить чтение этой главы.

Jakarta REST интегрирован с Инъецированием контекстом и зависимостей Jakarta (CDI), Jakarta Enterprise Beans и Jakarta Servlet.

Аннотации для полей и свойств компонентов классов ресурсов

Аннотации Jakarta REST для классов ресурсов позволяют извлекать определённые части или значения из унифицированного идентификатора ресурса (URI) или заголовка запроса.

Jakarta REST предоставляет аннотации, перечисленные в таблице 34-1.

Таблица 34-1 Дополнительные аннотации Jakarta REST

Аннотация	Описание
`@Context`	Инъецирует информацию в поле класса, свойство компонента или параметр метода
`@CookieParam`	Извлекает информацию из файлов cookie, объявленных в заголовке запроса cookie.
`@FormParam`	Извлекает информацию из запроса с типом MIME `application/x-www-form-urlencoded`
`@HeaderParam`	Извлекает значение заголовка
`@MatrixParam`	Извлекает значение параметра матрицы URI
`@PathParam`	Извлекает значение параметра шаблона URI
`@QueryParam`	Извлекает значение параметра запроса URI

Извлечение параметров пути

Шаблоны пути URI — это URI с переменными, встроенными в синтаксис URI. Аннотация `@PathParam` позволяет использовать переменные фрагменты пути URI при вызове метода.

В следующем фрагменте кода показано, как извлечь фамилию сотрудника, если указан адрес его электронной почты:

```java
@Path("/employees/{firstname}.{lastname}@{domain}.com")
public class EmpResource {

    @GET
    @Produces("text/xml")
    public String getEmployeelastname(@PathParam("lastname") String lastName) {
      ...
    }
}
```

В этом примере аннотация `@Path` определяет переменные URI (или параметры пути) `{firstname}`, `{lastname}` и `{domain}`. `@PathParam` в параметре метода запроса извлекает фамилию из адреса электронной почты.

Если ваш HTTP-запрос `GET /employees/john.doe@example.com`, значение «doe» вставляется в `{lastname}`.

Вы можете указать несколько параметров пути в одном URI.

Вы можете объявить регулярное выражение с помощью переменной URI. Например, если требуется, чтобы фамилия состояла только из строчных и прописных символов, вы можете объявить следующее регулярное выражение:

```java
@Path("/employees/{firstname}.{lastname[a-zA-Z]*}@{domain}.com")
```

Если фамилия не соответствует регулярному выражению, возвращается ответ 404.

Извлечение параметров запроса

Используйте аннотацию `@QueryParam` для извлечения параметров запроса из URI запроса.

Например, для запроса всех сотрудников, которые присоединились в течение определённого диапазона лет, используйте сигнатуру метода, подобную следующей:

```java
@Path("/employees/")
@GET
public Response getEmployees(
        @DefaultValue("2017") @QueryParam("minyear") int minyear,
        @DefaultValue("2020") @QueryParam("maxyear") int maxyear)
    {...}
```

Этот фрагмент кода определяет два параметра запроса: `minyear` и `maxyear`. Следующий HTTP-запрос будет запрашивать всех сотрудников, нанятых в период с 2017 по 2020 год:

```http
GET /employees?maxyear=2020&minyear=2017
```

Аннотация `@DefaultValue` определяет значение по умолчанию, которое следует использовать, если для параметров запроса не указаны значения. По умолчанию Jakarta REST присваивает значение null для значений `Object` и значение 0 для примитивных типов данных. Вы можете использовать аннотацию `@DefaultValue`, чтобы явно указать значений по умолчанию для параметров.

Извлечение данных формы

Используйте аннотацию `@FormParam` для извлечения параметров формы из форм HTML. Например, следующая форма принимает имя, адрес сотрудника и имя его руководителя:

```html
<FORM action="http://example.com/employees/" method="post">
  <p>
    <fieldset>
      Employee name: <INPUT type="text" name="empname" tabindex="1">
      Employee address: <INPUT type="text" name="empaddress" tabindex="2">
      Manager name: <INPUT type="text" name="managername" tabindex="3">
    </fieldset>
  </p>
</FORM>
```

Используйте следующий фрагмент кода, чтобы извлечь имя руководителя из этой HTML-формы:

```java
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(@FormParam("managername") String managername) {
    // Сохранение значения
    ...
}
```

Чтобы получить сопоставление имён параметров формы со значениями, используйте фрагмент кода:

```java
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(MultivaluedMap<String, String> formParams) {
    // Сохранение сообщения
}
```

Извлечение Java-типа запроса или ответа

Аннотация `jakarta.ws.rs.core.Context` извлекает типы Java, связанные с запросом или ответом.

Интерфейс `jakarta.ws.rs.core.UriInfo` предоставляет информацию о компонентах URI запроса. В следующем фрагменте кода показано, как получить отображение имён параметров запроса и пути их значениям:

```java
@GET
public String getParams(@Context UriInfo ui) {
    MultivaluedMap<String, String> queryParams = ui.getQueryParameters();
    MultivaluedMap<String, String> pathParams = ui.getPathParameters();
}
```

Интерфейс `jakarta.ws.rs.core.HttpHeaders` предоставляет сведения о заголовках запроса и файлах cookie. В следующем фрагменте кода показано, как получить сопоставление имён параметров заголовка и cookie со значениями:

```java
@GET
public String getHeaders(@Context HttpHeaders hh) {
    MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();
    MultivaluedMap<String, Cookie> pathParams = hh.getCookies();
}
```

Валидация данных ресурса с Bean Validation

Jakarta REST поддерживает Bean Validation для валидации классов ресурсов Jakarta REST. Эта поддержка состоит из:

- Добавление аннотаций ограничений к параметрам метода ресурса
- Обеспечение валидности данных объекта, когда объект передаётся в качестве параметра

Использование аннотаций ограничений на методах ресурсов

Аннотации ограничений Bean Validation могут применяться к параметрам для ресурса. Сервер проверит параметры и либо передаст их, либо выбросит `jakarta.validation.ValidationException`.

JAVA

```
@POST
@Path("/createUser")
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
public void createUser(@NotNull @FormParam("username") String username,
                       @NotNull @FormParam("firstName") String firstName,
                       @NotNull @FormParam("lastName") String lastName,
                       @Email @FormParam("email") String email) {
    ...
}
```

В предыдущем примере предустановленное ограничение `@NotNull` применяется к полям формы `username`, `firstName` и `lastName`. Другое предустановленное ограничение `@Email` проверяет на корректность форматирование адреса электронной почты, указанного в поле формы `email`.

Ограничения также могут применяться к полям в классе ресурсов.

JAVA

```
@Path("/createUser")
public class CreateUserResource {
  @NotNull
  @FormParam("username")
  private String username;

  @NotNull
  @FormParam("firstName")
  private String firstName;

  @NotNull
  @FormParam("lastName")
  private String lastName;

  @Email
  @FormParam("email")
  private String email;

  ...
}
```

В этом примере те же ограничения, которые были применены к параметрам метода в предыдущем примере, применяются к полям класса. Поведение одинаково в обоих примерах.

Ограничения могут также применяться к свойствам JavaBeans класса ресурсов путём добавления аннотаций ограничений к get-методу.

JAVA

```
@Path("/createuser")
public class CreateUserResource {
  private String username;

  @FormParam("username")
  public void setUsername(String username) {
    this.username = username;
  }

  @NotNull
  public String getUsername() {
    return username;
  }
  ...
}
```

Ограничения могут также применяться на уровне класса ресурсов. В следующем примере `@PhoneRequired` — это пользовательское ограничение, которое гарантирует, что пользователь вводит хотя бы один номер телефона. То есть `homePhone` или `mobilePhone` может быть null, но не оба сразу.

JAVA

```
@Path("/createUser")
@PhoneRequired
public class CreateUserResource {
  @FormParam("homePhone")
  private Phone homePhone;

  @FormParam("mobilePhone")
  private Phone mobilePhone;
  ...
}
```

Валидация данных объекта

Классы, которые содержат аннотации ограничений валидации, могут использоваться в параметрах метода в классе ресурса. Чтобы провалидировать эти классы сущностей, используйте аннотацию `@Valid` для параметра метода. Например, следующий пользовательский класс содержит как стандартные, так и пользовательские ограничения валидации.

JAVA

```
@PhoneRequired
public class User {
  @NotNull
  private String username;

  private Phone homePhone;

  private Phone mobilePhone;
  ...
}
```

Этот класс сущности используется в качестве параметра метода ресурса.

JAVA

```
@Path("/createUser")
public class CreateUserResource {
  ...
  @POST
  @Consumers(MediaType.APPLICATION_XML)
  public void createUser(@Valid User user) {
    ...
  }
  ...
}
```

Аннотация `@Valid` обеспечивает валидацию класса сущности во время выполнения. Дополнительные пользовательские ограничения также могут инициировать валидацию сущности.

JAVA

```
@Path("/createUser")
public class CreateUserResource {
  ...
  @POST
  @Consumers(MediaType.APPLICATION_XML)
  public void createUser(@ActiveUser User user) {
    ...
  }
  ...
}
```

В предыдущем примере пользовательское ограничение `@ActiveUser` применяется к классу `User` в дополнение к ограничениям `@PhoneRequired` и `@NotNull`, определённым в классе сущности.

Если метод ресурса возвращает класс сущности, валидация может быть запущена путём применения `@Valid` или любой другой определённой пользователем аннотации ограничения к методу ресурса.

JAVA

```
@Path("/getUser")
public class GetUserResource {
  ...
  @GET
  @Path("{username}")
  @Produces(MediaType.APPLICATION_XML)
  @ActiveUser
  @Valid
  public User getUser(@PathParam("username") String username) {
    // поиск объекта User
    return user;
  }
  ...
}
```

Как и в предыдущем примере, ограничение `@ActiveUser` применяется к возвращённому классу сущностей, а также к ограничениям `@PhoneRequired` и `@NotNull`, определённым в классе сущности.

Коды обработки исключений и ответов

Если выбрасывается `jakarta.validation.ValidationException` или любой подкласс `ValidationException`, кроме `ConstraintValidationException`, среда выполнения Jakarta REST ответит на запрос клиента кодом статуса HTTP 500 (Internal Server Error).

Если выбрасывается `ConstraintValidationException`, среда выполнения Jakarta REST ответит клиенту одним из следующих кодов состояния HTTP:

- `500` (Internal Server Error), если возникла исключительная ситуация при валидации возвращённого методом типа
- `400` (Bad Request) во всех остальных случаях

Подресурсы и разрешение ресурсов времени выполнения

Вы можете использовать класс ресурсов для обработки только части запроса URI. Корневой ресурс может реализовать подресурсы, которые могут обрабатывать оставшуюся часть пути URI.

Метод класса ресурсов, аннотированный `@Path`, является либо методом подресурса, либо указателем подресурса.

- Метод подресурса используется для обработки запросов к подресурсу соответствующего ресурса.
- Указатель подресурса используется для поиска подресурсов соответствующего ресурса.

Методы подресурсов

Метод подресурса обрабатывает HTTP-запрос напрямую. Метод должен быть аннотирован указателем метода запроса, таким как `@GET` или `@POST`, в дополнение к `@Path`. Метод вызывается для URI запроса, который соответствует шаблону URI, созданному путём объединения шаблона URI класса ресурсов с шаблоном URI метода.

В следующем фрагменте кода показано, как метод подресурса можно использовать для извлечения фамилии сотрудника, когда указан адрес электронной почты сотрудника:

JAVA

```java
@Path("/employeeinfo")
public class EmployeeInfo {

    public employeeinfo() {}

    @GET
    @Path("/employees/{firstname}.{lastname}@{domain}.com")
    @Produces("text/xml")
    public String getEmployeeLastName(@PathParam("lastname") String lastName) {
       ...
    }
}
```

Метод `getEmployeeLastName` возвращает `doe` для следующего запроса `GET`:

HTTP

```http
GET /employeeinfo/employees/john.doe@example.com
```

Указатели подресурсов

Указатель подресурса возвращает объект, который будет обрабатывать HTTP-запрос. Метод не должен быть аннотирован указателем метода запроса. Вы должны объявить указатель подресурса в классе подресурса, так как только указатели подресурсов используются для разрешения ресурсов времени выполнения.

Следующий фрагмент кода демонстрирует указатель подресурса:

JAVA

```java
// Корневой класс
@Path("/employeeinfo")
public class EmployeeInfo {

    // Локатор подресурса: получение подресурса Employee
    // по пути /employeeinfo/employees/{empid}
    @Path("/employees/{empid}")
    public Employee getEmployee(@PathParam("empid") String id) {
        // Получение объекта Employee на основе параметра пути id
        Employee emp = ...;
        ...
        return emp;
    }
}

// Класс подресурса
public class Employee {

    // Метод подресурса: возвращает фамилию сотрудника
    @GET
    @Path("/lastname")
    public String getEmployeeLastName() {
        ...
        return lastName;
    }
}
```

В этом фрагменте кода метод `getEmployee` является указателем подресурса, предоставляющего объект `Employee`, обслуживающий запросы для `lastname`.

Если ваш HTTP-запрос `GET /employeeinfo/employees/as209/`, метод `getEmployee` возвращает объект `Employee` с идентификатором `as209`. Во время выполнения Jakarta REST отправляет запрос `GET /employeeinfo/employee/as209/lastname` методу `getEmployeeLastName`. Метод `getEmployeeLastName` извлекает и возвращает фамилию сотрудника с идентификатором `as209`.

Интеграция Jakarta REST с Jakarta Enterprise Beans и CDI

Jakarta REST работает с Jakarta Enterprise Beans и инъецированием контекстов и зависимостей Jakarta (CDI).

В общем, чтобы Jakarta REST работала с EJB, нужно аннотировать класс компонента `@Path` для преобразования его в корневой класс ресурсов. Вы можете использовать аннотацию `@Path` для сессионных компонентов без сохранения состояния и синглтонов POJO.

В следующем фрагменте кода показаны сессионный компонент без сохранения состояния и компонент-синглтон, преобразованные в классы корневых ресурсов Jakarta REST.

JAVA

```
@Stateless
@Path("stateless-bean")
public class StatelessResource {...}

@Singleton
@Path("singleton-bean")
public class SingletonResource {...}
```

Сессионные компоненты также могут использоваться для подресурсов.

Jakarta REST и CDI имеют немного разные модели компонентов. По умолчанию корневые классы ресурсов Jakarta REST управляются в области видимости запроса и для указания области видимости не требуется аннотаций. Managed-бины CDI, аннотированные `@RequestScoped` или `@ApplicationScoped`, могут быть преобразованы в классы ресурсов Jakarta REST.

В следующем фрагменте кода показан класс ресурсов Jakarta REST.

JAVA

```
@Path("/employee/{id}")
public class Employee {
    public Employee(@PathParam("id") String id) {...}
}

@Path("{lastname}")
public final class EmpDetails {...}
```

Следующий фрагмент кода показывает этот класс ресурсов Jakarta REST, преобразованный в компонент CDI. Бины должны быть проксируемыми, поэтому классу `Employee` требуется неприватный конструктор без параметров, а класс `EmpDetails` не должен быть `final`.

JAVA

```
@Path("/employee/{id}")
@RequestScoped
public class Employee {
    public Employee() {...}

    @Inject
    public Employee(@PathParam("id") String id) {...}
}

@Path("{lastname}")
@RequestScoped
public class EmpDetails {...}
```

Условные HTTP-запросы

Jakarta REST обеспечивает поддержку условных HTTP-запросов `GET` и `PUT`. Условные запросы `GET` помогают понизить требования к пропускной способности за счёт повышения эффективности обработки клиентских запросов.

Запрос `GET` может вернуть ответ 304 (Not Modified), если представление не изменилось со времени предыдущего запроса. Например, веб-сайт может вернуть ответы 304 для всех своих статических изображений, которые не изменились со времени предыдущего запроса.

Запрос `PUT` может вернуть ответ 412 (Precondition Failed), если представление было изменено с момента последнего запроса. Условный `PUT` поможет избежать проблемы потерянного обновления.

Условные HTTP-запросы могут использоваться с заголовками `Last-Modified` и `ETag`. Заголовок `Last-Modified` может представлять дату с детализацией в одну секунду.

JAVA

```
@Path("/employee/{joiningdate}")
public class Employee {

    Date joiningdate;

    @GET
    @Produces("application/xml")
    public Employee(@PathParam("joiningdate") Date joiningdate,
                    @Context Request req,
                    @Context UriInfo ui) {

        this.joiningdate = joiningdate;
        ...
        this.tag = computeEntityTag(ui.getRequestUri());
        if (req.getMethod().equals("GET")) {
            Response.ResponseBuilder rb = req.evaluatePreconditions(tag);
            if (rb != null) {
                throw new WebApplicationException(rb.build());
            }
        }
    }
}
```

В этом фрагменте кода конструктор класса `Employee` вычисляет тег сущности из URI запроса и вызывает метод `request.evaluatePreconditions` с этим тегом. Если клиентский запрос возвращает заголовок `If-none-match` со значением, имеющим тот же тег сущности, который был вычислен, `evaluate.Preconditions` возвращает предварительно заполненный ответ с кодом состояния 304 и набором тегов сущностей, которые могут быть созданы и возвращены.

Согласование содержимого во время выполнения

Аннотации `@Produces` и `@Consumes` обрабатывают согласование статического содержимого в Jakarta REST. Эти аннотации определяют настройки содержимого сервера. Заголовки HTTP `Accept`, `Content-Type` и `Accept-Language` определяют параметры согласования содержимого клиента.

Дополнительные сведения о заголовках HTTP для согласования содержимого см. HTTP/1.1 — Content Negotiation (https://www.w3.org/Protocols/rfc2616/rfc2616-sec12.html).

Следующий фрагмент кода показывает настройки содержимого сервера:

JAVA
```java
@Produces("text/plain")
@Path("/employee")
public class Employee {

    @GET
    public String getEmployeeAddressText(String address) {...}

    @Produces("text/xml")
    @GET
    public String getEmployeeAddressXml(Address address) {...}
}
```

Метод `getEmployeeAddressText` вызывается для HTTP-запроса, который выглядит следующим образом:

HTTP
```
GET /employee
Accept: text/plain
```

Это даст следующий ответ:

```
500 Oracle Parkway, Redwood Shores, CA
```

Метод `getEmployeeAddressXml` вызывается для HTTP-запроса, который выглядит следующим образом:

HTTP
```
GET /employee
Accept: text/xml
```

Это даст следующий ответ:

XML
```xml
<address street="500 Oracle Parkway, Redwood Shores, CA" country="USA"/>
```

При согласовании статического содержимого вы также можете определить несколько типов содержимого и MIME для клиента и сервера.

JAVA
```java
@Produces("text/plain", "text/xml")
```

Помимо согласования статического содержимого, Jakarta REST также поддерживает согласование содержимого во время выполнения с помощью объектов `jakarta.ws.rs.core.Variant` и `Request`. Класс `Variant` определяет представление ресурса согласования контента. Каждый объект класса `Variant` может содержать тип MIME, язык и кодировку. Объект `Variant` определяет представление ресурса, которое поддерживается сервером. Класс `Variant.VariantListBuilder` используется для создания списка вариантов представления.

В следующем фрагменте кода показано, как создать список вариантов представления ресурсов:

JAVA
```
List<Variant> vs = Variant.mediatypes("application/xml", "application/json")
        .languages("en", "fr").build();
```

Этот фрагмент кода вызывает метод `build` класса `VariantListBuilder`. Класс `VariantListBuilder` вызывается при вызове методов `mediatypes`, `languages` или `encodings`. Метод `build` создаёт серию представлений ресурсов. Список `Variant`, созданный методом `build`, содержит все возможные комбинации элементов, указанных в методах `mediatypes`, `languages` и `encodings`.

В этом примере размер объекта `vs`, как определено в этом фрагменте кода, равен 4, а содержимое выглядит следующим образом:

JAVA
```
[["application/xml","en"], ["application/json","en"],
    ["application/xml","fr"],["application/json","fr"]]
```

Метод `jakarta.ws.rs.core.Request.selectVariant` принимает список объектов `Variant` и выбирает объект `Variant`, соответствующий HTTP-запросу. Этот метод сравнивает свой список объектов `Variant` с заголовками `Accept`, `Accept-Encoding`, `Accept-Language` и `Accept-Charset` HTTP-запроса.

В следующем фрагменте кода показано, как использовать метод `selectVariant` для выбора наиболее приемлемого `Variant` из значений в клиентском запросе:

JAVA
```
@GET
public Response get(@Context Request r) {
    List<Variant> vs = ...;
    Variant v = r.selectVariant(vs);
    if (v == null) {
        return Response.notAcceptable(vs).build();
    } else {
        Object rep = selectRepresentation(v);
        return Response.ok(rep, v);
    }
}
```

Метод `selectVariant` возвращает объект `Variant`, который соответствует запросу, или null, если совпадений не найдено. В этом фрагменте кода, если метод возвращает значение null, создаётся объект `Response` неприемлемого ответа. В противном случае возвращается объект `Response` со статусом OK, содержащий представление сущности в форме `Object` и `Variant`.

Использование Jakarta REST с Jakarta XML Binding

Jakarta XML Binding — это технология связывания XML с Java, которая упрощает разработку веб-сервисов, выполняя преобразования между XMD-схемой и объектами Java. Схема XML определяет элементы данных и структуру документа XML. Вы можете использовать Jakarta XML Binding API и инструменты для установления сопоставлений между Java-классами и XML-схемой. Jakarta XML Binding предоставляет инструменты, которые позволяют конвертировать XML-документы в объекты Java и обратно.

Используя Jakarta XML Binding, вы можете управлять объектами следующими способами.

- Вы можете начать с определения схемы XML (XSD) и использовать `xjc` — инструмент компилятора схемы Jakarta XML Binding — чтобы создать набор классов Java с аннотациями Jakarta XML Binding, которые отображаются на элементы и типы данных из схемы XSD.
- Вы можете начать с набора классов Java и использовать `schemagen` — инструмент генератора схемы Jakarta XML Binding — для создания XML-схемы.

- После того, как сопоставление между XML-схемой и классами Java установлено, вы можете использовать среду выполнения Jakarta XML Binding для сериализациии XML-документов в объекты Java и десериализации их обратно и использовать полученные классы Java для сборки приложения веб-сервиса.

XML является распространённым форматом, который RESTful сервисы принимают и в котором отдают документы. Чтобы десериализовать и сериализовать XML, вы можете представлять запросы и ответы с помощью аннотированных объектов Jakarta XML Binding. Ваше приложение Jakarta REST может использовать объекты Jakarta XML Binding для управления данными XML. Объекты Jakarta XML Binding могут использоваться как параметры сущности запроса и сущности ответа. Среда выполнения Jakarta REST включает стандартные интерфейсы поставщика `MessageBodyReader` и `MessageBodyWriter` для чтения и записи объектов Jakarta XML Binding в качестве сущностей.

Jakarta REST обеспечивает доступ к сервисам путём публикации ресурсов. Ресурсы — это простые классы Java с некоторыми дополнительными аннотациями Jakarta REST. Эти аннотации выражают следующее:

- Путь к ресурсу (URL, который используется для доступа к нему)
- Метод HTTP, который используется для вызова определённого метода (например, `GET` или `POST`)
- Тип MIME, который метод принимает в запросе или использует в ответе

Когда вы определяете ресурсы для своего приложения, учитывайте типы данных, которые хотите предоставить. Возможно, у вас уже есть реляционная база данных, содержащая информацию, которую вы хотите предоставить пользователям. Или у вас может быть статический контент, который не находится в базе данных, но должен распространяться как ресурсы. Using Jakarta REST, you can distribute content from multiple sources. RESTful веб-сервисы могут использовать различные типы форматов ввода/вывода для запроса и ответа. Пример `customer`, описанный в Приложение customer, использует XML.

Ресурсы имеют представления. Представление ресурса — это содержимое в HTTP-сообщении, которое отправляется или возвращается из ресурса с использованием URI. Каждое представление, поддерживаемое ресурсом, имеет соответствующий тип MIME. Например, если ресурс собирается возвращать содержимое в формате XML, можно использовать `application/xml` в качестве связанного типа MIME в HTTP-сообщении. В зависимости от требований приложения ресурсы могут возвращать представления в одном формате или в нескольких. Jakarta REST предоставляет аннотации `@Consumes` и `@Produces` для объявления типов MIME, допустимых методу ресурсов для чтения и записи.

Jakarta REST также отображает типы Java с представлениями ресурсов и из них с помощью провайдеров сущностей. Провайдер сущностей `MessageBodyReader` считывает объект запроса и десериализует его в объект Java. Провайдер сущностей `MessageBodyWriter` сериализует из объекта Java в объект ответа. Например, если в качестве параметра объекта запроса используется значение `String`, провайдер сущностей `MessageBodyReader` десериализует тело запроса в новую `String`. Если тип Jakarta XML Binding используется в качестве возвращаемого типа для метода ресурса, `MessageBodyWriter` сериализует объект Jakarta XML Binding в тело ответа.

По умолчанию среда выполнения Jakarta REST пытается создать и использовать класс `JAXBContext` для классов Jakarta XML Binding. Однако, если класс `JAXBContext` по умолчанию не подходит, вы можете предоставить класс `JAXBContext` для приложения, используя интерфейс провайдера Jakarta REST `ContextResolver`.

В следующих разделах объясняется, как использовать Jakarta XML Binding с методами ресурсов REST Jakarta.

Использование объектов Java для моделирования пользовательских данных

Если у вас нет определения схемы XML для данных, которые вы хотите предоставить, можете смоделировать свои данные как классы Java, добавить аннотации Jakarta XML Binding к этим классам и использовать Jakarta XML Binding для создания схемы XML для данных. Например, если данные, которые вы хотите предоставить, представляют собой набор продуктов, и каждый продукт имеет идентификатор, имя, описание и цену, можете смоделировать его как класс Java следующим образом:

JAVA

```java
@XmlRootElement(name="product")
@XmlAccessorType(XmlAccessType.FIELD)
public class Product {

    @XmlElement(required=true)
    protected int id;
    @XmlElement(required=true)
    protected String name;
    @XmlElement(required=true)
    protected String description;
    @XmlElement(required=true)
    protected int price;

    public Product() {}

    // get- и set- методы
    // ...
}
```

Запустите генератор схемы Jakarta XML Binding в командной строке, чтобы сгенерировать соответствующее определение схемы XML:

SHELL

```shell
schemagen Product.java
```

Эта команда создаёт схему XML в виде файла `.xsd`:

XML

```xml
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:element name="product" type="product"/>

    <xs:complexType name="product">
      <xs:sequence>
        <xs:element name="id" type="xs:int"/>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="description" type="xs:string"/>
        <xs:element name="price" type="xs:int"/>
      </xs:sequence>
    <xs:complexType>
</xs:schema>
```

Получив это сопоставление, вы можете создавать объекты `Product` в своём приложении, возвращать их и использовать в качестве параметров в методах ресурсов Jakarta REST. Среда выполнения Jakarta REST использует Jakarta XML Binding для преобразования XML-данных из запроса в объект `Product` и преобразования объекта `Product` в XML-данные для ответа. Следующий класс ресурсов предоставляет простой пример:

JAVA

```java
@Path("/product")
public class ProductService {
    @GET
    @Path("/get")
    @Produces("application/xml")
    public Product getProduct() {
        Product prod = new Product();
        prod.setId(1);
        prod.setName("Mattress");
        prod.setDescription("Queen size mattress");
        prod.setPrice(500);
        return prod;
    }

    @POST
    @Path("/create")
    @Consumes("application/xml")
    public Response createProduct(Product prod) {
        // Обработка или сохранение product и возвращение ответа
        // ...
    }
}
```

Некоторые IDE, такие как IDE NetBeans, автоматически запускают генератор схемы в процессе сборки, если вы добавляете в проект классы Java, имеющие аннотации Jakarta XML Binding. Подробный пример см. в разделе Приложение customer. Пример `customer` содержит более сложные взаимосвязи между классами Java, которые моделируют данные, что приводит к более иерархическому представлению XML.

Начинаем с определения существующей схемы XML

Если у вас уже есть определение схемы XML в файле `.xsd` для данных, которые вы хотите предоставить, используйте инструмент компилятора схемы Jakarta XML Binding. Рассмотрим простой пример файла `.xsd`:

XML

```xml
<xs:schema targetNamespace="http://xml.product"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified"
           xmlns:myco="http://xml.product">
  <xs:element name="product" type="myco:Product"/>
  <xs:complexType name="Product">
    <xs:sequence>
      <xs:element name="id" type="xs:int"/>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="description" type="xs:string"/>
      <xs:element name="price" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Запустите утилиту компиляции схемы в командной строке следующим образом:

SHELL

```shell
xjc Product.xsd
```

Эта команда генерирует исходный код для классов Java, которые соответствуют типам, определённым в файле `.xsd`. Утилита компиляции схемы создаёт класс Java для каждого элемента `complexType` из файла `.xsd`. Поля каждого сгенерированного Java-класса совпадают с элементами внутри соответствующего `complexType`, и класс содержит get- и set- методы для этих полей.

В этом случае утилита компиляции схемы генерирует классы `product.xml.Product` и `product.xml.ObjectFactory`. Класс `Product` содержит аннотации Jakarta XML Binding, а его поля соответствуют полям в `.xsd`:

JAVA

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Product", propOrder = {
    "id",
    "name",
    "description",
    "price"
})
public class Product {
    protected int id;
    @XmlElement(required = true)
    protected String name;
    @XmlElement(required = true)
    protected String description;
    protected int price;

    // Setter and getter methods
    // ...
}
```

Вы можете создавать объекты класса `Product` из своего приложения (например, из базы данных). Созданный класс `product.xml.ObjectFactory` содержит метод, позволяющий преобразовывать эти объекты в элементы Jakarta XML Binding, которые могут быть возвращены как XML внутри методов ресурсов Jakarta REST:

JAVA

```
@XmlElementDecl(namespace = "http://xml.product", name = "product")
public JAXBElement<Product> createProduct(Product value) {
    return new JAXBElement<Product>(_Product_QNAME, Product.class, null, value);
}
```

В следующем коде показано, как использовать сгенерированные классы для возврата элемента Jakarta XML Binding в виде XML в методе ресурсов Jakarta REST:

JAVA

```
@Path("/product")
public class ProductService {
    @GET
    @Path("/get")
    @Produces("application/xml")
    public JAXBElement<Product> getProduct() {
        Product prod = new Product();
        prod.setId(1);
        prod.setName("Mattress");
        prod.setDescription("Queen size mattress");
        prod.setPrice(500);
        return new ObjectFactory().createProduct(prod);
    }
}
```

Для методов ресурсов `@POST` и `@PUT` вы можете использовать объект `Product` непосредственно в качестве параметра. Jakarta REST отображает XML-данные из запроса в объект `Product`.

JAVA

```java
@Path("/product")
public class ProductService {
    @GET
    // ...

    @POST
    @Path("/create")
    @Consumes("application/xml")
    public Response createProduct(Product prod) {
        // Обработка или сохранение product и возвращение ответа
        // ...
    }
}
```

Использование JSON с Jakarta REST и Jakarta XML Binding

Jakarta REST может автоматически читать и писать XML с помощью Jakarta XML Binding, но он также может работать с данными JSON. JSON — это простой текстовый формат для обмена данными, пришедший из JavaScript. Для предыдущих примеров XML-представление продукта

XML

```xml
<product>
  <id>1</id>
  <name>Mattress</name>
  <description>Queen size mattress</description>
  <price>500</price>
</product>
```

Эквивалентное представление JSON

JSON

```json
{
    "id":"1",
    "name":"Mattress",
    "description":"Queen size mattress",
    "price":500
}
```

Вы можете добавить тип MIME `application/json` или `MediaType.APPLICATION_JSON` к аннотации `@Produces` в методах ресурсов для получения ответов с данными JSON:

JAVA

```java
@GET
@Path("/get")
@Produces({"application/xml","application/json"})
public Product getProduct() { ... }
```

В этом примере ответом по умолчанию является XML, но если клиент делает запрос `GET` , который включает такой заголовок, ответом будет объект JSON:

HTTP

```http
Accept: application/json
```

Методы ресурсов также могут принимать данные JSON для аннотированных классов Jakarta XML Binding:

JAVA

```java
@POST
@Path("/create")
@Consumes({"application/xml","application/json"})
public Response createProduct(Product prod) { ... }
```

При отправке данных JSON с запросом `POST` клиент должен включать следующий заголовок:

```HTTP
Content-Type: application/json
```

Приложение customer

В этом разделе описывается, как создать и запустить пример `customer`. Это приложение представляет собой RESTful веб-сервис, который использует Jakarta XML Binding для выполнения операций создания, чтения, изменения и удаления (CRUD) для определённого объекта.

Пример приложения `customer` находится в каталоге *`tut-install`*`/examples/jaxrs/customer/`. См. главу 2, *Использование примеров учебника* для получения базовой информации о сборке и запуске примеров.

Обзор customer

Исходные файлы этого приложения находятся в *`tut-install`*`/examples/jaxrs/customer/src/main/java/`. Приложение состоит из трёх частей.

- Классы сущностей `Customer` и `Address`. Эти классы моделируют данные приложения и содержат аннотации Jakarta XML Binding.
- Класс ресурсов `CustomerService`. Этот класс содержит методы ресурсов Jakarta REST, которые выполняют операции с объектами `Customer`, представленными в виде данных XML или JSON, с использованием Jakarta XML Binding. Смотрите Класс CustomerService для подробностей.
- Сессионный компонент `CustomerBean`, который является вспомогательным компонентом для веб-клиента. `CustomerBean` использует клиентский API Jakarta REST для вызова методов `CustomerService`.

Пример приложения `customer` показывает, как моделировать объекты данных как классы Java с аннотациями Jakarta XML Binding.

Классы сущностей Customer и Address

Следующий класс представляет адрес клиента:

JAVA

```java
@Entity
@Table(name="CUSTOMER_ADDRESS")
@XmlRootElement(name="address")
@XmlAccessorType(XmlAccessType.FIELD)
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @XmlElement(required=true)
    protected int number;

    @XmlElement(required=true)
    protected String street;

    @XmlElement(required=true)
    protected String city;

    @XmlElement(required=true)
    protected String province;

    @XmlElement(required=true)
    protected String zip;

    @XmlElement(required=true)
    protected String country;

    public Address() { }

    // get- и set- методы
    // ...
}
```

Аннотация `@XmlRootElement(name="address")` отображает этот класс в XML-элемент `address`. Аннотация `@XmlAccessorType(XmlAccessType.FIELD)` указывает, что все поля этого класса по умолчанию связаны с XML. Аннотация `@XmlElement(required=true)` указывает, что элемент должен присутствовать в представлении XML.

Следующий класс представляет клиента:

JAVA

```java
@Entity
@Table(name="CUSTOMER_CUSTOMER")
@NamedQuery(
    name="findAllCustomers",
    query="SELECT c FROM Customer c " +
          "ORDER BY c.id"
)
@XmlRootElement(name="customer")
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @XmlAttribute(required=true)
    protected int id;

    @XmlElement(required=true)
    protected String firstname;

    @XmlElement(required=true)
    protected String lastname;

    @XmlElement(required=true)
    @OneToOne
    protected Address address;

    @XmlElement(required=true)
    protected String email;

    @XmlElement (required=true)
    protected String phone;

    public Customer() {...}

    // get- и set- методы
    // ...
}
```

Класс `Customer` содержит те же аннотации Jakarta XML Binding, что и предыдущий класс, за исключением аннотации `@XmlAttribute(required=true)`, которая сопоставляет свойство с атрибутом XML-элемента, представляющим класс.

Класс `Customer` содержит свойство, тип которого является другой сущностью, класс `Address`. Этот механизм позволяет определять в коде Java иерархические отношения между сущностями без необходимости писать файл `.xsd` самостоятельно.

Jakarta XML Binding генерирует следующее определение схемы XML для двух предыдущих классов:

XML

```xml
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="address" type="address"/>
  <xs:element name="customer" type="customer"/>

  <xs:complexType name="address">
    <xs:sequence>
      <xs:element name="id" type="xs:long" minOccurs="0"/>
      <xs:element name="number" type="xs:int"/>
      <xs:element name="street" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="province" type="xs:string"/>
      <xs:element name="zip" type="xs:string"/>
      <xs:element name="country" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="customer">
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
      <xs:element ref="address"/>
      <xs:element name="email" type="xs:string"/>
      <xs:element name="phone" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:int" use="required"/>
  </xs:complexType>
</xs:schema>
```

Класс CustomerService

Класс `CustomerService` имеет метод `createCustomer`, который создаёт ресурс клиента на основе класса `Customer` и возвращает URI для нового ресурса.

JAVA

```java
@Stateless
@Path("/Customer")
public class CustomerService {
    public static final Logger logger =
            Logger.getLogger(CustomerService.class.getCanonicalName());
    @PersistenceContext
    private EntityManager em;
    private CriteriaBuilder cb;

    @PostConstruct
    private void init() {
        cb = em.getCriteriaBuilder();
    }
    ...
    @POST
    @Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Response createCustomer(Customer customer) {

        try {
            long customerId = persist(customer);
            return Response.created(URI.create("/" + customerId)).build();
        } catch (Exception e) {
            logger.log(Level.SEVERE,
                    "Error creating customer for customerId {0}. {1}",
                    new Object[]{customer.getId(), e.getMessage()});
            throw new WebApplicationException(e,
                    Response.Status.INTERNAL_SERVER_ERROR);
        }
    }
    ...
    private long persist(Customer customer) {
        try {
            Address address = customer.getAddress();
            em.persist(address);
            em.persist(customer);
        } catch (Exception ex) {
            logger.warning("Something went wrong when persisting the customer");
        }
        return customer.getId();
    }
}
```

Ответ, возвращённый клиенту, имеет URI для вновь созданного ресурса. Тип возвращаемого значения — это тело объекта, отображаемое из свойства ответа с кодом состояния, указанным в свойстве status ответа. `WebApplicationException` — это `RuntimeException`, который используется для переноса соответствующего кода состояния ошибки HTTP, такого как 404, 406, 415 или 500.

Аннотации `@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})` и `@Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})` устанавливают типы MIME для запросов и ответов для использования соответствующего клиента MIME. Эти аннотации могут применяться к методу ресурса, классу ресурса или даже провайдеру сущностей. Если эти аннотации не используются, Jakarta REST позволяет использовать любой тип MIME (`"*/*"`).

В следующем фрагменте кода показана реализация методов `getCustomer` и `findById`. Метод `getCustomer` использует аннотацию `@Produces` и возвращает объект `Customer`, который преобразуется в представление XML или JSON в зависимости от заголовка `Accept`, указанного клиентом.

JAVA

```
@GET
@Path("{id}")
@Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public Customer getCustomer(@PathParam("id") String customerId) {
    Customer customer = null;

    try {
        customer = findById(customerId);
    } catch (Exception ex) {
        logger.log(Level.SEVERE,
                "Error calling findCustomer() for customerId {0}. {1}",
                new Object[]{customerId, ex.getMessage()});
    }
    return customer;
}
...
private Customer findById(String customerId) {
    Customer customer = null;
    try {
        customer = em.find(Customer.class, customerId);
        return customer;
    } catch (Exception ex) {
        logger.log(Level.WARNING,
                "Couldn't find customer with ID of {0}", customerId);
    }
    return customer;
}
```

Использование клиента Jakarta REST в классах CustomerBean

Используйте клиентское API Jakarta REST чтобы написать клиента для примера `customer` .

Класс EJB-компонента `CustomerBean` вызывает клиентское API Jakarta REST для тестирования веб-сервиса `CustomerService`:

JAVA

```java
@Named
@Stateless
public class CustomerBean {
    protected Client client;
    private static final Logger logger =
            Logger.getLogger(CustomerBean.class.getName());

    @PostConstruct
    private void init() {
        client = ClientBuilder.newClient();
    }

    @PreDestroy
    private void clean() {
        client.close();
    }

    public String createCustomer(Customer customer) {
        if (customer == null) {
            logger.log(Level.WARNING, "customer is null.");
            return "customerError";
        }
        String navigation;
        Response response =
                client.target("http://localhost:8080/customer/webapi/Customer")
                .request(MediaType.APPLICATION_XML)
                .post(Entity.entity(customer, MediaType.APPLICATION_XML),
                        Response.class);
        if (response.getStatus() == Status.CREATED.getStatusCode()) {
            navigation = "customerCreated";
        } else {
            logger.log(Level.WARNING, "couldn''t create customer with " +
                    "id {0}. Status returned was {1}",
                    new Object[]{customer.getId(), response.getStatus()});
            navigation = "customerError";
        }
        return navigation;
    }

    public String retrieveCustomer(String id) {
        String navigation;
        Customer customer =
                client.target("http://localhost:8080/customer/webapi/Customer")
                .path(id)
                .request(MediaType.APPLICATION_XML)
                .get(Customer.class);
        if (customer == null) {
            navigation = "customerError";
        } else {
            navigation = "customerRetrieved";
        }
        return navigation;
    }

    public List<Customer> retrieveAllCustomers() {
        List<Customer> customers =
                client.target("http://localhost:8080/customer/webapi/Customer")
                .path("all")
                .request(MediaType.APPLICATION_XML)
                .get(new GenericType<List<Customer>>() {});
        return customers;
    }
}
```

Этот клиент использует методы `POST` и `GET`.

Все эти коды состояния HTTP указывают на успешное выполнение: 201 для `POST`, 200 для `GET` и 204 для `DELETE`. Дополнительная информация о значениях кодов состояния HTTP приведена на веб-странице https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

Запуск customer

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска приложения `customer`.

Сборка, упаковка и развёртывание customer с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/jaxrs
   ```

4. Выберите каталог `customer`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `customer` и выберите **Сборка**.

   Эта команда собирает и упаковывает приложение в WAR-файл `customer.war`, расположенный в каталоге `target`. Затем WAR-файл развёртывается в GlassFish Server.
7. Откройте веб-клиент в браузере по следующему URL:

   ```
   http://localhost:8080/customer/
   ```

   Веб-клиент позволяет создавать и просматривать клиентов.

Сборка, упаковка и развёртывание customer с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/jaxrs/customer/
   ```

3. Введите следующую команду:

   ```
   mvn install
   ```
   SHELL

   Эта команда собирает и упаковывает приложение в WAR-файл `customer.war`, расположенный в каталоге `target`. Затем WAR-файл развёртывается в GlassFish Server.
4. Откройте веб-клиент в браузере по следующему URL:

   ```
   http://localhost:8080/customer/
   ```

   Веб-клиент позволяет создавать и просматривать клиентов.

Часть VII: Enterprise-бины

В части VII рассматриваются Enterprise-бины.

Глава 35. Enterprise-бины

EJB-компоненты — это компоненты Jakarta EE, которые реализуют технологию Jakarta Enterprise Beans. EJB-компоненты работают в EJB-контейнере — среде выполнения в GlassFish Server (см. Типы контейнеров). Несмотря на то, что EJB-контейнер прозрачен для разработчика приложений, он предоставляет сервисы системного уровня, такие как транзакции и безопасность, своим EJB-компонентам. Эти сервисы позволяют быстро создавать и развёртывать Enterprise-бины, которые составляют ядро транзакционных приложений Jakarta EE.

Что такое Enterprise-бин?

Написанный на Java Enterprise-бин является серверным компонентом и инкапсулирует бизнес-логику приложения. Бизнес-логика — это код, который реализует назначение приложения. Например, в приложении для управления запасами Enterprise-бины могут реализовывать бизнес-логику в методах `checkInventoryLevel` и `orderProduct`. С помощью этих методов клиенты могут получить доступ к сервисам инвентаризации, предоставляемым приложением.

Преимущества Enterprise-бинов

По нескольким причинам Enterprise-бины упрощают разработку больших распределённых приложений. Во-первых, поскольку EJB-контейнер предоставляет EJB-компонентам сервисы системного уровня, разработчик компонентов может сосредоточиться на решении бизнес-задач. EJB-Контейнер, а не разработчик компонента, отвечает за сервисы системного уровня, такие как управление транзакциями и авторизация пользователей.

Во-вторых, поскольку бины, а не клиенты содержат бизнес-логику приложения, разработчик клиента может сосредоточиться на представлении клиента. Разработчик клиента не должен кодировать процедуры, которые реализуют бизнес-правила или обращаются к базам данных. В результате клиенты становятся тоньше, что особенно важно для клиентов, работающих на небольших устройствах.

В-третьих, поскольку Enterprise-бины являются переносимыми компонентами, из них можно скомпоновать новые приложения. При условии, что они используют стандартные API, эти приложения могут работать на любом совместимом с Jakarta EE сервере.

Когда использовать Enterprise-бины

Использование Enterprise-бинов может стать хорошим выбором, если приложение имеет любое из следующих требований.

- Приложение должно быть масштабируемым. Чтобы удовлетворить растущее число пользователей, может потребоваться распределить компоненты приложения на нескольких компьютерах. Не только Enterprise-бины приложения могут работать на разных компьютерах, но и их расположение останется прозрачным для клиентов.
- Транзакции должны обеспечивать целостность данных. Enterprise-бины поддерживают транзакции — механизмы, которые управляют одновременным доступом к общим объектам.
- Приложение будет иметь множество клиентов. Всего несколькими строками кода удалённые клиенты могут находить Enterprise-бины. Эти клиенты могут быть тонкими, разнообразными и многочисленными.

Типы Enterprise-бинов

Таблица 35-1 суммирует два типа Enterprise-бинов. В следующих разделах каждый тип обсуждается более подробно.

Таблица 35-1. Типы Enterprise-бинов

Тип Enterprise-бина	Назначение
Сессия	Выполняет задание для клиента. При желании может реализовывать веб-сервис
Управляемый сообщениями	Действует в качестве слушателя для определённого типа обмена сообщениями, такого как Jakarta Messaging

Что такое сессионный бин?

Сессионный компонент инкапсулирует бизнес-логику, которая может вызываться программно локальным, удалённым клиентом или клиентом веб-сервиса. Чтобы получить доступ к приложению, развёрнутому на сервере, клиент вызывает методы сессионного компонента. Сессионный компонент выполняет работу для своего клиента, скрывая сложность выполняемых задач на стороне сервера.

Сессионный компонент не является персистентным. (То есть его данные не сохраняются в базе данных.)

Примеры кода см. главе 37 *Запуск примеров Enterprise-бинов*.

Типы сессионных бинов

Сессионные компоненты могут быть одного из трёх типов: с сохранением состояния, без сохранения состояния и синглтон.

Сессионные бины, сохраняющие состояние

Состояние объекта состоит из значений его переменных. В сессионном бине с состоянием переменные объекта представляют состояние уникальной сессии клиента/компонента. Поскольку клиент взаимодействует («разговаривает») со своим компонентом, это состояние часто называют диалоговым состоянием.

Как следует из названия, сессионный компонент похож на интерактивную сессию. Сессионный компонент не является общим. У него может быть только один клиент, точно так же, как у интерактивной сессии может быть только один пользователь. Когда клиент завершает работу, его сессионный компонент завершает работу и больше не связан с клиентом.

Состояние сохраняется в течение сессии клиент/компонент. Если клиент удаляет компонент, сессия завершается, и состояние исчезает. Однако этот временный характер состояния не является проблемой, поскольку, когда разговор между клиентом и компонентом заканчивается, нет необходимости сохранять состояние.

Сессионные бины, не сохраняющие состояние

Сессионный компонент без сохранения состояния не поддерживает диалоговое состояние с клиентом. Когда клиент вызывает методы бина, не сохраняющего состояние, переменные объекта бина могут содержать состояние, специфичное для этого клиента, но только на время вызова. Когда метод завершён, специфичное для клиента состояние не должно сохраняться. Однако клиенты могут изменять состояние переменных объекта в компонентах без сохранения состояния, собранных в пулы, и это состояние сохраняется до следующего вызова этого компонента. За исключением вызова метода, все объекты компонента без сохранения состояния эквивалентны, что позволяет EJB-контейнеру назначать любой объект любому клиенту. То есть состояние сессионного компонента без сохранения состояния должно применяться ко всем клиентам.

Поскольку сессионные компоненты без сохранения состояния могут поддерживать несколько клиентов, они могут обеспечить лучшую масштабируемость для приложений, которым требуется большое количество клиентов. Как правило, приложению требуется меньше сессионных компонентов без сохранения состояния, чем сессионных компонентов с сохранением состояния, для поддержки одинакового количества клиентов.

Сессионный компонент без сохранения состояния может реализовывать веб-сервис, но сессионный компонент с сохранением состояния не может.

Сессионные бины-синглтоны

Сессионный компонент-синглтон создаётся один раз для каждого приложения и существует в течение жизненного цикла приложения. Сессионные бины-синглтоны предназначены для ситуаций, в которых один EJB-компонент может использоваться клиентами совместно и одновременно.

Сессионные бины-синглтоны предлагают схожую с сессионными бинами без сохранения состояния функциональность, но отличаются от них тем, что в приложении имеется только один сессионный бин-синглтон, в отличие от пула сессионных бинов без состояния, каждый из которых может отвечать на запрос клиента. Как и сессионные компоненты без сохранения состояния, сессионные компоненты-синглтоны могут реализовывать конечные точки веб-сервиса.

Сессионные бины-синглтоны поддерживают своё состояние между вызовами клиентов, но не обязаны этого делать при сбоях или остановках сервера.

Приложения, которые используют сессионный компонент-синглтон, могут указать, что объект-синглтон должен создаваться при запуске приложения, что позволяет синглтону выполнять задачи инициализации для приложения. Синглтон также может выполнять задачи очистки при завершении работы приложения, поскольку синглтон будет работать на протяжении всего жизненного цикла приложения.

Когда использовать сессионные бины

Сессионные компоненты с сохранением состояния подходят, если выполняется любое из следующих условий.

- Состояние компонента представляет собой взаимодействие между компонентом и конкретным клиентом.
- Бин должен содержать информацию о клиенте через вызовы методов.
- Бин является посредником между клиентом и другими компонентами приложения, предоставляя клиенту упрощённое представление.
- За кулисами компонент управляет рабочим процессом нескольких Enterprise-бинов.

Чтобы повысить производительность, вы можете выбрать сессионный компонент без сохранения состояния, если он имеет какую-либо из этих характеристик.

- Состояние бина не имеет данных для конкретного клиента.
- В одном вызове метода компонент выполняет общую задачу для всех клиентов. Например, вы можете использовать сессионный компонент без сохранения состояния для отправки электронного письма, подтверждающего онлайн-заказ.
- Бин реализует веб-сервис.

Сессионные компоненты-синглтоны подходят в следующих случаях.

- Состояние должно быть общим для всего приложения.
- Доступ к одному Enterprise-бину должен осуществляться несколькими потоками одновременно.

- Приложению требуется Enterprise-бин для выполнения задач при запуске и завершении работы приложения.
- Бин реализует веб-сервис.

Что такое бин, управляемый сообщениями?

Бин, управляемый сообщениями, — это Enterprise-бин, который позволяет приложениям Jakarta EE асинхронно обрабатывать сообщения. This type of bean normally acts as a Jakarta Messaging message listener, which is similar to an event listener but receives Jakarta Messaging messages instead of events. Сообщения могут отправляться любым компонентом Jakarta EE (клиентское приложение, EJB-компонент или веб-компонент), Jakarta Messaging или системой, не использующей Jakarta EE. Компоненты, управляемые сообщениями, могут обрабатывать сообщения Jakarta Messaging или другие виды сообщений.

Что отличает бины, управляемые сообщениями, от сессионных?

Наиболее заметное различие между компонентами, управляемыми сообщениями, и сессионными компонентами заключается в том, что клиенты не получают доступ к компонентам, управляемым сообщениями, через интерфейсы. Интерфейсы описаны в разделе Доступ к Enterprise-бинам. В отличие от сессионного компонента, управляемый сообщениями компонент имеет только класс компонента.

В некотором отношении управляемый сообщениями компонент напоминает сессионный компонент без сохранения состояния.

- Объекты компонента, управляемого сообщениями, не сохраняют данные или состояние диалога для конкретного клиента.
- Все объекты компонента, управляемого сообщениями, эквивалентны, что позволяет EJB-контейнеру назначать сообщение любому объекту компонента, управляемого сообщениями. Контейнер может объединять эти объекты в пулы для одновременной обработки потоков сообщений.
- Один управляемый сообщениями компонент может обрабатывать сообщения от нескольких клиентов.

Переменные объекта управляемого сообщениями компонента могут содержать некоторое состояние при обработке клиентских сообщений, например, соединение Jakarta Messaging, открытое соединение с базой данных или ссылку на объект EJB-компонента.

Клиентские компоненты не используют поиск JNDI для доступа к управляемым сообщениями компонентам и не вызывают их методы непосредственно. Вместо этого клиент получает доступ к управляемому сообщениям bean через, например, Jakarta Messaging, отправляя сообщения адресату сообщения, для которого управляемый сообщениями класс компонента является `MessageListener`. Вы назначаете пункт назначения бина, управляемого сообщениями, во время развёртывания, используя ресурсы GlassFish Server.

Бины, управляемые сообщениями, имеют следующие характеристики.

- Они выполняются при получении одного сообщения клиента.
- Они вызываются асинхронно.
- Они относительно недолговечны.
- Они не представляют напрямую общие данные в базе данных, но они могут получить доступ и обновить эти данные.
- Они могут быть осведомлены о транзакции.
- Они не сохраняют состояние.

Когда приходит сообщение, контейнер вызывает метод `onMessage` объекта, управляемого сообщениями, для обработки сообщения. Метод `onMessage` обычно приводит сообщение к одному из пяти типов сообщений Jakarta Messaging и обрабатывает его в соответствии с бизнес-логикой приложения. Метод `onMessage` может вызывать вспомогательные методы или может вызывать сессионный компонент для обработки информации в сообщении или для сохранения её в базе данных.

Сообщение может быть доставлено управляемому сообщениями компоненту в контексте транзакции, поэтому все операции в методе `onMessage` являются частью одной транзакции. Если обработка сообщения откатывается, сообщение будет доставлено. Дополнительные сведения см. в Получение сообщений асинхронно с использованием компонента, управляемого сообщениями и главе 55 *Транзакции*.

Когда использовать бины, управляемые сообщениями

Сессионные компоненты позволяют отправлять сообщения Jakarta Messaging и получать их синхронно, но не асинхронно. Чтобы избежать связывания ресурсов сервера, не используйте блокировку синхронного приёма в серверном компоненте. Как правило, сообщения Jakarta Messaging не следует отправлять или получать синхронно. Для асинхронного получения сообщений используйте бин, управляемый сообщениями.

Доступ к Enterprise-бинам

> Материал в этом разделе применим только к сессионным компонентам, но не к компонентам, управляемым сообщениями. Поскольку компоненты, управляемые сообщениями, используют другую программную модель, они не имеют интерфейсов или представлений без интерфейса, которые допускали бы доступ клиента.

Клиенты получают доступ к Enterprise-бинам через представление без интерфейса или через бизнес-интерфейс. Представление Enterprise-бина без интерфейса предоставляет клиентам публичные методы класса реализации Enterprise-бина. Клиенты, использующие представление без интерфейса Enterprise-бина, могут вызывать любые публичные методы в классе реализации Enterprise-бина или любых его родительских классах. Бизнес-интерфейс — это стандартный интерфейс Java, который содержит бизнес-методы Enterprise-бина.

Клиент может получить доступ к сессионному компоненту только через методы, определённые в бизнес-интерфейсе компонента, или через общедоступные методы Enterprise-бина, который не имеет представления интерфейса. Бизнес-интерфейс или представление без интерфейса определяет представление клиента Enterprise-бина. Все остальные аспекты Enterprise-бина (реализации методов и параметры развёртывания) скрыты от клиента.

Хорошо продуманные интерфейсы и представления без интерфейса упрощают разработку и обслуживание приложений Jakarta EE. Чистые интерфейсы и представления без интерфейса не только защищают клиентов от любых сложностей слоя бизнес-логики, но также позволяют EJB-компонентам изменяться внутри, не затрагивая клиентов. Например, при изменении реализации бизнес-метода сессионного компонента нет необходимости изменять клиентский код. Но если изменены сигнатуры методов в интерфейсах, то, возможно, придётся внести изменения и в клиентский код. Поэтому важно тщательно проектировать интерфейсы и представления без интерфейса, чтобы изолировать клиентов от возможных изменений в Enterprise-бинах.

Сессионные компоненты могут иметь более одного бизнес-интерфейса. Сессионные компоненты могут (но не обязаны) реализовать свой бизнес-интерфейс или интерфейсы.

Использование Enterprise-бинов в клиентах

Клиент Enterprise-бина получает ссылку на объект Enterprise-бина посредством инъецирования зависимости с аннотацией Java или поиска JNDI с использованием синтаксиса Java Naming and Directory Interface для поиска Enterprise-бина.

Инъецирование зависимостей — это самый простой способ получения ссылки на Enterprise-бин. Клиенты, которые работают в управляемой сервером среде Jakarta EE, веб-приложениях Jakarta Faces, RESTful веб-сервисах, других Enterprise-бинах или клиентах приложениях Jakarta EE поддерживают инъецирование зависимостей аннотацией `jakarta.ejb.EJB`.

Приложения, выполняющиеся вне среды окружения Jakarta EE, например, приложения Java SE, должны выполнять явный поиск. JNDI поддерживает глобальный синтаксис для идентификации компонентов Jakarta EE, чтобы упростить этот явный поиск.

Переносимый синтаксис JNDI

Переносимый поиск JNDI использует три пространства имён JNDI: `java:global`, `java:module` и `java:app`.

- Пространство имён JNDI `java:global` — это переносимый способ поиска удалённых Enterprise-бинов с помощью механизма JNDI. Адреса JNDI имеют следующую форму:

  ```
  java:global[/application name]/module name/enterprise bean name[/interface name]
  ```

  Имя приложения и имя модуля по умолчанию соответствуют именам приложения и модуля без расширения файла. Имена приложений требуются, только если приложение упаковано в EAR. Имя интерфейса требуется только в том случае, если Enterprise-бин реализует более одного бизнес-интерфейса.

- Пространство имён `java:module` используется для поиска локальных Enterprise-бинов внутри одного и того же модуля. Адреса JNDI, использующие пространство имён `java:module`, имеют следующую форму:

  ```
  java:module/enterprise bean name/[interface name]
  ```

  Имя интерфейса требуется только в том случае, если Enterprise-бин реализует более одного бизнес-интерфейса.

- Пространство имён `java:app` используется для поиска локальных Enterprise-бинов, упакованных в одном приложении. То есть Enterprise-бин упакован в файл EAR, содержащий несколько модулей Jakarta EE. Адреса JNDI, использующие пространство имён `java:app`, имеют следующую форму:

  ```
  java:app[/module name]/enterprise bean name[/interface name]
  ```

  Имя модуля необязательно. Имя интерфейса требуется только в том случае, если Enterprise-бин реализует более одного бизнес-интерфейса.

Например, если Enterprise-бин `MyBean` упакован в архив веб-приложения `myApp.war`, имя модуля будет `myApp`. Имя переносимого JNDI: `java:module/MyBean`. Эквивалентное имя JNDI, использующее пространство имён `java:global`: `java:global/myApp/MyBean`.

Выбор удалённого или локального доступа

При разработке приложения Jakarta EE одним из первых решений, которое вы принимаете, является тип клиентского доступа, разрешённого Enterprise-бинами: удалённый, локальный или веб-сервис.

Разрешить ли локальный или удалённый доступ зависит от следующих факторов.

- Сильная или слабая связность бинов: сильносвязные бины зависят друг от друга. Например, если сессионный компонент, обрабатывающий заказы на продажу, вызывает сессионный компонент, который отправляет клиенту сообщение с подтверждением, эти компоненты сильно связаны. Сильносвязные бины — хорошие кандидаты для локального доступа. Поскольку они подходят друг к другу как логическая единица, они, как правило, часто вызывают друг друга и выигрывают от повышения производительности, которое возможно при локальном доступе.
- Тип клиента: если к Enterprise-бину обращаются клиентские приложения, он должен разрешить удалённый доступ. В производственной среде эти клиенты почти всегда работают на компьютерах, отличных от GlassFish Server. Если клиентами Enterprise-бина являются веб-компоненты или другие Enterprise-бины, тип доступа зависит от того, как вы хотите распространять свои компоненты.
- Распределение компонентов: приложения Jakarta EE являются масштабируемыми, поскольку их серверные компоненты могут быть распределены по нескольким компьютерам. Например, в распределённом приложении сервер, на котором работают веб-компоненты, может быть не тем, на котором развёрнуты Enterprise-бины, к которым они получают доступ. В этом распределённом сценарии Enterprise-бины должны разрешать удалённый доступ.
- Производительность: из-за таких факторов, как задержки в сети, удалённые вызовы могут быть медленнее локальных. С другой стороны, при распределении компонентов между различными серверами можно повысить общую производительность приложения. Оба эти утверждения являются обобщениями. Производительность может варьироваться в разных операционных средах. Тем не менее, нужно помнить, как дизайн приложения может повлиять на производительность.

Если нет уверенности, какой тип доступа должен иметь Enterprise-бин, выбирайте удалённый доступ. Это решение даёт больше гибкости. Остаётся возможность в будущем распределить компоненты в соответствии с растущими требованиями к приложению.

Хотя это редко встречается, Enterprise-бин может разрешить как удалённый, так и локальный доступ. В этом случае либо бизнес-интерфейс компонента должен быть аннотирован `@Remote` или `@Local`, либо класс компонента должен явно обозначать бизнес-интерфейсы аннотациями `@Remote` и `@Local`. Один и тот же бизнес-интерфейс не может быть как локальным, так и удалённым бизнес-интерфейсом.

Локальные клиенты

Локальный клиент имеет следующие характеристики.

- Он должен работать в том же приложении, что и Enterprise-бин, к которому он обращается.
- Это может быть веб-компонент или другой Enterprise-бин.
- Для локального клиента местоположение Enterprise-бина, к которому он обращается, не является прозрачным.

Представление Enterprise-бина без интерфейса является локальным представлением. Публичные методы класса реализации Enterprise-бина доступны локальным клиентам, которые обращаются к представлению Enterprise-бина без интерфейса. Enterprise-бины, которые используют представление без интерфейса, не реализуют бизнес-интерфейс.

Локальный бизнес-интерфейс определяет бизнес-методы и методы жизненного цикла компонента. Если бизнес-интерфейс бина не аннотирован `@Local` или `@Remote`, и если класс бина не указывает интерфейс аннотациями `@Local` или `@Remote`, бизнес-интерфейс по умолчанию является локальным интерфейсом.

Чтобы создать Enterprise-бин, разрешающий только локальный доступ, вы можете, но не обязаны, выполнить одно из следующих действий.

- Создайте класс реализации Enterprise-бина, который не реализует бизнес-интерфейс, указывая, что компонент предоставляет представление без интерфейса клиентам. Например:

```java
@Session
public class MyBean { ... }
```

- Аннотируйте бизнес-интерфейс Enterprise-бина как интерфейс `@Local` . Например:

```java
@Local
public interface InterfaceName { ... }
```

- Укажите интерфейс класса компонента в аннотации `@Local` и укажите имя интерфейса. Например:

```java
@Local(InterfaceName.class)
public class BeanName implements InterfaceName { ... }
```

Доступ к локальным Enterprise-бинам с использованием представления без интерфейса

Клиентский доступ к Enterprise-бину, который предоставляет локальное представление без интерфейса, осуществляется посредством инъецирования зависимостей или поиска JNDI.

- Чтобы получить ссылку на представление корпоративного компонента без интерфейса через инъецирование зависимостей, используйте аннотацию `jakarta.ejb.EJB` и укажите класс реализации Enterprise-бина:

```java
@EJB
ExampleBean exampleBean;
```

- Чтобы получить ссылку на представление без интерфейса для Enterprise-бина через поиск JNDI, используйте метод `lookup` интерфейса `javax.naming.InitialContext` :

```java
ExampleBean exampleBean = (ExampleBean)
        InitialContext.lookup("java:module/ExampleBean");
```

Клиенты не используют оператор `new` для получения нового объекта Enterprise-бина, который использует представление без интерфейса.

Доступ к локальным Enterprise-бинам, реализующим бизнес-интерфейсы

Клиентский доступ к Enterprise-бинам, которые реализуют локальные бизнес-интерфейсы, осуществляется посредством инъецирования зависимостей или поиска JNDI.

- Чтобы получить ссылку на локальный бизнес-интерфейс Enterprise-бина через инъецирование зависимостей, используйте метод `jakarta.ejb.EJB` и укажите имя локального бизнес-интерфейса Enterprise-бина:

```java
@EJB
Example example;
```

- Чтобы получить ссылку на локальный бизнес-интерфейс Enterprise-бина с помощью поиска JNDI, используйте метод `lookup` интерфейса `javax.naming.InitialContext` :

```java
ExampleLocal example = (ExampleLocal)
         InitialContext.lookup("java:module/ExampleLocal");
```

Удалённые клиенты

Удалённый клиент Enterprise-бина имеет следующие особенности.

- Он может работать на другой машине и на другой JVM, чем Enterprise-бин, к которому он обращается. (Не обязательно работать на другой JVM.)
- Это может быть веб-компонент, клиентское приложение или другой Enterprise-бин.
- Для удалённого клиента местоположение Enterprise-бина прозрачно.
- Enterprise-бин должен реализовывать бизнес-интерфейс. То есть удалённые клиенты не могут получить доступ к Enterprise-бину через представление без интерфейса.

Чтобы создать Enterprise-бин, который разрешает удалённый доступ, вы должны сделать одно из двух:

- Аннотировать бизнес-интерфейс Enterprise-бина аннотацией `@Remote`:

```java
@Remote
public interface InterfaceName { ... }
```

- Аннотировать класс компонента аннотацией `@Remote`, указав бизнес-интерфейс или интерфейсы:

```java
@Remote(InterfaceName.class)
public class BeanName implements InterfaceName { ... }
```

Удалённый интерфейс определяет бизнес- методы и методы жизненного цикла, специфичные для компонента. Например, удалённый интерфейс компонента `BankAccountBean` может иметь бизнес-методы `deposit` и `credit`. На рисунке 35-1 показано, как интерфейс управляет клиентским представлением Enterprise-бина.

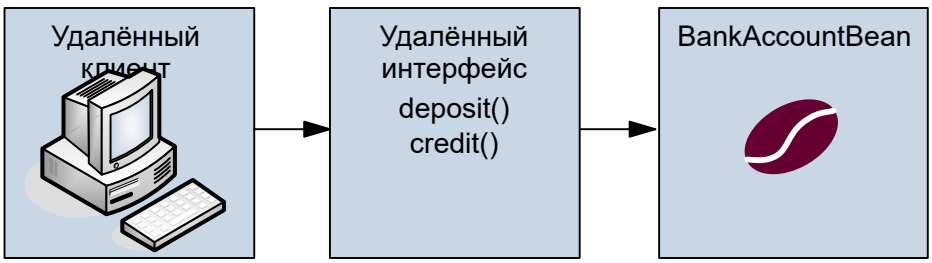

Рис. 35-1. Интерфейсы для Enterprise-бина с удалённым доступом

Клиентский доступ к Enterprise-бину, который реализует удалённый бизнес-интерфейс, осуществляется посредством инъецирования зависимостей или поиска JNDI.

- Чтобы получить ссылку на удалённый бизнес-интерфейс Enterprise-бина с помощью инъецирования зависимостей, используйте метод `jakarta.ejb.EJB` и укажите имя удалённого бизнес-интерфейса Enterprise-бина:

```java
@EJB
Example example;
```

- Чтобы получить ссылку на удалённый бизнес-интерфейс Enterprise-бина через поиск JNDI, используйте метод `lookup` интерфейса `javax.naming.InitialContext`:

```java
ExampleRemote example = (ExampleRemote)
        InitialContext.lookup("java:global/myApp/ExampleRemote");
```

Клиенты веб-сервисов

Клиент веб-сервиса может получить доступ к приложению Jakarta EE двумя способами. Во-первых, клиент может получить доступ к веб-сервису, созданному с помощью XML веб-сервисов Jakarta. (Дополнительные сведения о Jakarta XML Web Services см. в главе 31 *Создание веб-сервисов с Jakarta XML Web Services*.) Во-вторых, клиент веб-сервиса может вызывать бизнес-методы сессионного компонента без состояния. Бины сообщения не могут быть доступны клиентам веб-сервиса.

При условии, что он использует корректные протоколы (SOAP, HTTP, WSDL), любой клиент веб-сервиса может получить доступ к сессионному компоненту без сохранения состояния независимо от того, написан ли этот клиент на Java. Клиент даже не «знает», какая технология реализует сервис: сессионный компонент без сохранения состояния, XML веб-сервис Jakarta или что-то другое. Кроме того, Enterprise-бины и веб-компоненты могут быть клиентами веб-сервисов. Эта гибкость позволяет интегрировать приложения Jakarta EE с веб-сервисами.

Клиент веб-сервиса обращается к сессионному компоненту без сохранения состояния через класс реализации конечной точки веб-сервиса. По умолчанию все публичные методы в классе бина доступны для клиентов веб-сервисов. Аннотация `@WebMethod` может использоваться для настройки поведения методов веб-сервиса. Если для аннотирования методов класса компонента используется аннотация `@WebMethod`, клиентам веб-сервиса доступны только те методы, которые аннотированы `@WebMethod`.

Пример кода см. в разделе Пример веб-сервиса: helloservice.

Параметры метода и доступ

Тип доступа влияет на параметры методов бина, которые вызываются клиентами. Следующие разделы относятся не только к параметрам метода, но и к возвращаемым значениям.

Изоляция

Параметры удалённых вызовов более изолированы, чем параметры локальных вызовов. При удалённых вызовах клиент и компонент работают с разными копиями объекта параметра. Если клиент изменяет значение объекта, значение копии в компоненте не изменяется. Этот уровень изоляции может помочь защитить компонент, если клиент случайно изменил данные.

При локальном вызове и клиент, и компонент могут изменять один и тот же объект параметра. Вообще говоря, вы не должны полагаться на этот побочный эффект локальных вызовов. Возможно, когда-нибудь вы захотите распределить свои компоненты, заменив локальные вызовы удалёнными.

Как и в случае с удалёнными клиентами, клиенты веб-сервисов работают с копиями параметров, отличными от компонентов, реализующих веб-сервис.

Детальность доступа к данным

Поскольку удалённые вызовы, скорее всего, будут медленнее, чем локальные, параметры в удалённых методах должны быть относительно крупными. Крупный объект содержит больше данных, чем мелкий, поэтому требуется меньше вызовов для доступа. По той же причине параметры методов, вызываемых клиентами веб-сервисов, также должны быть крупными.

Содержимое Enterprise-бина

Для разработки Enterprise-бина вы должны предоставить следующие файлы.

- Класс Enterprise-бина: реализует бизнес-методы Enterprise-бина и Callback-методы жизненного цикла.
- Бизнес-интерфейсы. Определите бизнес-методы, реализуемые классом EJB. Бизнес-интерфейс не требуется, если Enterprise-бин предоставляет локальное представление без интерфейса.
- Вспомогательные классы: другие классы, необходимые классу Enterprise-бина, такие как классы исключений и служебные классы.

Упакуйте программные артефакты из предыдущего списка либо в JAR-файл EJB (автономный модуль, в котором хранится EJB-компонент), либо в модуль архива веб-приложения (WAR). Дополнительную информацию см. в Упаковка Enterprise-бинов в модули EJB JAR и Упаковка Enterprise-бинов в модули WAR.

Соглашения об именовании Enterprise-бинов

Поскольку Enterprise-бины состоят из нескольких частей, полезно следовать соглашениям об именовании в ваших приложениях. Таблица 35-2 суммирует соглашения для примеров бинов в этом руководстве.

Таблица 35-2 Соглашения об именовании Enterprise-бинов

Элемент	**Синтаксис**	**Пример**
Имя Enterprise-бина	*name*Bean	`AccountBean`
Класс Enterprise-бина	*name*Bean	`AccountBean`
Бизнес-интерфейс	*name*	`Account`

Жизненные циклы Enterprise-бинов

Enterprise-бин проходит через различные этапы в течение своего жизненного цикла. Каждый тип Enterprise-бина (сессионный с сохранением состояния, сессионный без сохранения состояния, сессионный синглтон, управляемый сообщениями) имеет свой жизненный цикл.

Следующие описания относятся к методам, которые объясняются вместе с примерами кода в следующих двух главах. Если вы новичок в Enterprise-бинах, вам стоит пропустить этот раздел и сначала выполнить примеры кода.

Жизненный цикл сессионного бина с сохранением состояния

Рисунок 35-2 иллюстрирует этапы, через которые сессионный компонент с сохранением состояния проходит через свою жизнь. Клиент инициирует жизненный цикл, получая ссылку на сессионный компонент с состоянием. Контейнер выполняет инъецирование всех зависимостей, а затем вызывает метод, аннотированный `@PostConstruct`, если таковой имеется. Теперь компонент готов к вызову его бизнес-методов клиентом.

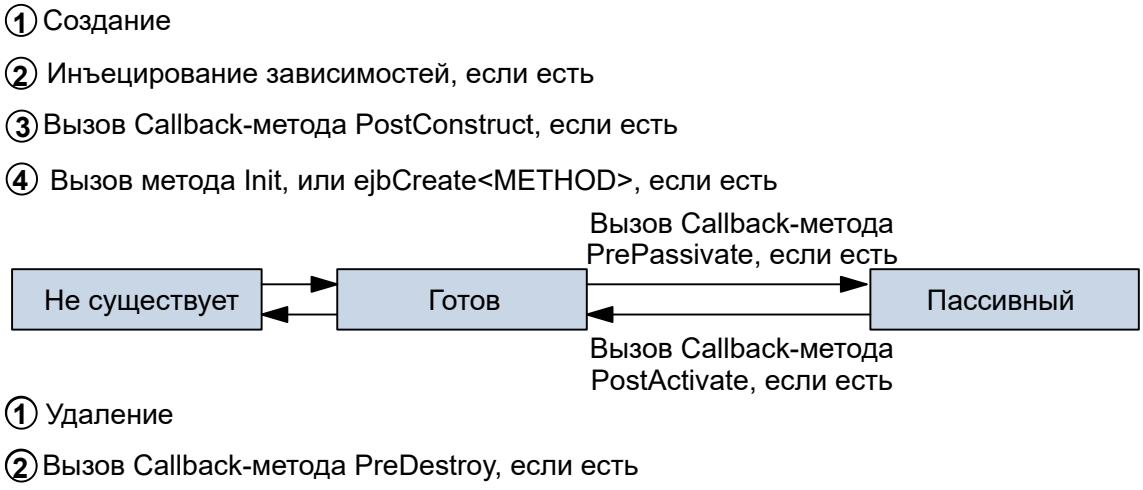

Рис. 35-2. Жизненный цикл сессионного компонента с сохранением состояния

На стадии готовности EJB-контейнер может решить деактивировать (пассивировать) компонент, переместив его из памяти во вторичное хранилище. (Как правило, EJB-контейнер при выборе компонента для пассивации использует наименее восстребованные компоненты.) EJB-контейнер вызывает метод, аннотированный `@PrePassivate`, если таковой имеется, непосредственно перед его пассивацией. Если клиент вызывает бизнес-метод компонента, когда он находится в стадии пассивации, EJB-контейнер активирует компонент, вызывает метод с аннотацией `@PostActivate`, если он есть, и затем перемещает его в стадию котовности.

В конце жизненного цикла клиент вызывает метод, аннотированный `@Remove`, а EJB-контейнер вызывает метод, аннотированный `@PreDestroy`, если таковой имеется. Объект компонента готов к сборке мусора.

Ваш код управляет вызовом только одного метода жизненного цикла — аннотированного `@Remove`. Все другие методы на рисунке 35-2 вызываются EJB-контейнером. См. главу 56 *Адаптеры и контракты ресурсов* для получения дополнительной информации.

Жизненный цикл сессионного бина без сохранения состояния

Поскольку сессионный компонент без сохранения состояния никогда не деактивируется, его жизненный цикл состоит только из двух этапов: несуществующего и готового к вызову бизнес-методов. Рисунок 35-3 иллюстрирует этапы сессионного компонента без сохранения состояния.

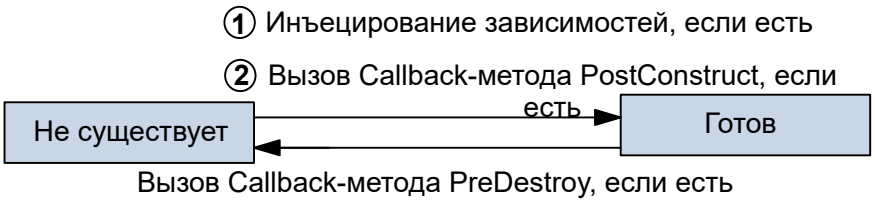

Рис. 35-3. Жизненный цикл сессионного компонента без состояния или синглтона

EJB-контейнер обычно создаёт и поддерживает пул сессионных компонентов без состояния, начиная жизненный цикл сессионного компонента без состояния. Контейнер выполняет инъецирование всех зависимостей, а затем вызывает аннотированный `@PostConstruct` метод, если он существует. Теперь компонент готов к тому, чтобы его бизнес-методы вызывались клиентом.

В конце жизненного цикла EJB-контейнер вызывает метод, аннотированный `@PreDestroy`, если он существует. Объект компонента готов к сборке мусора.

Жизненный цикл сессионного бина-синглтона

Как и сессионный компонент без сохранения состояния, сессионный компонент-синглтон никогда не деактивируется и имеет только два этапа: не существует и готов к вызову бизнес-методов, как показано на рис. 35-3.

EJB-контейнер инициирует жизненный цикл сессионного компонента-синглтона путём создания объекта синглтона. Если синглтон помечен аннотацией `@Startup`, это происходит при развёртывании приложения. Контейнер выполняет инъецирование всех зависимостей и затем вызывает метод, аннотированный `@PostConstruct`, если он существует. Теперь сессионный компонент-синглтон готов к тому, чтобы его бизнес-методы вызывались клиентом.

В конце жизненного цикла EJB-контейнер вызывает метод, аннотированный `@PreDestroy`, если он существует. Сессионный компонент-синглтон теперь готов для сборки мусора.

Жизненный цикл бина, управляемого сообщениями

Рисунок 35-4 иллюстрирует этапы жизненного цикла бина, управляемого сообщениями.

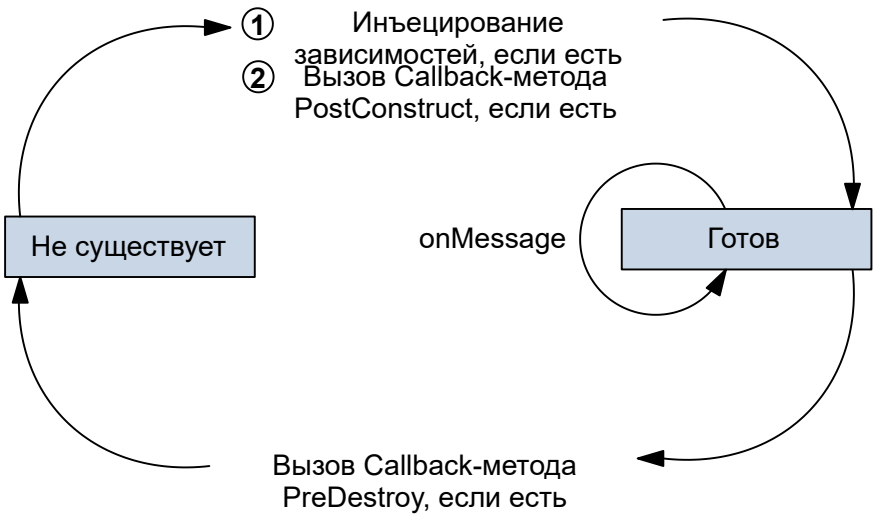

Рис. 35-4. Жизненный цикл бина, управляемого сообщениями

EJB-контейнер обычно создаёт пул объектов компонентов, управляемых сообщениями. Для каждого компонента EJB-контейнер выполняет эти задачи.

1. Если управляемый сообщениями компонент использует инъецирование зависимостей, контейнер инъецирует зависимости до инстанцирования объекта.
2. Контейнер вызывает аннотированный `@PostConstruct` метод, если таковой имеется.

Как и сессионный компонент без сохранения состояния, управляемый сообщениями компонент никогда не деактивируется и имеет только два состояния: не существует и готов к приёму сообщений.

В конце жизненного цикла контейнер вызывает аннотированный `@PreDestroy` метод, если таковой имеется. Объект компонента готов к сборке мусора.

Дополнительная информация об Enterprise-бинах

Дополнительные сведения о Jakarta Enterprise Beans см. в спецификации Jakarta Enterprise Beans 4.0: https://jakarta.ee/specifications/enterprise-beans/4.0/

Глава 36. Начало работы с Enterprise-бинами

В этой главе показано, как разработать, развернуть и запустить простое приложение Jakarta EE с именем `converter`, которое обращается к EJB-компоненту для выполнения бизнес-логики. Целью `converter` является конвертация валют между японской иеной, евро и долларами США. Приложение `converter` состоит из Enterprise-бина, выполняющего расчёт, и веб-клиента.

Начало работы с Enterprise-бинами

Вот обзор выполняемых шагов:

1. Создайте Enterprise-бин `ConverterBean`.
2. Создайте веб-клиент.
3. Разверните `converter` на сервере.
4. Используя браузер, запустите веб-клиент.

Прежде чем продолжить, убедитесь, что вы сделали следующее:

- Прочли главу 1 *Обзор*
- Ознакомились с Enterprise-бинами (см. Часть VII «Enterprise-бины»)
- Запустили сервер (см. Запуск и остановка GlassFish Server)

Создание Enterprise-бина

В нашем примере Enterprise-бин — это сессионный компонент без сохранения состояния `ConverterBean`. Исходный код для `ConverterBean` находится в каталоге *`tut-install`*`/examples/ejb/converter/src/main/java/`.

Создание `ConverterBean` требует следующих шагов:

1. Кодирование класса реализации компонента (предоставляется исходный код)
2. Компиляция исходного кода

Кодирование класса Enterprise-бина

Класс EJB для этого примера — `ConverterBean`. Этот класс реализует два бизнес-метода: `dollarToYen` и `yenToEuro`. Поскольку класс Enterprise-бина не реализует бизнес-интерфейс, Enterprise-бин предоставляет локальное представление без интерфейса. Публичные методы в классе Enterprise-бина доступны клиентам, которые получают ссылку на `ConverterBean`. Исходный код для класса `ConverterBean` выглядит следующим образом:

JAVA

```java
package ee.jakarta.tutorial.converter.ejb;

import java.math.BigDecimal;
import jakarta.ejb.*;

@Stateless
public class ConverterBean {
    private BigDecimal yenRate = new BigDecimal("83.0602");
    private BigDecimal euroRate = new BigDecimal("0.0093016");

    public BigDecimal dollarToYen(BigDecimal dollars) {
        BigDecimal result = dollars.multiply(yenRate);
        return result.setScale(2, BigDecimal.ROUND_UP);
    }

    public BigDecimal yenToEuro(BigDecimal yen) {
        BigDecimal result = yen.multiply(euroRate);
        return result.setScale(2, BigDecimal.ROUND_UP);
    }
}
```

Обратите внимание на аннотацию `@Stateless`, которая декорирует класс Enterprise-бина. Эта аннотация даёт контейнеру понять, что `ConverterBean` является сессионным компонентом без сохранения состояния.

Создание веб-клиента converter

Веб-клиент содержится в классе сервлетов в каталоге *`tut-install`*`/examples/ejb/converter/src/main/java/`:

```
converter/web/ConverterServlet.java
```

Сервлет Jakarta — это веб-компонент, который отвечает на запросы HTTP.

Класс `ConverterServlet` использует инъецирование зависимостей для получения ссылки на `ConverterBean`. Аннотация `jakarta.ejb.EJB` добавляется к объявлению приватной переменной свойства `converter`, который имеет тип `ConverterBean`. `ConverterBean` предоставляет локальное представление без интерфейса, поэтому класс реализации Enterprise-бина является типом переменной:

JAVA

```java
@WebServlet(urlPatterns="/")
public class ConverterServlet extends HttpServlet {
  @EJB
  ConverterBean converter;
  ...
}
```

Когда пользователь вводит сумму, которая будет конвертирована в иены и евро, сумма извлекается из параметров запроса. Тогда вызываются методы `ConverterBean.dollarToYen` и `ConverterBean.yenToEuro`:

JAVA

```
...
try {
  String amount = request.getParameter("amount");
  if (amount != null && amount.length() > 0) {
    // конвертация amount в BigDecimal из параметра запроса
    BigDecimal d = new BigDecimal(amount);
    // вызов метода ConverterBean.dollarToYen() для получение amount
    // в йенах
    BigDecimal yenAmount = converter.dollarToYen(d);

    // вызов метода ConverterBean.yenToEuro() для получение amount
    // в евро
    BigDecimal euroAmount = converter.yenToEuro(yenAmount);
    ...
  }
  ...
}
```

Результаты отображаются пользователю.

Запуск converter

Теперь вы готовы скомпилировать класс Enterprise-бина (`ConverterBean.java`) и класс сервлета (`ConverterServlet.java`) и упаковать скомпилированные классы в WAR-файл. Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска примера `converter` .

Запуск converter с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/ejb
   ```

4. Выберите каталог `converter` .
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `converter` и выберите **Сборка**.
7. Откройте веб-браузер по следующему URL:

   ```
   http://localhost:8080/converter
   ```

8. На странице Servlet ConverterServlet введите `100` в поле и нажмите «Отправить».

   Откроется вторая страница, показывающая преобразованные значения.

Запуск converter с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/ejb/converter/
   ```

3. Введите следующую команду:

   SHELL

   ```
   mvn install
   ```

Эта команда компилирует исходные файлы для Enterprise-бина и сервлета, упаковывает проект в модуль WAR (`converter.war`) и развёртывает WAR на сервере. Для получения дополнительной информации о Maven см. Сборка примеров.

4. Откройте веб-браузер по следующему URL:

   ```
   http://localhost:8080/converter
   ```

5. На странице Servlet ConverterServlet введите `100` в поле и нажмите «Отправить».

   Откроется вторая страница, показывающая преобразованные значения.

Изменение приложения Jakarta EE

GlassFish Server поддерживает итеративную разработку. Всякий раз при внесении изменений в приложение Jakarta EE вы должны повторно развернуть его.

Изменение файла класса

Чтобы изменить файл класса в Enterprise-бине, измените исходный код, перекомпилируйте его и повторно разверните приложение. Например, чтобы обновить обменный курс в бизнес-методе `dollarToYen` класса `ConverterBean` , выполните следующие действия.

Процедура изменения `ConverterServlet` аналогична.

1. Отредактируйте `ConverterBean.java` и сохраните файл.
2. Перекомпилируйте исходный файл.
   a. Чтобы перекомпилировать `ConverterBean.java` в IDE NetBeans, кликните правой кнопкой мыши проект `converter` и выберите «Выполнить».

      Это перекомпилирует файл `ConverterBean.java` , заменяет старый файл класса в каталоге сборки и повторно развёртывает приложение в GlassFish Server.

   b. Перекомпилируйте `ConverterBean.java` , используя Maven.

      i. В окне консоли перейдите в каталог *`tut-install`*`/examples/ejb/converter/` .

      ii. Введите следующую команду:

      ```
      mvn install
      ```
      SHELL

      Эта команда переупаковывает и развёртывает приложение.

Глава 37. Запуск примеров Enterprise-бина

В этой главе описаны примеры Jakarta Enterprise Beans. Сессионные компоненты предоставляют простой, но эффективный способ инкапсулировать бизнес-логику в приложении. Доступ к ним можно получить с удалённых клиентов Java, клиентов веб-сервисов и компонентов, работающих на одном сервере.

Обзор примеров Jakarta Enterprise Beans

В главе 36 *Начало работы с Enterprise-бинами* был создан сессионный компонент без сохранения состояния с именем `ConverterBean`. В этой главе рассматривается исходный код ещё четырёх сессионных компонентов:

- `CartBean`: сессионный компонент с сохранением состояния, к которому обращается удалённый клиент
- `CounterBean`: сессионный компонент-синглтон
- `HelloServiceBean`: сессионный компонент без сохранения состояния, который реализует веб-сервис
- `TimerSessionBean`: сессионный компонент без сохранения состояния, который устанавливает таймер

Пример cart

Пример `cart` представляет корзину покупок в книжном онлайн-магазине и использует сессионный компонент с состоянием для управления операциями с корзиной покупок. Клиент бина может добавить книгу в корзину, удалить книгу или получить содержимое корзины. Для сборки `cart` понадобится следующий код:

- Класс сессионного компонента (`CartBean`)
- Удалённый бизнес-интерфейс (`Cart`)

Все сессионные компоненты имеют соответствующий им класс Java. Все Enterprise-бины, которые разрешают удалённый доступ, должны иметь удалённый бизнес-интерфейс. Для удовлетворения потребностей конкретного приложения Enterprise-бину также могут потребоваться некоторые вспомогательные классы. Сессионный компонент `CartBean` использует два вспомогательных класса — `BookException` и `IdVerifier`, которые обсуждаются в разделе Вспомогательные классы.

Исходный код для этого примера находится в каталоге *`tut-install`*`/examples/ejb/cart/`.

Бизнес-интерфейс

Бизнес-интерфейс `Cart` — это интерфейс Java, который определяет все бизнес-методы, реализованные в классе компонента. Если класс бина реализует один интерфейс, этот интерфейс считается бизнес-интерфейсом. Бизнес-интерфейс является локальным, если он не аннотирован `jakarta.ejb.Remote`. Аннотация `jakarta.ejb.Local` в этом случае необязательна.

Класс бина может реализовывать более одного интерфейса. В этом случае бизнес-интерфейсы должны быть либо явно аннотированы `@Local` или `@Remote`, либо указаны путём аннотирования класса компонента аннотациями `@Local` или `@Remote`. Однако следующие интерфейсы исключаются при определении того, реализует ли класс бина более одного интерфейса:

- `java.io.Serializable`
- `java.io.Externalizable`
- Любой из интерфейсов, определённых в пакете `jakarta.ejb`

Исходный код бизнес-интерфейса `Cart` выглядит следующим образом:

JAVA

```
package ee.jakarta.tutorial.cart.ejb;

import cart.util.BookException;
import java.util.List;
import jakarta.ejb.Remote;

@Remote
public interface Cart {
    public void initialize(String person) throws BookException;
    public void initialize(String person, String id) throws BookException;
    public void addBook(String title);
    public void removeBook(String title) throws BookException;
    public List<String> getContents();
    public void remove();
}
```

Класс сессионного бина

Класс сессионного компонента для этого примера называется `CartBean`. Как и любой сессионный компонент с состоянием, класс `CartBean` должен соответствовать следующим требованиям.

- Класс аннотирован `@Stateful`.
- Класс реализует бизнес-методы, определённые в бизнес-интерфейсе.

Сессионные компоненты с сохранением состояния могут также выполнять следующие действия.

- Реализовать бизнес-интерфейс — интерфейс Java Реализовать бизнес-интерфейс — хорошая практика.
- Реализовать любые дополнительные Callback-методы жизненного цикла, аннотированные `@PostConstruct`, `@PreDestroy`, `@PostActivate` и `@PrePassivate`.
- Реализовать любые необязательные бизнес-методы, аннотированные `@Remove`.

Исходный код для класса `CartBean` выглядит следующим образом:

JAVA

```java
package ee.jakarta.tutorial.cart.ejb;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import ee.jakarta.tutorial.cart.util.BookException;
import ee.jakarta.tutorial.cart.util.IdVerifier;
import jakarta.ejb.Remove;
import jakarta.ejb.Stateful;

@Stateful
public class CartBean implements Cart {
    String customerId;
    String customerName;
    List<String> contents;

    @Override
    public void initialize(String person) throws BookException {
        if (person == null) {
            throw new BookException("Null person not allowed.");
        } else {
            customerName = person;
        }
        customerId = "0";
        contents = new ArrayList<>();
    }

    @Override
    public void initialize(String person, String id)
                throws BookException {
        if (person == null) {
            throw new BookException("Null person not allowed.");
        } else {
            customerName = person;
        }

        IdVerifier idChecker = new IdVerifier();
        if (idChecker.validate(id)) {
            customerId = id;
        } else {
            throw new BookException("Invalid id: " + id);
        }

        contents = new ArrayList<>();
    }

    @Override
    public void addBook(String title) {
        contents.add(title);
    }

    @Override
    public void removeBook(String title) throws BookException {
        boolean result = contents.remove(title);
        if (result == false) {
            throw new BookException("\"" + title + " not in cart.");
        }
    }

    @Override
    public List<String> getContents() {
        return contents;
    }

    @Remove
    @Override
    public void remove() {
```

```
        contents = null;
    }
}
```

Callback-методы жизненного цикла

Метод в классе бина может быть объявлен как Callback-метод жизненного цикла, добавлением в метод следующих аннотаций.

- `jakarta.annotation.PostConstruct` : методы, аннотированные `@PostConstruct` , вызываются контейнером на новых объектах компонента после завершения инъецирования всех зависимостей и до того, как первый бизнес-метод Enterprise-бина будет вызван.
- `jakarta.annotation.PreDestroy` : методы, аннотированные `@PreDestroy` , вызываются после завершения любого аннотированного `@Remove` метода и до того, как контейнер удалит объект Enterprise-бина.
- `jakarta.ejb.PostActivate` : методы, аннотированные `@PostActivate` вызываются контейнером после того, как контейнер перемещает компонент из вторичного хранилища в активный статус.
- `jakarta.ejb.PrePassivate` : методы, аннотированные `@PrePassivate` , вызываются контейнером до того, как он пассивирует Enterprise-бин, то есть контейнер временно удаляет компонент из среды и сохраняет его во вторичное хранилище.

Callback-методы жизненного цикла должны возвращать `void` и не иметь параметров.

Бизнес-методы

Основная цель сессионного компонента — запуск бизнес-задач для клиента. Клиент вызывает бизнес-методы для ссылки на объект, который он получает при инъецирования зависимости или поиске JNDI. С точки зрения клиента, бизнес-методы работают локально, хотя они выполняются удалённо в сессионном компоненте. В следующем фрагменте кода показано, как программа `CartClient` вызывает бизнес-методы:

JAVA
```
cart.initialize("Duke DeEarl", "123");
...
cart.addBook("Bel Canto");
 ...
List<String> bookList = cart.getContents();
...
cart.removeBook("Gravity's Rainbow");
```

Класс `CartBean` реализует бизнес-методы в следующем коде:

JAVA
```
@Override
public void addBook(String title) {
   contents.add(title);
}

@Override
public void removeBook(String title) throws BookException {
   boolean result = contents.remove(title);
   if (result == false) {
      throw new BookException("\"" + title + "not in cart.");
   }
}

@Override
public List<String> getContents() {
   return contents;
}
```

Сигнатура бизнес-метода должна соответствовать следующим правилам.

- Имя метода не должно начинаться с `ejb`, чтобы избежать конфликтов с Callback-методами, определёнными архитектурой Jakarta Enterprise Beans. Например, вы не можете вызвать бизнес-метод `ejbCreate` или `ejbActivate`.
- Модификатор доступа должен быть `public`.
- Если бин разрешает удалённый доступ через удалённый бизнес-интерфейс, аргументы и возвращаемые типы должны быть допустимыми типами для API удалённого вызова методов Java (RMI).
- Если объект EJB является конечной точкой Jakarta XML Web Services, аргументы и типы возврата для методов с аннотациями `@WebMethod` должны иметь допустимые типы для Jakarta XML Web Services.
- Если объект EJB является ресурсом RESTful веб-сервиса Jakarta, то аргументы и типы возвращаемых данных для методов ресурсов должны иметь допустимые типы для RESTful веб-сервиса Jakarta.
- Модификатор не должен быть `static` или `final`.

Предложение `throws` может содержать исключения, которые вы определяете для своего приложения. Например, метод `removeBook` выдает `BookException`, если книга отсутствует в корзине.

Чтобы указать на проблему системного уровня, такую как невозможность подключения к базе данных, бизнес-метод должен выбросить `jakarta.ejb.EJBException`. Контейнер не будет обёртывать (wrap) исключения приложения, такие как `BookException`. Поскольку `EJBException` является дочерним классом `RuntimeException`, не нужно добавлять его в `throws` бизнес-метода.

Метод @Remove

Бизнес-методы, аннотированные `jakarta.ejb.Remove` в классе сессионного компонента с сохранением состояния, могут быть вызваны клиентами Enterprise-бина для удаления объекта компонента. Контейнер удалит Enterprise-бин после завершения метода `@Remove` независимо от того, завершился он нормально или с ошибкой.

В `CartBean` метод `remove` помечен аннотацией `@Remove`:

JAVA

```
@Remove
@Override
public void remove() {
    contents = null;
}
```

Вспомогательные классы

Сессионный компонент `CartBean` имеет два вспомогательных класса: `BookException` и `IdVerifier`. `BookException` выбрасывается методом `removeBook`, а `IdVerifier` валидирует `customerId` в одном из методов `create`. Вспомогательные классы могут находиться в файле EJB JAR, который содержит класс Enterprise-бина. Или в WAR-файле, если Enterprise-бин упакован в WAR. Или EAR-файле, содержащем файл EJB JAR, WAR-файл или отдельный JAR-файл библиотеки. В `cart` вспомогательные классы включены в библиотечный JAR, используемый клиентским приложением, и EJB JAR.

Запуск cart

Теперь вы готовы скомпилировать удалённый интерфейс (`Cart.java`), класс Enterprise-бина (`CartBean.java`), клиентский класс (`CartClient.java`) и вспомогательные классы (`BookException.java` и `IdVerifier.java`).

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска приложения `cart`.

Запуск cart с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/ejb
   ```

4. Выберите каталог `cart`.
5. Выберите чекбокс **Открыть требуемые проекты**.
6. Нажмите Открыть проект.
7. На вкладке **Проекты** кликните правой кнопкой мыши проект `cart` и выберите **Сборка**.

   Команда собирает и упаковывает приложение в `cart.ear`, расположенный в *`tut-install`*`/examples/ejb/cart/cart-ear/target/`, и развёртывает этот файл EAR на вашем экземпляре GlassFish Server.

   Вы увидите выходные данные приложения-клиента `cart-app-client` на вкладке Вывод:

   ```
   ...
   Retrieving book title from cart: Infinite Jest
   Retrieving book title from cart: Bel Canto
   Retrieving book title from cart: Kafka on the Shore
   Removing "Gravity's Rainbow" from cart.
   Caught a BookException: "Gravity's Rainbow" not in cart.
   ```

Запуск cart с помощью Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/ejb/cart/
   ```

3. Введите следующую команду:

   ```
   mvn install
   ```
   SHELL

   Эта команда компилирует и упаковывает приложение в файл EAR, `cart.ear`, расположенный в каталоге `target`, и развёртывает EAR в GlassFish Server.

   Затем клиентские заглушки извлекаются и запускаются. Это эквивалентно выполнению следующей команды:

   ```
   appclient -client cart-ear/target/cart-earClient.jar
   ```
   SHELL

   Клиентский JAR `cart-earClient.jar` содержит клиентский класс приложения, вспомогательный класс `BookException` и бизнес-интерфейс `Cart`.

   При запуске клиента контейнер клиентского приложения инъецирует все ссылки на компоненты, объявленные в классе клиентского приложения, в данном случае ссылку на Enterprise-бин `Cart`.

В окне терминала вы увидите вывод `cart-app-client`:

```
...
Retrieving book title from cart: Infinite Jest
Retrieving book title from cart: Bel Canto
Retrieving book title from cart: Kafka on the Shore
Removing "Gravity's Rainbow" from cart.
Caught a BookException: "Gravity's Rainbow" not in cart.
```

Сессионный компонент-синглтон counter

Пример `counter` демонстрирует, как создать сессионный компонент-синглтон.

Создание сессионного компонента-синглтона

Аннотация `jakarta.ejb.Singleton` указывает, что класс Enterprise-бина является компонентом-синглтоном:

JAVA

```java
@Singleton
public class SingletonBean { ... }
```

Инициализация сессионных компонентов-синглтонов

EJB-контейнер отвечает за определение момента инициализации объекта сессионного компонента-синглтона, если класс реализации этого компонента не аннотирован `jakarta.ejb.Startup`. В этом случае, иногда называемом ранней инициализацией, EJB-контейнер должен инициализировать сессионный компонент-синглтон при запуске приложения. Сессионный компонент-синглтон инициализируется до того, как EJB-контейнер доставляет клиентские запросы любым Enterprise-бинам приложения. Это позволяет сессионному компоненту-синглтону выполнять, например, задачи запуска приложения.

Следующий сессионный компонент-синглтон хранит состояние приложения и будет инициализирован рано:

JAVA

```java
@Startup
@Singleton
public class StatusBean {
  private String status;

  @PostConstruct
  void init {
    status = "Ready";
  }
  ...
}
```

Иногда для инициализации данных для приложения используются несколько сессионных EJB-компонентов, поэтому их следует инициализировать в определённом порядке. В этих случаях используйте аннотацию `jakarta.ejb.DependsOn` для объявления зависимостей запуска сессионного компонента-синглтона. Атрибут `value` аннотации `@DependsOn` представляет собой одну или несколько строк, которые задают имя целевого сессионного компонента-синглтона. Если более одного зависимого компонента-синглтона указано в `@DependsOn`, порядок их перечисления не обязательно является порядком, в котором EJB-контейнер будет инициализировать целевые сессионные компоненты-синглтоны.

Так, сессионный компонент-синглтон `PrimaryBean` должен быть запущен первым:

JAVA

```java
@Singleton
public class PrimaryBean { ... }
```

`SecondaryBean` зависит от `PrimaryBean`:

JAVA

```
@Singleton
@DependsOn("PrimaryBean")
public class SecondaryBean { ... }
```

Это гарантирует, что EJB-контейнер инициализирует `PrimaryBean` раньше `SecondaryBean`.

Следующий сессионный компонент-синглтон `TertiaryBean` зависит от `PrimaryBean` и `SecondaryBean`:

JAVA

```
@Singleton
@DependsOn({"PrimaryBean", "SecondaryBean"})
public class TertiaryBean { ... }
```

`SecondaryBean` с помощью аннотации `@DependsOn` требует, чтобы его инициализация выполнялась после инициализации `PrimaryBean`. В этом случае EJB-контейнер сначала инициализирует `PrimaryBean`, затем `SecondaryBean` и, наконец, `TertiaryBean`.

Если, однако, `SecondaryBean` явно не зависит от `PrimaryBean`, EJB-контейнер может сначала инициализировать либо `PrimaryBean`, либо `SecondaryBean`. Таким образом, EJB-контейнер может инициализировать синглтоны в следующем порядке: `SecondaryBean`, `PrimaryBean`, `TertiaryBean`.

Управление параллельным доступом в сессионном компоненте-синглтоне

Дизайн сессионных компонентов-синглтонов позволяет одновременный доступ к нему множества клиентов. Клиенту синглтона требуется только ссылка на синглтон для вызова любых бизнес-методов, предоставляемых синглтоном, и ему не нужно беспокоиться о любых других клиентах, которые могут одновременно вызывать бизнес-методы в одном и том же синглтоне.

При создании сессионного компонента-синглтона одновременный доступ к бизнес-методам синглтона можно контролировать двумя способами: параллелизм, управляемый контейнером, и параллелизм, упраляемый Managed-бином.

Аннотация `jakarta.ejb.ConcurrencyManagement` используется для указания управляемого контейнером или управляемого компонентом параллелизма для синглтона. Для `@ConcurrencyManagement` атрибут type должен иметь значение `jakarta.ejb.ConcurrencyManagementType.CONTAINER` или `jakarta.ejb.ConcurrencyManagementType.BEAN`. Если в классе реализации синглтона нет аннотации `@ConcurrencyManagement`, используется EJB-контейнер по умолчанию для параллелизма, управляемого контейнером.

Управляемый контейнером параллелизм

Если синглтон использует управляемый контейнером параллелизм, EJB-контейнер контролирует доступ клиентов к бизнес-методам синглтона. Аннотация `jakarta.ejb.Lock` и тип `jakarta.ejb.LockType` используются для указания уровня доступа бизнес-методов синглтона или методов `@Timeout`. Значениями `LockType` являются `READ` и `WRITE`.

Аннотируйте бизнес-метод или метод таймера с помощью `@Lock(LockType.READ)`, если к методу можно одновременно обращаться или совместно использовать его несколькими клиентами. Аннотируйте бизнес-метод или метод таймера с помощью `@Lock(LockType.WRITE)`, если при выполнении сессионного компонента-синглтона одним из клиентов он должен блокироваться для остальных клиентов. Как правило, аннотация `@Lock(LockType.WRITE)` используется, когда клиенты изменяют состояние синглтона.

Аннотирование класса синглтона с помощью `@Lock` указывает, что все бизнес-методы и любые методы тайм-аута синглтона будут использовать указанный тип блокировки, если они явно не установят тип блокировки с помощью аннотации `@Lock` уровня метода. Если в классе синглтона нет аннотации `@Lock`, тип блокировки по умолчанию `@Lock(LockType.WRITE)` применяется ко всем бизнес-методам и методам тайм-аута.

В следующем примере показано, как использовать аннотации `@ConcurrencyManagement`, `@Lock(LockType.READ)` и `@Lock(LockType.WRITE)` для синглтона, который использует управляемый контейнером параллелизм.

Хотя по умолчанию в синглтонах используется управляемый контейнером параллелизм, аннотация `@ConcurrencyManagement(CONTAINER)` может быть добавлена на уровне класса синглтона, чтобы явно установить тип управления параллелизмом:

JAVA

```java
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
@Singleton
public class ExampleSingletonBean {
  private String state;

  @Lock(LockType.READ)
  public String getState() {
    return state;
  }

  @Lock(LockType.WRITE)
  public void setState(String newState) {
    state = newState;
  }
}
```

Метод `getState` может быть доступен многим клиентам одновременно, потому что он аннотирован с помощью `@Lock(LockType.READ)`. Однако когда вызывается метод `setState`, все методы в `ExampleSingletonBean` будут заблокированы для других клиентов, поскольку `setState` помечен `@Lock(LockType.WRITE)`. Это не позволяет двум клиентам одновременно пытаться изменить переменную `state` в `ExampleSingletonBean`.

Методы `getData` и `getStatus` в следующем синглтоне имеют тип `READ`, а метод `setStatus` имеет тип `WRITE`:

JAVA

```java
@Singleton
@Lock(LockType.READ)
public class SharedSingletonBean {
  private String data;
  private String status;

  public String getData() {
    return data;
  }

  public String getStatus() {
    return status;
  }

  @Lock(LockType.WRITE)
  public void setStatus(String newStatus) {
    status = newStatus;
  }
}
```

Если метод имеет тип блокировки `WRITE`, клиентский доступ ко всем методам синглтона блокируется до тех пор, пока текущий клиент не завершит свой вызов метода или не истечёт время ожидания доступа. Когда происходит тайм-аут, EJB-контейнер выбрасывает `jakarta.ejb.ConcurrentAccessTimeoutException`. Аннотация `jakarta.ejb.AccessTimeout` используется для указания количества миллисекунд до истечения тайм-аута. Добавление `@AccessTimeout` на уровне класса синглтона указывает значение времени ожидания доступа для всех методов в синглтоне, если только метод явно не переопределяет значение по умолчанию своей собственной аннотацией `@AccessTimeout`.

Аннотация `@AccessTimeout` может применяться к методам `@Lock(LockType.READ)` и `@Lock(LockType.WRITE)`. Аннотация `@AccessTimeout` имеет один обязательный элемент `value` и один необязательный элемент `unit`. По умолчанию `value` указывается в миллисекундах. Чтобы изменить единицу `value`, установите для `unit` одну из констант `java.util.concurrent.TimeUnit`: `NANOSECONDS`, `MICROSECONDS`, `MILLISECONDS` или `SECONDS`.

В следующем синглтоне значение тайм-аута доступа по умолчанию составляет 120 000 миллисекунд или 2 минуты. Метод `doTediousOperation` переопределяет время ожидания доступа по умолчанию и устанавливает значение 360 000 миллисекунд или 6 минут:

JAVA

```java
@Singleton
@AccessTimeout(value=120000)
public class StatusSingletonBean {
  private String status;

  @Lock(LockType.WRITE)
  public void setStatus(String new Status) {
    status = newStatus;
  }

  @Lock(LockType.WRITE)
  @AccessTimeout(value=360000)
  public void doTediousOperation {
    ...
  }
}
```

Следующий синглтон имеет значение тайм-аута доступа по умолчанию 60 секунд, указанное с помощью константы `TimeUnit.SECONDS`:

JAVA

```java
@Singleton
@AccessTimeout(value=60, unit=TimeUnit.SECONDS)
public class StatusSingletonBean { ... }
```

Параллелизм, управляемый Managed-бином

Синглтоны, использующие управляемый Managed-бином параллелизм, обеспечивают полный параллельный доступ ко всем бизнес-методам и методам тайм-аута в синглтоне. Разработчик синглтона отвечает за синхронизацию состояния синглтона между всеми клиентами. Разработчикам, которые создают синглтоны с управляемым бином параллелизмом, разрешается использовать примитивы синхронизации Java, такие как `synchronization` и `volatile`, для предотвращения ошибок при одновременном доступе.

Добавьте аннотацию `@ConcurrencyManagement` с типом, установленным в `ConcurrencyManagementType.BEAN` на уровне класса синглтона, чтобы указать параллелизм, управляемый Managed-бином:

JAVA

```java
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
@Singleton
public class AnotherSingletonBean { ... }
```

Обработка ошибок в сессионном компоненте-синглтоне

Если сессионный компонент-синглтон обнаруживает ошибку при инициализации контейнером EJB-компонента, объект компонента будет уничтожен.

В отличие от других Enterprise-бинов, после инициализации объекта сессионного компонента-синглтона он не уничтожается, если методы бизнес-процесса или жизненного цикла синглтона выбрасывают системные исключения. Это гарантирует, что один и тот же объект синглтона используется на протяжении всего жизненного цикла приложения.

Пример архитектуры counter

Пример `counter` состоит из сессионного компонента-синглтона `CounterBean` и веб-интерфейса Jakarta Faces Facelets.

`CounterBean` — это простой синглтон с одним методом `getHits`, который возвращает целое число, представляющее количество обращений к веб-странице. Вот код `CounterBean`:

JAVA

```java
package ee.jakarta.tutorial.counter.ejb;

import jakarta.ejb.Singleton;

/**
 * CounterBean - простой сессионный компонент-синглтон, выводящий кол-во
 * кликов на страницу.
 */
@Singleton
public class CounterBean {
    private int hits = 1;

    // Увеличить и вернуть новое значение кликов
    public int getHits() {
        return hits++;
    }
}
```

Аннотация `@Singleton` помечает `CounterBean` как сессионный компонент-синглтон. `CounterBean` использует локальное представление без интерфейса.

`CounterBean` использует значения метаданных EJB-контейнера по умолчанию для синглтонов, чтобы упростить их реализацию. В классе нет аннотации `@ConcurrencyManagement`, поэтому по умолчанию применяется управляемый контейнером параллельный доступ. В классе или бизнес-методе нет аннотации `@Lock`, поэтому по умолчанию `@Lock(WRITE)` применяется к единственному бизнес-методу `getHits`.

Следующая версия `CounterBean` функционально эквивалентна предыдущей версии:

JAVA

```java
package ee.jakarta.tutorial.counter.ejb;

import jakarta.ejb.Singleton;
import jakarta.ejb.ConcurrencyManagement;
import static jakarta.ejb.ConcurrencyManagementType.CONTAINER;
import jakarta.ejb.Lock;
import jakarta.ejb.LockType.WRITE;

/**
 * CounterBean - простой сессионный компонент-синглтон, выводящй кол-во
 * кликов на страницу.
 */
@Singleton
@ConcurrencyManagement(CONTAINER)
public class CounterBean {
    private int hits = 1;

    // Увеличить и вернуть новое значение
    @Lock(WRITE)
    public int getHits() {
        return hits++;
    }
}
```

Веб-интерфейс `counter` состоит из Managed-бина Jakarta Faces `Count.java`, который используется XHTML-файлами Facelets `template.xhtml` и `index.xhtml`. Managed-бин Jakarta Faces `Count` получает ссылку на `CounterBean` посредством инъецирования зависимостей. `Count` определяет свойство `hitCount` JavaBeans. Когда get-метод `getHitCount` вызывается из файлов XHTML, вызывается метод `CounterBean.getHits` для возврата текущего числа обращений к странице.

Вот класс Managed-бина `Count`:

JAVA

```java
@Named
@ConversationScoped
public class Count implements Serializable {
    @EJB
    private CounterBean counterBean;

    private int hitCount;

    public Count() {
        this.hitCount = 0;
    }

    public int getHitCount() {
        hitCount = counterBean.getHits();
        return hitCount;
    }

    public void setHitCount(int newHits) {
        this.hitCount = newHits;
    }
}
```

Файлы `template.xhtml` и `index.xhtml` используются для визуализации представления Facelets, которое отображает количество совпадений для этого представления. В файле `index.xhtml` используется оператор языка выражений `#{count.hitCount}` для доступа к свойству `hitCount` Managed-бина `Count`. Вот содержимое `index.xhtml`:

XML

```xml
<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <ui:composition template="/template.xhtml">
        <ui:define name="title">
            This page has been accessed #{count.hitCount} time(s).
        </ui:define>
        <ui:define name="body">
            Hooray!
        </ui:define>
    </ui:composition>
</html>
```

Запуск counter

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска примера `counter`.

Запуск counter в IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/ejb
   ```

4. Выберите каталог `counter`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `counter` и выберите **Запуск**.

   Веб-браузер откроет URL-адрес http://localhost:8080/counter, который отображает количество совпадений.
7. Перезагрузите страницу, чтобы увидеть увеличение количества просмотров.

Запуск counter с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/ejb/counter/
   ```

3. Введите следующую команду:

   SHELL

   ```
   mvn install
   ```

   Эта команда собирает и развёртывает `counter` в GlassFish Server.
4. В веб-браузере введите следующий URL:

   ```
   http://localhost:8080/counter
   ```

5. Перезагрузите страницу, чтобы увидеть увеличение количества просмотров.

Пример веб-сервиса: helloservice

В этом примере демонстрируется простой веб-сервис, который генерирует ответ на основе информации, полученной от клиента. `HelloServiceBean` — это сессионный компонент без сохранения состояния, который реализует единственный метод `sayHello`. Этот метод соответствует методу `sayHello`, который вызывается клиентом, описанным в Простой клиент XML веб-сервиса.

Класс реализации конечной точки веб-сервиса

`HelloServiceBean` — это класс реализации конечной точки, обычно основной программный артефакт для конечных точек веб-сервиса Enterprise-бина. Класс реализации конечной точки веб-сервиса имеет следующие требования.

- Класс должен быть аннотирован либо `jakarta.jws.WebService`, либо `jakarta.jws.WebServiceProvider`.
- Реализующий класс может явно ссылаться на SEI через элемент `endpointInterface` аннотации `@WebService`, но не обязательно. Если `endpointInterface` не указан в `@WebService`, SEI неявно определяется для реализующего класса.
- Бизнес-методы реализующего класса должны быть публичными и не должны объявляться `static` или `final`.
- Бизнес-методы, которые доступны для клиентов веб-сервисов, должны быть аннотированы `jakarta.jws.WebMethod`.
- Бизнес-методы, предоставляемые клиентам веб-сервиса, должны иметь параметры и возвращаемые типы, совместимые с Jakarta XML Binding. См. список типов данных Jakarta XML Binding по умолчанию в разделе Типы, поддерживаемые XML веб-сервисами.
- Реализующий класс не должен быть объявлен `final` и не должен быть `abstract`.
- Реализующий класс должен иметь публичный конструктор по умолчанию.
- Класс конечной точки должен быть аннотирован `@Stateless`.
- Реализующий класс не должен определять метод `finalize`.
- Реализующий класс может использовать аннотации `jakarta.annotation.PostConstruct` или `jakarta.annotation.PreDestroy` для Callback-методов событий жизненного цикла.

  Метод `@PostConstruct` вызывается контейнером до того, как реализующий класс начинает отвечать клиентам веб-сервиса.

  Метод `@PreDestroy` вызывается контейнером перед удалением конечной точки.

Класс реализации сессионного компонента без сохранения состояния

Класс `HelloServiceBean` реализует метод `sayHello`, который аннотируется `@WebMethod`. Исходный код для класса `HelloServiceBean` выглядит следующим образом:

JAVA

```java
package ee.jakarta.tutorial.helloservice.ejb;

import jakarta.ejb.Stateless;
import jakarta.jws.WebMethod;
import jakarta.jws.WebService;

@Stateless
@WebService
public class HelloServiceBean {
    private final String message = "Hello, ";

    public void HelloServiceBean() {}

    @WebMethod
    public String sayHello(String name) {
        return message + name + ".";
    }
}
```

Запуск helloservice

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки и развёртывания примера `helloservice`. Затем вы можете использовать Консоль администрирования для тестирования методов конечной точки веб-сервиса.

Сборка, упаковка и развёртывание helloservice с использованием IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/ejb
   ```

4. Выберите каталог `helloservice`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `helloservice` и выберите **Сборка**.

   Это создаёт и упаковывает приложение в `helloservice.ear`, расположенный в *`tut-install`*`/examples/ejb/helloservice/target/`, и развёртывает этот EAR-файл в GlassFish Server.

Сборка, упаковка и развёртывание helloservice с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/ejb/helloservice/
   ```

3. Введите следующую команду:

   SHELL

   ```
   mvn install
   ```

   Команда компилирует исходные файлы и упаковывает приложение в JAR-файл EJB, расположенный по каталоге *`tut-install`*`/examples/ejb/helloservice/target/helloservice.jar`. Затем JAR-файл EJB развёртывается в GlassFish Server.

После развёртывания GlassFish Server создаёт дополнительные артефакты, необходимые для вызова веб-сервиса, включая файл WSDL.

Тестирование сервиса без клиента

Консоль администрирования GlassFish Server позволяет тестировать методы конечной точки веб-сервиса. Чтобы проверить метод `sayHello` у `HelloServiceBean`, выполните следующие действия.

1. Откройте **Консоль администрирования** по следующему URL в браузере:

   ```
   http://localhost:4848/
   ```

2. В дереве навигации выберите узел **Приложения**.
3. В таблице **Приложения** кликните ссылку `helloservice`.
4. В таблице **Модули и компоненты** кликните ссылку **Просмотр конечной точки**.
5. На странице **Информация о конечной точке веб-сервиса** кликните ссылку **Tester**:

   ```
   /HelloServiceBeanService/HelloServiceBean?Tester
   ```

6. На вкладке **Тестовые ссылки веб-сервиса** кликните незащищённую ссылку (указывающую порт 8080).
7. На вкладке **Тестер веб-сервиса HelloServiceBeanService**, в разделе **Методы** введите имя в качестве параметра метода `sayHello`.
8. Кликните **sayHello**.

   Откроется страница вызова метода sayHello. В разделе Method returned вы увидите ответ от конечной точки.

Использование сервиса таймера

Приложения, моделирующие бизнес-процессы, часто полагаются на синхронизированные уведомления. Сервис таймера контейнера EJB позволяет планировать уведомления по времени для всех типов EJB, за исключением компонентов EJB с сохранением состояния. Вы можете запланировать своевременное уведомление в соответствии с расписанием календаря, в определённое время, по истечении определённого времени или через определённые интервалы времени. Например, вы можете настроить таймеры на срабатывание в 10:30 утра 23 мая, через 30 дней или каждые 12 часов.

EJB-таймеры являются либо программными, либо автоматическими. Программные таймеры устанавливаются путём явного вызова одного из методов создания таймера интерфейса `TimerService`. Автоматические таймеры создаются после успешного развёртывания Enterprise-бина, содержащего метод, аннотированный `jakarta.ejb.Schedule` или `jakarta.ejb.Schedules`.

Создание календарных выражений таймера

Таймеры могут быть установлены в соответствии с календарным расписанием, выраженным с использованием синтаксиса, аналогичного утилите UNIX `cron`. Как программные, так и автоматические таймеры могут использовать календарные выражения таймера. Таблица 37-1 показывает атрибуты календарных выражений таймера.

Таблица 37-1 Атрибуты таймера на основе календаря

Атрибут	Описание	Значение по умолчанию	Допустимые значения и примеры

Атрибут	Описание	Значение по умолчанию	Допустимые значения и примеры
`second`	Одна или несколько секунд в течение минуты	`0`	`0` to `59`. Например: `second="30"`.
`minute`	Одна или несколько минут в течение часа	`0`	`0` to `59`. Например: `minute="15"`.
`hour`	Один или несколько часов в течение дня	`0`	`0` to `23`. Например: `hour="13"`.
`dayOfWeek`	Один или несколько дней в течение недели	`*`	`0` до `7` (0 и 7 относятся к воскресенью). Например: `dayOfWeek="3"`. `Sun`, `Mon`, `Tue`, `Wed`, `Thu`, `Fri`, `Sat`. Например: `dayOfWeek="Mon"`.
`dayOfMonth`	Один или несколько дней в течение месяца	`*`	`1` to `31`. Например: `dayOfMonth="15"`. `-7` до `-1` (отрицательное число означает n-й день или дни до конца месяца). Например: `dayOfMonth="-3"`. `Last`. Например: `dayOfMonth="Last"`. [`1st`, `2nd`, `3rd`, `4th`, `5th`, `Last`] [`Sun`, `Mon`, `Tue`, `Wed`, `Thu`, `Fri`, `Sat`]. Например: `dayOfMonth="2nd Fri"`.
`month`	Один или несколько месяцев в течение года	`*`	`1` to `12`. Например: `month="7"`. `Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, `Dec`. Например: `month="July"`.
`year`	Конкретный календарный год	`*`	Четырёхзначный календарный год. Например: `year="2011"`.

Указание нескольких значений в календарных выражениях

Вы можете указать несколько значений в календарных выражениях, как описано в следующих разделах.

Использование подстановочных знаков в календарных выражениях

Установка атрибута в символ звездочки (`*`) представляет все допустимые значения для атрибута.

Следующее выражение представляет каждую минуту:

```java
minute="*"
```

Следующее выражение представляет каждый день недели:

```java
dayOfWeek="*"
```

Указание списка значений

Чтобы указать два значения и более для атрибута, используйте запятую (`,`) для разделения значений. Диапазон значений допускается как часть списка. Подстановочные знаки и интервалы, однако, не допускаются.

Дубликаты в списке игнорируются.

Следующее выражение устанавливает день недели во вторник и четверг:

```java
dayOfWeek="Tue, Thu"
```

Следующее выражение представляет 4:00 утра, каждый час с 9:00 утра до 5:00 вечера. используя диапазон, и 10:00 вечера:

```java
hour="4,9-17,22"
```

Указание диапазона значений

Используйте тире (`-`), чтобы указать диапазон значений для атрибута. Члены диапазона не могут быть подстановочными знаками, списками или интервалами. Диапазон в форме `x-x` эквивалентен однозначному выражению `x` . Диапазон вида `x-y` , где `x` больше `y` , равноценно выражению `x-`*`maximumvalue`* , *`minimumvalue`*`-y` . То есть выражение начинается с `x` , возвращается к началу допустимых значений и продолжается до `y` .

Следующее выражение представляет с 9:00 до 17:00:

```java
hour="9-17"
```

Следующее выражение представляет период с пятницы до понедельника:

```java
dayOfWeek="5-1"
```

Следующее выражение представляет период с двадцать пятого дня месяца до конца месяца и с начала месяца до пятого дня месяца:

```java
dayOfMonth="25-5"
```

Это эквивалентно следующему выражению:

```java
dayOfMonth="25-Last,1-5"
```

Указание интервалов

Косая черта (`/`) ограничивает атрибут начальной точкой и интервалом и используется для указания каждых `N` секунд, минут или часов в течение минуты, часа или дня. Для выражения вида `x/y` `x` представляет начальную точку, а `y` представляет интервал. Подстановочный знак может использоваться в позиции `x` интервала и эквивалентен установке `x` в `0`.

Интервалы могут быть установлены только для атрибутов `second`, `minute` и `hour`.

Следующее выражение представляет каждые 10 минут в течение часа:

```java
minute="*/10"
```

Это эквивалентно:

```java
minute="0,10,20,30,40,50"
```

Следующее выражение представляет каждые 2 часа, начиная с полудня:

```java
hour="12/2"
```

Программные таймеры

Когда срабатывает программный таймер, контейнер вызывает метод, аннотированный `@Timeout` в классе реализации компонента. Метод `@Timeout` содержит бизнес-логику, которая обрабатывает событие срабатывания.

Метод @Timeout

Методы, аннотированные `@Timeout` в классе Enterprise-бина, должны возвращать `void` и, при необходимости, принимать `jakarta.ejb.Timer` в качестве единственного параметра. Они не могут генерировать исключения приложений:

```java
@Timeout
public void timeout(Timer timer) {
    System.out.println("TimerBean: timeout occurred");
}
```

Создание программных таймеров

Чтобы создать таймер, компонент вызывает один из методов `create` интерфейса `TimerService`. Эти методы позволяют создавать таймеры однократного действия, интервалов или календаря.

Для таймеров однократного действия или с интервалом срабатывание таймера может быть выражено либо как длительность, либо как абсолютное время. Длительность выражается в виде количества миллисекунд до срабатывания события тайм-аута. Чтобы указать абсолютное время, создайте объект `java.util.Date` и передайте его методу `TimerService.createSingleActionTimer` или `TimerService.createTimer`.

Следующий код устанавливает программный таймер, который срабатывает через 1 минуту (60000 миллисекунд):

```java
long duration = 60000;
Timer timer =
    timerService.createSingleActionTimer(duration, new TimerConfig());
```

Следующий код устанавливает программный таймер, который срабатывает в 12:05. 1 мая 2015 г., указанное как `java.util.Date`:

```java
SimpleDateFormatter formatter =
    new SimpleDateFormatter("MM/dd/yyyy 'at' HH:mm");
Date date = formatter.parse("05/01/2015 at 12:05");
Timer timer = timerService.createSingleActionTimer(date, new TimerConfig());
```

Для таймеров на основе календаря срабатывание таймера выражается в виде объекта `jakarta.ejb.ScheduleExpression`, переданного в качестве параметра методу `TimerService.createCalendarTimer`. Класс `ScheduleExpression` представляет календарные выражения таймера и имеет методы, соответствующие атрибутам, описанным в Создание календарных выражений таймера.

Следующий код создаёт программный таймер с использованием вспомогательного класса `ScheduleExpression`:

```java
ScheduleExpression schedule = new ScheduleExpression();
schedule.dayOfWeek("Mon");
schedule.hour("12-17, 23");
Timer timer = timerService.createCalendarTimer(schedule);
```

Дополнительные сведения о сигнатурах методов см. в документации по API `TimerService` по ссылке https://jakarta.ee/specifications/platform/9/apidocs/jakarta/ejb/TimerService.html.

Компонент, описанный в Примере timersession, создаёт таймер следующим образом:

```java
Timer timer = timerService.createTimer(intervalDuration,
        "Created new programmatic timer");
```

В примере `timersession` метод, вызывающий `createTimer`, вызывается в бизнес-методе, который вызывается клиентом.

Таймеры являются персистентным по умолчанию. Если сервер выключен или выходит из строя, персистентные таймеры сохраняются и снова становятся активными при перезапуске сервера. Если время срабатывания персистентного таймера наступает, когда сервер не работает, контейнер будет вызывать метод `@Timeout` при перезапуске сервера.

Неперсистентные программные таймеры создаются путём вызова `TimerConfig.setPersistent(false)` и передачи объекта `TimerConfig` одному из методов создания таймера.

Параметры `Date` и `long` методов `createTimer` представляют время с разрешением в миллисекундах. Однако, поскольку сервис таймера не предназначен для приложений реального времени, вызов Callback-метод `@Timeout` может не выполняться с точностью до миллисекунды. Сервис таймера предназначен для бизнес-приложений, которые обычно измеряют время в часах, днях или дольше.

Автоматические таймеры

EJB-контейнер создаёт автоматические таймеры, когда развёртывается EJB-компонент, содержащий методы, аннотированные `@Schedule` или `@Schedules`. В Enterprise-бине может быть несколько методов автоматического тайм-аута, в отличие от программного таймера, который позволяет использовать только один метод с аннотацией `@Timeout` в классе Enterprise-бина.

Автоматические таймеры можно настроить аннотациями или дескриптором развёртывания `ejb-jar.xml`.

Добавление аннотации `@Schedule` к Enterprise-бину помечает этот метод как метод таймера, срабатывающий по календарному расписанию, указанному в атрибутах `@Schedule`.

Аннотация `@Schedule` содержит элементы, которые соответствуют выражениям календаря, подробно изложенным в Создание календарных выражений таймера и `persistent`, `info` и `timezone`.

Необязательный элемент `persistent` принимает логическое значение и используется для указания того, должен ли автоматический таймер выдерживать перезапуск сервера или сбой. По умолчанию все автоматические таймеры являются персистентными.

Необязательный элемент `timezone` используется для указания того, что автоматический таймер связан с конкретным часовым поясом. Если установлено, этот элемент будет вычислять все выражения таймера по отношению к указанному часовому поясу, независимо от часового пояса, в котором выполняется EJB-контейнер. По умолчанию все установленные автоматические таймеры относятся к часовому поясу по умолчанию на сервере.

Необязательный элемент `info` используется для установки информационного описания таймера. Информация о таймере может быть получена позже с помощью `Timer.getInfo`.

Следующий метод таймера использует `@Schedule` для установки расписания срабатывания на полночь каждого воскресенья:

```java
@Schedule(dayOfWeek="Sun", hour="0")
public void cleanupWeekData() { ... }
```

Аннотация `@Schedules` используется для установки методу таймера нескольких календарных выражений.

Следующий метод таймера использует аннотацию `@Schedules` для установки нескольких календарных выражений. Первое выражение устанавливает таймер, который срабатывает в последний день каждого месяца. Второе выражение устанавливает таймер, который срабатывает каждую пятницу в 11:00 вечера.

```java
@Schedules ({
    @Schedule(dayOfMonth="Last"),
    @Schedule(dayOfWeek="Fri", hour="23")
})
public void doPeriodicCleanup() { ... }
```

Отмена и сохранение таймеров

Таймеры могут быть отменены следующими событиями.

- Когда одноразовый таймер срабатывает, EJB-контейнер вызывает связанный метод таймера и отменяет таймер.
- Когда компонент вызывает метод `cancel` интерфейса `Timer`, контейнер отменяет таймер.

Если метод вызывается на отменённом таймере, контейнер выбрасывает `jakarta.ejb.NoSuchObjectLocalException`.

Чтобы сохранить объект `Timer` для использования в будущем, вызовите его метод `getHandle` и сохраните объект `TimerHandle` в базе данных. (Объект `TimerHandle` является сериализуемым.) Чтобы восстановить объект `Timer`, извлеките дескриптор из базы данных и вызовите `getTimer` для дескриптора. Объект

`TimerHandle` не может быть аргументом при вызове метода удалённого интерфейса или веб-сервиса. Другими словами, удалённые клиенты и клиенты веб-сервисов не могут получить доступ к объекту `TimerHandle` компонента. Для локальных клиентов такого ограничения нет.

Получение информации о таймере

В дополнение к определению методов `cancel` и `getHandle` интерфейс `Timer` определяет методы для получения информации о таймерах:

JAVA

```
public long getTimeRemaining();
public java.util.Date getNextTimeout();
public java.io.Serializable getInfo();
```

Метод `getInfo` возвращает объект, который был последним параметром вызова `createTimer` . Например, во фрагменте кода `createTimer` предыдущего раздела этот информационный параметр является объектом `String` со значением `created timer` .

Чтобы получить все активные таймеры компонента, вызовите метод `getTimers` интерфейса `TimerService` . Метод `getTimers` возвращает коллекцию объектов `Timer` .

Транзакции и таймеры

Enterprise-бин обычно создаёт таймер в транзакции. Если эта транзакция откатывается, создание таймера также откатывается. Аналогично, если бин отменяет таймер в транзакции и транзакция откатывается, отмена таймера также откатывается. В этом случае длительность таймера сбрасывается, как если бы отмена никогда не происходила.

В бинах, использующих транзакции, управляемые контейнером, метод `@Timeout` обычно имеет атрибут транзакции `Required` или `RequiresNew` , чтобы сохранить целостность транзакции. С этими атрибутами EJB-контейнер начинает новую транзакцию перед вызовом метода `@Timeout` . Если транзакция откатывается, контейнер будет вызывать метод `@Timeout` как минимум ещё один раз.

Пример timersession

Исходный код для этого примера находится в каталоге *`tut-install`*`/examples/ejb/timersession/src/main/java/` .

`TimerSessionBean` — это сессионный компонент-синглтон, который показывает, как установить автоматический таймер и программный таймер. В следующем листинге исходного кода `TimerSessionBean` методы `setTimer` и `@Timeout` используются для установки программного таймера. Объект `TimerService` инъецируется контейнером при создании компонента. Поскольку это бизнес-метод, `setTimer` доступен для локального представления без интерфейса `TimerSessionBean` и может вызываться клиентом. В этом примере клиент вызывает `setTimer` с интервалом в 8000 миллисекунд или 8 секунд. Метод `setTimer` создаёт новый таймер, вызывая метод `createTimer` для `TimerService` . Теперь, когда таймер установлен, EJB-контейнер будет вызывать метод `programmaticTimeout` компонента `TimerSessionBean` по истечении таймера примерно через 8 секунд:

JAVA

```
...
    public void setTimer(long intervalDuration) {
        logger.log(Level.INFO,
                "Setting a programmatic timeout for {0} milliseconds from now.",
                intervalDuration);
        Timer timer = timerService.createTimer(intervalDuration,
                "Created new programmatic timer");
    }

    @Timeout
    public void programmaticTimeout(Timer timer) {
        this.setLastProgrammaticTimeout(new Date());
        logger.info("Programmatic timeout occurred.");
    }
...
```

`TimerSessionBean` также имеет автоматический таймер с методом `automaticTimeout`. Автоматический таймер срабатывает каждую 1 минуту и устанавливается календарным выражением таймера аннотацией `@Schedule`:

JAVA

```
...
    @Schedule(minute = "*/1", hour = "*", persistent = false)
    public void automaticTimeout() {
        this.setLastAutomaticTimeout(new Date());
        logger.info("Automatic timeout occured");
    }
...
```

`TimerSessionBean` также имеет два бизнес-метода: `getLastProgrammaticTimeout` и `getLastAutomaticTimeout`. Клиенты вызывают эти методы, чтобы получить дату и время последнего тайм-аута для программного таймера и автоматического таймера соответственно.

Вот исходный код для класса `TimerSessionBean`:

JAVA

```java
package ee.jakarta.tutorial.timersession.ejb;

import java.util.Date;
import java.util.logging.Level;
import java.util.logging.Logger;
import jakarta.annotation.Resource;
import jakarta.ejb.Schedule;
import jakarta.ejb.Singleton;
import jakarta.ejb.Startup;
import jakarta.ejb.Timeout;
import jakarta.ejb.Timer;
import jakarta.ejb.TimerService;

@Singleton
@Startup
public class TimerSessionBean {
    @Resource
    TimerService timerService;

    private Date lastProgrammaticTimeout;
    private Date lastAutomaticTimeout;

    private static final Logger logger =
            Logger.getLogger("timersession.ejb.TimerSessionBean");

    public void setTimer(long intervalDuration) {
        logger.log(Level.INFO,
                "Setting a programmatic timeout for {0} milliseconds from now.",
                intervalDuration);
        Timer timer = timerService.createTimer(intervalDuration,
                "Created new programmatic timer");
    }

    @Timeout
    public void programmaticTimeout(Timer timer) {
        this.setLastProgrammaticTimeout(new Date());
        logger.info("Programmatic timeout occurred.");
    }

    @Schedule(minute = "*/1", hour = "*", persistent = false)
    public void automaticTimeout() {
        this.setLastAutomaticTimeout(new Date());
        logger.info("Automatic timeout occured");
    }

    public String getLastProgrammaticTimeout() {
        if (lastProgrammaticTimeout != null) {
            return lastProgrammaticTimeout.toString();
        } else {
            return "never";
        }
    }

    public void setLastProgrammaticTimeout(Date lastTimeout) {
        this.lastProgrammaticTimeout = lastTimeout;
    }

    public String getLastAutomaticTimeout() {
        if (lastAutomaticTimeout != null) {
            return lastAutomaticTimeout.toString();
        } else {
            return "never";
        }
    }

    public void setLastAutomaticTimeout(Date lastAutomaticTimeout) {
        this.lastAutomaticTimeout = lastAutomaticTimeout;
```

```
    }
}
```

Минимальное время ожидания по умолчанию для GlassFish Server составляет 1000 миллисекунд или 1 секунда. Если нужно установить значение тайм-аута меньше 1000 миллисекунд, измените значение параметра Minimum Delivery Interval в Консоли администрирования.

Чтобы изменить минимальное значение времени ожидания, в Консоли администрирования разверните узел Конфигурации, затем разверните server-config, выберите EJB-контейнер и перейдите на вкладку Сервис EJB-таймера. Введите новое значение времени ожидания в поле «Minimum Delivery Interval» и нажмите «Сохранить».

Минимальное практическое значение для `minimum-delivery-interval-in-millis` составляет около 10 миллисекунд из-за ограничений виртуальной машины.

Запуск timersession

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска примера `timersession`.

Запуск timersession с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/ejb
   ```

4. Выберите каталог `timersession`.
5. Нажмите **Открыть проект**.
6. В меню **Запуск** выберите **Запуск проекта**.

   Это создаёт и упаковывает приложение в WAR-файл timersession.war, расположенный в каталоге *`tut-install`*`/examples/ejb/timersession/target/`, и развёртывает этот WAR-файл в GlassFish Server, а затем запускает веб-клиент.

Сборка, упаковка и развёртывание timersession с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/ejb/timersession/
   ```

3. Введите следующую команду:

   ```
   mvn install
   ```
   SHELL

   Это создаёт и упаковывает приложение в WAR-файл timersession.war, расположенный в каталоге *`tut-install`*`/examples/ejb/timersession/target/`, и развёртывает этот WAR-файл в GlassFish Server.

Запуск веб-клиента

1. Откройте веб-браузер по следующему URL:

   ```
   http://localhost:8080/timersession
   ```

2. Нажмите Set Timer, чтобы установить программный таймер.
3. Подождите немного и нажмите кнопку «Обновить» в браузере.

   Вы увидите дату и время последних программных и автоматических тайм-аутов.

   Чтобы просмотреть сообщения, регистрирующиеся при срабатывании таймера, откройте файл `server.log`, расположенный в каталоге *`domain-dir`*`/logs/`.

Обработка исключений

Исключения, выбрасываемые Enterprise-бинами, делятся на две категории: системные и прикладные.

Системное исключение указывает на проблему с сервисами, которые поддерживают приложение. Например, невозможно получить соединение с внешним ресурсом или не найден инъецированный ресурс. Если он сталкивается с проблемой системного уровня, Enterprise-бин должен выбросить `jakarta.ejb.EJBException`. Поскольку `EJBException` является дочерним классом `RuntimeException`, не обязательно указывать его в предложении `throws` объявления метода. Если генерируется системное исключение, контейнер Jakarta Enterprise Beans может уничтожить объект компонента. Следовательно, системная исключительная ситуация не может быть обработана клиентской программой компонента, а требует вмешательства системного администратора.

Исключение приложения сигнализирует об ошибке в бизнес-логике Enterprise-бина. Исключения приложений, как правило, являются исключениями, которые вы сами кодировали, например, `BookException`, создаваемое бизнес-методами примера `CartBean`. Когда Enterprise-бин генерирует исключение приложения, контейнер не переносит его в другое исключение. Клиент должен иметь возможность обрабатывать любое исключение приложения, которое он получает.

Если в транзакции возникает системное исключение, EJB-контейнер откатывает транзакцию. Однако, если исключение приложения выдаётся в транзакции, контейнер не откатывает транзакцию.

Глава 38. Использование встроенного EJB-контейнера

В этой главе показано, как использовать встроенный EJB-контейнер для запуска приложения EJB-компонентов в среде Java SE вне сервера Jakarta EE.

Обзор встроенного EJB-контейнера

Встроенный EJB-контейнер используется для доступа к компонентам EJB из клиентского кода, выполняемого в среде Java SE. Контейнер и клиентский код выполняются в одной виртуальной машине. Встроенный EJB-контейнер обычно используется для тестирования Enterprise-бинов без необходимости их развёртывания на сервере.

Большинство сервисов, присутствующих в контейнере Enterprise-бина на сервере Jakarta EE, доступны во встроенном контейнере Enterprise-бина, включая инъецирование, управляемые контейнером транзакции и безопасность. EJB-компоненты работают одинаково как во встроенных средах, так и в среде Jakarta EE, поэтому один и тот же EJB-компонент может быть повторно использован как в автономных, так и в сетевых приложениях.

Разработка встраиваемых приложений Enterprise-бинов

Все встраиваемые контейнеры EJB поддерживают функции, перечисленные в табл. 38-1.

Таблица 38-1. Необходимые функции Enterprise-бина во встраиваемом контейнере

Функция Enterprise-бина	**Описание**
Локальные сессионные бины	Локальное и неинтерфейсное представление компонентов без сохранения состояния, с сохранением состояния и синглтон. Доступ ко всем методам синхронный. Сессионные компоненты не должны быть конечными точками веб-сервиса.
Транзакции	Управляемые контейнером и компонентом транзакции.
Безопасность	Декларативная и программная безопасность.
Interceptor-ы	Interceptor-ы уровня класса и уровня метода для сессионных компонентов.

Функция Enterprise-бина	Описание
Дескриптор развёртывания	Необязательный дескриптор развёртывания `ejb-jar.xml` с теми же правилами переопределения для EJB-контейнера на серверах Jakarta EE.

Поставщикам контейнеров разрешено поддерживать полный набор функций в Enterprise-бинах, но приложения, использующие встроенный контейнер, не будут переносимыми, если они используют функции Enterprise-бинов, не перечисленные в таблице 38-1, такие как сервис таймера, сессионные компоненты в качестве конечных точек веб-сервиса или удалённые бизнес-интерфейсы.

Запуск встроенных приложений

Встроенный контейнер, компоненты Enterprise-бина и клиент выполняются в одной виртуальной машине с использованием одного и того же classpath. В результате разработчики могут запустить приложение, которое использует встроенный контейнер, как обычное приложение Java SE, следующим образом:

```shell
java -classpath mySessionBean.jar:containerProviderRuntime.jar:myClient.jar \
com.example.ejb.client.Main
```

В приведённом выше примере `mySessionBean.jar` является JAR-модулем EJB, содержащим локальный сессионный компонент без состояния, `containerProviderRuntime.jar` — это JAR-файл, предоставляемый поставщиком EJB-компонента, который содержит необходимые классы времени выполнения для встроенного контейнера, а `myClient.jar` — JAR, содержащий приложение Java SE, которое вызывает бизнес-методы в сессионном компоненте через встроенный контейнер.

Исполняемый JAR-файл GlassFish Server `glassfish-embedded-all.jar` включает классы для встроенного контейнера.

Создание EJB-контейнера

Абстрактный класс `jakarta.ejb.embedded.EJBContainer` представляет EJB-контейнер и включает в себя фабричные методы для создания объекта контейнера. Метод `EJBContainer.createEJBContainer` используется для создания и инициализации объекта встроенного контейнера.

В следующем фрагменте кода показано, как создать встроенный контейнер, который инициализируется с настройками по умолчанию поставщика контейнера:

```java
EJBContainer ec = EJBContainer.createEJBContainer();
```

По умолчанию встроенный контейнер будет искать classpath виртуальных машин для модулей Enterprise-бинов: каталоги, содержащие дескриптор развёртывания `META-INF/ejb-jar.xml` , каталоги, содержащие файл классов с Enterprise-бинов с аннотациями компонентов (такими как `@Stateless`) или файлы JAR, содержащие дескриптор развёртывания `ejb-jar.xml` или файл класса с аннотацией Enterprise-бина. Любые совпадающие записи считаются модулями EJB в одном и том же приложении. Как только все допустимые модули Enterprise-бинов будут найдены в classpath, контейнер начнет инициализацию модулей. Когда метод `createEJBContainer` успешно возвращается, клиентское приложение может получить ссылки на представление клиента любого модуля EJB, найденного встроенным контейнером.

Альтернативная версия метода `EJBContainer.createEJBContainer` принимает `Map` свойств и настроек для настройки объекта встраиваемого контейнера:

```java
Properties props = new Properties();
props.setProperty(...);
...
EJBContainer ec = EJBContainer.createEJBContainer(props);
```

Явное указание модулей Enterprise-бина для инициализации

Разработчики могут точно указать, какие модули Enterprise-бина будут инициализированы встроенным контейнером. Чтобы явно указать модули Enterprise-бина, инициализированные встроенным контейнером, установите свойство `EJBContainer.MODULES` .

Модули могут находиться либо в classpath виртуальных машин, в котором выполняются встроенный контейнер и клиентский код, либо вне classpath виртуальных машин.

Чтобы указать модули в classpath виртуальных машин, задайте для `EJBContainer.MODULES` значение `String` , чтобы указать одно имя модуля, или массив `String` , содержащий имена модулей. Встроенный контейнер ищет classpath виртуальной машины для модулей Enterprise-бина, соответствующих указанным именам:

```java
Properties props = new Properties();
props.setProperty(EJBContainer.MODULES, "mySessionBean");
EJBContainer ec = EJBContainer.createEJBContainer(props);
```

Чтобы указать модули Enterprise-бина вне classpath виртуальной машины, задайте для `EJBContainer.MODULES` объект `java.io.File` или массив объектов `File` , Каждый объект `File` ссылается на JAR-файл EJB или каталог, содержащий расширенный JAR-файл EJB:

```java
Properties props = new Properties();
File ejbJarFile = new File(...);
props.setProperty(EJBContainer.MODULES, ejbJarFile);
EJBContainer ec = EJBContainer.createEJBContainer(props);
```

Поиск ссылок на сессионный компонент

Чтобы получить ссылки на сессионные компоненты в приложении, используя встроенный контейнер:

1. Используйте объект `EJBContainer` , чтобы получить объект `javax.naming.Context` .

   Вызовите метод `EJBContainer.getContext` , чтобы получить объект `Context` :

   ```java
   EJBContainer ec = EJBContainer.createEJBContainer();
   Context ctx = ec.getContext();
   ```

   Ссылки на сессионные компоненты могут быть получены с использованием переносимого синтаксиса JNDI, подробно описанного в Переносимый синтаксис JNDI. Например, чтобы получить ссылку на `MySessionBean` , локальный сессионный компонент с представлением без интерфейса, используйте следующий код:

   ```java
   MySessionBean msb = (MySessionBean)
               ctx.lookup("java:global/mySessionBean/MySessionBean");
   ```

Завершение работы контейнера с Enterprise-бинами

Чтобы закрыть встроенный контейнер:

1. Из клиента вызовите метод `close` объекта `EJBContainer`.

```java
EJBContainer ec = EJBContainer.createEJBContainer();
...
ec.close();
```

> Хотя клиентам не требуется закрывать объекты `EJBContainer`, это освобождает ресурсы, занятые встроенным контейнером. Это особенно важно, когда виртуальная машина, на которой запущено клиентское приложение, превышает время жизни клиентского приложения.

Приложение standalone

Приложение `standalone` демонстрирует, как создать объект встроенного EJB-контейнера в тестовом классе JUnit и вызвать бизнес-метод сессионного компонента.

Обзор приложения standalone

Тестирование бизнес-методов Enterprise-бина в модульном тесте позволяет разработчикам использовать бизнес-логику приложения отдельно от других уровней приложения, таких как уровень представления, без необходимости развёртывания приложения на сервере Jakarta EE.

Пример `standalone` имеет два основных компонента: `StandaloneBean`, сессионный компонент без сохранения состояния и `StandaloneBeanTest`, тестовый класс JUnit, который действует как клиент для `StandaloneBean` с использованием встроенного контейнера.

`StandaloneBean` — это простой сессионный компонент, представляющий локальное представление без интерфейса с бизнес-методом `returnMessage`, возвращающим строку «Greetings!»:

```java
@Stateless
public class StandaloneBean {

    private static final String message = "Greetings!";

    public String returnMessage() {
        return message;
    }

}
```

`StandaloneBeanTest` вызывает `StandaloneBean.returnMessage` и проверяет правильность возвращённого сообщения. Во-первых, встроенный объект контейнера и начальный контекст создаются в методе `setUp`, который аннотирован `org.junit.Before` для указания, что метод должен выполняться перед тестовыми методами:

```java
@Before
public void setUp() {
    ec = EJBContainer.createEJBContainer();
    ctx = ec.getContext();
}
```

Метод `testReturnMessage` аннотирован `org.junit.Test` для указания, что метод включает в себя модульный тест. Он получает ссылку на `StandaloneBean` через `Context` и вызывает `StandaloneBean.returnMessage`. Полученный результат сравнивается с ожидаемым с использованием

`assertEquals` от JUnit. Если `StandaloneBean.returnMessage` возвращает строку "Greetings!", то тест проходит:

JAVA

```
@Test
public void testReturnMessage() throws Exception {
    logger.info("Testing standalone.ejb.StandaloneBean.returnMessage()");
    StandaloneBean instance = (StandaloneBean)
            ctx.lookup("java:global/classes/StandaloneBean");
    String expResult = "Greetings!";
    String result = instance.returnMessage();
    assertEquals(expResult, result);
}
```

Наконец, метод `tearDown` , аннотированный `org.junit.After` для указания того, что метод должен быть выполнен после выполнения всех модульных тестов, закрывает объект встроенного контейнера:

JAVA

```
@After
public void tearDown() {
    if (ec != null) {
        ec.close();
    }
}
```

Запуск приложения standalone с использованием IDE NetBeans

1. В меню **Файл** выберите **Открыть проект**.
2. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/ejb
   ```

3. Выберите каталог `standalone` и кликните **Открыть проект**.
4. На вкладке **Проекты** кликните правой кнопкой мыши `standalone` и выберите **Test**.

   Это выполнит тестовый класс JUnit `StandaloneBeanTest` . На вкладке «Вывод» отображается ход теста и журнал вывода.

Запуск приложения standalone с использованием Maven

1. В окне терминала перейдите в:

   ```
   tut-install/examples/ejb/standalone/
   ```

2. Введите следующую команду:

   SHELL

   ```
   mvn install
   ```

   Эта команда компилирует и упаковывает приложение в файл JAR и выполняет тестовый класс JUnit `StandaloneBeanTest` .

Глава 39. Использование асинхронного вызова методов в сессионных компонентах

В этой главе обсуждается, как реализовать асинхронные бизнес-методы в сессионных компонентах и вызывать их из клиентов Enterprise-бинов.

Вызов асинхронных методов

Сессионные компоненты могут реализовывать асинхронные методы, бизнес-методы, в которых управление возвращается клиенту EJB-контейнером перед вызовом метода в объекте сессионного компонента. Затем клиенты могут использовать API параллелизма Java SE для получения результата, отмены вызова и проверки исключений. Асинхронные методы обычно используются для длительных операций, для задач с интенсивным использованием процессора, для фоновых задач, для увеличения пропускной способности приложения или для увеличения времени отклика приложения, если результат вызова метода не требуется немедленно.

Когда клиент сессионного компонента вызывает типичный не асинхронный бизнес-метод, управление не возвращается клиенту, пока метод не завершится. Однако клиенты, вызывающие асинхронные методы, немедленно получают контроль, возвращаемый им EJB-контейнером. Это позволяет клиенту выполнять другие задачи во время выполнения вызова метода. Если метод возвращает результат, этим результатом является реализация интерфейса `java.util.concurrent.Future<V>`, где «V» — тип значения результата. Интерфейс `Future<V>` определяет методы, которые клиент может использовать для проверки завершения вычислений, ожидания завершения вызова, получения окончательного результата и отмены вызова.

Создание асинхронного бизнес-метода

Аннотируйте бизнес-метод с `jakarta.ejb.Asynchronous`, чтобы обозначить его как асинхронный, или примените `@Asynchronous` на уровне класса, чтобы отметить все бизнес-методы сессионного бина как асинхронные. Методы сессионных компонентов, которые предоставляют веб-сервисы, не могут быть асинхронными.

Асинхронные методы должны возвращать либо `void`, либо реализацию интерфейса `Future<V>`. Асинхронные методы, которые возвращают `void`, не могут объявлять исключения приложения. Но асинхронные методы, возвращающие `Future<V>`, могут объявлять исключения приложения. Например:

JAVA

```
@Asynchronous
public Future<String> processPayment(Order order) throws PaymentException { ... }
```

Этот метод попытается обработать оплату заказа и вернёт статус в виде `String`. Даже если процессор обработки платежей выполняется значительное время, клиент может продолжить работу и отобразить результат после окончательной обработки.

Класс `jakarta.ejb.AsyncResult<V>` — это конкретная реализация интерфейса `Future<V>`, предоставляемая как вспомогательный класс для возврата асинхронных результатов. `AsyncResult` имеет конструктор со строковым параметром-результатом, что упрощает создание реализаций `Future<V>`. Например, метод `processPayment` будет использовать `AsyncResult` для возврата статуса в виде `String`:

JAVA

```
@Asynchronous
public Future<String> processPayment(Order order) throws PaymentException {
    ...
    String status = ...;
    return new AsyncResult<String>(status);
}
```

Результат возвращается в EJB-контейнер, а не непосредственно клиенту, и EJB-контейнер делает результат доступным для клиента. Сессионный компонент может проверить, запросил ли клиент отмену вызова, вызвав метод `jakarta.ejb.SessionContext.wasCancelled`. Например:

JAVA

```
@Asynchronous
public Future<String> processPayment(Order order) throws PaymentException {
    ...
    if (SessionContext.wasCancelled()) {
        // очистка
    } else {
        // обработка платежа
    }
    ...
}
```

Вызов асинхронных методов из Enterprise-бина

Сессионные компоненты вызывают асинхронные методы точно так же, как не асинхронные бизнес-методы. Если асинхронный метод возвращает результат, клиент получает объект `Future<V>` сразу после вызова метода. Этот объект можно использовать для получения окончательного результата, отмены вызова, проверки завершения вызова, проверки наличия каких-либо исключений во время обработки и проверки отмены вызова.

Получение окончательного результата из вызова асинхронного метода

Клиент может получить результат, используя один из методов `Future<V>.get`. Если обработка не была завершена сессионным компонентом, обрабатывающим вызов, то вызов одного из методов `get` приведёт к тому, что клиент остановит выполнение до завершения вызова. Используйте метод `Future<V>.isDone`, чтобы определить, завершена ли обработка перед вызовом одного из методов `get`.

Метод `get()` возвращает результат в виде типа, указанного в значении типа объекта `Future<V>`. Например, вызов `Future<String>.get()` вернёт объект `String`. Если вызов метода был отменён, вызовы `get()` приводят к генерированию `java.util.concurrent.CancellationException`. Если в результате вызова во время обработки сессионным компонентом возникла исключительная ситуация, вызовы `get()` приводят к выбрасыванию `java.util.concurrent.ExecutionException`. Причину `ExecutionException` можно узнать, вызвав метод `ExecutionException.getCause`.

Метод `get(long timeout, java.util.concurrent.TimeUnit unit)` похож на метод `get()`, но позволяет клиенту установить значение тайм-аута. Если превышено значение времени ожидания, генерируется `java.util.concurrent.TimeoutException`. См. Javadoc для класса `TimeUnit` для доступных единиц времени, чтобы указать значение времени ожидания.

Отмена асинхронного вызова метода

Вызовите метод `cancel(boolean mayInterruptIfRunning)` в объекте `Future<V>`, чтобы попытаться отменить вызов метода. Метод `cancel` возвращает `true`, если отмена была успешной, и `false`, если вызов метода не может быть отменён.

Если вызов нельзя отменить, параметр `mayInterruptIfRunning` используется для предупреждения объекта сессионного компонента, на котором выполняется вызов метода, о том, что клиент попытался отменить вызов. Если для `mayInterruptIfRunning` установлено значение `true`, вызовы `SessionContext.wasCancelled` объектом сессионного компонента возвращают `true`. Если `mayInterruptIfRunning` установить `false`, вызовы `SessionContext.wasCancelled` объектом сессионного компонента вернут `false`.

Метод `Future<V>.isCancelled` используется для проверки отмены вызова метода перед завершением вызова асинхронного метода путём вызова `Future<V>.cancel`. Метод `isCancelled` возвращает `true`, если вызов был отменён.

Проверка состояния вызова асинхронного метода

Метод `Future<V>.isDone` возвращает `true`, если объект сессионного компонента завершил обработку вызова метода. Метод `isDone` возвращает `true`, если асинхронный вызов метода завершился нормально, был отменён или привёл к исключению. То есть `isDone` указывает только то, завершил ли сессионный компонент обработку вызова.

Приложение async

Пример `async` демонстрирует, как определить асинхронный бизнес-метод для сессионного компонента и вызвать его из веб-клиента. Этот пример содержит два модуля.

- Веб-приложение (`async-war`), которое содержит сессионный компонент без сохранения состояния и интерфейс Jakarta Faces. Сессионный компонент без состояния `MailerBean` определяет асинхронный метод `sendMessage`, который использует API Jakarta Mail для отправки электронной почты на указанный адрес электронной почты.
- Вспомогательная программа Java SE (`async-smtpd`), которая имитирует SMTP-сервер. Эта программа прослушивает TCP-порт 3025 для запросов SMTP и печатает сообщения электронной почты в стандартный вывод (вместо их доставки).

В следующем разделе описывается архитектура модуля `async-war`.

Архитектура модуля async-war

Модуль `async-war` состоит из одного сессионного компонента без сохранения состояния `MailerBean` и внешнего интерфейса веб-приложения Jakarta Faces, который использует теги Facelets в файлах XHTML для отображения пользователю формы ввода адреса электронной почты получателя. Статус письма обновится, когда письмо будет отправлено.

Сессионный компонент `MailerBean` инъецирует ресурс Jakarta Mail, используемый для отправки сообщения электронной почты на адрес, указанный пользователем. Сообщение создаётся, изменяется и отправляется с использованием API Jakarta Mail. Сессионный компонент выглядит так:

JAVA

```java
@Named
@Stateless
public class MailerBean {
    @Resource(name="mail/myExampleSession")
    private Session session;
    private static final Logger logger =
            Logger.getLogger(MailerBean.class.getName());

    @Asynchronous
    public Future<String> sendMessage(String email) {
        String status;
        try {
            Properties properties = new Properties();
            properties.put("mail.smtp.port", "3025");
            session = Session.getInstance(properties);
            Message message = new MimeMessage(session);
            message.setFrom();
            message.setRecipients(Message.RecipientType.TO,
                    InternetAddress.parse(email, false));
            message.setSubject("Test message from async example");
            message.setHeader("X-Mailer", "Jakarta Mail");
            DateFormat dateFormatter = DateFormat
                    .getDateTimeInstance(DateFormat.LONG, DateFormat.SHORT);
            Date timeStamp = new Date();
            String messageBody = "This is a test message from the async "
                    + "example of the Jakarta EE Tutorial. It was sent on "
                    + dateFormatter.format(timeStamp)
                    + ".";
            message.setText(messageBody);
            message.setSentDate(timeStamp);
            Transport.send(message);
            status = "Sent";
            logger.log(Level.INFO, "Mail sent to {0}", email);
        } catch (MessagingException ex) {
            logger.severe("Error in sending message.");
            status = "Encountered an error: " + ex.getMessage();
            logger.severe(ex.getMessage());
        }
        return new AsyncResult<>(status);
    }
}
```

Инъецированный ресурс Jakarta Mail можно настроить в Консоли администрирования GlassFish Server, административными командами GlassFish Server или с помощью файла конфигурации ресурса, прилагаемого к приложению. Конфигурация ресурса может быть изменена во время выполнения администратором GlassFish Server для использования другого почтового сервера или транспортного протокола.

Веб-клиент состоит из шаблона Facelets `template.xhtml`, двух страниц Facelets `index.xhtml` и `response.xhtml` и Managed-бина Jakarta Faces `MailerManagedBean`. Файл `index.xhtml` содержит форму для целевого адреса электронной почты. Когда пользователь отправляет форму, вызывается метод `MailerManagedBean.send`. Этот метод использует инъецированный объект сессионного компонента `MailerBean` для вызова `MailerBean.sendMessage`. Результат отправляется в представление Facelets `response.xhtml`.

Запуск async

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска примера `async`.

Запуск приложения async с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).

2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне **Открыть проект** перейдите к:

```
tut-install/examples/ejb
```

4. Выберите каталог `async`, затем выберите **Открыть требуемые проекты** и кликните **Открыть проект**.
5. На вкладке **Проекты** кликните правой кнопкой мыши проект `async-smtpd` и выберите **Запуск**.

   Симулятор SMTP-сервера начинает принимать соединения. На вкладке вывода async-smptd отображается следующее сообщение:

```
[Test SMTP server listening on port 3025]
```

6. На вкладке **Проекты** кликните правой кнопкой мыши проект `async-war` и выберите **Сборка**.

   Эта команда настраивает ресурс Jakarta Mail с помощью административной команды GlassFish Server, а также собирает, упаковывает и развёртывает модуль `async-war`.
7. Откройте следующий URL в окне веб-браузера:

```
http://localhost:8080/async-war
```

8. В окне веб-браузера введите адрес электронной почты и нажмите «Отправить письмо».

   Бин без сохранения состояния `MailerBean` использует API Jakarta Mail для доставки электронной почты на симулятор SMTP-сервера. В окне вывода async-smptd в IDE NetBeans отображается полученное сообщение электронной почты, включая его заголовки.
9. Чтобы остановить симулятор SMTP-сервера, нажмите кнопку X в правой части строки состояния в IDE NetBeans.
10. Удалите ресурс сессии Jakarta Mail.
    a. На вкладке «Сервисы» разверните узел «Серверы», а затем разверните узел «Сервер GlassFish».
    b. Разверните узел «Ресурсы», затем разверните узел «Сессии Jakarta Mail».
    c. Кликните правой кнопкой мыши mail/myExampleSession и выберите Unregister.

Запуск async с помощью Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

```
tut-install/examples/ejb/async/async-smtpd/
```

3. Введите следующую команду для сборки и упаковки симулятора SMTP-сервера:

```
mvn install
```
SHELL

4. Введите следующую команду для запуска симулятора сервера STMP:

```
mvn exec:java
```
SHELL

   Появится следующее сообщение:

```
[Test SMTP server listening on port 3025]
```

Держите это окно терминала открытым.

5. В новом окне терминала перейдите по ссылке:

```
tut-install/examples/ejb/async/async-war
```

6. Введите следующую команду для настройки ресурса Jakarta Mail, а также для сборки, упаковки и развёртывания модуля `async-war`:

```
mvn install
```
SHELL

7. Откройте следующий URL в окне веб-браузера:

```
http://localhost:8080/async-war
```

8. В окне веб-браузера введите адрес электронной почты и нажмите «Отправить письмо».

   Бин без сохранения состояния `MailerBean` использует API Jakarta Mail для доставки электронной почты на симулятор SMTP-сервера. Полученное сообщение электронной почты появляется в первом окне терминала, включая его заголовки.

9. Чтобы остановить симулятор SMTP-сервера, закройте окно терминала, в котором вы дали команду на запуск симулятора STMP-сервера.

10. Чтобы удалить ресурс сессии Jakarta Mail, введите следующую команду:

```
asadmin delete-mail-resource mail/myExampleSession
```
SHELL

Часть VIII: Персистентность

В части VIII говорится о Jakarta Persistence.

Глава 40. Введение в Jakarta Persistence

В этой главе приводится описание Jakarta Persistence.

Обзор Jakarta Persistence

Jakarta Persistence предоставляет разработчикам Java объектно-реляционное отображение для управления реляционными данными в приложениях Java. Jakarta Persistence состоит из четырёх областей:

- Персистентность Jakarta
- Язык запросов
- Jakarta Persistence Criteria API
- Метаданные объектно-реляционного отображения

Сущности

Сущность — это легковесный объект персистентной области. Как правило, сущность представляет таблицу в реляционной базе данных, и каждый объект сущности соответствует строке в этой таблице. Основным программным артефактом сущности является класс сущности, хотя сущности могут использовать вспомогательные классы.

Персистентное состояние сущности проявляется через персистентные поля и персистентные свойства. Эти поля или свойства используют аннотации объектно-реляционного отображения для отображения сущностей и отношений сущностей с реляционными данными в хранилище данных.

Требования к классам сущностей

Класс сущности должен соответствовать следующим требованиям.

- Класс должен быть аннотирован `jakarta.persistence.Entity`.
- Класс должен иметь открытый или защищённый (protected) конструктор без аргументов. Класс может иметь другие конструкторы.
- Класс не должен быть объявлен `final`. Методы и персистентные переменные объекта не должны быть объявлены `final`.
- Если объект сущности передаётся по значению как отдельный объект, например, через удалённый бизнес-интерфейс сессионного компонента, класс должен реализовать интерфейс `Serializable`.
- Сущности могут расширять классы сущностей и не-сущностей, а классы не-сущностей могут расширять классы сущностей.
- Персистентные переменные должны быть иметь приватный, защищённый (protected) или пакетный доступ, и доступ к ним можно получить только через методы класса сущности. Клиенты должны получить доступ к состоянию объекта с помощью методов доступа или бизнес-методов.

Персистентные поля и свойства в классах сущностей

Доступ к персистентному состоянию объекта можно получить через переменные или свойства объекта. Поля или свойства должны иметь следующие типы Java:

- Java примитивы
- `java.lang.String`

- Другие сериализуемые типы, в том числе:
  - Обёртки (wrapper) примитивов Java
  - `java.math.BigInteger`
  - `java.math.BigDecimal`
  - `java.util.Date`
  - `java.util.Calendar`
  - `java.sql.Date`
  - `java.sql.Time`
  - `java.sql.TimeStamp`
  - Пользовательские сериализуемые типы
  - `byte[]`
  - `Byte[]`
  - `char[]`
  - `Character[]`
- Перечислимые типы
- Другие объекты и/или коллекции объектов
- Встраиваемые классы

Сущности могут использовать персистентные поля, персистентные свойства или их комбинацию. Если аннотации сопоставления применяются к переменным объекта сущности, сущность использует персистентные поля. Если аннотации сопоставления применяются к методам получения объекта для свойств стиля JavaBeans, объект использует персистентные свойства.

Персистентные поля

Если класс сущности использует персистентные поля, среда выполнения персистентности напрямую обращается к переменным объекта класса сущности. Все поля, не аннотированные `jakarta.persistence.Transient` или ключевым словом Java `transient`, будут сохранены в хранилище данных. Аннотации объектно-реляционного отображения должны применяться к переменным объекта.

Персистентные свойства

Если объект использует персистентные свойства, он должен следовать соглашениям о методах компонентов JavaBeans. Свойства стиля JavaBeans используют get- и set- методы, которые обычно именуются в соответствии с именами переменных объекта класса сущности. Для каждого персистентного свойства *`property`* типа *`Type`* сущности, существует get-метод `get`*`Property`* и set-метод `set`*`Property`*. Если свойство является логическим, может использоваться `is`*`Property`* вместо `get`*`Property`*. Например, если объект `Customer` использует персистентные свойства и имеет приватную переменную объекта с именем `firstName`, класс определяет методы `getFirstName` и `setFirstName` для получения и установки состояния переменной `firstName` объекта.

Сигнатуры метода для однозначных персистентных свойств следующие:

```java
Type getProperty()
void setProperty(Type type)
```

Аннотации объектно-реляционного отображения для персистентных свойств должны быть применимы к get-методам. Аннотации отображения нельзя применять к полям или свойствам, аннотированным `@Transient` или помеченным `transient`.

Использование коллекций в полях и свойствах сущностей

Персистентные поля и свойства со значением коллекции должны использовать поддерживаемые интерфейсы коллекции Java независимо от того, использует объект персистентные поля или свойства. Можно использовать следующие интерфейсы коллекций:

- `java.util.Collection`
- `java.util.Set`
- `java.util.List`
- `java.util.Map`

Если класс сущности использует персистентные поля, тип в сигнатурах предыдущего метода должен быть одним из этих типов коллекции. Универсальные варианты этих типов коллекций также могут быть использованы. Например, если у него есть персистентное свойство, содержащее набор телефонных номеров, сущность `Customer` будет иметь следующие методы:

JAVA

```
Set<PhoneNumber> getPhoneNumbers() { ... }
void setPhoneNumbers(Set<PhoneNumber>) { ... }
```

Если поле или свойство сущности состоит из коллекции базовых типов или встраиваемых классов, используйте аннотацию `jakarta.persistence.ElementCollection` для поля или свойства.

Атрибуты `targetClass` и `fetch` аннотации `@ElementCollection`. Атрибут `targetClass` указывает имя класса базового или встраиваемого класса и является необязательным, если поле или свойство определяются с использованием обобщений (generics) Java. Необязательный атрибут `fetch` указывает, следует использовать раннее или отложенное извлечение коллекции, используя константы `jakarta.persistence.FetchType` `LAZY` или `EAGER` соответственно. По умолчанию коллекция будет извлекаться отложенно (lazy).

Следующая сущность, `Person`, имеет персистентное поле `nicknames`, которое представляет собой коллекцию классов `String` и для которой будет использоваться раннее (eager) извлечение. Элемент `targetClass` не является обязательным, потому что он использует обобщённые значения (generics) при определении поля:

JAVA

```
@Entity
public class Person {
    ...
    @ElementCollection(fetch=EAGER)
    protected Set<String> nickname = new HashSet();
    ...
}
```

Коллекции элементов сущностей и отношений могут быть представлены коллекциями `java.util.Map`. `Map` состоит из ключа и значения.

При использовании элементов или отношений `Map` применяются следующие правила.

- Ключ или значение `Map` могут быть базовым типом Java, встраиваемым классом или сущностью.

- Если значение `Map` является встраиваемым классом или базовым типом, используйте аннотацию `@ElementCollection`.
- Если значение `Map` является сущностью, используйте аннотацию `@OneToMany` или `@ManyToMany`.
- Используйте тип `Map` только для одной из сторон двунаправленных отношений.

Если тип ключа `Map` является базовым типом языка Java, используйте аннотацию `jakarta.persistence.MapKeyColumn`, чтобы установить отображение столбца для ключа. По умолчанию атрибут `name` аннотации `@MapKeyColumn` имеет вид *RELATIONSHIP-FIELD/PROPERTY-NAME* `_KEY`. Например, если имя ссылочного поля отношения `image`, значением по умолчанию атрибута `name` будет `IMAGE_KEY`.

Если типом ключа `Map` является сущность, используйте аннотацию `jakarta.persistence.MapKeyJoinColumn`. Если для установки сопоставления необходимо несколько столбцов, используйте аннотацию `jakarta.persistence.MapKeyJoinColumns`, чтобы включить несколько аннотаций `@MapKeyJoinColumn`. Если `@MapKeyJoinColumn` отсутствует, имя столбца сопоставления по умолчанию установлено в *RELATIONSHIP-FIELD/PROPERTY-NAME* `_KEY`. Например, если имя поля отношения — `employee`, атрибут `name` по умолчанию — `EMPLOYEE_KEY`.

Если generic-типы языка Java не используются в поле отношения или свойстве, класс ключа должен быть явно установлен аннотацией `jakarta.persistence.MapKeyClass`.

Если параметр `Map` является первичным ключом, персистентным полем или свойством сущности `Map`, используйте аннотацию `jakarta.persistence.MapKey`. Аннотации `@MapKeyClass` и `@MapKey` нельзя использовать для одного поля или свойства.

Если значение `Map` является базовым типом Java или встраиваемым классом, оно будет отображено как таблица коллекции в базе данных. Если обобщённые (generic) типы не используются, для атрибута `targetClass` аннотации `@ElementCollection` должен быть указан тип значения `Map`.

Если значение `Map` является сущностью и частью отношения «многие ко многим» или однонаправленного отношения «один ко многим», оно будет отображено на промежуточную таблицу в базе данных. Однонаправленное отношение «один ко многим», в котором используется `Map`, также может быть отображено с помощью аннотации `@JoinColumn`.

Если объект является частью двунаправленного отношения «один ко многим/многие к одному», он будет отображён в таблице объекта, представляющего значение `Map`. Если обобщённые (generic) типы не используются, атрибут `targetEntity` в аннотациях `@OneToMany` и `@ManyToMany` должен быть установлен на тип значения `Map`.

Валидация персистентных полей и свойств

Jakarta Bean Validation предоставляет механизм валидации данных приложения. Bean Validation интегрирован в контейнеры Jakarta EE, что позволяет использовать одну и ту же логику валидации на любом уровне корпоративного приложения.

Ограничения Bean Validation могут применяться к классам постоянных сущностей, встраиваемым классам и отображаемым родительским классам. По умолчанию Persistence provider будет автоматически выполнять валидацию сущностей с персистентными полями или свойствами, аннотированными ограничениями Bean Validation сразу после событий `PrePersist`, `PreUpdate` и `PreRemove` жизненного цикла.

Ограничения Bean Validation — это аннотации, применяемые к полям или свойствам классов Java. Bean Validation предоставляет набор ограничений, а также API для определения пользовательских ограничений. Пользовательские ограничения могут быть комбинациями ограничений по умолчанию или новыми

ограничениями, которые не используют ограничения по умолчанию. Каждое ограничение связано по крайней мере с одним классом валидатора, который проверяет значение поля или свойства. Разработчики пользовательских ограничений также должны предоставить класс валидатора для ограничения.

Ограничения Bean Validation применяются к персистентным полям или свойствам персистентных классов. При добавлении ограничений Bean Validation используйте ту же стратегию доступа, что и для персистентного класса. То есть, если персистентный класс использует доступ к полям, примените аннотации ограничения Bean Validation к полям класса. Если класс использует доступ к свойству, примените ограничения на get-методы.

Таблица 23-1 перечисляет встроенные ограничения Bean Validation, определённые в пакете `jakarta.validation.constraints`.

Все встроенные ограничения, перечисленные в таблице 23-1, имеют соответствующую аннотацию *`ConstraintName`*`.List` для группировки нескольких ограничений одного типа на том же поле или свойстве. Например, следующее персистентное поле имеет два ограничения `@Pattern`:

JAVA

```
@Pattern.List({
    @Pattern(regexp="..."),
    @Pattern(regexp="...")
})
```

Следующий класс сущности, `Contact`, имеет ограничения Bean Validation, примененные к его персистентным полям:

JAVA

```
@Entity
public class Contact implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @NotNull
    protected String firstName;
    @NotNull
    protected String lastName;
    @Pattern(regexp = "[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\\."
            + "[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@"
            + "(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.)+[a-z0-9]"
            + "(?:[a-z0-9-]*[a-z0-9])?",
            message = "{invalid.email}")
    protected String email;
    @Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
            message = "{invalid.phonenumber}")
    protected String mobilePhone;
    @Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
            message = "{invalid.phonenumber}")
    protected String homePhone;
    @Temporal(jakarta.persistence.TemporalType.DATE)
    @Past
    protected Date birthday;
    ...
}
```

Аннотация `@NotNull` в полях `firstName` и `lastName` указывает, что эти поля теперь обязательны для заполнения. Если создаётся новый объект `Contact`, где `firstName` или `lastName` не были инициализированы, Bean Validation выдаст ошибку валидации. Аналогично, если ранее созданный объект `Contact` был изменён так, что `firstName` или `lastName` имеют значение null, будет выдана ошибка валидации.

К полю `email` применено ограничение `@Pattern` со сложным регулярным выражением, которое соответствует большинству допустимых адресов электронной почты. Если значение `email` не соответствует этому регулярному выражению, будет выдана ошибка валидации.

Поля `homePhone` и `mobilePhone` имеют одинаковые ограничения `@Pattern`. Регулярное выражение соответствует 10-значным телефонным номерам США и Канады в формате `( xxx ) xxx - xxxx`.

Поле `birthday` аннотировано ограничением `@Past`, чтобы гарантировать, что значение `birthday` будет в прошлом.

Первичные ключи в сущностях

Каждый объект имеет уникальный идентификатор объекта. Например, объект клиента может быть идентифицирован по номеру клиента. Уникальный идентификатор или первичный ключ позволяет клиентам находить конкретный объект. Каждый объект должен иметь первичный ключ. Сущность может иметь простой или составной первичный ключ.

Простые первичные ключи используют аннотацию `jakarta.persistence.Id` для обозначения свойства или поля первичного ключа.

Составные первичные ключи используются, когда первичный ключ состоит из более чем одного атрибута, который соответствует набору отдельных персистентных свойств или полей. Составные первичные ключи должны быть определены в классе первичных ключей. Составные первичные ключи обозначаются аннотациями `jakarta.persistence.EmbeddedId` и `jakarta.persistence.IdClass`.

Первичный ключ или свойство или поле составного первичного ключа должны быть одного из следующих типов языка Java:

- Java примитивы
- Обёртки (wrapper) примитивов Java
- `java.lang.String`
- `java.util.Date` (временной тип должен быть `DATE`)
- `java.sql.Date`
- `java.math.BigDecimal`
- `java.math.BigInteger`

Типы с плавающей точкой никогда не должны использоваться в первичных ключах. Если вы используете сгенерированный первичный ключ, учтите, что только целочисленные типы являются переносимыми.

Класс первичного ключа должен соответствовать следующим требованиям.

- Модификатор доступа класса должен быть `public`.
- Свойства класса первичного ключа должны быть `public` или `protected`, если используется доступ на основе свойств.
- Класс должен иметь публичный конструктор по умолчанию.
- Класс должен реализовывать методы `hashCode()` и `equals(Object other)`.
- Класс должен быть сериализуемым.

- Составной первичный ключ должен включать несколько полей или свойств класса сущности или должен быть представлен как встраиваемый класс.
- Если класс сопоставлен с несколькими полями или свойствами класса сущности, имена и типы полей или свойств первичного ключа в классе первичного ключа должны совпадать с именами класса сущности.

Следующий класс первичного ключа является составным ключом, и поля `customerOrder` и `itemId` вместе однозначно идентифицируют объект сущности:

JAVA

```
public final class LineItemKey implements Serializable {
    private Integer customerOrder;
    private int itemId;

    public LineItemKey() {}

    public LineItemKey(Integer order, int itemId) {
        this.setCustomerOrder(order);
        this.setItemId(itemId);
    }

    @Override
    public int hashCode() {
        return ((this.getCustomerOrder() == null
                ? 0 : this.getCustomerOrder().hashCode())
                ^ ((int) this.getItemId()));
    }

    @Override
    public boolean equals(Object otherOb) {
        if (this == otherOb) {
            return true;
        }
        if (!(otherOb instanceof LineItemKey)) {
            return false;
        }
        LineItemKey other = (LineItemKey) otherOb;
        return ((this.getCustomerOrder() == null
                ? other.getCustomerOrder() == null : this.getCustomerOrder()
                .equals(other.getCustomerOrder()))
                && (this.getItemId() == other.getItemId()));
    }

    @Override
    public String toString() {
        return "" + getCustomerOrder() + "-" + getItemId();
    }
    /* get- и set- методы */
}
```

Множественность в отношениях сущностей

Отношения между сущностями бывают следующих типов.

- Один-к-одному: каждый объект связан с одним другим объектом. Например, для моделирования физического хранилища, в котором каждая корзина содержит один виджет, `StorageBin` и `Widget` будут иметь отношение один к одному. Отношения «один к одному» используют аннотацию `jakarta.persistence.OneToOne` в соответствующем персистентном свойстве или поле.
- Один-ко-многим: объект сущности может быть связан с несколькими объектами других сущностей. Например, заказ на продажу может содержать несколько позиций. В приложении заказа `CustomerOrder` будет иметь отношение «один ко многим» с `LineItem`. Отношения «один ко многим» используют аннотацию `jakarta.persistence.OneToMany` в соответствующем персистентном свойстве или поле.

- Многие-к-одному: несколько объектов сущности могут быть связаны с одним объектом другой сущности. Эта множественность является противоположностью отношения «один ко многим». В только что упомянутом примере отношение к `CustomerOrder` с точки зрения `LineItem` является «многие к одному». Отношения «многие к одному» используют аннотацию `jakarta.persistence.ManyToOne` в соответствующем персистентном свойстве или поле.
- Многие-ко-многим: объекты сущности могут быть связаны с несколькими объектами друг друга. Например, на каждом курсе колледжа есть много студентов, и каждый студент может пройти несколько курсов. Поэтому в заявке на зачисление `Course` и `Student` будут иметь отношение «многие ко многим». Отношения «многие ко многим» используют аннотацию `jakarta.persistence.ManyToMany` в соответствующем персистентном свойстве или поле.

Направление в отношениях сущностей

Отношения могут быть двунаправленными или однонаправленными. Двунаправленное отношение имеет как собственную сторону, так и обратную сторону. Однонаправленные отношения имеют только свою сторону. Сторона-владелец связи определяет, как среда выполнения персистентности обновляет связь в базе данных.

Двунаправленные отношения

В двунаправленном отношении каждый объект имеет поле отношения или свойство, которое ссылается на другой объект. Через поле или свойство отношения код класса сущности может получить доступ к связанному объекту. Если у сущности есть связанное поле, говорят, что сущность «знает» о своём родственном объекте. Например, если `CustomerOrder` знает, какие у него есть объекты `LineItem` и если `LineItem` знает, к какому `CustomerOrder` он принадлежит, то отношение этих сущностей двунаправленное.

Двунаправленные отношения должны следовать следующим правилам.

- Обратная сторона двунаправленного отношения должна ссылаться на сторону владельца, используя элемент `mappedBy` аннотаций `@OneToOne`, `@OneToMany` или `@ManyToMany`. Элемент `mappedBy` обозначает свойство или поле в объекте, который является владельцем отношения.
- Двусторонняя связь «многие к одному» не должна определять элемент `mappedBy`. Множественная сторона всегда является владельцем отношения.
- Для двунаправленных отношений «один к одному» сторона-владелец соответствует стороне, которая содержит соответствующий внешний ключ.
- Для двунаправленных отношений «многие ко многим» любая сторона может быть стороной-владельцем.

Однонаправленные отношения

В однонаправленном отношении только один объект имеет поле или свойство отношения, которое относится к другому. Например, `LineItem` будет иметь поле отношения, которое идентифицирует `Product`, но `Product` не будет иметь поле отношения или свойство для `LineItem`. Другими словами, `LineItem` знает о `Product`, но `Product` не знает, какие объекты `LineItem` ссылаются на него.

Запросы и направление отношений

Язык запросов Jakarta Persistence и запросы Criteria API часто перемещаются по взаимосвязям. Направление отношения определяет, может ли запрос перемещаться от одного объекта к другому. Например, запрос может перейти от `LineItem` к `Product`, но не может перейти в противоположном направлении. Для `CustomerOrder` и `LineItem` запрос может перемещаться в обоих направлениях, поскольку эти два объекта имеют двунаправленное отношение.

Каскадные операции и отношения

Объекты, которые используют отношения, часто имеют зависимости от существования другого объекта в отношениях. Например, позиция является частью заказа. Если заказ удален, позиция также должна быть удалена. Это называется отношением каскадного удаления.

Перечислимый тип `jakarta.persistence.CascadeType` определяет каскадные операции, которые применяются в элементе `cascade` аннотаций отношений. Таблица 40-1 перечисляет каскадные операции для объектов.

Таблица 40-1 Каскадные операции для сущностей

Каскадная операция	**Описание**
`ALL`	Все каскадные операции будут применены к сущности, связанной с родительской сущностью. `All` эквивалентно указанию `cascade={DETACH, MERGE, PERSIST, REFRESH, REMOVE}`
`DETACH`	Если родительский объект отсоединён от контекста персистентности, связанный объект также будет отсоединён.
`MERGE`	Если родительский объект объединён с контекстом персистентности, связанный объект также будет объединён.
`PERSIST`	Если родительский объект сохраняется в контексте персистентности, связанный объект также будет сохраняться.
`REFRESH`	Если родительский объект обновляется в текущем контексте персистентности, связанный объект также будет обновляться.
`REMOVE`	Если родительский объект удаляется из текущего контекста персистентности, связанный объект также будет удалён.

Отношения каскадного удаления задаются с помощью указания `cascade=REMOVE` для отношений `@OneToOne` и `@OneToMany` . Например:

JAVA

```java
@OneToMany(cascade=REMOVE, mappedBy="customer")
public Set<CustomerOrder> getOrders() { return orders; }
```

Удаление потерянных объектов

Когда целевой объект в отношении «один-к-одному» или «один-ко-многим» удаляется, часто желательно каскадно удалить и связанные с ним объекты. Такие связанные объекты считаются потерянными, а атрибут `orphanRemoval` может использоваться для указания того, что такие потерянные объекты должны быть удалены. Например, если заказ содержит много позиций, и одна из них удалена из заказа, удалённая позиция считается потерянной. Если для `orphanRemoval` установлено значение `true` , сущность позиции будет удалена при удалении позиции из заказа.

Атрибут `orphanRemoval` в `@OneToMany` и `@OneToOne` принимает логическое значение и по умолчанию имеет значение false.

В следующем примере каскадно удаляется потерянный объект `order` при удалении объекта `customer` :

JAVA

```
@OneToMany(mappedBy="customer", orphanRemoval="true")
public List<CustomerOrder> getOrders() { ... }
```

Встраиваемые классы в сущностях

Встраиваемые классы используются для представления состояния объекта, но не имеют собственного персистентного идентификатора в отличие от классов объектов. Объекты встраиваемого класса используют идентификатор объекта, которому они принадлежат. Встраиваемые классы существуют только как состояние другого объекта. Сущность может иметь однозначные атрибуты или атрибуты-коллекции встраиваемых классов.

Встраиваемые классы имеют те же правила, что и классы сущностей, но аннотируются `jakarta.persistence.Embeddable` вместо `@Entity`.

Следующий встраиваемый класс, `ZipCode`, имеет поля `zip` и `plusFour`:

JAVA

```
@Embeddable
public class ZipCode {
    String zip;
    String plusFour;
    ...
}
```

Этот встраиваемый класс используется сущностью `Address`:

JAVA

```
@Entity
public class Address {
    @Id
    protected long id
    String street1;
    String street2;
    String city;
    String province;
    @Embedded
    ZipCode zipCode;
    String country;
    ...
}
```

Сущности, имеющие встраиваемые классы в своём составе, могут аннотировать поле или свойство аннотацией `jakarta.persistence.Embedded`, но не обязаны этого делать.

Встраиваемые классы могут сами использовать другие встраиваемые классы для представления своего состояния. Они также могут содержать коллекции базовых типов Java или других встраиваемых классов. Встраиваемые классы также могут содержать отношения к другим объектам или коллекциям объектов. Если у встраиваемого класса есть такая связь, то связь идёт от целевой сущности или коллекции сущностей к сущности, которой принадлежит встраиваемый класс.

Наследование сущностей

Сущности поддерживают наследование классов, полиморфные ассоциации и полиморфные запросы. Классы сущностей могут расширять классы не-сущностей, а классы не-сущностей могут расширять классы сущностей. Классы сущностей могут быть как абстрактными, так и конкретными.

Приложение `roster` демонстрирует наследование объектов, как описано в Наследование сущностей в приложении roster.

Абстрактные сущности

Абстрактный класс может быть объявлен как сущность путём указания аннотации `@Entity` для класса. Абстрактные сущности подобны конкретным сущностям, но не могут быть инстанцированы.

Выборка абстрактных сущностей происходит так же, как и конкретных сущностей. Если абстрактная сущность является целью запроса, запрос работает со всеми конкретными дочерними классами абстрактной сущности:

JAVA

```
@Entity
public abstract class Employee {
    @Id
    protected Integer employeeId;
    ...
}
@Entity
public class FullTimeEmployee extends Employee {
    protected Integer salary;
    ...
}
@Entity
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;
}
```

Сопоставленные родительские классы

Сущности могут наследовать от родительских классов, которые содержат информацию о персистентном состоянии и отображении, но не являются сущностями. То есть родительский класс не аннотирован `@Entity` и не будет распознан как сущность в Persistence provider. Эти родительские классы чаще всего используются, когда имеется информация о состоянии и отображении, общая для нескольких классов сущностей.

Сопоставленные суперклассы указываются путем украшения класса аннотацией `jakarta.persistence.MappedSuperclass`:

JAVA

```
@MappedSuperclass
public class Employee {
    @Id
    protected Integer employeeId;
    ...
}
@Entity
public class FullTimeEmployee extends Employee {
    protected Integer salary;
    ...
}
@Entity
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;
    ...
}
```

Для сопоставленных родительских классов нельзя сделать выборку и нельзя использовать в операциях `EntityManager` или `Query`. Вы должны использовать дочерние классы сущности сопоставленного родительского класса в операциях `EntityManager` или `Query`. Сопоставленные родительские классы не могут быть объектами отношений сущностей. Сопоставленные родительские классы могут быть абстрактными или конкретными.

Сопоставленные родительские классы не имеют соответствующих таблиц в хранилище данных. Объекты, которые наследуются от сопоставленного родительского класса, определяют mapping на таблицы. Например, в предыдущем примере кода были бы таблицы `FULLTIMEEMPLOYEE` и `PARTTIMEEMPLOYEE`, но таблицы `EMPLOYEE` нет.

Родительские классы, не являющиеся сущностями

Родительские классы сущностей могут не являться сущностями, и эти родительские классы могут быть абстрактными или конкретными. Родительские классы, не являющиеся сущностями, не имеют персистентного состояния, и любое состояние, унаследованное от такого родительского класса классом сущностей, не является персистентным. Не-сущностные родительские классы нельзя использовать в операциях `EntityManager` или `Query`. Любые аннотации отображения или отношений в родительских классах — не-сущностях игнорируются.

Стратегии отображения при наследовании сущностей

Вы можете настроить, как провайдер Jakarta Persistence сопоставляет унаследованные объекты с хранилищем данных, аннотируя корневой класс иерархии `jakarta.persistence.Inheritance`. Следующие стратегии используются для отображения данных сущности на базу данных:

- Единая таблица на иерархию классов
- По таблице на конкретный класс сущностей
- Стратегия «объединения», при которой поля и свойства, специфичные для дочерних классов, отображаются на таблицу, отличную от таблицы для полей и свойств родительского класса.

Стратегия настраивается путём установки элемента `strategy` в `@Inheritance` в одну из опций, определённых перечисляемым типом `jakarta.persistence.InheritanceType`:

JAVA

```
public enum InheritanceType {
    SINGLE_TABLE,
    JOINED,
    TABLE_PER_CLASS
};
```

Стратегия по умолчанию, `InheritanceType.SINGLE_TABLE`, используется, если аннотация `@Inheritance` не указана в корневом классе иерархии сущностей.

Стратегия "Единая таблица на иерархию классов"

С помощью этой стратегии, которая соответствует `InheritanceType.SINGLE_TABLE` по умолчанию, все классы в иерархии отображаются в одну таблицу в базе данных. Эта таблица имеет столбец дискриминатора, содержащий значение, идентифицирующее дочерний класс, которому принадлежит объект, представленный строкой.

Столбец дискриминатора, элементы которого показаны в Таблице 40-2, можно указать аннотацией `jakarta.persistence.DiscriminatorColumn` в корне иерархии классов сущностей.

Таблица 40-2 @DiscriminatorColumn Elements

Тип	Название	Описание
`String`	`name`	Имя столбца, который будет использоваться в качестве столбца дискриминатора. По умолчанию используется значение `DTYPE`. Этот элемент не является обязательным.

Тип	Название	Описание
`DiscriminatorType`	`discriminatorType`	Тип столбца, который будет использоваться в качестве столбца дискриминатора. По умолчанию используется значение `DiscriminatorType.STRING`. Этот элемент не является обязательным.
`String`	`columnDefinition`	Фрагмент SQL для использования при создании столбца дискриминатора. Значение по умолчанию создаётся Persistence provider-ом и зависит от реализации. Этот элемент не является обязательным.
`String`	`length`	Длина столбца для типов дискриминаторов на основе `String`. Этот элемент игнорируется, если тип дискриминатора не-`String`. По умолчанию установлено значение 31. Этот элемент не является обязательным.

Перечисляемый тип `jakarta.persistence.DiscriminatorType` используется для указания типа столбца дискриминатора в базе данных, задав для элемента `discriminatorType` одно из значений `@DiscriminatorColumn`. `DiscriminatorType` определяется следующим образом:

JAVA

```
public enum DiscriminatorType {
    STRING,
    CHAR,
    INTEGER
};
```

Если `@DiscriminatorColumn` не указан в корне иерархии сущностей и требуется столбец дискриминатора, Persistence provider принимает имя столбца по умолчанию `DTYPE` и тип столбца `DiscriminatorType.STRING`.

Аннотация `jakarta.persistence.DiscriminatorValue` может использоваться для установки значения, введённого в столбец дискриминатора для каждой сущности в иерархии классов. Аннотация `@DiscriminatorValue` может быть установлена только конкретным классам сущностей.

Если `@DiscriminatorValue` не указан для объекта в иерархии классов, в котором используется столбец дискриминатора, Persistence provider предоставит значение по умолчанию, зависящее от реализации. Если элемент `discriminatorType` в `@DiscriminatorColumn` равен `DiscriminatorType.STRING`, значением по умолчанию является имя объекта.

Эта стратегия обеспечивает хорошую поддержку полиморфных отношений между сущностями и запросами, которые охватывают всю иерархию классов сущностей. Однако эта стратегия требует, чтобы столбцы, которые содержат состояние дочерних классов, могли принимать значение null.

Стратегия "По таблице на конкретный класс сущностей"

В этой стратегии, которая соответствует `InheritanceType.TABLE_PER_CLASS`, каждый конкретный класс сопоставляется с отдельной таблицей в базе данных. Все поля или свойства в классе, включая унаследованные поля или свойства, сопоставляются со столбцами в таблице класса в базе данных.

Эта стратегия плохо поддерживает полиморфные отношения и обычно требует либо запросов SQL `UNION`, либо отдельных запросов SQL для каждого дочернего класса для запросов, которые охватывают всю иерархию классов сущностей.

Поддержка этой стратегии не является обязательной и может поддерживаться не всеми поставщиками Jakarta Persistence. Провайдер Jakarta Persistence по умолчанию в GlassFish Server не поддерживает эту стратегию.

Стратегия объединения дочерних классов

В этой стратегии, которая соответствует `InheritanceType.JOINED`, корень иерархии классов представлен одной таблицей, и каждый дочерний класс имеет отдельную таблицу, которая содержит только те поля, которые относятся к этому дочернему классу. То есть таблица дочернего класса не содержит столбцы для унаследованных полей или свойств. Таблица дочернего класса также имеет столбец или столбцы, которые представляют её первичный ключ, который является внешним ключом первичного ключа таблицы родительского класса.

Эта стратегия обеспечивает хорошую поддержку полиморфных отношений, но требует выполнения одной или нескольких операций соединения при создании объектов дочерних классов сущностей. Это может привести к низкой производительности для обширных иерархий классов. Точно так же запросы, которые покрывают всю иерархию классов, требуют операций соединения между таблицами дочерних классов, что приводит к снижению производительности.

Некоторые провайдеры Jakarta Persistence, включая провайдера по умолчанию в GlassFish Server, требуют столбец дискриминатора, который соответствует корневой сущности при использовании стратегии объединения дочерних классов. Если вы не используете автоматическое создание таблиц в своём приложении, убедитесь, что таблица базы данных настроена правильно для значений по умолчанию для столбца дискриминатора, или используйте аннотацию `@DiscriminatorColumn` для соответствия схеме вашей базы данных. Для получения информации о столбцах дискриминатора см. Единая таблица на иерархию классов.

Управление сущностями

Сущности управляются entity manager-ом, который представлен объектом `jakarta.persistence.EntityManager`. Каждый объект `EntityManager` связан с контекстом персистентности: набором объектов сущностей, которые существуют в хранилище данных. Контекст персистентности определяет область, в которой создаются, сохраняются и удаляются объекты сущностей. Интерфейс `EntityManager` определяет методы, которые используются для взаимодействия с контекстом персистентности.

Интерфейс EntityManager

API `EntityManager` создаёт и удаляет объекты сущностей, осуществляет выборку объектов сущности по первичному ключу сущности и позволяет выполнять запросы к сущностям.

Управляемые контейнером Entity Manager-ы

При использовании entity manager-а, управляемого контейнером, контекст персистентности объекта `EntityManager` автоматически распространяется контейнером на все компоненты приложения, использующие объект `EntityManager` в рамках одной транзакции Jakarta.

Транзакции в Jakarta обычно включают вызовы компонентов приложения. Для завершения транзакции Jakarta этим компонентам обычно требуется доступ к единому контексту персистентности. Это происходит, когда `EntityManager` инъецируется в компоненты приложения аннотацией `jakarta.persistence.PersistenceContext`. Контекст персистентности автоматически распространяется вместе с текущей транзакцией Jakarta, а ссылки `EntityManager`, имеющие одинаковый юнит персистентности, предоставляют доступ к контексту персистентности в этой транзакции. Благодаря автоматическому распространению (propagating) контекста персистентности компонентам приложения не

нужно передавать ссылки на объекты `EntityManager` друг другу, чтобы вносить изменения в одной транзакции. Контейнер Jakarta EE управляет жизненным циклом entity manager-ов, управляемых контейнером.

Чтобы получить объект `EntityManager`, инъецируйте EntityManager в компонент приложения:

JAVA

```
@PersistenceContext
EntityManager em;
```

Управляемые приложением Entity Manager-ы

С другой стороны, с помощью entity manager-а, управляемого приложением, контекст персистентности не распространяется на компоненты приложения, а жизненный цикл объектов `EntityManager` управляется приложением.

Управляемые приложением entity manager-ы используются, когда приложению требуется доступ к контексту персистентности, который не распространяется с транзакцией Jakarta между объектами `EntityManager` в определённом юните персистентности. В этом случае каждый `EntityManager` создаёт новый изолированный контекст персистентности. `EntityManager` и связанный с ним контекст персистентности создаются и уничтожаются приложением явным образом. Они также используются, когда прямое инъецирование объектов `EntityManager` невозможно, поскольку объекты `EntityManager` не являются потокобезопасными. Объекты `EntityManagerFactory` являются потокобезопасными.

В этом случае приложения создают объекты `EntityManager` с помощью метода `createEntityManager` из `jakarta.persistence.EntityManagerFactory`.

Чтобы получить объект `EntityManager`, сначала нужно получить объект `EntityManagerFactory`, инъецировав его в компонент приложения аннотацией `jakarta.persistence.PersistenceUnit`:

JAVA

```
@PersistenceUnit
EntityManagerFactory emf;
```

Затем получите `EntityManager` из объекта `EntityManagerFactory`:

JAVA

```
EntityManager em = emf.createEntityManager();
```

Entity manager-ы, управляемые приложением, не распространяют автоматически контекст транзакций Jakarta. Таким приложениям необходимо вручную получить доступ к transaction manager-у Jakarta и добавить информацию о разграничении транзакций при выполнении операций с объектами. Интерфейс `jakarta.transaction.UserTransaction` определяет методы для запуска, фиксации и отката транзакций. Инъецируйте объект `UserTransaction`, создав переменную объекта с аннотацией `@Resource`:

JAVA

```
@Resource
UserTransaction utx;
```

Чтобы начать транзакцию, вызовите метод `UserTransaction.begin`. Когда все операции с объектами сущностей завершены, вызовите метод `UserTransaction.commit` для фиксации транзакции. Метод `UserTransaction.rollback` используется для отката текущей транзакции.

В следующем примере показано, как управлять транзакциями в приложении, в котором используется EntityManager, управляемый приложением:

JAVA

```
@PersistenceUnit
EntityManagerFactory emf;
EntityManager em;
@Resource
UserTransaction utx;
...
em = emf.createEntityManager();
try {
    utx.begin();
    em.persist(SomeEntity);
    em.merge(AnotherEntity);
    em.remove(ThirdEntity);
    utx.commit();
} catch (Exception e) {
    utx.rollback();
}
```

Выборка сущностей с помощью EntityManager

Метод `EntityManager.find` используется для выборки объектов из хранилища данных по их первичному ключу:

JAVA

```
@PersistenceContext
EntityManager em;
public void enterOrder(int custID, CustomerOrder newOrder) {
    Customer cust = em.find(Customer.class, custID);
    cust.getOrders().add(newOrder);
    newOrder.setCustomer(cust);
}
```

Управление жизненным циклом объекта сущности

Объекты сущности управляются с помощью операций объекта `EntityManager`. Объекты сущностей могут находиться в одном из четырёх состояний: новое, управляемое (managed), отсоединённое (detached) или удалённое (removed).

- Новые объекты сущностей ещё не сохранены в хранилище и не связаны с контекстом персистентности.
- Управляемые объекта уже сохранены в хранилище и связаны с контекстом персистентности.
- Отсоединённые объекты сущности сохранены в хранилище, но в настоящее время не связаны с контекстом персистентности.
- Удалённые объекты сохранены в хранилище, связаны с контекстом персистентности и отмечены для удаления из хранилища.

Сохранение объектов сущности

Новые объекты сущностей становятся управляемыми и сохраняются в хранилище либо путём вызова метода `persist`, либо с помощью каскадной операции `persist`, вызываемой из связанных сущностей, которые имеют элементы `cascade=PERSIST` или `cascade=ALL`, установленные в аннотациях отношений. Это означает, что данные объекта сохраняются в базе данных, когда транзакция, связанная с операцией `persist`, завершена. Если объект уже управляется, операция `persist` игнорируется, хотя операция `persist` будет каскадно связана со связанными объектами, у которых элемент `cascade` установлен в `PERSIST` или `ALL` в аннотации отношений. Если `persist` вызывается для удалённого объекта, объект становится управляемым. Если объект отсоединён, тогда либо выполнение `persist` выбросит исключение `IllegalArgumentException`, либо фиксация транзакции завершится неудачно. Следующий метод выполняет операцию `persist`:

JAVA

```
@PersistenceContext
EntityManager em;
...
public LineItem createLineItem(CustomerOrder order, Product product,
        int quantity) {
    LineItem li = new LineItem(order, product, quantity);
    order.getLineItems().add(li);
    em.persist(li);
    return li;
}
```

Операция `persist` распространяется на все объекты, связанные с вызывающим объектом, у которых для элемента `cascade` установлено значение `ALL` или `PERSIST` в аннотации отношений:

JAVA

```
@OneToMany(cascade=ALL, mappedBy="order")
public Collection<LineItem> getLineItems() {
    return lineItems;
}
```

Удаление объектов сущности

Управляемые объекты сущностей удаляются путём вызова метода `remove` или каскадной операции `remove`, вызываемой из связанных сущностей, которые имеют `cascade=REMOVE` или `cascade=ALL` в аннотации отношений. Если метод `remove` вызывается для нового объекта, операция `remove` игнорируется, хотя `remove` будет каскадно переходить к связанным объектам, которые имеют элемент `cascade`, установленный в значение `REMOVE` или `ALL` в аннотации отношения. Если `remove` вызывается для отсоединённого объекта, то либо `remove` сгенерирует `IllegalArgumentException`, либо фиксация транзакции завершится неудачно. Если вызывается для уже удалённого объекта, `remove` будет игнорироваться. Данные сущности будут удалены из хранилища после завершения транзакции или в результате операции `flush`.

В следующем примере все объекты `LineItem`, связанные с заказом, также удаляются, так как `CustomerOrder.getLineItems` имеет `cascade=ALL`, установленный в аннотации отношения:

JAVA

```
public void removeOrder(Integer orderId) {
    try {
        CustomerOrder order = em.find(CustomerOrder.class, orderId);
        em.remove(order);
    }...
}
```

Синхронизация данных объекта сущности с базой данных

Состояние сохранённых объектов синхронизируется с базой данных, когда транзакция, с которой связан этот объект, фиксируется. Если управляемый объект находится в двунаправленном отношении с другим управляемым объектом, данные будут сохранены в зависимости от стороны-владельца отношения.

Чтобы принудительно синхронизировать управляемый объект с хранилищем данных, вызовите метод `flush` объекта `EntityManager`. Если объект связан с другим объектом, а аннотация отношения имеет элемент `cascade`, установленный в `PERSIST` или `ALL`, данные связанного объекта будут синхронизированы с хранилищем данных при вызове `flush`.

Если объект удаляется, вызов `flush` удалит данные объекта из хранилища данных.

Юниты персистентности

Юнит персистентности определяет набор всех классов сущностей, которые управляются объектами `EntityManager` в приложении. Этот набор классов сущностей представляет данные, содержащиеся в одном хранилище данных.

Юниты персистентности определяются в файле конфигурации `persistence.xml`. Ниже приведён пример файла `persistence.xml`:

```xml
<persistence>
    <persistence-unit name="OrderManagement">
        <description>This unit manages orders and customers.
            It does not rely on any vendor-specific features and can
            therefore be deployed to any persistence provider.
        </description>
        <jta-data-source>jdbc/MyOrderDB</jta-data-source>
        <jar-file>MyOrderApp.jar</jar-file>
        <class>com.widgets.CustomerOrder</class>
        <class>com.widgets.Customer</class>
    </persistence-unit>
</persistence>
```

Этот файл определяет юнит персистентности с именем `OrderManagement`, который использует источник данных `jdbc/MyOrderDB`, поддерживающий транзакции Jakarta. Элементы `jar-file` и `class` определяют управляемые классы: классы сущностей, встраиваемые классы и сопоставленные родительские классы. Элемент `jar-file` определяет JAR-файлы, которые видны упакованному юниту персистентности, которые содержат управляемые классы сущностей, тогда как элементы `class` перечисляют управляемые классы сущностей явным образом.

Элементы `jta-data-source` (для источников данных, поддерживающих Jakarta Transactions) и `non-jta-data-source` (для источников данных, не поддерживающих Jakarta Transactions), определяют глобальное имя источника данных JNDI, который будет использоваться контейнером.

JAR-файл или каталог, в каталоге `META-INF` которого содержится `persistence.xml`, называется корнем юнита персистентности. Область видимости юнита персистентности определяется корнем юнита персистентности. Каждый юнит персистентности идентифицируется по имени, уникальному для области видимости юнита персистентности.

Юниты персистентности могут быть упакованы как часть WAR-файла или EJB-JAR или могут быть упакованы в JAR-файл, который затем может быть включён в файл WAR или EAR.

- Если вы упаковываете юнит персистентности как набор классов в JAR-файл EJB, `persistence.xml` следует поместить в каталог `META-INF` EJB JAR.
- Если вы упакуете юнит персистентности как набор классов в файле WAR, файл `persistence.xml` должен находиться в каталоге `WEB-INF/classes/META-INF` файла WAR.
- Если вы упакуете юнит персистентности в файл JAR, который будет включён в WAR-файл или EAR, файл JAR должен находиться в одном из следующих каталогов
    - Каталог `WEB-INF/lib` WAR
    - Каталог одной из библиотек EAR-файла

> В Java Persistence API 1.0 файлы JAR могут находиться в корне файла EAR в качестве корня юнита персистентности. Это больше не поддерживается. Переносимые приложения должны использовать каталог библиотеки EAR-файла в качестве корня юнита персистентности.

Выборка объектов сущностей

Jakarta Persistence предоставляет следующие методы для запроса сущностей.

- Язык запросов Jakarta Persistence (Jakarta Persistence query language — JPQL) — это язык на основе строк, похожий на SQL, который используется для выборки объектов сущностей и их отношений. См. главу 42 *Язык запросов Jakarta Persistence* для получения дополнительной информации.
- Criteria API используется для создания типобезопасных (typesafe) запросов с использованием API Java для выборки объектов сущностей и их отношений. См. главу 43 *Использование Criteria API для создания запросов* для получения дополнительной информации.

Как JPQL, так и Criteria API имеют свои преимущества и недостатки.

JPQL-запросы длиной всего несколько строк обычно более лаконичны и более читабельны, чем запросы Criteria. Разработчикам, знакомым с SQL, будет легко понять синтаксис JPQL. Именованные запросы JPQL могут быть определены в классе сущности с помощью аннотации Java или в дескрипторе развёртывания приложения. Однако запросы JPQL не являются типобезопасными (typesafe) и требуют преобразования при получении результата запроса от entity manager-а. Это означает, что ошибки приведения типов не могут быть обнаружены во время компиляции. JPQL-запросы не поддерживают открытые (open-ended) параметры.

Запросы Criteria позволяют определить запрос в слое бизнес-логики приложения. Хотя это также возможно при использовании динамических запросов JPQL, запросы Criteria обеспечивают более высокую производительность, поскольку динамические запросы JPQL должны анализироваться при каждом вызове. Запросы Criteria являются типобезопасными и поэтому не требуют приведения, как это необходимо делать с запросами JPQL. Criteria API — это ещё один API Java, и он не требует от разработчиков изучения ещё одного синтаксиса языка запросов. Запросы Criteria, как правило, более подробны, чем запросы JPQL, и требуют от разработчика создания нескольких объектов и выполнения операций над этими объектами перед отправкой запроса entity manager-у.

Создание схемы базы данных

Persistence provider может быть настроен на автоматическое создание таблиц базы данных, загрузку данных в таблицы и удаление таблиц во время развёртывания приложения с использованием стандартных свойств в дескрипторе развёртывания приложения. Эти задачи обычно используются на этапе разработки, но не для рабочей базы данных.

Ниже приведён пример дескриптора развёртывания `persistence.xml`, который предписывает persistence provider-у удалить все артефакты, создать новые и загрузить данные из предоставленного сценария при развёртывании приложения:

XML

```xml
<persistence version="3.0" xmlns="https://jakarta.ee/xml/ns/persistence"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
 https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd">
  <persistence-unit name="examplePU" transaction-type="JTA">
    <jta-data-source>java:global/ExampleDataSource</jta-data-source>
    <properties>
        <property name="jakarta.persistence.schema-generation.database.action"
                  value="drop-and-create"/>
        <property name="jakarta.persistence.schema-generation.create-source"
                  value="script"/>
        <property name="jakarta.persistence.schema-generation.create-script-source"
                  value="META-INF/sql/create.sql" />
        <property name="jakarta.persistence.sql-load-script-source"
                  value="META-INF/sql/data.sql" />
        <property name="jakarta.persistence.schema-generation.drop-source"
                  value="script" />
        <property name="jakarta.persistence.schema-generation.drop-script-source"
                  value="META-INF/sql/drop.sql" />
    </properties>
  </persistence-unit>
</persistence>
```

Настройка приложения для создания или удаления таблиц базы данных

Свойство `jakarta.persistence.schema-generation.database.action` используется для указания действия, предпринимаемого persistence provider-ом при развёртывании приложения. Если свойство не установлено, persistence provider не будет создавать или удалять артефакты базы данных.

Таблица 40-3 Действия по созданию схемы

Значение настройки	**Описание**
`none`	Создание и удаление схемы не производится.
`создание`	Persistence provider создаст артефакты базы данных при развёртывании приложения. Артефакты останутся неизменными после повторного развёртывания приложения.
`drop-and-create`	Все артефакты в базе данных будут удалены, и persistence provider создаст артефакты базы данных при развёртывании.
`drop`	Все артефакты в базе данных будут удалены при развёртывании приложения.

В этом примере persistence provider удалит все оставшиеся артефакты базы данных, а затем создаст артефакты при развёртывании приложения:

XML

```xml
<property name="jakarta.persistence.schema-generation.database.action"
          value="drop-and-create"/>
```

По умолчанию объектно-реляционные метаданные в юните персистентности используются для создания артефактов базы данных. Вы также можете предоставить сценарии, используемые provider-ом для создания и удаления артефактов базы данных. Свойства `jakarta.persistence.schema-generation.create-source` и

`jakarta.persistence.schema-generation.drop-source` управляют тем, как провайдер будет создавать и удалять артефакты базы данных.

Таблица 40-4. Настройки для создания и удаления свойств источника

Значение настройки	Описание
`metadata`	Используйте объектно-реляционные метаданные в приложении для создания или удаления артефактов базы данных.
`script`	Укажите скрипт для создания или удаления артефактов базы данных.
`metadata-then-script`	Используйте комбинацию объектно-реляционных метаданных, а затем предоставленный сценарий для создания или удаления артефактов базы данных.
`script-then-metadata`	Используйте комбинацию предоставленного сценария, а затем метаданные объектно-реляционных метаданных для создания и удаления артефактов базы данных.

В этом примере persistence provider будет использовать сценарий, упакованный в приложении, для создания артефактов базы данных:

```xml
<property name="jakarta.persistence.schema-generation.create-source"
          value="script"/>
```

При указании `create-source` или `drop-source` укажите расположение скрипта с помощью свойствв `jakarta.persistence.schema-geneteration.create-script-source` или `jakarta.persistence.schema-generation.drop script-source`. Расположение скрипта относительно корня юнита персистентности:

```xml
<property name="jakarta.persistence.schema-generation.create-script-source"
          value="META-INF/sql/create.sql" />
```

В приведённом выше примере для `create-script-source` задан файл SQL с именем `create.sql` в каталоге `META-INF/sql` относительно корня юнита персистентности.

Загрузка данных с использованием сценариев SQL

Если требуется заполнить таблицы базы данных данными перед загрузкой приложения, укажите расположение скрипта загрузки в свойстве `jakarta.persistence.sql-load-script-source`. Местоположение, указанное в этом свойстве, указывает относительный путь от юнита персистентности.

В этом примере сценарий загрузки представляет собой файл `data.sql` в каталоге `META-INF/sql` относительно корня юнита персистентности:

```xml
<property name="jakarta.persistence.sql-load-script-source"
          value="META-INF/sql/data.sql" />
```

Дополнительная информация о персистентности

Дополнительные сведения о Jakarta Persistence см.

- Спецификация Jakarta Persistence 3.0 API:
  https://jakarta.ee/specifications/persistence/3.0/
- EclipseLink, реализация Jakarta Persistence в GlassFish Server:
  https://www.eclipse.org/eclipselink/
- Вики-документация EclipseLink:
  https://wiki.eclipse.org/EclipseLink

Глава 41. Запуск примеров персистентности

В этой главе объясняется, как использовать Jakarta Persistence. Материал здесь сосредоточен на исходном коде и настройках трёх примеров.

Обзор примеров персистентности

Первый пример `order` представляет собой приложение, которое использует сессионный компонент с сохранением состояния для управления объектами, связанными с системой заказов. Второй пример `roster` представляет собой приложение, которое управляет спортивной системой сообщества. Третий пример `address-book` представляет собой веб-приложение, в котором хранятся контактные данные. В этой главе предполагается, что вы знакомы с концепциями, описанными в разделе Введение в Jakarta Persistence.

Приложение order

Приложение `order` представляет собой простое приложение для инвентаризации и заказа товаров для ведения каталога деталей и размещения подробного заказа этих деталей. В приложении есть сущности, которые представляют детали, поставщиков, заказы и их позиции. Доступ к этим сущностям осуществляется с помощью сессионного компонента с сохранением состояния, который содержит бизнес-логику приложения. Простой сессионный компонент-синглтон инициализирует сущности при развёртывании приложения. Веб-приложение Facelets управляет данными и отображает содержимое каталога.

Информация, содержащаяся в заказе, может быть разделена на элементы. Какой номер заказа? Какие детали включены в заказ? Из каких деталей состоит эта деталь? Кто производит деталь? Каковы технические характеристики детали? Есть ли какие-либо схемы для детали? Приложение `order` является упрощённой версией системы заказов, которая имеет ответы на все эти вопросы.

Приложение `order` состоит из одного модуля WAR, который включает в себя классы Enterprise-бинов, сущностей, вспомогательные классы, а также XHTML Facelets.

Схема базы данных в Derby для `order` показана на рис. 41-1.

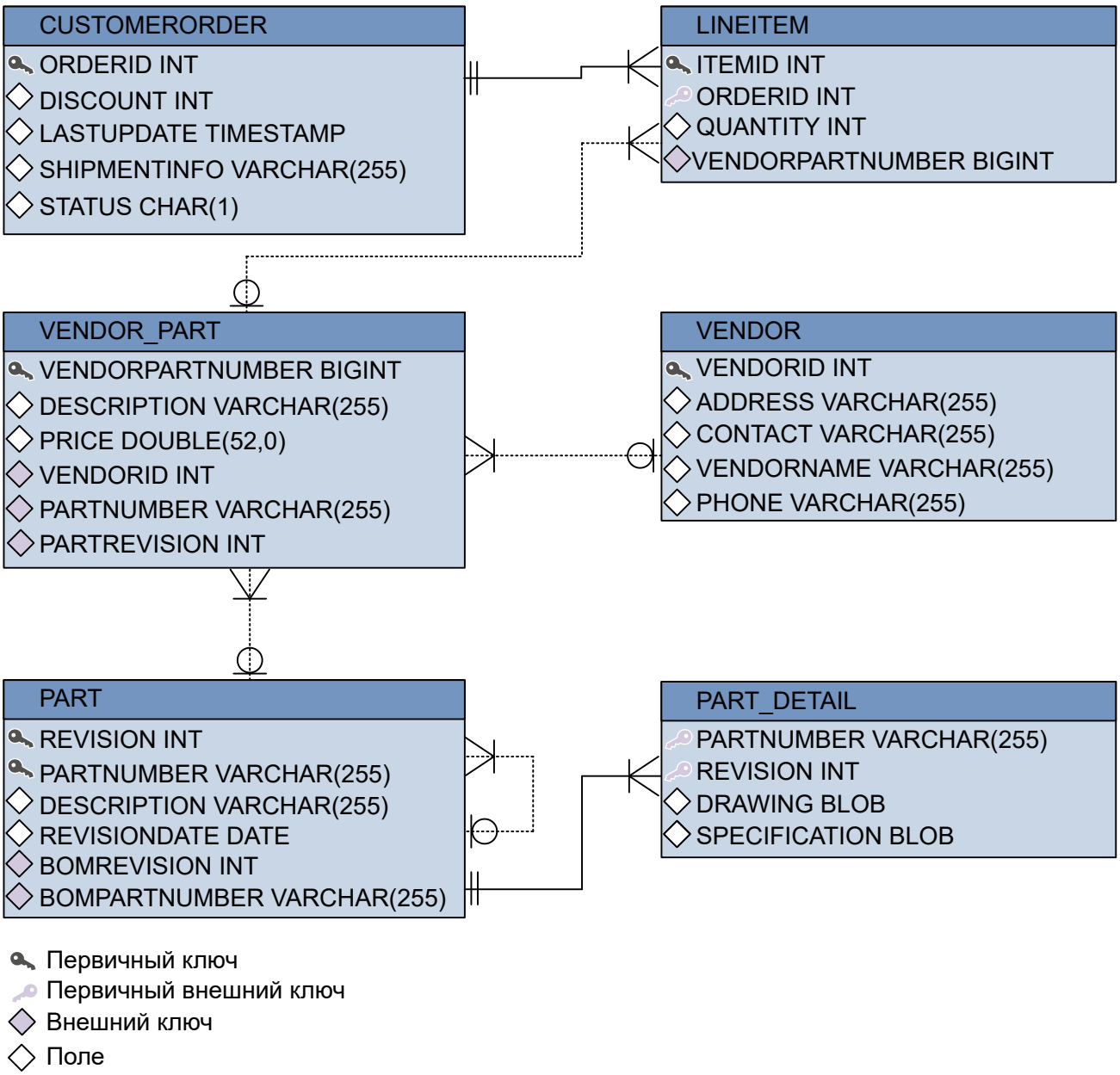

Рис. 41-1. Схема базы данных для приложения order

В этой диаграмме для простоты префикс `PERSISTENCE_ORDER_` опущен в именах таблиц.

Взаимоотношения сущностей в приложении order

Приложение `order` демонстрирует несколько типов отношений сущностей: ссылки на себя, один-к-одному, один-ко-многим, многие-к-одному и однонаправленные.

Отношения ссылки на себя

Относящиеся ссылки на себя возникают между полями в одной и той же сущности. `Part` имеет поле `bomPart`, которое имеет отношение один-ко-многим с полем `parts`, которое также находится в `Part`. То есть сложная деталь может состоять из множества составляющих её более простых деталей.

Первичный ключ для `Part` является составным первичным ключом, комбинацией полей `partNumber` и `revision`. Этот ключ сопоставляется со столбцами `PARTNUMBER` и `REVISION` в таблице `PERSISTENCE_ORDER_PART`:

JAVA

```
...
@ManyToOne
@JoinColumns({
    @JoinColumn(name="BOMPARTNUMBER", referencedColumnName="PARTNUMBER"),
    @JoinColumn(name="BOMREVISION", referencedColumnName="REVISION")
})
public Part getBomPart() {
    return bomPart;
}
...
@OneToMany(mappedBy="bomPart")
public Collection<Part> getParts() {
    return parts;
}
...
```

Отношения один-к-одному

`Part` имеет поле `vendorPart`, которое связано отношением один-к-одному с полем `part` у `VendorPart`. То есть каждая деталь имеет ровно одну соответствующую ей деталь поставщика, и наоборот.

Вот отображение отношения в `Part`:

JAVA

```
@OneToOne(mappedBy="part")
public VendorPart getVendorPart() {
    return vendorPart;
}
```

Вот отображение отношения в `VendorPart`:

JAVA

```
@OneToOne
@JoinColumns({
    @JoinColumn(name="PARTNUMBER", referencedColumnName="PARTNUMBER"),
    @JoinColumn(name="PARTREVISION", referencedColumnName="REVISION")
})
public Part getPart() {
    return part;
}
```

Обратите внимание: поскольку `Part` использует составной первичный ключ, аннотация `@JoinColumns` используется для сопоставления столбцов в таблице `PERSISTENCE_ORDER_VENDOR_PART` со столбцами в `PERSISTENCE_ORDER_PART`. Столбец `PERSISTENCE_ORDER_VENDOR_PART` таблицы `PARTREVISION` ссылается на столбец `PERSISTENCE_ORDER_PART` таблицы `REVISION`.

Отношения один-ко-многим, сопоставленные с перекрывающимися первичными и внешними ключами

`CustomerOrder` имеет поле `lineItems`, которое связано отношением один-ко-многим с полем `LineItem` из `customerOrder`. То есть каждый заказ имеет одну или несколько позиций.

`LineItem` использует составной первичный ключ, который состоит из полей `orderId` и `itemId`. Этот составной первичный ключ сопоставляется со столбцами `ORDERID` и `ITEMID` в таблице `PERSISTENCE_ORDER_LINEITEM`. `ORDERID` — это внешний ключ для столбца `ORDERID` в таблице `PERSISTENCE_ORDER_CUSTOMERORDER`. Это означает, что столбец `ORDERID` отображается дважды: один раз как поле первичного ключа `orderId` и второй раз в качестве поля отношения `order`.

Вот отображение отношений в `CustomerOrder`:

JAVA

```
@OneToMany(cascade=ALL, mappedBy="customerOrder")
public Collection<LineItem> getLineItems() {
    return lineItems;
}
```

Вот отображение отношений в `LineItem`:

JAVA

```
@Id
@ManyToOne
@JoinColumn(name="ORDERID")
public CustomerOrder getCustomerOrder() {
    return customerOrder;
}
```

Однонаправленные отношения

`LineItem` имеет поле `vendorPart`, которое связано однонаправленным отношением многие-к-одному с `VendorPart`. То есть в целевой сущности нет поля для этой связи:

JAVA

```
@JoinColumn(name="VENDORPARTNUMBER")
@ManyToOne
public VendorPart getVendorPart() {
    return vendorPart;
}
```

Первичные ключи в order

Приложение `order` использует несколько типов первичных ключей: простые (однозначные) первичные ключи, сгенерированные первичные ключи и составные первичные ключи.

Сгенерированные первичные ключи

`VendorPart` использует сгенерированное значение первичного ключа. Таким образом, приложение не устанавливает значение первичного ключа для объектов сущности, а вместо этого полагается на persistence provider для генерации значений первичного ключа. Аннотация `@GeneratedValue` используется для указания того, что объекты сущности будут использовать сгенерированный первичный ключ.

В `VendorPart` следующий код задаёт параметры для генерации значений первичного ключа:

JAVA

```
@TableGenerator(
    name="vendorPartGen",
    table="PERSISTENCE_ORDER_SEQUENCE_GENERATOR",
    pkColumnName="GEN_KEY",
    valueColumnName="GEN_VALUE",
    pkColumnValue="VENDOR_PART_ID",
    allocationSize=10)
@Id
@GeneratedValue(strategy=GenerationType.TABLE, generator="vendorPartGen")
public Long getVendorPartNumber() {
    return vendorPartNumber;
}
```

Аннотация `@TableGenerator` используется вместе с элементом `@GeneratedValue ытrategy=TABLE`. Таким образом, стратегия, используемая для генерации первичных ключей, заключается в использовании таблицы в базе данных. Аннотация `@TableGenerator` используется для настройки параметров таблицы генератора. Элемент name устанавливает имя генератора. Для `VendorPart` это `vendorPartGen`.

Таблица `PERSISTENCE_ORDER_SEQUENCE_GENERATOR`, двумя столбцами которой являются `GEN_KEY` и `GEN_VALUE`, будет хранить сгенерированные значения первичного ключа. Эту таблицу можно использовать для генерации первичных ключей других сущностей, поэтому элемент `pkColumnValue` установлен в `VENDOR_PART_ID`, чтобы отличать сгенерированные первичные ключи этой сущности от сгенерированных первичных ключей других сущностей. Элемент `allocSize` указывает величину, которую нужно увеличить при выделении значений первичного ключа. В этом случае каждый первичный ключ `VendorPart` будет увеличиваться на 10.

Поле первичного ключа `vendorPartNumber` имеет тип `Long`, так как поле сгенерированного первичного ключа должно быть целочисленным.

Составные первичные ключи

Составной первичный ключ состоит из нескольких полей и соответствует требованиям, описанным в Первичные ключи в сущностях. Чтобы использовать составной первичный ключ, вы должны создать класс-обёртку (wrapper).

В `order` две сущности используют составные первичные ключи: `Part` и `LineItem`.

- `Part` использует класс-обёртку (wrapper) `PartKey`. Первичный ключ `Part` представляет собой комбинацию номера детали и номера редакции. `PartKey` инкапсулирует этот первичный ключ.
- `LineItem` использует класс `LineItemKey`. Первичный ключ `LineItem` представляет собой комбинацию номера заказа и номера позиции. `LineItemKey` инкапсулирует этот первичный ключ.

Это класс-обёртка (wrapper) `LineItemKey` составного первичного ключа:

JAVA

```java
package ee.jakarta.tutorial.order.entity;

import java.io.Serializable;

public final class LineItemKey implements Serializable {

    private Integer customerOrder;
    private int itemId;

    public LineItemKey() {}

    public LineItemKey(Integer order, int itemId) {
        this.setCustomerOrder(order);
        this.setItemId(itemId);
    }

    @Override
    public int hashCode() {
        return ((this.getCustomerOrder() == null
                ? 0 : this.getCustomerOrder().hashCode())
                ^ ((int) this.getItemId()));
    }

    @Override
    public boolean equals(Object otherOb) {
        if (this == otherOb) {
            return true;
        }
        if (!(otherOb instanceof LineItemKey)) {
            return false;
        }
        LineItemKey other = (LineItemKey) otherOb;
        return ((this.getCustomerOrder() == null
                ? other.getCustomerOrder == null : this.getOrderId()
                .equals(other.getCustomerOrder()))
                && (this.getItemId == oother.getItemId()));
    }

    @Override
    public String toString() {
        return "" + getCustomerOrder() + "-" + getItemId();
    }

    public Integer getCustomerOrder() {
        return customerOrder;
    }

    public void setCustomerOrder(Integer order) {
        this.customerOrder = order;
    }

    public int getItemId() {
        return itemId;
    }

    public void setItemId(int itemId) {
        this.itemId = itemId;
    }
}
```

Аннотация `@IdClass` используется для указания класса первичного ключа в классе сущности. В `LineItem` `@IdClass` используется следующим образом:

JAVA

```
@IdClass(LineItemKey.class)
@Entity
...
public class LineItem implements Serializable {
    ...
}
```

Два поля в `LineItem` помечены аннотацией `@Id`, чтобы пометить эти поля как часть составного первичного ключа:

JAVA

```
@Id
public int getItemId() {
    return itemId;
}
...
@Id
@ManyToOne
@JoinColumn(name="ORDERID")
public CustomerOrder getCustomerOrder() {
    return customerOrder;
}
```

Для `customerOrder` вы также используете аннотацию `@JoinColumn` для указания имени столбца в таблице и того, что этот столбец является перекрывающимся внешним ключом, указывающим на `PERSISTENCE_ORDER_CUSTOMERORDER` столбец `ORDERID` таблицы (см. отношение «один ко многим», сопоставленное перекрывающимся первичным и внешним ключам). То есть `customerOrder` будет установлен сущностью `CustomerOrder`.

В конструкторе `LineItem` номер позиции (`LineItem.itemId`) устанавливается с помощью метода `CustomerOrder.getNextId`:

JAVA

```
public LineItem(CustomerOrder order, int quantity, VendorPart vendorPart) {
    this.customerOrder = order;
    this.itemId = order.getNextId();
    this.quantity = quantity;
    this.vendorPart = vendorPart;
}
```

`CustomerOrder.getNextId` подсчитывает количество текущих позиций, добавляет 1 и возвращает это число:

JAVA

```
@Transient
public int getNextId() {
    return this.lineItems.size() + 1;
}
```

`Part` требует аннотации `@Column` для двух полей, которые образуют составной первичный ключ `Part`, поскольку составной первичный ключ `Part` является перекрывающимся первичным и внешним ключом:

JAVA

```java
@IdClass(PartKey.class)
@Entity
...
public class Part implements Serializable {
    ...
    @Id
    @Column(nullable=false)
    public String getPartNumber() {
        return partNumber;
    }
    ...
    @Id
    @Column(nullable=false)
    public int getRevision() {
        return revision;
    }
    ...
}
```

Сущности соответствует более чем одна таблица базы данных

Поля `Part` соответствуют нескольким таблицам базы данных: `PERSISTENCE_ORDER_PART` и `PERSISTENCE_ORDER_PART_DETAIL`. Таблица `PERSISTENCE_ORDER_PART_DETAIL` содержит спецификацию и схемы для детали. Аннотация `@SecondaryTable` используется для указания дополнительной таблицы:

JAVA

```java
...
@Entity
@Table(name="PERSISTENCE_ORDER_PART")
@SecondaryTable(name="PERSISTENCE_ORDER_PART_DETAIL", pkJoinColumns={
    @PrimaryKeyJoinColumn(name="PARTNUMBER",
        referencedColumnName="PARTNUMBER"),
    @PrimaryKeyJoinColumn(name="REVISION",
        referencedColumnName="REVISION")
})
public class Part implements Serializable {
    ...
}
```

`PERSISTENCE_ORDER_PART_DETAIL` и `PERSISTENCE_ORDER_PART` используют одни и те же значения первичного ключа. Элемент `pkJoinColumns` в `@SecondaryTable` используется для указания того, что столбцы первичного ключа `PERSISTENCE_ORDER_PART_DETAIL` являются внешними ключами для `PERSISTENCE_ORDER_PART`, Аннотация `@PrimaryKeyJoinColumn` задаёт имена столбцов первичного ключа и указывает, к какому столбцу в первичной таблице относится этот столбец. В этом случае имена столбцов первичного ключа для `PERSISTENCE_ORDER_PART_DETAIL` и `PERSISTENCE_ORDER_PART` совпадают: `PARTNUMBER` и `REVISION`, соответственно.

Каскадные операции в order

Объекты сущностей, связанные отношениями с объектами других сущностей, часто зависят от существования этого другого объекта отношений. Например, если позиция является частью заказа и если заказ удаляется, то позиция также должна быть удалена. Это называется отношением каскадного удаления.

В `order` в отношениях сущностей есть две зависимости каскадного удаления. Если `CustomerOrder`, с которым связан `LineItem`, удалён, `LineItem` также следует удалить. Если `Vendor`, с которым связан `VendorPart`, удалён, `VendorPart` также следует удалить.

Вы указываете каскадные операции для отношений сущностей, устанавливая элемент `cascade` на обратной стороне (не стороне-владельце) отношения. Каскадный элемент имеет значение `ALL` в случае `CustomerOrder.lineItems`. Это означает, что все персистентные операции (удаление, обновление и т. д.) каскадируются от заказов к позициям.

Вот отображение отношений в `CustomerOrder`:

JAVA
```
@OneToMany(cascade=ALL, mappedBy="customerOrder")
public Collection<LineItem> getLineItems() {
    return lineItems;
}
```

Вот отображение отношений в `LineItem`:

JAVA
```
@Id
@ManyToOne
@JoinColumn(name="ORDERID")
public CustomerOrder getCustomerOrder() {
    return customerOrder;
}
```

Типы данных BLOB и CLOB в order

Таблица `PARTDETAIL` в базе данных содержит столбец `DRAWING` типа `BLOB`. `BLOB` обозначает большие бинарные объекты, которые используются для хранения бинарных данных, таких как изображение. Столбец `DRAWING` отображается в поле `Part.drawing` типа `java.io.Serializable`. Аннотация `@Lob` используется для обозначения того, что поле является большим объектом:

JAVA
```
@Column(table="PERSISTENCE_ORDER_PART_DETAIL")
@Lob
public Serializable getDrawing() {
    return drawing;
}
```

`PERSISTENCE_ORDER_PART_DETAIL` также содержит столбец `SPECIFICATION` типа `CLOB`. `CLOB` обозначает большие символьные объекты, которые используются для хранения строковых данных, слишком больших для хранения в столбце `VARCHAR`. `SPECIFICATION` отображается в поле `Part.specification` типа `java.lang.String`. Аннотация `@Lob` также используется здесь для обозначения того, что поле является большим объектом:

JAVA
```
@Column(table="PERSISTENCE_ORDER_PART_DETAIL")
@Lob
public String getSpecification() {
    return specification;
}
```

Оба эти поля используют аннотацию `@Column` и задают элемент `table` для вторичной таблицы.

Временны́е типы в order

Персистентное свойство `CustomerOrder.lastUpdate` имеет тип `java.util.Date`, отображается в поле базы данных `PERSISTENCE_ORDER_CUSTOMERORDER.LASTUPDATE`, которое имеет тип SQL `TIMESTAMP`. Чтобы обеспечить правильное сопоставление между этими типами, необходимо использовать аннотацию `@Temporal` с правильным временны́м типом, указанным в элементе `@Temporal`. Элементы `@Temporal` имеют тип `jakarta.persistence.TemporalType`. Возможные значения

- `DATE`, который отображается на `java.sql.Date`
- `TIME`, который отображается на `java.sql.Time`
- `TIMESTAMP`, который отображается на `java.sql.Timestamp`

Вот соответствующий раздел `CustomerOrder`:

```java
@Temporal(TIMESTAMP)
public Date getLastUpdate() {
    return lastUpdate;
}
```

Управление сущностями в приложении order

Сессионный компонент с состоянием `RequestBean` содержит бизнес-логику и управляет объектами `order`. `RequestBean` использует аннотацию `@PersistenceContext` для извлечения объекта entity manager-а, который используется для управления сущностями `order` в бизнес-методах `RequestBean`:

```java
@PersistenceContext
private EntityManager em;
```

Этот объект `EntityManager` управляется контейнером, поэтому контейнер заботится обо всех транзакциях, связанных с управлением сущностями `order`.

Создание сущностей

Бизнес-метод `RequestBean.createPart` создаёт новый объект сущности `Part`. Метод `EntityManager.persist` используется для сохранения вновь созданного объекта в базе данных:

```java
Part part = new Part(partNumber,
        revision,
        description,
        revisionDate,
        specification,
        drawing);
em.persist(part);
```

Сессионный компонент-синглтон `ConfigBean` используется для инициализации данных в `order`. `ConfigBean` помечен тегом `@Startup`, который указывает, что EJB-контейнер должен создавать `ConfigBean` при развёртывании `order`. Метод `createData` аннотируется `@PostConstruct` и создаёт начальные сущности, используемые `order`, вызывая бизнес-методы `RequestBean`.

Выборка сущностей

Бизнес-метод `RequestBean.getOrderPrice` возвращает цену заказа по его `orderId`. Метод `EntityManager.find` используется для извлечения сущности из базы данных:

```java
CustomerOrder order = em.find(CustomerOrder.class, orderId);
```

Первый аргумент `EntityManager.find` является классом сущности, а второй — первичным ключом.

Установка отношений между сущностями

Бизнес-метод `RequestBean.createVendorPart` создаёт `VendorPart`, связанный с конкретным объектом `Vendor`. Метод `EntityManager.persist` используется для сохранения вновь созданного объекта `VendorPart` в базе данных, а методы `VendorPart.setVendor` и `Vendor.Vendor.setVendorPart` используются для связи `VendorPart` с `Vendor`:

```java
PartKey pkey = new PartKey();
pkey.setPartNumber(partNumber);
pkey.setRevision(revision);

Part part = em.find(Part.class, pkey);

VendorPart vendorPart = new VendorPart(description, price, part);
em.persist(vendorPart);

Vendor vendor = em.find(Vendor.class, vendorId);
vendor.addVendorPart(vendorPart);
vendorPart.setVendor(vendor);
```

Использование запросов

Бизнес-метод `RequestBean.adjustOrderDiscount` обновляет скидку, применяемую ко всем заказам. Этот метод использует именованный запрос `findAllOrders`, определённый в `CustomerOrder`:

```java
@NamedQuery(
    name="findAllOrders",
    query="SELECT co FROM CustomerOrder co " +
          "ORDER BY co.orderId"
)
```

Для выполнения запроса используется метод `EntityManager.createNamedQuery`. Поскольку запрос возвращает `List` всех заказов, используется метод `Query.getResultList`:

```java
List orders = em.createNamedQuery(
        "findAllOrders")
        .getResultList();
```

Бизнес-метод `RequestBean.getTotalPricePerVendor` возвращает общую стоимость всех деталей для конкретного поставщика. Этот метод использует именованный параметр `id`, определённый в именованном запросе `findTotalVendorPartPricePerVendor`, определённом в `VendorPart`:

```java
@NamedQuery(
    name="findTotalVendorPartPricePerVendor",
    query="SELECT SUM(vp.price) " +
    "FROM VendorPart vp " +
    "WHERE vp.vendor.vendorId = :id"
)
```

При выполнении запроса метод `Query.setParameter` используется для установки именованному параметру `id` значения `vendorId` для параметра `RequestBean.getTotalPricePerVendor`:

```java
return (Double) em.createNamedQuery(
    "findTotalVendorPartPricePerVendor")
    .setParameter("id", vendorId)
    .getSingleResult();
```

Метод `Query.getSingleResult` используется для этого запроса, поскольку запрос возвращает одно значение.

Удаление сущностей

Бизнес-метод `RequestBean.removeOrder` удаляет данный заказ из базы данных. Этот метод использует метод `EntityManager.remove` для удаления объекта из базы данных:

JAVA

```
CustomerOrder order = em.find(CustomerOrder.class, orderId);
em.remove(order);
```

Запуск order

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска приложения `order`. Сначала вы создадите таблицы базы данных в Apache Derby.

Запуск order с использованием IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. Если сервер базы данных ещё не запущен, запустите его, следуя инструкциям в Запуск и остановка Apache Derby.
3. В меню **Файл** выберите **Открыть проект**.
4. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/persistence
   ```

5. Выберите каталог `order`.
6. Нажмите **Открыть проект**.
7. На вкладке **Проекты** кликните правой кнопкой мыши проект `order` и выберите **Запуск**.

   Среда IDE NetBeans открывает веб-браузер по следующему URL:

   ```
   http://localhost:8080/order/
   ```

Запуск order с помощью Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. Если сервер базы данных ещё не запущен, запустите его, следуя инструкциям в Запуск и остановка Apache Derby.
3. В окне терминала перейдите в:

   ```
   tut-install/examples/persistence/order/
   ```

4. Введите следующую команду:

   SHELL

   ```
   mvn install
   ```

   Компиляция исходных файлов и пакетов выполняется в WAR-файл, расположенный в *`tut-install`*`/examples/persistence/order/target/order.war`. Затем WAR-файл будет развёрнут в GlassFish Server.
5. Чтобы создать и обновить данные заказа, откройте веб-браузер по следующему URL:

   ```
   http://localhost:8080/order/
   ```

Приложение roster

Приложение `roster` поддерживает списки команд для игроков в спортивных лигах. Приложение состоит из четырёх компонентов: сущностей Jakarta Persistence (`Player` , `Team` и `League`), сессионного компонента с состоянием (`RequestBean`), клиента приложения (`RosterClient`) и трёх вспомогательных классов (`PlayerDetails` , `TeamDetails` и `LeagueDetails`).

Функционально `roster` аналогичен приложению `order` с тремя новыми функциями: отношение «многие ко многим», наследование сущностей и автоматическое создание таблицы при развёртывании.

Схема базы данных в Apache Derby для приложения `roster` показана на рис. 41-2.

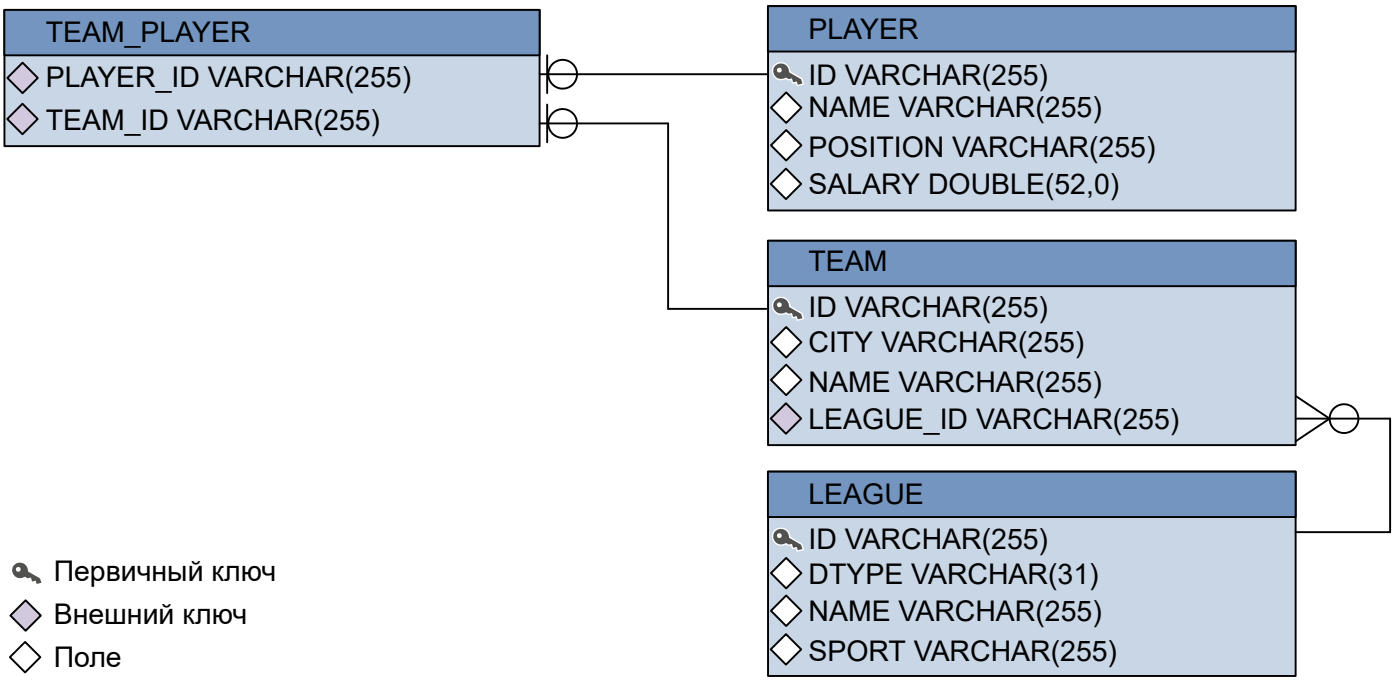

Рис. 41-2. Схема базы данных для приложения roster

В этой диаграмме для простоты префикс `PERSISTENCE_ROSTER_` опущен в именах таблиц.

Отношения в roster

Система оздоровительного спорта имеет следующие отношения.

- Игрок может быть во многих командах.
- В команде может быть много игроков.
- Команда находится в одной лиге.
- В лиге много команд.

В `roster` эта система отражается следующими отношениями между сущностями `Player` , `Team` и `League` .

- Между `Player` и `Team` существует отношение «многие ко многим».
- Между `Team` и `League` существует отношение «многие к одному».

Отношения «многие ко многим» в roster

Отношение «многие ко многим» между `Player` и `Team` определяется с помощью аннотации `@ManyToMany`. В `Team.java` метод `getPlayers` аннотирован `@ManyToMany`:

JAVA

```
@ManyToMany
@JoinTable(
    name="PERSISTENCE_ROSTER_TEAM_PLAYER",
    joinColumns=
        @JoinColumn(name="TEAM_ID", referencedColumnName="ID"),
    inverseJoinColumns=
        @JoinColumn(name="PLAYER_ID", referencedColumnName="ID")
)
public Collection<Player> getPlayers() {
    return players;
}
```

Аннотация `@JoinTable` используется для указания таблицы базы данных, которая будет связывать идентификаторы игроков с идентификаторами команд. Сущность, которая определяет `@JoinTable`, является владельцем связи, поэтому сущность `Team` является владельцем связи с сущностью `Player`. Поскольку `roster` использует автоматическое создание таблицы во время развёртывания, контейнер создаст таблицу соединения с именем `PERSISTENCE_ROSTER_TEAM_PLAYER`.

`Player` — это обратная сторона (не владелец) отношения с `Team`. Как отношения один-к-одному и многие-к-одному, обратная сторона отмечена элементом `mappedBy` в аннотации отношения. Поскольку отношения между `Player` и `Team` являются двунаправленными, выбор объекта-владельца отношения произвольный.

В `Player.java` метод `getTeams` аннотирован `@ManyToMany`:

JAVA

```
@ManyToMany(mappedBy="players")
public Collection<Team> getTeams() {
    return teams;
}
```

Наследование сущностей в roster

Приложение `roster` показывает использование наследования сущностей, как описано в Наследование сущностей.

Сущность `League` в `roster` является абстрактной сущностью с двумя конкретными дочерними классами: `SummerLeague` и `WinterLeague`. Поскольку `League` является абстрактным классом, его объект не может быть создан:

JAVA

```
@Entity
@Table(name = "PERSISTENCE_ROSTER_LEAGUE")
public abstract class League implements Serializable { ... }
```

Вместо этого при создании лиги клиенты используют `SummerLeague` или `WinterLeague`. `SummerLeague` и `WinterLeague` наследуют персистентные свойства, определённые в `League`, и добавляют только конструктор, который проверяет, что спортивный параметр соответствует типу спорта, разрешённому этой сезонной лигой. Например, вот сущность `SummerLeague`:

JAVA

```
...
@Entity
public class SummerLeague extends League implements Serializable {

    /** Creates a new instance of SummerLeague */
    public SummerLeague() {
    }

    public SummerLeague(String id, String name, String sport)
            throws IncorrectSportException {
        this.id = id;
        this.name = name;
        if (sport.equalsIgnoreCase("swimming") ||
                sport.equalsIgnoreCase("soccer") ||
                sport.equalsIgnoreCase("basketball") ||
                sport.equalsIgnoreCase("baseball")) {
            this.sport = sport;
        } else {
            throw new IncorrectSportException("Sport is not a summer sport.");
        }
    }
}
```

Приложение `roster` использует стратегию отображения по умолчанию `InheritanceType.SINGLE_TABLE`, поэтому аннотация `@Inheritance` не требуется. Если вы хотите использовать другую стратегию отображения, укажите для `League` аннотацию `@Inheritance` и укажите стратегию отображения в элементе `strategy`:

JAVA

```
@Entity
@Inheritance(strategy=JOINED)
@Table(name="PERSISTENCE_ROSTER_LEAGUE")
public abstract class League implements Serializable { ... }
```

Приложение `roster` использует для столбца дискриминатора имя по умолчанию, поэтому аннотация `@DiscriminatorColumn` не требуется. Поскольку используется автоматическое создание таблиц в `roster`, Persistence provider создаст столбец дискриминатора с именем `DTYPE` в таблице `PERSISTENCE_ROSTER_LEAGUE`, в котором будут храниться имя унаследованного лица, используемого для создания лиги. При желании использовать другое имя для столбца дискриминатора, укажите для `League` аннотацию `@DiscriminatorColumn` и установите элемент `name`:

JAVA

```
@Entity
@DiscriminatorColumn(name="DISCRIMINATOR")
@Table(name="PERSISTENCE_ROSTER_LEAGUE")
public abstract class League implements Serializable { ... }
```

Запросы Criteria в roster

Приложение `roster` использует запросы Criteria, в отличие от запросов JPQL, используемых в `order`. Запросы Criteria записаны на Java, типобезопасны, определены в слое бизнес-логики приложения `roster` в сессионном компоненте с состоянием `RequestBean`.

Классы метамодели в roster

Классы метамодели моделируют атрибуты объекта и используются запросами Criteria для перехода к атрибутам объекта. Каждый класс сущности в `roster` имеет соответствующий класс метамодели, сгенерированный при компиляции, с тем же именем пакета, что и сущность, и добавленным с символом подчеркивания (`_`). Например, `roster.entity.Player` имеет соответствующий класс метамодели, `roster.entity.Player_`.

Каждое персистентное поле или свойство в классе сущности имеет соответствующий атрибут в классе метамодели сущности. Для сущности `Player` соответствующий класс метамодели выглядит следующим образом:

JAVA

```
@StaticMetamodel(Player.class)
public class Player_ {
    public static volatile SingularAttribute<Player, String> id;
    public static volatile SingularAttribute<Player, String> name;
    public static volatile SingularAttribute<Player, String> position;
    public static volatile SingularAttribute<Player, Double> salary;
    public static volatile CollectionAttribute<Player, Team> teams;
}
```

Получение объекта CriteriaBuilder в RequestBean

Интерфейс `CriteriaBuilder` определяет методы для создания объектов запроса Criteria и создания выражений для изменения этих объектов запроса. `RequestBean` создаёт объект `CriteriaBuilder` с помощью метода `init`, аннотированного `@PostConstruct`:

JAVA

```
@PersistenceContext
private EntityManager em;
private CriteriaBuilder cb;

@PostConstruct
private void init() {
    cb = em.getCriteriaBuilder();
}
```

Объект `EntityManager` инъецируется во время выполнения, а затем этот объект `EntityManager` используется для создания объекта `CriteriaBuilder` путём вызова `getCriteriaBuilder`. Объект `CriteriaBuilder` создаётся в методе `@PostConstruct`, чтобы гарантировать, что объект `EntityManager` был инъецирован EJB-контейнером.

Создание запросов Criteria в бизнес-методах RequestBean

Многие бизнес-методы в `RequestBean` определяют запросы Criteria. Один бизнес-метод, `getPlayersByPosition`, возвращает список игроков, которые занимают определённую позицию в команде:

JAVA

```
public List<PlayerDetails> getPlayersByPosition(String position) {
    logger.info("getPlayersByPosition");
    List<Player> players = null;

    try {
        CriteriaQuery<Player> cq = cb.createQuery(Player.class);
        if (cq != null) {
            Root<Player> player = cq.from(Player.class);

            // установка предложение where
            cq.where(cb.equal(player.get(Player_.position), position));
            cq.select(player);
            TypedQuery<Player> q = em.createQuery(cq);
            players = q.getResultList();
        }
        return copyPlayersToDetails(players);
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
}
```

Объект запроса создаётся путём вызова метода `createQuery` объекта `CriteriaBuilder` с типом, установленным в `Player`, поскольку запрос вернёт список игроков.

Корень запроса, базовая сущность, по которой запрос будет перемещаться для поиска атрибутов сущности и связанных сущностей, создаётся путём вызова метода `from` объекта запроса. Это устанавливает предложение `FROM` запроса.

Предложение `WHERE`, установленное путём вызова метода `where` объекта запроса, ограничивает результаты запроса в соответствии с условиями выражения. Метод `CriteriaBuilder.equal` сравнивает два выражения. В `getPlayersByPosition`, атрибут `position` класса метамодели `Player_`, доступ к которому вызывается методом `get` корня запроса, сравнивается с параметром `position`, переданным в `getPlayersByPosition`.

Предложение `SELECT` запроса устанавливается путём вызова метода `select` объекта запроса. Запрос вернёт сущности `Player`, поэтому корневой объект запроса передаётся в качестве параметра `select`.

Объект запроса подготавливается к выполнению путём вызова `EntityManager.createQuery`, который возвращает объект `TypedQuery<T>` с типом запроса, в данном случае `Player`. Этот типизированный объект запроса используется для выполнения запроса, который происходит при вызове метода `getResultList` и возвращении коллекции `List<Player>`.

Автоматическая генерация таблиц в roster

При развёртывании GlassFish Server автоматически удалит и создаст таблицы базы данных, используемые `roster`. Для этого свойству `jakarta.persistence.schema-generation.database.action` присваивается значение `drop-and-create` в файле `persistence.xml`:

XML

```
<persistence version="3.0"
    xmlns="https://jakarta.ee/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
        https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd">
  <persistence-unit name="em" transaction-type="JTA">
    <jta-data-source>java:comp/DefaultDataSource</jta-data-source>
    <properties>
      <property name="jakarta.persistence.schema-generation.database.action"
                value="drop-and-create"/>
    </properties>
  </persistence-unit>
</persistence>
```

Запуск roster

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска приложения `roster`.

Запуск roster с использованием IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. Если сервер базы данных ещё не запущен, запустите его, следуя инструкциям в Запуск и остановка Apache Derby.
3. В меню **Файл** выберите **Открыть проект**.
4. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/persistence
   ```

5. Выберите каталог `roster` .
6. Выберите чекбокс **Открыть требуемые проекты**.
7. Нажмите **Открыть проект**.
8. На вкладке **Проекты** кликните правой кнопкой мыши проект `roster` и выберите **Сборка**.

   Это скомпилирует, упакует и развернёт EAR в GlassFish Server.

   На вкладке «Вывод» вы увидите следующий частичный вывод клиентского приложения:

```
List all players in team T2:
P6 Ian Carlyle goalkeeper 555.0
P7 Rebecca Struthers midfielder 777.0
P8 Anne Anderson forward 65.0
P9 Jan Wesley defender 100.0
P10 Terry Smithson midfielder 100.0

List all teams in league L1:
T1 Honey Bees Visalia
T2 Gophers Manteca
T5 Crows Orland

List all defenders:
P2 Alice Smith defender 505.0
P5 Barney Bold defender 100.0
P9 Jan Wesley defender 100.0
P22 Janice Walker defender 857.0
P25 Frank Fletcher defender 399.0
```

Запуск roster с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. Если сервер базы данных ещё не запущен, запустите его, следуя инструкциям в Запуск и остановка Apache Derby.
3. В окне терминала перейдите в:

```
tut-install/examples/persistence/roster/roster-ear/
```

4. Введите следующую команду:

```
mvn install
```
SHELL

   Команда компилирует исходные файлы и упаковывает приложение в файл EAR, расположенный в *`tut-install`*`/examples/persistence/roster/target/roster.ear` . Затем файл EAR развёртывается в GlassFish Server. Затем GlassFish Server будет удалять и создавать таблицы базы данных во время развёртывания, как указано в `persistence.xml` .

   После успешного развёртывания EAR клиентские заглушки извлекаются, а клиентское приложение запускается с помощью приложения appclient, входящего в состав GlassFish Server.

   Вы увидите вывод, который начинается следующим образом:

```
[echo] running application client container.
[exec] List all players in team T2:
[exec] P6 Ian Carlyle goalkeeper 555.0
[exec] P7 Rebecca Struthers midfielder 777.0
[exec] P8 Anne Anderson forward 65.0
[exec] P9 Jan Wesley defender 100.0
[exec] P10 Terry Smithson midfielder 100.0

[exec] List all teams in league L1:
[exec] T1 Honey Bees Visalia
[exec] T2 Gophers Manteca
[exec] T5 Crows Orland

[exec] List all defenders:
[exec] P2 Alice Smith defender 505.0
[exec] P5 Barney Bold defender 100.0
[exec] P9 Jan Wesley defender 100.0
[exec] P22 Janice Walker defender 857.0
[exec] P25 Frank Fletcher defender 399.0
```

Приложение address-book

Приложение `address-book` — это простое веб-приложение, в котором хранятся контактные данные. Он использует класс сущности `Contact`, который использует Jakarta Bean Validation для валидации данных, хранящихся в персистентных атрибутах сущности, как описано в Валидация персистентных полей и свойств.

Ограничения валидации бинов в address-book

Сущность `Contact` использует ограничения `@NotNull`, `@Pattern` и `@Past` для персистентных атрибутов.

Ограничение `@NotNull` помечает атрибут как обязательное поле. Атрибут должен быть установлен в значение не-null, прежде чем объект будет сохранён или изменён. Bean Validation выдаст ошибку валидации, если атрибут будет null при сохранении или изменении.

Ограничение `@Pattern` определяет регулярное выражение, которому должно соответствовать значение атрибута, прежде чем объект сущности будет сохранён или изменён. Это ограничение имеет два различных использования в `address-book`.

- Регулярное выражение, объявленное аннотацией `@Pattern` на поле `email` совпадает с адресами электронной почты имени формы `@` domain name `.` top level domain, допуская только разрешённые символы для адресов электронной почты. Например, `username@example.com` пройдет валидацию, как и `firstname.lastname@mail.example.com`. Однако `firstname,lastname@example.com`, который содержит недопустимый символ запятой в локальном имени, не пройдёт валидацию.
- Поля `mobilePhone` и `homePhone` аннотированы `@Pattern`, которая определяет регулярное выражение для сопоставления номеров телефонов в формате `(` xxx `)` xxx - xxxx.

Ограничение `@Past` применяется к полю дня рождения, которое должно быть `java.util.Date` в прошлом.

Вот соответствующие части класса сущности `Contact`:

JAVA

```
@Entity
public class Contact implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @NotNull
    protected String firstName;
    @NotNull
    protected String lastName;
    @Pattern(regexp = "[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\\."
            + "[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@"
            + "(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.)+[a-z0-9]"
            + "(?:[a-z0-9-]*[a-z0-9])?",
            message = "{invalid.email}")
    protected String email;
    @Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
            message = "{invalid.phonenumber}")
    protected String mobilePhone;
    @Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
            message = "{invalid.phonenumber}")
    protected String homePhone;
    @Temporal(jakarta.persistence.TemporalType.DATE)
    @Past
    protected Date birthday;
    ...
}
```

Указание сообщений об ошибках для ограничений в address-book

Некоторые из ограничений в сущности `Contact` указывают необязательное сообщение:

JAVA

```
@Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
        message = "{invalid.phonenumber}")
protected String homePhone;
```

Необязательный элемент сообщения в ограничении `@Pattern` переопределяет сообщение валидации по умолчанию. Сообщение может быть указано напрямую:

JAVA

```
@Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
        message = "Invalid phone number!")
protected String homePhone;
```

Ограничения в `Contact`, однако, являются строками в пакете ресурсов `ValidationMessages.properties`, в разделе *`tut-install`*`/examples/persistence/address-book/src/java/`. Это позволяет размещать сообщения валидации в одном файле свойств и легко локализовывать сообщения. Переопределённые сообщения Bean Validation должны быть помещены в bundle-ресурс с именем `ValidationMessages.properties` в пакете по умолчанию, при этом локализованные bundle-ресурсы имеют вид `ValidationMessages_`*`locale-prefix`*`.properties`. Например, `ValidationMessages_ es.properties` — это bundle-ресурс, используемый в испаноязычных локалях.

Валидация корректности ввода контактов из приложения Jakarta Faces

Приложение `address-book` использует веб-интерфейс Jakarta Faces, чтобы позволить пользователям вводить контактные данные. Хотя Jakarta Faces имеет механизм валидации данных формы ввода с использованием тегов в файлах Facelets XHTML, `address-book` не использует эти теги валидации. Ограничения валидации бинов в Managed-бинах Jakarta Faces, в данном случае в сущности `Contact`, автоматически запускают валидацию при отправке форм.

В следующем фрагменте кода из файла `Create.xhtml` Facelets показаны некоторые формы ввода для создания новых объектов `Contact`:

```xml
<h:form>
    <table columns="3" role="presentation">
        <tr>
            <td><h:outputLabel value="#{bundle.CreateContactLabel_firstName}"
                               for="firstName" /></td>
            <td><h:inputText id="firstName"
                             value="#{contactController.selected.firstName}"
                             title="#{bundle.CreateContactTitle_firstName}"/>
            </td>
            <td><h:message for="firstName" /></td>
        </tr>
        <tr>
            <td><h:outputLabel value="#{bundle.CreateContactLabel_lastName}"
                               for="lastName" /></td>
            <td><h:inputText id="lastName"
                             value="#{contactController.selected.lastName}"
                             title="#{bundle.CreateContactTitle_lastName}" />
            </td>
            <td><h:message for="lastName" /></td>
        </tr>
        ...
    </table>
</h:form>
```

Теги `<h:inputText>` `firstName` и `lastName` связаны с атрибутами в объекте `selected` сущности `Contact` в сессионном компоненте без сохранения состояния `ContactController`. Каждый тег `<h:inputText>` имеет связанный тег `<h:message>`, который будет отображать сообщения об ошибках валидации. Однако форма не требует каких-либо тегов валидации Jakarta Faces.

Запуск address-book

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска приложения `address-book`.

Запуск address-book с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. Если сервер базы данных ещё не запущен, запустите его, следуя инструкциям в Запуск и остановка Apache Derby.
3. В меню **Файл** выберите **Открыть проект**.
4. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/persistence
   ```

5. Выберите каталог `address-book`.
6. Нажмите **Открыть проект**.
7. На вкладке **Проекты** кликните правой кнопкой мыши проект `address-book` и выберите **Запуск**.

   После развёртывания приложения появится окно веб-браузера по следующему URL:

   ```
   http://localhost:8080/address-book/
   ```

8. Нажмите Показать все элементы контакта, затем Создать новый контакт. Введите значения в поля. Затем нажмите Сохранить.

   Если любое из введённых значений нарушает ограничения в `Contact`, рядом с полем рядом с полем появится сообщение об ошибке с неверными значениями.

Запуск address-book с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. Если сервер базы данных ещё не запущен, запустите его, следуя инструкциям в Запуск и остановка Apache Derby.
3. В окне терминала перейдите в:

   ```
   tut-install/examples/persistence/address-book/
   ```

4. Введите следующую команду:

   ```
   mvn install
   ```
   SHELL

   Эта команда компилирует и упаковывает приложение `address-book` в WAR. WAR-файл затем развёртывается в GlassFish Server.
5. Откройте окно веб-браузера и введите следующий URL:

   ```
   http://localhost:8080/address-book/
   ```

6. Нажмите Показать все элементы контакта, затем Создать новый контакт. Введите значения в поля. Затем нажмите Сохранить.

   Если любое из введённых значений нарушает ограничения в `Contact`, рядом с полем рядом с полем появится сообщение об ошибке с неверными значениями.

Глава 42. Язык запросов Jakarta Persistence

В этой главе описывается язык запросов Jakarta Persistence, который определяет запросы для сущностей и их персистентное состояние. Язык запросов позволяет писать переносимые запросы, которые работают независимо от основного хранилища данных.

Обзор языка запросов Jakarta Persistence

Язык запросов использует абстрактные схемы персистентности сущностей, включая их отношения, для своей модели данных и определяет операторы и выражения на основе этой модели данных. Объём запроса охватывает абстрактные схемы связанных сущностей, которые упакованы в один юнит персистентности. Язык запросов использует SQL-подобный синтаксис для выборки объектов или значений по типам сущностей схемы абстракций и связей между ними.

Эта глава опирается на материал, представленный в предыдущих главах. Концептуальные сведения см. в главе 40 *Введение в Jakarta Persistence*. Примеры кода см. в главе 41 *Выполнение примеров персистентности*.

Терминология языка запросов

Следующий список определяет некоторые термины, упомянутые в этой главе.

- Абстрактная схема: схема персистентных абстракций (персистентных сущностей, их состояние и отношения между ними), с которыми работают запросы. Язык запросов переводит запросы по этой схеме персистентных абстракций в запросы, которые выполняются в схеме базы данных, на которую отображаются объекты.
- Тип схемы абстракций: тип, по которому в схеме абстракций вычисляется персистентное свойство сущности. То есть любое персистентное поле или свойство в объекте имеет соответствующее поле состояния того же типа в схеме абстракций. Тип схемы абстракций для сущности формируется из класса сущности и информации метаданных, предоставленной аннотациями языка Java.
- Форма Бэкуса-Наура (BNF): нотация, которая описывает синтаксис языков высокого уровня. Синтаксические диаграммы в этой главе представлены в формате BNF.
- Навигация: обход отношений в выражении языка запросов. Оператор навигации — точка.
- Выражение пути: выражение, которое перемещается к полю состояния или отношения объекта сущности.
- Поле состояния: персистентное поле объекта сущности.
- Поле отношения: персистентное поле сущности, тип которого является типом связанной сущности в схеме абстракций.

Создание запросов с использованием языка запросов Jakarta Persistence

Методы `EntityManager.createQuery` и `EntityManager.createNamedQuery` используются для обращения к хранилищу данных с помощью запросов Jakarta Persistence.

Метод `createQuery` используется для создания динамических запросов, заданными непосредственно в бизнес-логике приложения:

JAVA

```
public List findWithName(String name) {
return em.createQuery(
    "SELECT c FROM Customer c WHERE c.name LIKE :custName")
    .setParameter("custName", name)
    .setMaxResults(10)
    .getResultList();
}
```

Метод `createNamedQuery` используется для создания статических запросов или запросов, которые определены в метаданных с помощью аннотации `jakarta.persistence.NamedQuery`. Элемент `name` в `@NamedQuery` указывает имя запроса, которое будет присвоено в методе `createNamedQuery`. Элемент `query` в `@NamedQuery` является запросом:

JAVA

```
@NamedQuery(
    name="findAllCustomersWithName",
    query="SELECT c FROM Customer c WHERE c.name LIKE :custName"
)
```

Вот пример `createNamedQuery`, который использует `@NamedQuery`:

JAVA

```
@PersistenceContext
public EntityManager em;
...
customers = em.createNamedQuery("findAllCustomersWithName")
    .setParameter("custName", "Smith")
    .getResultList();
```

Именованные параметры в запросах

Именованные параметры — это параметры запроса с префиксом двоеточия (`:`). Именованные параметры в запросе связываются с аргументом следующим образом:

JAVA

```
jakarta.persistence.Query.setParameter(String name, Object value)
```

В следующем примере аргумент `name` бизнес-метода `findWithName` связывается с именованным параметром `:custName` запроса путём вызова `Query.setParameter`:

JAVA

```
public List findWithName(String name) {
    return em.createQuery(
        "SELECT c FROM Customer c WHERE c.name LIKE :custName")
        .setParameter("custName", name)
        .getResultList();
}
```

Именованные параметры чувствительны к регистру и могут использоваться как динамическими, так и статическими запросами.

Позиционные параметры в запросах

Можно использовать в запросах позиционные параметры вместо именованных. Позиционные параметры имеют префикс знака вопроса (`?`), за которым следует числовая позиция параметра в запросе. Метод `Query.setParameter(int position, Object value)` используется для установки значений параметров.

В следующем примере бизнес-метод `findWithName` переписывается для использования входных параметров:

JAVA

```
public List findWithName(String name) {
    return em.createQuery(
        "SELECT c FROM Customer c WHERE c.name LIKE ?1")
        .setParameter(1, name)
        .getResultList();
}
```

Входные параметры нумеруются начиная с 1. Входные параметры чувствительны к регистру и могут использоваться как динамическими, так и статическими запросами.

Упрощённый синтаксис языка запросов

В этом разделе кратко описывается синтаксис языка запросов, чтобы можно было быстро перейти к Примерам запросов. Когда вы будете готовы узнать о синтаксисе более подробно, см. Полный синтаксис языка запросов.

Выражение SELECT

Запрос на выборку состоит из шести частей (предложений): `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING` и `ORDER BY`. Обязательные предложения `SELECT` и `FROM`, а `WHERE`, `GROUP BY`, `HAVING` и `ORDER BY` являются необязательными. Вот высокоуровневый BNF-синтаксис запроса:

JAVA

```
QL_statement ::= select_clause from_clause
  [where_clause][groupby_clause][having_clause][orderby_clause]
```

Синтаксис BNF определяет следующие предложения.

- Предложение `SELECT` определяет типы объектов или значений, возвращаемых запросом.
- Предложение `FROM` определяет область запроса, объявляя одну или несколько идентификационных переменных, на которые можно ссылаться в предложениях `SELECT` и `WHERE`. Идентификационная переменная представляет собой один из следующих элементов:
  - Имя в схеме абстракций для сущности
  - Элемент отношения коллекции
  - Элемент однозначных отношений
  - Член коллекции, которая является множественной стороной отношения один-ко-многим
- Предложение `WHERE` является условным выражением, которое фильтрует объекты или значения, возвращаемые запросом. Хотя это предложение является необязательным, в большинстве запросов оно присутствует.
- Предложение `GROUP BY` группирует результаты запроса в соответствии с набором свойств.
- Предложение `HAVING` используется с предложением `GROUP BY` для дальнейшей фильтрации результатов запроса в соответствии с условным выражением.
- Предложение `ORDER BY` сортирует объекты или значения, возвращаемые запросом, в указанном порядке.

Выражения обновления и удаления данных

Операторы обновления и удаления обеспечивают массовые операции над наборами объектов. Эти операторы имеют следующий синтаксис:

JAVA

```
update_statement :: = update_clause [where_clause]
delete_statement :: = delete_clause [where_clause]
```

Предложения update_clause и delete_clause определяют тип объектов, которые будут обновлены или удалены. Предложение `WHERE` может использоваться для ограничения объёма операции обновления или удаления.

Примеры запросов

Следующие запросы выполняются к сущности `Player` приложения `roster`, которая описана в приложении roster.

Простые запросы

Если вы не знакомы с языком запросов, эти простые запросы станут хорошей отправной точкой.

Основной запрос на выборку данных

```SQL
SELECT p
FROM Player p
```

- Полученные данные: все игроки.
- Описание: предложение `FROM` объявляет переменную `p`, опуская необязательное ключевое слово `AS`. Если бы было включено ключевое слово `AS`, предложение было бы записано следующим образом:

  ```SQL
  FROM Player AS p
  ```

  Элемент `Player` в схеме абстракций соответствует сущности `Player`.
- Смотрите также: Идентификационные переменные.

Устранение повторяющихся значений

```SQL
SELECT DISTINCT p
FROM Player p
WHERE p.position = ?1
```

- Полученные данные: игроки с позицией, указанной параметром запроса.
- Описание: ключевое слово `DISTINCT` удаляет повторяющиеся значения.

  Предложение `WHERE` фильтрует игроков путём проверки их `position`, персистентного поля сущности `Player`. Элемент `?1` обозначает входной параметр запроса.
- Смотрите также: Входные параметры и Ключевое слово DISTINCT.

Использование именованных параметров

```SQL
SELECT DISTINCT p
FROM Player p
WHERE p.position = :position AND p.name = :name
```

- Полученные данные: игроки с указанными позицией и именем.
- Описание: элементы `position` и `name` являются персистентными полями сущности `Player`. Предложение `WHERE` сравнивает значения этих полей с именованными параметрами запроса, установленными с помощью метода `Query.setNamedParameter`. Язык запросов обозначает именованный входной параметр, используя двоеточие (`:`), за которым следует идентификатор. Первый входной параметр — `:position`, второй — `:name`.

Запросы, ведущие к связанным объектам

На языке запросов выражение может перемещаться по связанным объектам. Эти выражения являются основным отличием языка запросов Jakarta Persistence от SQL. Запросы перемещают к связанным объектам, тогда как SQL объединяет таблицы.

Простой запрос с отношениями

```SQL
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
```

- Полученные данные: все игроки, играющие за команду (неважно какую).
- Описание: предложение `FROM` объявляет две переменные: `p` и `t`. Переменная `p` представляет сущность `Player`, а переменная `t` представляет связанную сущность `Team`. Объявление для `t` ссылается на ранее объявленную переменную `p`. Ключевое слово `IN` означает, что `teams` является набором связанных сущностей. Выражение `p.teams` перемещается от `Player` к связанной с ним `Team`. Точка в выражении `p.teams` является оператором навигации.

  Тот же запрос может быть написан с использованием оператора `JOIN`:

  ```SQL
  SELECT DISTINCT p
  FROM Player p JOIN p.teams t
  ```

  Или так:

  ```SQL
  SELECT DISTINCT p
  FROM Player p
  WHERE p.team IS NOT EMPTY
  ```

Переход к однозначным полям отношений

Используйте оператор предложения `JOIN`, чтобы перейти к однозначному полю отношений:

```SQL
SELECT t
FROM Team t JOIN t.league l
WHERE l.sport = 'soccer' OR l.sport ='football'
```

В этом примере запрос вернёт все команды из футбольной лиги.

Обход отношений с входным параметром

```SQL
SELECT DISTINCT p
FROM Player p, IN (p.teams) AS t
WHERE t.city = :city
```

- Полученные данные: игроки, команды которых находятся в указанном городе.
- Описание: этот запрос похож на предыдущий пример, но добавляет входной параметр. Ключевое слово `AS` в предложении `FROM` является необязательным. В предложении `WHERE` точка, предшествующая персистентной переменной `city`, является разделителем, а не оператором навигации. Строго говоря, выражения могут перемещаться к полям отношений (связанным объектам), но не к персистентным полям. Чтобы получить доступ к персистентному полю, выражение использует точку в качестве разделителя.

  Выражения не могут выходить за пределы (или уточнять) полей отношений, которые являются коллекциями. В синтаксисе выражения поле со значением коллекции является терминальным символом. Поскольку поле `teams` является коллекцией, предложение `WHERE` не может указывать `p.teams.city`

(недопустимое выражение).

- Смотрите также: Выражения пути.

Обход множественных отношений

```SQL
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league = :league
```

- Полученные данные: игроки указанной лиги.
- Описание: выражения в этом запросе перемещаются по двум отношениям. Выражение `p.teams` перемещается по отношению `Player - Team`, а выражение `t.league` перемещается по отношению `Team - League`.

В других примерах входными параметрами являются объекты `String`. В этом примере параметр является объектом типа `League`. Этот тип соответствует полю отношения `league` в выражении сравнения предложения `WHERE`.

Навигация по связанным полям

```SQL
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league.sport = :sport
```

- Полученные данные: игроки, участвующие в указанном виде спорта.
- Описание: персистентное поле `sport` принадлежит сущности `League`. Чтобы добраться до поля `sport`, запрос должен сначала перейти от сущности `Player` к `Team` (`p.teams`) а затем из `Team` в организацию `League` (`t.league`). Поскольку это не коллекция, за полем отношений `league` может следовать персистентное поле `sport`.

Запросы с другими условными выражениями

Каждое предложение `WHERE` должно указывать условное выражение одного из нескольких видов. В предыдущих примерах условные выражения — это выражения сравнения, которые проверяют равенство. Следующие примеры демонстрируют и другие виды условных выражений. Для описания всех условных выражений см. Выражение WHERE.

Выражение LIKE

```SQL
SELECT p
FROM Player p
WHERE p.name LIKE 'Mich%'
```

- Полученные данные: все игроки, чьи имена начинаются с "Mich".
- Описание: выражение `LIKE` использует символы подстановки для выборки строк, соответствующих шаблону подстановки. В этом случае запрос использует выражение `LIKE` и подстановочный знак `%`, чтобы найти всех игроков, имена которых начинаются со строки «Mich». Например, «Michael» и «Michelle» оба соответствуют шаблону подстановочных знаков.
- Смотрите также: Выражения LIKE.

Выражение IS NULL

```SQL
SELECT t
FROM Team t
WHERE t.league IS NULL
```

- Полученные данные: все команды, не связанные ни с одной лигой.
- Описание: выражение `IS NULL` можно использовать для проверки, установлена ли связь между двумя сущностями. В этом случае запрос проверяет, связаны ли команды с какой-либо лигой, и возвращает команды, которые не имеют лигу.
- Смотрите также: выражения сравнения NULL и значения NULL.

Выражение IS EMPTY

```SQL
SELECT p
FROM Player p
WHERE p.teams IS EMPTY
```

- Полученные данные: все игроки без команды.
- Описание: поле отношения `teams` объекта `Player` является коллекцией. Если игрок не принадлежит ни к одной команде, коллекция `teams` пуста, а условное выражение равно `TRUE`.
- Смотрите также: Выражения сравнения с пустой коллекцией.

Выражение BETWEEN

```SQL
SELECT DISTINCT p
FROM Player p
WHERE p.salary BETWEEN :lowerSalary AND :higherSalary
```

- Полученные данные: игроки, зарплаты которых находятся в указанном диапазоне.
- Описание: `BETWEEN` имеет три арифметических выражения: персистентное поле (`p.salary`) и два входных параметра (`:lowerSalary` и `:higherSalary`). Следующее выражение эквивалентно выражению `BETWEEN`:

  ```SQL
  p.salary >= :lowerSalary AND p.salary <= :higherSalary
  ```

- Смотрите также: Выражения BETWEEN.

Операторы сравнения

```SQL
SELECT DISTINCT p1
FROM Player p1, Player p2
WHERE p1.salary > p2.salary AND p2.name = :name
```

- Полученные данные: все игроки, чья зарплата выше зарплаты игрока с указанным именем.
- Описание: предложение `FROM` объявляет две переменные (`p1` и `p2`) типа `Player`. Две переменные необходимы, потому что предложение `WHERE` сравнивает зарплату одного игрока (`p2`) с зарплатой других игроков (`p1`).
- Смотрите также: Идентификационные переменные.

Массовые обновления и удаления

В следующих примерах показано, как использовать выражения `UPDATE` и `DELETE` в запросах. `UPDATE` и `DELETE` работают с несколькими объектами в соответствии с условием/условиями, установленными в предложении `WHERE`. Предложение `WHERE` в запросах `UPDATE` и `DELETE` соответствует тем же правилам, что и запросы `SELECT`.

Запросы на обновление данных

```SQL
UPDATE Player p
SET p.status = 'inactive'
WHERE p.lastPlayed < :inactiveThresholdDate
```

- Описание: этот запрос устанавливает статус группы игроков в `inactive`, если последняя игра игрока была раньше даты, указанной в `inactiveThresholdDate`.

Запросы удаления данных

```SQL
DELETE
FROM Player p
WHERE p.status = 'inactive'
AND p.teams IS EMPTY
```

- Описание: Этот запрос удаляет всех неактивных игроков без команды.

Полный синтаксис языка запросов

В этом разделе обсуждается синтаксис языка запросов, определённый в спецификации Jakarta Persistence 3.0, доступной по ссылке https://jakarta.ee/specifications/persistence/3.0/. Многое из следующего материала перефразирует или прямо цитирует спецификацию.

Символы BNF

Таблица 42-1 описывает символы BNF, используемые в этой главе.

Таблица 42-1 Обзор символов BNF

Символ	Описание
`::=`	Элемент слева от символа определяется конструкциями справа.
`*`	Предыдущая конструкция может встречаться любое количество раз.
`{…}`	Конструкции в фигурных скобках сгруппированы вместе.
`[…]`	Конструкции в квадратных скобках не являются обязательными.
`|`	Исключающее `OR`.
`жирный шрифт`	Ключевое слово. Хотя в диаграмме BNF они указаны прописными, ключевые слова не чувствительны к регистру.

Символ	Описание
Символ пробела	Символом пробела может быть пробел, горизонтальная табуляция или перевод строки.

Грамматика BNF языка запросов Jakarta Persistence

Вот полная диаграмма BNF для языка запросов:

SQL

```
QL_statement ::= select_statement | update_statement | delete_statement
select_statement ::= select_clause from_clause [where_clause] [groupby_clause]
    [having_clause] [orderby_clause]
update_statement ::= update_clause [where_clause]
delete_statement ::= delete_clause [where_clause]
from_clause ::=
    FROM identification_variable_declaration
        {, {identification_variable_declaration |
            collection_member_declaration}}*
identification_variable_declaration ::=
        range_variable_declaration { join | fetch_join }*
range_variable_declaration ::= abstract_schema_name [AS]
        identification_variable
join ::= join_spec join_association_path_expression [AS]
        identification_variable
fetch_join ::= join_specFETCH join_association_path_expression
association_path_expression ::=
        collection_valued_path_expression |
        single_valued_association_path_expression
join_spec::= [LEFT [OUTER] |INNER] JOIN
join_association_path_expression ::=
        join_collection_valued_path_expression |
        join_single_valued_association_path_expression
join_collection_valued_path_expression::=
    identification_variable.collection_valued_association_field
join_single_valued_association_path_expression::=
        identification_variable.single_valued_association_field
collection_member_declaration ::=
        IN (collection_valued_path_expression) [AS]
        identification_variable
single_valued_path_expression ::=
        state_field_path_expression |
        single_valued_association_path_expression
state_field_path_expression ::=
    {identification_variable |
    single_valued_association_path_expression}.state_field
single_valued_association_path_expression ::=
    identification_variable.{single_valued_association_field.}*
    single_valued_association_field
collection_valued_path_expression ::=
    identification_variable.{single_valued_association_field.}*
    collection_valued_association_field
state_field ::=
    {embedded_class_state_field.}*simple_state_field
update_clause ::=UPDATE abstract_schema_name [[AS]
    identification_variable] SET update_item {, update_item}*
update_item ::= [identification_variable.]{state_field |
    single_valued_association_field} = new_value
new_value ::=
     simple_arithmetic_expression |
    string_primary |
    datetime_primary |
    boolean_primary |
    enum_primary simple_entity_expression |
    NULL
delete_clause ::= DELETE FROM abstract_schema_name [[AS]
    identification_variable]
select_clause ::= SELECT [DISTINCT] select_expression {,
    select_expression}*
select_expression ::=
    single_valued_path_expression |
    aggregate_expression |
    identification_variable |
    OBJECT(identification_variable) |
    constructor_expression
constructor_expression ::=
    NEW constructor_name(constructor_item {,
```

```
    constructor_item}*)
constructor_item ::= single_valued_path_expression |
    aggregate_expression
aggregate_expression ::=
    {AVG |MAX |MIN |SUM} ([DISTINCT]
        state_field_path_expression) |
    COUNT ([DISTINCT] identification_variable |
        state_field_path_expression |
        single_valued_association_path_expression)
where_clause ::= WHERE conditional_expression
groupby_clause ::= GROUP BY groupby_item {, groupby_item}*
groupby_item ::= single_valued_path_expression
having_clause ::= HAVING conditional_expression
orderby_clause ::= ORDER BY orderby_item {, orderby_item}*
orderby_item ::= state_field_path_expression [ASC |DESC]
subquery ::= simple_select_clause subquery_from_clause
    [where_clause] [groupby_clause] [having_clause]
subquery_from_clause ::=
    FROM subselect_identification_variable_declaration
        {, subselect_identification_variable_declaration}*
subselect_identification_variable_declaration ::=
    identification_variable_declaration |
    association_path_expression [AS] identification_variable |
    collection_member_declaration
simple_select_clause ::= SELECT [DISTINCT]
    simple_select_expression
simple_select_expression::=
    single_valued_path_expression |
    aggregate_expression |
    identification_variable
conditional_expression ::= conditional_term |
    conditional_expression OR conditional_term
conditional_term ::= conditional_factor | conditional_term AND
    conditional_factor
conditional_factor ::= [NOT] conditional_primary
conditional_primary ::= simple_cond_expression |(
    conditional_expression)
simple_cond_expression ::=
    comparison_expression |
    between_expression |
    like_expression |
    in_expression |
    null_comparison_expression |
    empty_collection_comparison_expression |
    collection_member_expression |
    exists_expression
between_expression ::=
    arithmetic_expression [NOT] BETWEEN
        arithmetic_expressionAND arithmetic_expression |
    string_expression [NOT] BETWEEN string_expression AND
        string_expression |
    datetime_expression [NOT] BETWEEN
        datetime_expression AND datetime_expression
in_expression ::=
    state_field_path_expression [NOT] IN (in_item {, in_item}*
    | subquery)
in_item ::= literal | input_parameter
like_expression ::=
    string_expression [NOT] LIKE pattern_value [ESCAPE
        escape_character]
null_comparison_expression ::=
    {single_valued_path_expression | input_parameter} IS [NOT]
        NULL
empty_collection_comparison_expression ::=
    collection_valued_path_expression IS [NOT] EMPTY
collection_member_expression ::= entity_expression
    [NOT] MEMBER [OF] collection_valued_path_expression
exists_expression::= [NOT] EXISTS (subquery)
```

```
all_or_any_expression ::= {ALL |ANY |SOME} (subquery)
comparison_expression ::=
    string_expression comparison_operator {string_expression |
    all_or_any_expression} |
    boolean_expression {= |<> } {boolean_expression |
    all_or_any_expression} |
    enum_expression {= |<> } {enum_expression |
    all_or_any_expression} |
    datetime_expression comparison_operator
        {datetime_expression | all_or_any_expression} |
    entity_expression {= |<> } {entity_expression |
    all_or_any_expression} |
    arithmetic_expression comparison_operator
        {arithmetic_expression | all_or_any_expression}
comparison_operator ::= = |> |>= |< |<= |<>
arithmetic_expression ::= simple_arithmetic_expression |
    (subquery)
simple_arithmetic_expression ::=
    arithmetic_term | simple_arithmetic_expression {+ |- }
        arithmetic_term
arithmetic_term ::= arithmetic_factor | arithmetic_term {* |/ }
    arithmetic_factor
arithmetic_factor ::= [{+ |- }] arithmetic_primary
arithmetic_primary ::=
    state_field_path_expression |
    numeric_literal |
    (simple_arithmetic_expression) |
    input_parameter |
    functions_returning_numerics |
    aggregate_expression
string_expression ::= string_primary | (subquery)
string_primary ::=
    state_field_path_expression |
    string_literal |
    input_parameter |
    functions_returning_strings |
    aggregate_expression
datetime_expression ::= datetime_primary | (subquery)
datetime_primary ::=
    state_field_path_expression |
    input_parameter |
    functions_returning_datetime |
    aggregate_expression
boolean_expression ::= boolean_primary | (subquery)
boolean_primary ::=
    state_field_path_expression |
    boolean_literal |
    input_parameter
 enum_expression ::= enum_primary | (subquery)
enum_primary ::=
    state_field_path_expression |
    enum_literal |
    input_parameter
entity_expression ::=
    single_valued_association_path_expression |
        simple_entity_expression
simple_entity_expression ::=
    identification_variable |
    input_parameter
functions_returning_numerics::=
    LENGTH(string_primary) |
    LOCATE(string_primary, string_primary[,
        simple_arithmetic_expression]) |
    ABS(simple_arithmetic_expression) |
    SQRT(simple_arithmetic_expression) |
    MOD(simple_arithmetic_expression,
        simple_arithmetic_expression) |
    SIZE(collection_valued_path_expression)
```

```
functions_returning_datetime ::=
    CURRENT_DATE |
    CURRENT_TIME |
    CURRENT_TIMESTAMP
functions_returning_strings ::=
    CONCAT(string_primary, string_primary) |
    SUBSTRING(string_primary,
        simple_arithmetic_expression,
        simple_arithmetic_expression)|
    TRIM([[trim_specification] [trim_character] FROM]
        string_primary) |
    LOWER(string_primary) |
    UPPER(string_primary)
trim_specification ::= LEADING | TRAILING | BOTH
```

Выражение FROM

Предложение `FROM` определяет область запроса путём объявления идентификационных переменных.

Идентификаторы

Идентификатор — это последовательность из одного или нескольких символов. Первый символ должен быть допустимым первым символом (буквой, `$`, `_`) для идентификаторов языка Java. Каждый последующий символ должен быть допустимым символом (буквой, цифрой, `$`, `_`) для идентификаторов Java. (Подробнее см. документацию Java SE API для методов `isJavaIdentifierStart` и `isJavaIdentifierPart` класса `Character`.) Знак вопроса (`?`) является зарезервированным символом в языке запросов и не может использоваться в идентификаторе.

Идентификатор языка запросов чувствителен к регистру, с двумя исключениями:

- Ключевые слова
- Идентификационные переменные

Идентификатор не может совпадать с ключевым словом языка запросов. Вот список ключевых слов языка запросов:

ABS	ALL	AND	ANY
AS	ASC	AVG	BETWEEN
BIT_LENGTH	BOTH	BY	CASE
CHAR_LENGTH	CHARACTER_LENGTH	CLASS	COALESCE
CONCAT	COUNT	CURRENT_DATE	CURRENT_TIMESTAMP
DELETE	DESC	DISTINCT	ELSE
EMPTY	END	ENTRY	ESCAPE
EXISTS	FALSE	FETCH	FROM
GROUP	HAVING	IN	INDEX
INNER	IS	JOIN	KEY
LEADING	LEFT	LENGTH	LIKE

LOCATE	LOWER	MAX	MEMBER
MIN	MOD	NEW	NOT
NULL	NULLIF	OBJECT	OF
OR	ORDER	OUTER	POSITION
SELECT	SET	SIZE	SOME
SQRT	SUBSTRING	SUM	THEN
TRAILING	TRIM	TRUE	TYPE
UNKNOWN	UPDATE	UPPER	VALUE
WHEN	WHERE		

Не рекомендуется использовать ключевое слово SQL в качестве идентификатора, поскольку список ключевых в будущем слов может быть расширен и включить другие зарезервированные слова SQL.

Идентификационные переменные

Идентификационная переменная — это идентификатор, объявленный в предложении `FROM`. Хотя они могут ссылаться на идентификационные переменные, предложения `SELECT` и `WHERE` не могут их объявить. Все идентификационные переменные должны быть объявлены в предложении `FROM`.

Поскольку это идентификатор, идентификационная переменная подчиняется тем же соглашениям и ограничениям именования, что и идентификатор с тем исключением, что идентификационная переменная не зависит от регистра. Например, переменная идентификации не может совпадать с ключевым словом языка запросов. (Дополнительные правила именования см. в Идентификаторах.) Кроме того, в пределах данного юнита персистентности имя идентификационной переменной не должно совпадать с именем сущности или типом схемы абстракций.

Предложение `FROM` может содержать несколько объявлений, разделённых запятыми. Объявление может ссылаться на другую идентификационную переменную, объявленную ранее (слева). В следующем предложении `FROM` переменная `t` ссылается на ранее объявленную переменную `p`:

```SQL
FROM Player p, IN (p.teams) AS t
```

Даже если она не используется в предложении `WHERE`, объявление идентификационной переменной может повлиять на результаты запроса. Например, сравните следующие два запроса. Этот запрос возвращает всех игроков, независимо от их принадлежности к какой-либо команде:

```SQL
SELECT p
FROM Player p
```

Следующий запрос напротив, выбирает только игроков, принадлежащих к какой-либо команде, поскольку он объявляет идентификационную переменную `t`:

```SQL
SELECT p
FROM Player p, IN (p.teams) AS t
```

Следующий запрос возвращает те же результаты, что и предыдущий запрос, но предложение `WHERE` облегчает чтение:

```SQL
SELECT p
FROM Player p
WHERE p.teams IS NOT EMPTY
```

Идентификационная переменная всегда обозначает ссылку на одно значение, тип которого является типом выражения, используемого в объявлении. Существует два вида объявлений: переменная диапазона и член коллекции.

Объявления переменных диапазона

Чтобы объявить идентификационную переменную как тип схемы абстракций, укажите переменную диапазона. Другими словами, переменная идентификации может варьироваться по типу схемы абстракций. В следующем примере идентификационная переменная с именем `p` представляет тип `Player` схемы абстракций:

```SQL
FROM Player p
```

Объявление переменной диапазона может включать в себя необязательный оператор `$`:

```SQL
FROM Player AS p
```

Чтобы получить объекты, запрос обычно использует выражения пути для навигации по отношениям. Но для тех объектов, которые не могут быть получены с помощью навигации, вы можете использовать объявление переменной диапазона для обозначения начальной точки или корня запроса.

Если запрос сравнивает несколько значений одного и того же типа схемы абстракций, предложение `FROM` должно объявить несколько идентификационных переменных для схемы абстракций:

```SQL
FROM Player p1, Player p2
```

Пример такого запроса см. в разделе Операторы сравнения.

Объявления членов коллекции

В отношении «один ко многим» множественная сторона состоит из набора сущностей. Идентификационная переменная может представлять член этой коллекции. Чтобы получить доступ к элементу коллекции, выражение пути в объявлении переменной перемещается по отношениям в схеме абстракций. (Для получения дополнительной информации о выражениях пути см. Выражения пути.) Поскольку выражение пути может быть основано на другом выражении пути, навигация может проходить через несколько отношений. Смотрите Обход множественных отношений.

Объявление члена коллекции должно включать оператор `IN`, но может опускать необязательный оператор `$`.

В следующем примере сущность, представленная типом `Player` схемы абстракций, имеет поле отношения `teams`. Идентификационная переменная `t` представляет одного члена коллекции `teams`:

```SQL
FROM Player p, IN (p.teams) t
```

Объединения

Оператор `JOIN` используется для обхода отношений между сущностями и функционально аналогичен оператору `IN`.

В следующем примере запрос объединяет клиентов и заказы по отношению между ними:

```SQL
SELECT c
FROM Customer c JOIN c.orders o
WHERE c.status = 1 AND o.totalPrice > 10000
```

Ключевое слово `INNER` необязательно:

```SQL
SELECT c
FROM Customer c INNER JOIN c.orders o
WHERE c.status = 1 AND o.totalPrice > 10000
```

Эти примеры эквивалентны следующему запросу, в котором используется оператор `IN`:

```SQL
SELECT c
FROM Customer c, IN (c.orders) o
WHERE c.status = 1 AND o.totalPrice > 10000
```

Вы также можете присоединиться к однозначным отношениям:

```SQL
SELECT t
FROM Team t JOIN t.league l
WHERE l.sport = :sport
```

`LEFT JOIN` или `LEFT OUTER JOIN` извлекает набор объектов, в которых совпадающие значения в условии соединения могут отсутствовать. Ключевое слово `OUTER` необязательно:

```SQL
SELECT c.name, o.totalPrice
FROM CustomerOrder o LEFT JOIN o.customer c
```

`FETCH JOIN` — это операция объединения, которая возвращает связанные сущности в качестве побочного эффекта выполнения запроса. В следующем примере запрос возвращает набор отделов и, как побочный эффект, связанных сотрудников отделов, даже если сотрудники не были явно запрошены предложением `SELECT`:

```SQL
SELECT d
FROM Department d LEFT JOIN FETCH d.employees
WHERE d.deptno = 1
```

Выражения пути

Выражения пути являются важными конструкциями в синтаксисе языка запросов по нескольким причинам. Во-первых, выражения пути определяют пути навигации по отношениям в схеме абстракций. Эти определения пути влияют как на объём, так и на результаты запроса. Во-вторых, выражения пути могут появляться в любом из основных предложений запроса (`SELECT`, `DELETE`, `HAVING`, `UPDATE`, `WHERE`, `FROM`, `GROUP BY`, `ORDER BY`). Наконец, хотя большая часть языка запросов является подмножеством SQL, выражения пути являются расширениями, которых нет в SQL.

Примеры выражений пути

Здесь предложение `WHERE` содержит `single_valued_path_expression`. `p` является идентификационной переменной, а `salary` персистентным полем `Player`:

```SQL
SELECT DISTINCT p
FROM Player p
WHERE p.salary BETWEEN :lowerSalary AND :higherSalary
```

Здесь, предложение `WHERE` также содержит `single_valued_path_expression`. `t` является идентификационной переменной, `league` однозначным полем отношений, а `sport` персистентным полем `league`:

```SQL
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league.sport = :sport
```

Здесь предложение `WHERE` содержит `collection_valued_path_expression`. `p` является идентификационной переменной, а `teams` обозначает поле отношения со значением коллекции:

```SQL
SELECT DISTINCT p
FROM Player p
WHERE p.teams IS EMPTY
```

Типы выражений

Тип выражения пути — это тип объекта, представленного конечным элементом, который может быть одним из следующих:

- Персистентное поле
- Однозначное поле отношения
- Поле отношения с коллекцией

Например, выражения `p.salary` имеет тип `double`, потому что завершающее персистентное поле (`salary`) имеет тип `double`.

В выражении `p.teams` завершающий элемент является полем отношений (`teams`). Тип этого выражения является коллекцией объектов типа `Team` схемы абстракций. `Team` является типом схемы абстракций для сущности `Team` и отображается на эту сущность. Для получения дополнительной информации о сопоставлении типов абстрактных схем см. Возвращаемые типы.

Навигация

Выражение пути позволяет запросу перемещаться к связанным объектам. Завершающие элементы выражения определяют, разрешена ли навигация. Если выражение содержит однозначное поле отношения, навигация может продолжаться к объекту, связанному с полем. Однако выражение не может выходить за пределы персистентного поля или поля отношения со значением коллекции. Например, выражение `p.teams.league.sport` недопустимо, потому что `teams` является полем отношений со значением коллекции. Чтобы достичь поля `sport`, предложение `FROM` может определить идентификационную переменную `t` для поля `teams`:

```SQL
FROM Player AS p, IN (p.teams) t
WHERE t.league.sport = 'soccer'
```

Выражение WHERE

Предложение `WHERE` определяет условное выражение, которое фильтрует значения, возвращаемые запросом. Запрос возвращает все соответствующие значения из хранилища данных, для которого значение условного выражения `TRUE`. Предложение `WHERE` является необязательным, хотя обычно оно указывается. Если предложение `WHERE` опущено, запрос возвращает все значения. Синтаксис высокого уровня для предложения `WHERE` выглядит следующим образом:

```sql
where_clause ::= WHERE conditional_expression
```

Литералы

Существует четыре вида литералов: строковые, числовые, логические и перечисления.

- Строковые литералы: строковый литерал заключён в одинарные кавычки:

  ```java
  'Duke'
  ```

  Если строковый литерал содержит одну кавычку, вы указываете кавычку с помощью двух одинарных кавычек:

  ```java
  'Duke''s'
  ```

  Подобно `String` в Java, строковый литерал в языке запросов использует кодировку символов Unicode.

- Числовые литералы. Существует два типа числовых литералов: точные и приблизительные.
  - Точный числовой литерал — это числовое значение без десятичной точки, например 65, –233 и +12. Используя целочисленный синтаксис Java, точные числовые литералы поддерживают числа в диапазоне `long` Java.
  - Приблизительный числовой литерал — это числовое значение в научной нотации, например 57., –85.7 и +2.1. Используя синтаксис литерала Java с плавающей точкой, приближенные числовые литералы поддерживают числа в диапазоне `double` Java.
- Логические литералы: логический литерал имеет значение `TRUE` или `FALSE`. Эти ключевые слова не чувствительны к регистру.
- Литералы Enum: язык запросов Jakarta Persistence поддерживает использование литералов enum с использованием синтаксиса литерала Java enum. Имя класса enum должно быть указано как полное имя класса:

  ```sql
  SELECT e
  FROM Employee e
  WHERE e.status = com.example.EmployeeStatus.FULL_TIME
  ```

Входные параметры

Входные параметры могут быть именованными или позиционными.

- Именованный входной параметр обозначается двоеточием (`:`), за которым следует строка. Например, `:name`.
- Позиционный входной параметр обозначается знаком вопроса (`?`), за которым следует целое число. Например, первый входной параметр — это `?1`, второй — `?2` и так далее.

Следующие правила применяются к входным параметрам.

- Они могут использоваться только в предложении `WHERE` или `HAVING`.
- Позиционные параметры должны быть пронумерованы, начиная с 1.
- Именованные параметры и позиционные параметры не могут быть смешаны в одном запросе.
- Именованные параметры чувствительны к регистру.

Условные выражения

Предложение `WHERE` состоит из условного выражения, которое вычисляется слева направо в пределах уровня приоритета. Вы можете изменить порядок выполнения, используя скобки.

Операторы и их приоритет

Таблица 42-2 перечисляет операторы языка запросов в порядке убывания приоритета.

Таблица 42-2. Порядок приоритетов языка запросов

Тип	**Порядок приоритета**
Навигация	`.` (точка)
Арифметические	`+ -` (одинарный) `*/` (умножение и деление) `+ -` (сложение и вычитание)

Сравнения	`=`
	`>`
	`>=`
	`<`
	`=<`
	`<>` (не равно)
	`[NOT] BETWEEN`
	`[NOT] LIKE`
	`[NOT] IN`
	`IS [NOT] NULL`
	`IS [NOT] EMPTY`
	`[NOT] MEMBER OF`

Логические	`NOT`
	`AND`
	`OR`

Выражения BETWEEN

Выражение `BETWEEN` определяет, попадает ли арифметическое выражение в диапазон значений.

Эти два выражения эквивалентны:

```SQL
p.age BETWEEN 15 AND 19
p.age >= 15 AND p.age <= 19
```

Следующие два выражения также эквивалентны:

```SQL
p.age NOT BETWEEN 15 AND 19
p.age < 15 OR p.age > 19
```

Если арифметическое выражение имеет значение `NULL`, значение выражения `BETWEEN` неизвестно.

Выражения IN

Выражение `IN` определяет, принадлежит ли строка к набору строковых литералов или число принадлежит к набору числовых значений.

Выражение пути должно иметь строковое или числовое значение. Если выражение пути имеет значение `NULL`, значение выражения `IN` неизвестно.

В следующем примере выражение равно `TRUE`, если страна `UK`, но `FALSE`, если страна `Peru`:

```SQL
o.country IN ('UK', 'US', 'France')
```

Также могут использоваться входные параметры:

```SQL
o.country IN ('UK', 'US', 'France', :country)
```

Выражения LIKE

Выражение `LIKE` определяет, соответствует ли шаблон подстановки строке.

Выражение пути должно иметь строковое или числовое значение. Если это значение равно `NULL`, значение выражения `LIKE` неизвестно. Значением шаблона является строковый литерал, который может содержать символы подстановки. Подстановочный знак (`_`) обозначает любой отдельный символ. Подстановочный знак процента (`%`) представляет собой любое количество символов. Предложение `ESCAPE` определяет escape-символ для символов подстановки в значении шаблона. Таблица 42-3 показывает некоторые примеры выражений `LIKE`.

Таблица 42-3 Примеры выражения LIKE

Выражение	**TRUE**	**FALSE**
`address.phone LIKE '12%3'`	`'123'` `'12993'`	`'1234'`
`asentence.word LIKE 'l_se'`	`'lose'`	`'loose'`
`aword.underscored LIKE '\_%' ESCAPE '\'`	`'_foo'`	`'bar'`

Выражение	TRUE	FALSE
`address.phone NOT LIKE '12%3'`	`'1234'`	`'123'` `'12993'`

Выражения сравнения с NULL

Выражение сравнения с `NULL` проверяет, имеет ли однозначное выражение пути или входной параметр значение `NULL`. Обычно выражение сравнения с `NULL` используется для проверки, установлено ли однозначное отношение:

```SQL
SELECT t
FROM Team t
WHERE t.league IS NULL
```

Этот запрос выбирает все команды без установленной лиги. Обратите внимание, что следующий запрос не эквивалентен предыдущему:

```SQL
SELECT t
FROM Team t
WHERE t.league = NULL
```

Сравнение с `NULL` с использованием оператора равенства (=) всегда возвращает неизвестное значение, даже если связь не установлена. Второй запрос всегда будет возвращать пустой результат.

Выражения сравнения с пустой коллекцией

Выражение сравнения `IS [NOT] EMPTY` проверяет, нет ли в выражении пути с коллекцией значений элементов. Другими словами, он проверяет, установлено ли отношение со значением коллекции.

Если выражение пути со значением коллекции `NULL`, пустое выражение сравнения коллекции имеет значение `NULL`.

Вот пример, который находит все заказы, которые не имеют позиций:

```SQL
SELECT o
FROM CustomerOrder o
WHERE o.lineItems IS EMPTY
```

Выражения к членам коллекции

Выражение члена коллекции `[NOT] MEMBER [OF]` определяет, является ли значение членом коллекции. Значение и члены коллекции должны иметь одинаковый тип.

Если выражение пути с коллекцией или с одним значением неизвестно, выражение члена коллекции неизвестно. Если выражение пути со значением коллекции обозначает пустую коллекцию, выражение члена коллекции имеет значение `FALSE`.

Ключевое слово `OF` является необязательным.

В следующем примере проверяется, является ли позиция частью заказа:

```SQL
SELECT o
FROM CustomerOrder o
WHERE :lineItem MEMBER OF o.lineItems
```

Подзапросы

Подзапросы могут использоваться в предложении `WHERE` или `HAVING` запроса. Подзапросы должны быть заключены в круглые скобки.

Следующий пример находит всех клиентов, которые разместили более десяти заказов:

```SQL
SELECT c
FROM Customer c
WHERE (SELECT COUNT(o) FROM c.orders o) > 10
```

Подзапросы могут содержать выражения `EXISTS`, `ALL` и `ANY`.

- Выражения EXISTS: выражение `[NOT] EXISTS` используется с подзапросом и имеет значение true, только если результат подзапроса состоит из одного или нескольких значений. В противном случае false.

  Следующий пример находит всех сотрудников, чьи супруги также являются сотрудниками:

  ```SQL
  SELECT DISTINCT emp
  FROM Employee emp
  WHERE EXISTS (
      SELECT spouseEmp
      FROM Employee spouseEmp
      WHERE spouseEmp = emp.spouse)
  ```

- Выражения ALL и ANY: выражение `ALL` используется с подзапросом и имеет значение true, если все значения, возвращаемые подзапросом, являются истинными или если подзапрос пуст.

  Выражение `ANY` используется с подзапросом и является истинным, если некоторые из значений, возвращаемых подзапросом, являются истинными. Выражение `ANY` является ложным, если результат подзапроса пуст или если все возвращённые значения являются ложными. Ключевое слово `SOME` является синонимом `ANY`.

  Выражения `ALL` и `ANY` используются с операторами сравнения `=`, `<`, `<=`, `>`, `>=` и `<>`.

  Следующий пример находит всех сотрудников, чьи зарплаты выше, чем зарплаты менеджеров в отделе сотрудников:

  ```SQL
  SELECT emp
  FROM Employee emp
  WHERE emp.salary > ALL (
      SELECT m.salary
      FROM Manager m
      WHERE m.department = emp.department)
  ```

Функциональные выражения

Язык запросов включает несколько строковых, арифметических функций и функций даты и времени, которые могут использоваться в предложениях `SELECT`, `WHERE` или `HAVING` запроса. Функции перечислены в Таблице 42-4, Таблице 42-5 и Таблице 42-6.

В таблице 42-4 аргументы `start` и `length` имеют тип `int` и обозначают позиции в `String` аргумент. Нумерация позиций в строке начинается с 1.

Таблица 42-4 Строковые выражения

Синтаксис функции	Возвращаемый тип
`CONCAT(String, String)`	`String`
`LENGTH(String)`	`int`
`LOCATE(String, String [, start])`	`int`
`SUBSTRING(String, start, length)`	`String`
`TRIM([[LEADING|TRAILING|BOTH] char FROM] String)`	`String`
`LOWER(String)`	`String`
`UPPER(String)`	`String`

Функция `CONCAT` объединяет две строки в одну строку.

Функция `LENGTH` возвращает длину строки в символах в виде целого числа.

Функция `LOCATE` возвращает позицию указанной строки в строке. Эта функция возвращает первую позицию, в которой строка была найдена, как целое число. Первый аргумент — это строка, которую нужно найти. Второй аргумент — это строка, в которой выполняется поиск. Необязательный третий аргумент — это целое число, представляющее позицию в строке из второго аргумента, с которой должен выполняться поиск. По умолчанию `LOCATE` начинает поиск с начала строки из второго аргумента. То есть с `1`. Если строка не была найдена, `LOCATE` возвращает `0`.

Функция `SUBSTRING` возвращает строку, которая является подстрокой первого аргумента, основываясь на начальной позиции и длине.

Функция `TRIM` обрезает указанный символ от начала и/или конца строки. Если символ не указан, `TRIM` удаляет пробельные символы из строки. Если используется необязательная спецификация `LEADING`, `TRIM` удаляет только начальные символы из строки. Если используется необязательная спецификация `TRAILING`, `TRIM` удаляет только завершающие символы из строки. По умолчанию используется значение `BOTH`, которое удаляет начальные и конечные символы из строки.

Функции `LOWER` и `UPPER` преобразуют строку в нижний или верхний регистр соответственно.

В таблице 42-5 аргумент `number` может быть `int`, `float` или `double`.

Таблица 42-5 Арифметические выражения

Синтаксис функции	Возвращаемый тип
`ABS(number)`	`int`, `float`, or `double`
`MOD(int, int)`	`int`
`SQRT(double)`	`double`
`SIZE(Collection)`	`int`

Функция `ABS` принимает числовое выражение и возвращает число того же типа, что и аргумент.

Функция `MOD` возвращает остаток от деления первого аргумента на второй.

Функция `SQRT` возвращает квадратный корень из числа.

Функция `SIZE` возвращает целое число — количество элементов в данной коллекции.

В таблице 42-6 функции даты/времени возвращают дату, время или метку времени на сервере базы данных.

Таблица 42-6 Выражение даты/времени

Синтаксис функции	**Возвращаемый тип**
`CURRENT_DATE`	`java.sql.Date`
`CURRENT_TIME`	`java.sql.Time`
`CURRENT_TIMESTAMP`	`java.sql.Timestamp`

Выражения Case

Выражения Case изменяются в зависимости от условия, аналогично ключевому слову `case` Java. Ключевое слово `CASE` указывает начало выражения Case, а выражение завершается ключевым словом `END`. Ключевые слова `WHEN` и `THEN` определяют отдельные условия, а ключевое слово `ELSE` определяет условие по умолчанию, если не выполняется ни одно из других условий.

Следующий запрос выбирает имя человека и условную строку в зависимости от подтипа объекта `Person`. Если подтипом является `Student`, возвращается строка `kid`. Если подтипом является `Guardian` или `Staff`, возвращается строка `adult`. Если сущность является другим подтипом `Person`, возвращается строка unknown:

SQL

```
SELECT p.name
CASE TYPE(p)
    WHEN Student THEN 'kid'
    WHEN Guardian THEN 'adult'
    WHEN Staff THEN 'adult'
    ELSE 'unknown'
END
FROM Person p
```

Следующий запрос устанавливает скидку для различных типов клиентов. Покупатели золотого уровня получают скидку 20%, серебряного — 15%, бронзового — 10%, а все остальные получают скидку 5%:

SQL

```
UPDATE Customer c
SET c.discount =
    CASE c.level
        WHEN 'Gold' THEN 20
        WHEN 'Silver' THEN 15
        WHEN 'Bronze' THEN 10
        ELSE 5
    END
```

Значения NULL

Если ссылки указывает на несуществующий в персистентном хранилище объект, значением ссылки будет `NULL`. Для условных выражений, содержащих `NULL`, язык запросов использует семантику, определённую SQL92. Вкратце, эта семантика заключается в следующем.

- Если операция сравнения или арифметическая операция имеют неизвестное значение, она возвращает значение `NULL`.
- Два значения `NULL` не равны. Сравнение двух значений `NULL` даёт неизвестное значение.
- Проверка `IS NULL` преобразует персистентное поле `NULL` или однозначное поле отношения в `TRUE`. Проверка `IS NOT NULL` преобразует их в `FALSE`.
- Логические операторы и условные проверки используют трёхзначную логику, определённую в таблице 42-7 и таблице 42-8. (В этих таблицах T означает `TRUE`, F — `FALSE`, а U — неизвестно.)

Таблица 42-7 Логика оператора AND

AND	**T**	**F**	**U**
T	T	F	U
F	F	F	F
U	U	F	U

Таблица 42-8 Логика оператора OR

OR	**T**	**F**	**U**
T	T	T	T
F	T	F	U
U	T	U	U

Семантика равенства

На языке запросов можно сравнивать только значения одного типа. Однако из этого правила есть одно исключение: можно сравнивать точные и приблизительные числовые значения. При таком сравнении необходимое преобразование типов соответствует правилам Java.

Язык запросов обрабатывает сравниваемые значения, как если бы они были типами Java, а не как если бы они представляли типы в хранилище данных. Например, персистентное поле, которое может быть целым числом или `NULL`, должно быть обозначено как объект `Integer`, а не как примитив `int`. Это обозначение является обязательным, поскольку объект Java может быть `NULL`, а примитив — нет.

Две строки равны, только если они содержат одинаковую последовательность символов. Пробелы в начале и конце имеют значение. Например, строки `'abc'` и `'abc '` не равны.

Два объекта сущности одного и того же типа схемы абстракций равны, только если их первичные ключи имеют одинаковое значение. Таблица 42-9 показывает логику оператора отрицания, а таблица 42-10 показывает истинные значения условных тестов.

Таблица 42-9 Логика оператора NOT

NOT Value	Значение
T	F
F	T
U	U

Таблица 42-10 Условный тест

Условный тест	T	F	U
Выражение `IS TRUE`	T	F	F
Выражение `IS FALSE`	F	T	F
Выражение неизвестно	F	F	T

Выражение SELECT

Предложение `SELECT` определяет типы объектов или значений, возвращаемых запросом.

Возвращаемые типы

Возвращаемый тип предложения `SELECT` определяется типами результатов выражений select, содержащихся в нём. Если используется несколько выражений, результатом запроса является `Object[]`, а элементы в массиве соответствуют порядку выражений в предложении `SELECT` с соответствующими каждому выражению типами.

Предложение `SELECT` не может указывать выражение со значением коллекции. Например, предложение `SELECT p.teams` недопустимо, поскольку `teams` является коллекцией. Однако предложение в следующем запросе является допустимым, поскольку `t` является отдельным элементом коллекции `teams`:

```sql
SELECT t
FROM Player p, IN (p.teams) t
```

Следующий запрос является примером запроса с несколькими выражениями в предложении `SELECT`:

```sql
SELECT c.name, c.country.name
FROM customer c
WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

Этот запрос возвращает список элементов `Object[]`. Первый элемент массива представляет собой строку, обозначающую имя клиента, а второй элемент массива — строку, обозначающую название страны клиента.

Результат запроса может быть результатом вычисления статистической функции, указанной в таблице 42-11.

Таблица 42-11. Агрегатные функции в выражениях Select

Название	Возвращаемый тип	Описание
`AVG`	`Double`	Возвращает среднее значение полей
`COUNT`	`Long`	Возвращает общее количество результатов

Название	Возвращаемый тип	Описание
`MAX`	Тип поля	Возвращает максимальное значение в наборе результатов
`MIN`	Тип поля	Возвращает минимальное значение в наборе результатов
`SUM`	`Long` (для целочисленных полей) `Double` (для чисел с плавающей запятой) `BigInteger` (для полей типа `BigInteger`) `BigDecimal` (для полей типа `BigDecimal`)	Возвращает сумму всех значений в наборе результатов

Для выбора метода запроса с агрегатной функцией (`AVG`, `COUNT`, `MAX`, `MIN` или `SUM`) в предложении `SELECT` применяются следующие правила.

- Функции `AVG`, `MAX`, `MIN` и `SUM` возвращают `null`, если есть нет значений, к которым функция может быть применена.
- Функция `COUNT` возвращает 0, если нет значений, к которым функция может быть применена.

В следующем примере возвращается среднее количество заказов:

```SQL
SELECT AVG(o.quantity)
FROM CustomerOrder o
```

В следующем примере возвращается общая стоимость товаров, заказанных Roxane Coss:

```SQL
SELECT SUM(l.price)
FROM CustomerOrder o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

В следующем примере возвращается общее количество заказов:

```SQL
SELECT COUNT(o)
FROM CustomerOrder o
```

В следующем примере возвращается общее количество товаров с ценами в заказе Hal Incandenza:

```SQL
SELECT COUNT(l.price)
FROM CustomerOrder o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Incandenza' AND c.firstname = 'Hal'
```

Ключевое слово DISTINCT

Ключевое слово `DISTINCT` исключает повторяющиеся возвращаемые значения. Если запрос возвращает `java.util.Collection`, который разрешает дублирование, необходимо указать ключевое слово `DISTINCT`, чтобы исключить дубликаты.

Выражения Constructor

Выражения конструктора позволяют возвращать объекты Java, которые хранят элемент результата запроса вместо `Object[]`.

Следующий запрос создаёт объект `CustomerDetail` для `Customer`, соответствующий предложению `WHERE`. `CustomerDetail` хранит имя клиента и название страны клиента. Таким образом, запрос возвращает `List` объектов `CustomerDetail`:

```SQL
SELECT NEW com.example.CustomerDetail(c.name, c.country.name)
FROM customer c
WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

Выражение ORDER BY

Как следует из названия, предложение `ORDER BY` упорядочивает значения или объекты, возвращаемые запросом.

Если предложение `ORDER BY` содержит несколько элементов, последовательность элементов слева направо определяет упорядоченность от максимального к минимальному.

Ключевое слово `ASC` указывает восходящий порядок, значение по умолчанию, а ключевое слово `DESC` указывает нисходящий порядок.

При использовании предложения `ORDER BY` предложение `SELECT` должно возвращать упорядоченный набор объектов или значений. Нельзя упорядочивать значения или объекты для значений или объектов, не возвращаемых предложением `SELECT`. Например, следующий запрос является допустимым, поскольку в предложении `ORDER BY` используются объекты, возвращаемые предложением `SELECT`:

```SQL
SELECT o
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = 'CA'
ORDER BY o.quantity, o.totalcost
```

Следующий пример недопустим, поскольку в предложении `ORDER BY` используется значение, не возвращаемое предложением `SELECT`:

```SQL
SELECT p.product_name
FROM CustomerOrder o, IN(o.lineItems) l JOIN o.customer c
WHERE c.lastname = 'Faehmel' AND c.firstname = 'Robert'
ORDER BY o.quantity
```

Выражения GROUP BY и HAVING

Предложение `GROUP BY` позволяет группировать значения в соответствии с набором свойств.

Следующий запрос группирует клиентов по их стране и возвращает количество клиентов по стране:

```SQL
SELECT c.country, COUNT(c)
FROM Customer c GROUP BY c.country
```

Предложение `HAVING` используется с предложением `GROUP BY` для дальнейшей фильтрации возвращаемого результата запроса.

Следующий запрос группирует заказы по статусу их клиента и возвращает статус клиента плюс среднее значение `totalPrice` для всех заказов, где соответствующие клиенты имеют одинаковый статус. Кроме того, он рассматривает только клиентов со статусом `1`, `2` или `3`, поэтому заказы других клиентов не учитываются:

```sql
SELECT c.status, AVG(o.totalPrice)
FROM CustomerOrder o JOIN o.customer c
GROUP BY c.status HAVING c.status IN (1, 2, 3)
```

Глава 43. Использование Criteria API для создания запросов

Criteria API используется для определения запросов для сущностей и их персистентного состояния путём создания определяющих запросы объектов. Запросы Criteria написаны с использованием API Java, являются типобезопасными и переносимыми. Такие запросы работают независимо от хранилища данных.

Обзор Criteria и Metamodel API

Подобно JPQL, Criteria API основан на схеме абстракций персистентных сущностей, их отношений и встроенных объектов. Criteria API работает с этой абстрактной схемой, чтобы дать разработчикам возможность находить, изменять и удалять персистентные сущности, вызывая операции Jakarta Persistence. Metamodel API работает совместно с Criteria API для моделирования персистентных классов сущностей для запросов Criteria.

Criteria API и JPQL тесно связаны и предназначены для обеспечения аналогичных операций в их запросах. Разработчики, знакомые с синтаксисом JPQL, найдут схожие операции уровня объектов в Criteria API.

Следующий простой запрос Criteria возвращает все объекты сущности `Pet` в базе данных:

```java
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet);
TypedQuery<Pet> q = em.createQuery(cq);
List<Pet> allPets = q.getResultList();
```

Эквивалентный запрос JPQL

```sql
SELECT p
FROM Pet p
```

Этот запрос демонстрирует основные шаги для создания запроса Criteria.

1. Используйте объект `EntityManager` для создания объекта `CriteriaBuilder`.
2. Создайте объект запроса, создав объект интерфейса `CriteriaQuery`. Атрибуты этого объекта будут содержать детали запроса.
3. Установите корень запроса, вызвав метод `from` объекта `CriteriaQuery`.
4. Укажите тип результата запроса, вызвав метод `select` объекта `CriteriaQuery`.
5. Подготовьте запрос к выполнению, создав объект `TypedQuery<T>` и указав тип результата запроса.
6. Выполните запрос, вызвав метод `getResultList` объекта `TypedQuery<T>`. Поскольку этот запрос возвращает коллекцию сущностей, результат сохраняется в `List`.

В этой главе подробно обсуждаются все задачи каждого из шагов.

Чтобы создать объект `CriteriaBuilder`, вызовите метод `getCriteriaBuilder` объекта `EntityManager`:

```java
CriteriaBuilder cb = em.getCriteriaBuilder();
```

Используйте объект `CriteriaBuilder` для создания объекта запроса:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
```

Запрос вернёт объекты сущности `Pet`. Чтобы создать типобезопасный запрос, укажите тип запроса при создании объекта `CriteriaQuery`.

Вызовите метод `from` объекта запроса, чтобы задать условие `FROM` запроса и указать корень запроса:

```java
Root<Pet> pet = cq.from(Pet.class);
```

Вызовите метод `select` объекта запроса, передав его в корень запроса, чтобы задать предложение `SELECT` запроса:

```java
cq.select(pet);
```

Теперь используйте объект запроса, чтобы создать объект `TypedQuery<T>`, который может быть выполнен для источника данных. Изменения объекта запроса фиксируются для создания готового к выполнению запроса:

```java
TypedQuery<Pet> q = em.createQuery(cq);
```

Выполните этот типизированный объект запроса, вызвав его метод `getResultList`, поскольку этот запрос будет возвращать несколько объектов сущности. Следующий оператор сохраняет результаты в объекте коллекции `List<Pet>`:

```java
List<Pet> allPets = q.getResultList();
```

Использование Metamodel API для моделирования классов объектов

Используйте Metamodel API для создания метамодели управляемых сущностей в определённом юните персистентности. Для каждого класса сущности в конкретном пакете создаётся класс метамодели с завершающим подчеркиванием и атрибутами, которые соответствуют персистентным полям или свойствам класса сущности.

Следующий класс сущностей, `com.example.Pet`, имеет четыре персистентных поля: `id`, `name`, `color`, и `owners`:

```java
package com.example;
...
@Entity
public class Pet {
    @Id
    protected Long id;
    protected String name;
    protected String color;
    @ManyToOne
    protected Set<Person> owners;
    ...
}
```

Соответствующий класс метамодели выглядит следующим образом:

JAVA

```
package com.example;
...
@Static Metamodel(Pet.class)
public class Pet_ {

    public static volatile SingularAttribute<Pet, Long> id;
    public static volatile SingularAttribute<Pet, String> name;
    public static volatile SingularAttribute<Pet, String> color;
    public static volatile SetAttribute<Pet, Person> owners;
}
```

Запросы критериев используют класс метамодели и его атрибуты для ссылки на классы управляемых объектов и их персистентное состояние и взаимосвязи.

Использование классов метамодели

Классы метамодели, которые соответствуют классам сущностей, имеют следующий тип:

JAVA

```
jakarta.persistence.metamodel.EntityType<T>
```

Процессоры аннотаций обычно генерируют классы метамодели во время разработки или во время выполнения. Разработчики приложений, использующих запросы Criteria, могут выполнить одно из следующих действий:

- Сгенерировать статические классы метамодели с использованием процессора аннотаций persistence provider-a
- Получить класс метамодели, выполнив одно из следующих действий:
  - Вызовите метод `getModel` для корневого объекта запроса
  - Получите объект интерфейса `Metamodel` , а затем передайте тип объекта в его метод `entity` .

В следующем фрагменте кода показано, как получить класс метамодели сущности `Pet` , вызвав `Root<T>.getModel` :

JAVA

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
EntityType<Pet> Pet_ = pet.getModel();
```

В следующем фрагменте кода показано, как получить класс метамодели сущности `Pet` , сначала получив объект метамодели с помощью `EntityManager.getMetamodel` , а затем вызвав `entity` объекта метамодели:

JAVA

```
EntityManager em = ...;
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ = m.entity(Pet.class);
```

> Наиболее распространённым вариантом использования является создание типобезопасных статических классов метамодели во время разработки. Динамическое получение классов метамодели путём вызова `Root<T>.getModel` или `EntityManager.getMetamodel` , а затем метода `entity` , не является типобезопасным и не позволяет приложению вызывать имена персистентных полей и свойств в классе метамодели.

Использование Criteria и Metamodel API для создания типобезопасных запросов

Базовая семантика запроса Criteria состоит из предложений `SELECT`, `FROM` и необязательного `WHERE`, аналогичных запросу JPQL. Запросы Criteria устанавливают эти предложения с использованием объектов Java, поэтому запрос может быть создан типобезопасным способом.

Создание запроса Criteria

Для создания используется интерфейс `jakarta.persistence.criteria.CriteriaBuilder`

- Запросов Criteria
- Выбор
- Выражения
- Предикаты
- Упорядочивание

Чтобы получить объект интерфейса `CriteriaBuilder`, вызовите метод `getCriteriaBuilder` объекта `EntityManager` или `EntityManagerFactory`.

В следующем коде показано, как получить объект `CriteriaBuilder` с помощью метода `EntityManager.getCriteriaBuilder`:

```java
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
```

Запросы Criteria строятся путём получения объекта следующего интерфейса:

```java
jakarta.persistence.criteria.CriteriaQuery
```

Объекты `CriteriaQuery` определяют конкретный запрос, который будет перемещаться по одному или нескольким объектам. Получите объекты `CriteriaQuery`, вызвав один из методов `CriteriaBuilder.createQuery`. Чтобы создать типобезопасные запросы, вызовите метод `CriteriaBuilder.createQuery` следующим образом:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
```

Тип объекта `CriteriaQuery` должен быть равен ожидаемому типу результата запроса. В предыдущем коде тип объекта установлен в `CriteriaQuery<Pet>` для запроса, который найдёт объекты сущности `Pet`.

Следующий фрагмент кода создаёт объект `CriteriaQuery` для запроса, который возвращает `String`:

```java
CriteriaQuery<String> cq = cb.createQuery(String.class);
```

Корень запроса

Для конкретного объекта `CriteriaQuery` корневая сущность запроса, от которой происходит вся навигация, называется корнем запроса. Это похоже на предложение `FROM` в запросе JPQL.

Создайте корень запроса, вызвав метод `from` объекта `CriteriaQuery`. Аргументом метода `from` является либо класс сущности, либо объект `EntityType<T>` для сущности.

Следующий код устанавливает сущность `Pet` корнем запроса:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
```

Следующий код устанавливает класс `Pet` корнем запроса, используя объект `EntityType<T>`:

```java
EntityManager em = ...;
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ = m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet_);
```

Запросы Criteria могут иметь несколько корней запросов. Обычно это происходит, когда запрос перемещается по нескольким объектам.

Следующий код имеет два объекта `Root`:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet1 = cq.from(Pet.class);
Root<Pet> pet2 = cq.from(Pet.class);
```

Запрос отношений с помощью объединений

Для запросов, которые переходят к связанным классам сущностей, запрос должен определить соединение со связанной сущностью, вызвав один из методов `From.join` у корневого объекта запроса или другого объекта соединения. Методы `join` аналогичны ключевому слову `JOIN` в JPQL.

Цель объединения использует класс Metamodel типа `EntityType<T>`, чтобы указать персистентное поле или свойство присоединяемого объекта.

Методы `join` возвращают объект типа `Join<X, Y>`, где `X` — исходная сущность, а `Y` является целью объединения. В следующем фрагменте кода `Pet` является исходной сущностью, `Owner` — целью, а `Pet_` — статически сгенерированным классом метамодели:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);

Root<Pet> pet = cq.from(Pet.class);
Join<Pet, Owner> owner = pet.join(Pet_.owners);
```

Вы можете образовать цепочку объединений, чтобы перейти к связанным объектам целевой сущности, не создавая объект `Join<X, Y>` для каждого объединения:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);

Root<Pet> pet = cq.from(Pet.class);
Join<Owner, Address> address = pet.join(Pet_.owners).join(Owner_.addresses);
```

Навигация по пути в запросах Criteria

Объекты `Path`, которые используются в предложениях `SELECT` и `WHERE` запроса Criteria, могут быть запросами корневых объектов, объектами объединения или другими объектами `Path`. Используйте метод `Path.get`, чтобы перейти к атрибутам сущностей запроса.

Аргументом метода `get` является соответствующий атрибут класса метамодели сущности. Атрибут может быть либо однозначным атрибутом, указанным `@SingularAttribute` в классе Metamodel, либо атрибутом со значением коллекции, указанным одним из `@CollectionAttribute`, `@SetAttribute`, `@ListAttribute` или `@MapAttribute`.

Следующий запрос возвращает имена всех домашних животных в хранилище данных. Метод `get` вызывается у корня запроса `pet` с атрибутом `name` класса метамодели `Pet_` сущности `Pet` в качестве аргумента:

JAVA

```
CriteriaQuery<String> cq = cb.createQuery(String.class);

Root<Pet> pet = cq.from(Pet.class);
cq.select(pet.get(Pet_.name));
```

Ограничение результатов запроса Criteria

Условия, которые устанавливаются путём вызова метода `CriteriaQuery.where`, могут ограничивать результаты запроса к объекту `CriteriaQuery`. Вызов метода `where` аналогичен установке условия `WHERE` в запросе JPQL.

Метод `where` выполняет объекты интерфейса `Expression` для ограничения результатов в соответствии с условиями выражений. Для создания объектов `Expression` используются методы, определённые в интерфейсах `Expression` и `CriteriaBuilder`.

Методы интерфейса Expression

Объект `Expression` используется в предложениях `SELECT`, `WHERE` или `HAVING` запроса. Таблица 43-1 показывает условные методы, которые могут использоваться с объектами `Expression`.

Таблица 43-1. Условные методы в интерфейсе Expression

Метод	Описание
`isNull`	Проверяет, является ли выражение null
`isNotNull`	Проверяет, не является ли выражение null
`in`	Проверяет, находится ли выражение в списке значений

В следующем запросе используется метод `Expression.isNull`, чтобы найти всех домашних животных, у которых атрибут `color` равен null:

JAVA

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(pet.get(Pet_.color).isNull());
```

В следующем запросе используется метод `Expression.in` для поиска всех коричневых и чёрных питомцев:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(pet.get(Pet_.color).in("brown", "black"));
```

Метод `in` также может проверять, является ли атрибут членом коллекции.

Методы Expression в интерфейсе CriteriaBuilder

Интерфейс `CriteriaBuilder` определяет дополнительные методы для создания выражений. Эти методы соответствуют арифметическим, строковым операторам, операторам даты, времени, функции case и функциям JPQL. Таблица 43-2 показывает условные методы, которые могут использоваться с объектами `CriteriaBuilder`.

Таблица 43-2. Условные методы в интерфейсе CriteriaBuilder

Условный метод	**Описание**
`equal`	Проверяет, равны ли два выражения
`notEqual`	Проверяет, что два выражения не равны
`gt`	Проверяет, что первое числовое выражение больше второго
`ge`	Проверяет, что первое числовое выражение больше или равно второму
`lt`	Проверяет, что первое числовое выражение меньше второго
`le`	Проверяет, что первое числовое выражение меньше или равно второму
`between`	Проверяет, находится ли значение первого выражения между вторым и третьим выражениями
`like`	Проверяет, соответствует ли выражение заданному шаблону

В следующем коде используется метод `CriteriaBuilder.equal`:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get(Pet_.name), "Fido"));
```

В следующем коде используется метод `CriteriaBuilder.gt`:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Date someDate = new Date(...);
cq.where(cb.gt(pet.get(Pet_.birthday), date));
```

В следующем коде используется метод `CriteriaBuilder.between`:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Date firstDate = new Date(...);
Date secondDate = new Date(...);
cq.where(cb.between(pet.get(Pet_.birthday), firstDate, secondDate));
```

В следующем коде используется метод `CriteriaBuilder.like`:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.like(pet.get(Pet_.name), "*do"));
```

Чтобы указать несколько условных предикатов, используйте методы составных предикатов интерфейса `CriteriaBuilder`, как показано в таблице 43-3.

Таблица 43-3. Методы составных предикатов в интерфейсе CriteriaBuilder

Метод	Описание
`and`	Логическое AND двух булевых выражений
`or`	Логическое OR двух булевых выражений
`not`	Логическое отрицание данного булева выражения

В следующем коде показано использование составных предикатов в запросах:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get(Pet_.name), "Fido")
        .and(cb.equal(pet.get(Pet_.color), "brown")));
```

Управление результатами запроса Criteria

Для запросов, которые возвращают более одного результата, часто полезно организовать эти результаты. Интерфейс `CriteriaQuery` определяет следующие методы упорядочения и группировки:

- Метод `orderBy` упорядочивает результаты запроса в соответствии с атрибутами объекта
- Метод `groupBy` группирует результаты запроса вместе в соответствии с атрибутами объекта, а метод `having` ограничивает эти группы в соответствии с условием

Упорядочение результатов

Чтобы упорядочить результаты запроса, вызовите метод `CriteriaQuery.orderBy`, передав объект `Order`. Чтобы создать объект `Order`, вызовите метод `CriteriaBuilder.asc` или `CriteriaBuilder.desc`. Метод `asc` используется для упорядочивания результатов по возрастанию значения переданного параметра выражения. Метод `desc` используется для упорядочивания результатов по убыванию значения переданного параметра выражения. Следующий запрос показывает использование метода `desc`:

JAVA

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet);
cq.orderBy(cb.desc(pet.get(Pet_.birthday)));
```

В этом запросе результаты будут упорядочены по дню рождения питомца от наивысшего к низшему. То есть домашние животные, рождённые в декабре, появятся в результатах выборки раньше домашних животных, рождённых в мае.

Следующий запрос показывает использование метода `asc` :

JAVA

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Owner, Address> address = pet.join(Pet_.owners).join(Owner_.address);
cq.select(pet);
cq.orderBy(cb.asc(address.get(Address_.postalCode)));
```

В этом запросе результаты будут упорядочены по почтовому индексу владельца домашнего животного от самого низкого до самого высокого. То есть домашние животные, владелец которых живёт по почтовому индексу 10001, появятся в результатах выборки раньше домашних животных, владелец которых живёт по почтовому индексу 91000.

Если в `orderBy` передаётся более одного объекта `Order` , приоритет определяется порядком их появления в списке аргументов `orderBy` . Первый объект `Order` имеет приоритет.

Следующий код упорядочивает результаты по нескольким критериям:

JAVA

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Pet, Owner> owner = pet.join(Pet_.owners);
cq.select(pet);
cq.orderBy(cb.asc(owner.get(Owner_.lastName)), owner.get(Owner_.firstName)));
```

Результаты этого запроса будут упорядочены в алфавитном порядке по фамилии владельца домашнего животного, а затем по имени.

Группировка результатов

Метод `CriteriaQuery.groupBy` разделяет результаты запроса на группы. Чтобы установить эти группы, передайте выражение для `groupBy` :

JAVA

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.groupBy(pet.get(Pet_.color));
```

Этот запрос возвращает все сущности `Pet` и группирует результаты по цвету питомца.

Используйте метод `CriteriaQuery.having` в сочетании с `groupBy` для фильтрации групп. Метод `having` принимает условное выражение в качестве параметра, ограничивает результат запроса в соответствии с условным выражением:

JAVA

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.groupBy(pet.get(Pet_.color));
cq.having(cb.in(pet.get(Pet_.color)).value("brown").value("blonde"));
```

В этом примере запрос группирует возвращённые сущности `Pet` по цвету, как в предыдущем примере. Однако единственными возвращаемыми группами будут объекты `Pet`, для атрибута `color` которых установлено значение `brown` или `blonde`. То есть домашние животные серого цвета не попадут в результаты запроса.

Выполнение запросов

Чтобы подготовить запрос к выполнению, создайте объект `TypedQuery<T>` с типом результата запроса, передав объект `CriteriaQuery` в `EntityManager.createQuery`.

Чтобы выполнить запрос, вызовите `getSingleResult` или `getResultList` для объекта `TypedQuery<T>`.

Однозначные результаты запроса

Используйте метод `TypedQuery<T>.getSingleResult` для выполнения запросов, которые возвращают один объект в результате:

JAVA

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
...
TypedQuery<Pet> q = em.createQuery(cq);
Pet result = q.getSingleResult();
```

Результаты запроса с коллекцией

Используйте метод `TypedQuery<T>.getResultList` для выполнения запросов, которые возвращают коллекцию объектов:

JAVA

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
...
TypedQuery<Pet> q = em.createQuery(cq);
List<Pet> results = q.getResultList();
```

Глава 44. Создание и использование строковых запросов Criteria

В этой главе описывается, как создавать слабо типизированные строковые запросы Criteria

Обзор строковых запросов Criteria

Строковые запросы Criteria — это запросы Java, которые используют строки, а не строго типизированные объекты метамодели, для указания атрибутов сущности при обходе иерархии данных. Строковые запросы строятся аналогично запросам метамодели, могут быть статическими или динамическими и могут выражать те же типы запросов и операций, что и строго типизированные запросы метамодели.

Строго типизированные запросы метамодели являются предпочтительным методом построения запросов Criteria.

Основным преимуществом строковых запросов над запросами метамодели является возможность создавать запросы Criteria во время разработки без необходимости создавать статические классы метамодели или иным образом получать доступ к динамически генерируемым классам метамодели.

Основным недостатком строковых запросов является отсутствие типобезопасности. Эта проблема может привести к ошибкам во время выполнения из-за несоответствия типов и может быть обнаружена во время разработки, если вы используете строго типизированные запросы метамодели.

Для получения информации о создании запросов criteria см. главу 43 *Использование Criteria API для создания запросов*.

Создание строковых запросов

Чтобы создать строковый запрос, укажите имена атрибутов классов сущностей непосредственно в виде строк вместо указания атрибутов класса метамодели. Например, этот запрос находит все объекты сущности `Pet` , у которых значение атрибута `name` равно `Fido` :

JAVA

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get("name"), "Fido"));
```

Имя атрибута указывается в виде строки. Этот запрос соответствует следующему запросу метамодели:

JAVA

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ = m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get(Pet_.name), "Fido"));
```

> Ошибки несоответствия типов в строковых запросах не будут появляться до тех пор, пока код не будет выполнен, в отличие от вышеупомянутого запроса метамодели, где несоответствия типов будут обнаружены во время компиляции.

Объединения указываются так же:

JAVA

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Owner, Address> address = pet.join("owners").join("addresses");
```

Все условные выражения, выражения методов, методы навигации по путям и методы фильтрации результатов, используемые в запросах метамодели, также могут использоваться в строковых запросах. В каждом случае атрибуты указываются с использованием строк. Например, вот строковый запрос, который использует выражение `in`:

JAVA

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(pet.get("color").in("brown", "black"));
```

Вот строковый запрос, который упорядочивает результаты в порядке убывания даты:

JAVA

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet);
cq.orderBy(cb.desc(pet.get("birthday")));
```

Выполнение строковых запросов

Строковые запросы выполняются аналогично строго типизированным запросам Criteria. Сначала создайте объект `jakarta.persistence.TypedQuery`, передав объект запроса criteria методу `EntityManager.createQuery`, затем вызовите либо `getSingleResult`, либо `getResultList` объекта запроса для выполнения запроса:

JAVA

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get("name"), "Fido"));
TypedQuery<Pet> q = em.createQuery(cq);
List<Pet> results = q.getResultList();
```

Глава 45. Управление параллельным доступом к данным объекта с помощью блокировки

В этой главе описывается, как обрабатывать одновременный доступ к данным сущностей, а также стратегии блокировки, доступные разработчикам приложений Jakarta Persistence.

Обзор блокировки сущностей и параллелизма

Доступ к данным объектов сущностей осуществляется параллельно, если к данным в источнике данных обращаются одновременно несколько приложений. Убедитесь, что целостность данных сохраняется при параллельном доступе к ним.

Когда данные обновляются в таблицах базы данных в транзакции, persistence provider предполагает, что система управления базами данных будет поддерживать кратковременные блокировки чтения и долгосрочные блокировки записи для поддержания целостности данных. Большинство persistence provider-ов задерживают запись в базу данных до конца транзакции, за исключением случаев, когда приложение явно требует принудительной записи (то есть приложение вызывает метод `EntityManager.flush` или выполняет запросы с установленным режимом очистки в `AUTO`).

По умолчанию persistence provider-ы используют оптимистичную блокировку, когда перед фиксацией изменений в данных persistence provider проверяет, что никакая другая транзакция не изменила или не удалила данные с момента их чтения из базы данных. Это достигается с помощью столбца версии в таблице базы данных с соответствующим атрибутом версии в классе сущности. Когда запись изменяется, значение версии увеличивается. Исходная транзакция проверяет атрибут version, и если данные были изменены другой транзакцией, будет выброшен `jakarta.persistence.OptimisticLockException` и выполнен откат исходной транзакции. Когда приложение задаёт режимы оптимистичной блокировки, persistence provider проверяет, что конкретный объект не изменился, поскольку он был прочитан из базы данных, даже если данные объекта не были изменены.

Пессимистичная блокировка идёт дальше, чем оптимистичная. При пессимистичной блокировке persistence provider создаёт транзакцию, которая получает долговременную блокировку данных до её завершения, что не позволяет другим транзакциям изменять или удалять данные до тех пор, пока блокировка не будет завершена. Пессимистичная блокировка — лучшая стратегия, чем оптимистичная блокировка, когда к базовым данным часто обращаются и изменяют многие транзакции.

Использование пессимистичних блокировок для объектов, которые не подлежат частой модификации, может привести к снижению производительности приложения.

Использование оптимистичной блокировки

Используйте аннотацию `jakarta.persistence.Version`, чтобы обозначить персистентное поле или свойство как атрибут версии объекта. Атрибут версии включает объект для оптимистичного управления параллелизмом. Persistence provider считывает и обновляет атрибут версии, когда объект изменяется во время транзакции. Приложение может прочитать атрибут версии, но не должно изменять значение.

> Хотя некоторые persistence provider-ы могут поддерживать оптимистичную блокировку для объектов, которые не имеют атрибутов версии, переносимые приложения всегда должны использовать объекты с атрибутами версии при использовании оптимистичной блокировки. Если приложение пытается заблокировать объект, у которого нет атрибута версии, а persistence provider не поддерживает оптимистичную блокировку для объектов без контроля версий, будет выдано `PersistenceException`.

Аннотация `@Version` имеет следующие требования.

- Только один атрибут `@Version` может быть определён для каждой сущности.
- Атрибут `@Version` должен находиться в первичной таблице для сущности, сопоставленной с несколькими таблицами.
- Тип атрибута `@Version` должен быть одним из следующих: `int`, `Integer`, `long`, `long`, `short`, `Short` или `java.sql.Timestamp`.

В следующем фрагменте кода показано, как определить атрибут версии в объекте с персистентными полями:

```java
@Version
protected int version;
```

В следующем фрагменте кода показано, как определить атрибут версии в объекте с персистентными свойствами:

```java
@Version
protected Short getVersion() { ... }
```

Режимы блокировки

Приложение может повысить уровень блокировки для объекта, указав использование режимов блокировки. Режимы блокировки могут быть указаны для повышения уровня оптимистичной блокировки или для запросов на использование пессимистичних блокировок.

Использование режимов оптимистичной блокировки заставляет persistence provider проверять атрибуты версии для сущностей, которые были прочитаны (но не изменены) во время транзакции, а также для сущностей, которые были обновлены.

Использование режимов пессимистичной блокировки указывает, что persistence provider должен немедленно получить долгосрочные блокировки чтения или записи данных базы данных, соответствующих объекту сущности.

Можно задать режим блокировки для операции сущности, указав один из режимов блокировки, определённых в перечислимом типе `jakarta.persistence.LockModeType` в таблице 45-1.

Таблица 45-1. Режимы блокировки для параллельного доступа к объекту

Режим блокировки	**Описание**
`OPTIMISTIC`	Получение оптимистичной блокировки чтения для всех сущностей с атрибутами версии.

Режим блокировки	Описание
`OPTIMISTIC_FORCE_INCREMENT`	Получение оптимистичной блокировки чтения для всех объектов с атрибутами версии и увеличение значения атрибута версии.
`PESSIMISTIC_READ`	Немедленное получение долговременной блокировки чтения данных, чтобы предотвратить их изменение или удаление. Другие транзакции могут читать данные, пока поддерживается блокировка, но не могут изменять или удалять данные. Persistence provider-у разрешается получать блокировку записи в базу данных при запросе блокировки чтения, но не наоборот.
`PESSIMISTIC_WRITE`	Немедленное получение долговременной блокировки записи данных, чтобы предотвратить чтение, изменение или удаление данных.
`PESSIMISTIC_FORCE_INCREMENT`	Немедленное получение долговременной блокировки данных, чтобы предотвратить изменение или удаление данных, и увеличение атрибута версии у версионных сущностей.
`READ`	Синоним для `OPTIMISTIC`. Использование `LockModeType.OPTIMISTIC` предпочтительнее для новых приложений.
`WRITE`	Синоним для `OPTIMISTIC_FORCE_INCREMENT`. Использование `LockModeType.OPTIMISTIC_FORCE_INCREMENT` предпочтительнее для новых приложений.
`NONE`	Никакой дополнительной блокировки не произойдёт с данными в базе данных.

Установка режима блокировки

Чтобы указать режим блокировки, используйте один из следующих методов:

1. Вызовите метод `EntityManager.lock`, перейдя в один из режимов блокировки:

```java
EntityManager em = ...;
Person person = ...;
em.lock(person, LockModeType.OPTIMISTIC);
```

2. Вызовите один из методов `EntityManager.find`, которые принимают режим блокировки в качестве параметра:

JAVA

```
EntityManager em = ...;
String personPK = ...;
Person person = em.find(Person.class, personPK,
    LockModeType.PESSIMISTIC_WRITE);
```

3. Вызовите один из методов `EntityManager.refresh`, которые принимают режим блокировки в качестве параметра:

JAVA

```
EntityManager em = ...;
String personPK = ...;
Person person = em.find(Person.class, personPK);
...
em.refresh(person, LockModeType.OPTIMISTIC_FORCE_INCREMENT);
```

4. Вызовите метод `Query.setLockMode` или `TypedQuery.setLockMode`, передав режим блокировки в качестве параметра:

JAVA

```
Query q = em.createQuery(...);
q.setLockMode(LockModeType.PESSIMISTIC_FORCE_INCREMENT);
```

5. Добавьте элемент `lockMode` в аннотацию `@NamedQuery`:

JAVA

```
@NamedQuery(name="lockPersonQuery",
  query="SELECT p FROM Person p WHERE p.name LIKE :name",
  lockMode=PESSIMISTIC_READ)
```

Использование пессимистичной блокировки

Версионные объекты, а также объекты, которые не имеют атрибутов версии, могут быть заблокированы пессимистично.

Чтобы блокировать объекты пессимистично, установите режим блокировки на `PESSIMISTIC_READ`, `PESSIMISTIC_WRITE` или `PESSIMISTIC_FORCE_INCREMENT`.

Если пессимистичная блокировка не может быть получена для записей базы данных и неудачная блокировка данных приводит к откату транзакции, генерируется `PessimisticLockException`. Если пессимистичная блокировка не может быть получена, но сбой блокировки не приводит к откату транзакции, выдаётся `LockTimeoutException`.

Пессимистичная блокировка версионного объекта с помощью `PESSIMISTIC_FORCE_INCREMENT` приводит к увеличению атрибута версии, даже если данные объекта не изменены. При пессимистичной блокировке версионного объекта persistence provider будет выполнять проверки версий, которые происходят во время оптимистичной блокировки, и в случае сбоя проверки версии будет выдано `OptimisticLockException`. Попытка заблокировать неверсионную сущность с помощью `PESSIMISTIC_FORCE_INCREMENT` не переносима и может привести к `PersistenceException`, если persistence provider не поддерживает оптимистичных блокировок для неверсионных сущностей. Блокировка версионного объекта с помощью `PESSIMISTIC_WRITE` приводит к увеличению атрибута версии, если транзакция была успешно зафиксирована.

Тайм-ауты пессимистичной блокировки

Используйте свойство `jakarta.persistence.lock.timeout`, чтобы указать время в миллисекундах, в течение которого persistence provider должен ожидать получения блокировки таблиц базы данных. Если время, необходимое для получения блокировки, превышает значение этого свойства, будет выброшено

`LockTimeoutException`, но текущая транзакция не будет помечена для отката. Если для этого свойства установлено значение `0`, persistence provider должен выдать `LockTimeoutException`, если он не может немедленно получить блокировку.

> Переносимые приложения не должны полагаться на настройку `jakarta.persistence.lock.timeout`, потому что стратегия блокировки и база данных могут отменять установленное значение тайм-аута. Значение `jakarta.persistence.lock.timeout` является подсказкой, а не контрактом.

Это свойство можно установить программно, передав его методам `EntityManager`-а, которые позволяют указывать режимы блокировки: методы `Query.setLockMode` и `TypedQuery.setLockMode`, аннотация `@NamedQuery` и метод `Persistence.createEntityManagerFactory`. Оно также может быть установлено как свойство в дескрипторе развёртывания `persistence.xml`.

Если `jakarta.persistence.lock.timeout` установлен в нескольких местах, значение будет определяться в следующем порядке:

1. Аргумент одного из методов `EntityManager` или `Query`
2. Параметр в аннотации `@NamedQuery`
3. Аргумент метода `Persistence.createEntityManagerFactory`
4. Значение в дескрипторе развёртывания `persistence.xml`

Глава 46. Создание планов выполнения с помощью графов сущностей

В этой главе объясняется, как использовать графы сущностей для создания планов выборки для персистентных операций и запросов.

Обзор использования планов выполнения и графов сущностей

Графы сущностей — это шаблоны для определённого запроса или персистентной операции. Они используются при создании планов выполнения или групп персистентных полей, которые извлекаются одновременно. Разработчики приложений используют планы выборки для группировки связанных персистентных полей для повышения производительности во время выполнения.

По умолчанию поля или свойства объекта извлекаются отложенно (lazy). Разработчики указывают поля или свойства как часть плана выполнения, и persistence provider использовать для них раннее (eager) извлечение.

Например, приложение электронной почты, которое хранит сообщения в виде объектов сущности `EmailMessage`, отдаёт приоритет выборке одних полей над другими. Отправитель, тема и дата будут просматриваться чаще всего при просмотре почтового ящика и при отображении сообщения. Сущность `EmailMessage` имеет коллекцию связанных сущностей `EmailAttachment`. Из соображений производительности вложения не должны выбираться до тех пор, пока они не потребуются, но имена файлов вложения важны. Разработчик этого приложения может составить план выполнения, задав раннее (eager) извлечение важных полей из `EmailMessage` и `EmailAttachment`, в то же время отложенно (lazy) извлекая данные с более низким приоритетом.

Основы графа сущностей

Графы сущностей могут быть созданы аннотациями или в дескрипторе развёртывания статически или стандартными интерфейсами динамически.

Граф сущностей можно использовать с методом `EntityManager.find` или как часть запроса JPQL или Criteria, указав граф сущностей в качестве подсказки для операции или запроса.

Графы сущностей имеют атрибуты, соответствующие полям, для которых будет использоваться раннее (eager) извлечение во время выполнения `find` или операции запроса. Поля первичного ключа и версии класса сущностей всегда выбираются и не требуют явного добавления к графу сущностей.

Граф сущностей по умолчанию

По умолчанию все поля в сущности извлекаются отложенно, если атрибут `fetch` метаданных сущности не устанавлен в `jakarta.persistence.FetchType.EAGER`. Граф сущностей по умолчанию состоит из всех полей сущности, для которых установлено раннее (eager) извлечение.

Например, следующая сущность `EmailMessage` указывает, что для некоторых полей будет выполняться раннее (eager) извлечение:

JAVA

```java
@Entity
public class EmailMessage implements Serializable {
    @Id
    String messageId;
    @Basic(fetch=EAGER)
    String subject;
    String body;
    @Basic(fetch=EAGER)
    String sender;
    @OneToMany(mappedBy="message", fetch=LAZY)
    Set<EmailAttachment> attachments;
    ...
}
```

Граф сущностей по умолчанию для этой сущности будет содержать поля `messageId`, `subject` и `sender`, но не поля `body` и `attachments`.

Использование графов сущностей в персистентных операциях

Графы сущностей используются путём создания объекта интерфейса `jakarta.persistence.EntityGraph`, путём вызова либо `EntityManager.getEntityGraph` для именованных графов сущностей, либо `EntityManager.createEntityGraph` для создания динамических графов сущностей.

Именованный граф сущностей — это граф сущностей, определённый аннотацией `@NamedEntityGraph`, применяемой к классам сущностей или элементом `named-entity-graph` в дескрипторе развёртывания приложения. Именованные графы сущностей, определённые в дескрипторе развёртывания, переопределяют любые графы сущностей на основе аннотаций с тем же именем.

Созданный граф сущностей может быть либо графом выборки, либо графом загрузки.

Графы выборки

Чтобы указать граф выборки, установите свойство `jakarta.persistence.fetchgraph` при выполнении `EntityManager.find` или операции запроса. Граф выборки состоит только из полей, явно указанных в объекте `EntityGraph`, и игнорирует настройки графа сущностей по умолчанию.

В следующем примере граф сущностей по умолчанию игнорируется, и в динамически создаваемый граф выборки включается только поле `body`:

JAVA

```java
EntityGraph<EmailMessage> eg = em.createEntityGraph(EmailMessage.class);
eg.addAttributeNodes("body");
...
Properties props = new Properties();
props.put("jakarta.persistence.fetchgraph", eg);
EmailMessage message = em.find(EmailMessage.class, id, props);
```

Графы загрузки

Чтобы указать граф зугрузки, установите свойство `jakarta.persistence.loadgraph` при выполнении `EntityManager.find` или операции запроса. Граф загрузки состоит из полей, явно указанных в объекте `EntityGraph`, а также любых полей в графе сущностей по умолчанию.

В следующем примере динамически создаваемый граф загрузки содержит все поля в графе сущностей по умолчанию плюс поле `body`:

JAVA

```
EntityGraph<EmailMessage> eg = em.createEntityGraph(EmailMessage.class);
eg.addAttributeNodes("body");
...
Properties props = new Properties();
props.put("jakarta.persistence.loadgraph", eg);
EmailMessage message = em.find(EmailMessage.class, id, props);
```

Использование именованных графов сущностей

Именованные графы сущностей создаются с использованием аннотаций, применяемых к классам сущностей или элементу `named-entity-graph` и его подэлементам в дескрипторе развёртывания приложения. Persistence provider будет сканировать все именованные графы сущностей, определённые в аннотациях и в XML приложения. Набор сущностей именованного графа с использованием аннотации может быть переопределён с использованием `named-entity-graph`.

Применение аннотаций именованных графов сущностей к классам сущностей

Аннотация `jakarta.persistence.NamedEntityGraph` определяет один именованный граф сущности и применяется на уровне класса. Для класса можно определить несколько аннотаций `@NamedEntityGraph`, добавив их в аннотацию уровня класса `jakarta.persistence.NamedEntityGraphs`.

Аннотация `@NamedEntityGraph` должна быть применена к корню графа сущностей. То есть, если `EntityManager.find` или операция запроса имеет в качестве корневого объекта класс `EmailMessage`, именованный граф сущностей, используемый в операции, должен быть определён в классе `EmailMessage`:

JAVA

```
@NamedEntityGraph
@Entity
public class EmailMessage {
    @Id
    String messageId;
    String subject;
    String body;
    String sender;
}
```

В этом примере класс `EmailMessage` имеет аннотацию `@NamedEntityGraph` для определения именованного графа сущностей, который по умолчанию соответствует имени класса `EmailMessage`. Никакие поля не включены в аннотацию `@NamedEntityGraph` как атрибуты, и поля не снабжены метаданными для установки типа выборки, поэтому единственное поле, для которого будет выполняться раннее (eager) извлечение в графе загрузки или в графе выборки — это `messageId`.

Атрибуты именованного графа сущностей — это поля сущности, которые должны быть включены в граф сущностей. Добавьте поля в граф сущности, указав их в элементе `attributeNodes` аннотации `@NamedEntityGraph` с `jakarta.persistence.NamedAttributeNode`:

JAVA

```
@NamedEntityGraph(name="emailEntityGraph", attributeNodes={
    @NamedAttributeNode("subject"),
    @NamedAttributeNode("sender")
})
@Entity
public class EmailMessage { ... }
```

В этом примере имя именованного графа сущностей — `emailEntityGraph` включает в себя поля `subject` и `sender`.

Несколько определений `@NamedEntityGraph` могут быть применены к классу путём группировки их в аннотации `@NamedEntityGraphs`.

В следующем примере два графа сущностей определены в классе `EmailMessage`. Один из них предназначен для области предварительного просмотра, которая выбирает только отправителя, тему и текст сообщения. Другой для полного просмотра сообщения, включая любые вложения сообщения:

JAVA

```
@NamedEntityGraphs({
    @NamedEntityGraph(name="previewEmailEntityGraph", attributeNodes={
        @NamedAttributeNode("subject"),
        @NamedAttributeNode("sender"),
        @NamedAttributeNode("body")
    }),
    @NamedEntityGraph(name="fullEmailEntityGraph", attributeNodes={
        @NamedAttributeNode("sender"),
        @NamedAttributeNode("subject"),
        @NamedAttributeNode("body"),
        @NamedAttributeNode("attachments")
    })
})
@Entity
public class EmailMessage { ... }
```

Получение объектов EntityGraph из именованных графов сущностей

Используйте метод `EntityManager.getEntityGraph`, передавая имя именованного графа сущностей для получение объекта `EntityGraph` для этого графа сущностей:

JAVA

```
EntityGraph<EmailMessage> eg = em.getEntityGraph("emailEntityGraph");
```

Использование графов сущностей в запросах

Чтобы указать графы сущностей для типизированных и нетипизированных запросов, вызовите метод `setHint` для объекта запроса и укажите либо `jakarta.persistence.loadgraph`, либо `jakarta.persistence.fetchgraph` как имя свойства и объект `EntityGraph` как значение:

JAVA

```
EntityGraph<EmailMessage> eg = em.getEntityGraph("previewEmailEntityGraph");
List<EmailMessage> messages = em.createNamedQuery("findAllEmailMessages")
        .setParameter("mailbox", "inbox")
        .setHint("jakarta.persistence.loadgraph", eg)
        .getResultList();
```

В этом примере `previewEmailEntityGraph` используется для именованного запроса `findAllEmailMessages`.

Типизированные запросы используют ту же технику:

JAVA

```
EntityGraph<EmailMessage> eg = em.getEntityGraph("previewEmailEntityGraph");

CriteriaQuery<EmailMessage> cq = cb.createQuery(EmailMessage.class);
Root<EmailMessage> message = cq.from(EmailMessage.class);
TypedQuery<EmailMessage> q = em.createQuery(cq);
q.setHint("jakarta.persistence.loadgraph", eg);
List<EmailMessage> messages = q.getResultList();
```

Глава 47. Использование кэша второго уровня в приложениях Jakarta Persistence

В этой главе объясняется, как изменить параметры режима кэширования второго уровня для повышения производительности приложений, использующих Jakarta Persistence.

Обзор кэша второго уровня

Кэш второго уровня — это локальное хранилище данных сущностей, управляемых persistence provider-ом для повышения производительности приложений. Кэш второго уровня помогает повысить производительность, избегая дорогостоящих вызовов базы данных, сохраняя данные объекта локально для приложения. Кэш второго уровня обычно прозрачен для приложения, так как он управляется persistence provider-ом и лежит в основе контекста персистентности приложения. То есть приложение считывает и фиксирует данные с помощью обычных операций entity manager-а, не зная о кэше.

> Persistence provider-ы не обязаны поддерживать кэш второго уровня. Переносимые приложения не должны полагаться на поддержку кэша второго уровня со стороны persistence provider-а.

Кэш второго уровня для юнита персистентности может быть сконфигурирован в один из нескольких режимов работы. Следующие настройки режима кэширования определяются Jakarta Persistence.

Таблица 47-1. Настройки режима кэширования для кэша второго уровня

Настройка режима кэширования	**Описание**
`ALL`	Все данные сущности сохраняются в кэше второго уровня для этого юнита персистентности.
`NONE`	Данные не кэшируются для юнита персистентности. Persistence provider не должен кэшировать какие-либо данные.
`ENABLE_SELECTIVE`	Включено кэширование для сущностей, для которых оно явно установлено аннотациями `@Cacheable`.
`DISABLE_SELECTIVE`	Включено кэширование для всех сущностей, кроме тех, которым оно явно было запрещено аннотацией `@Cacheable(false)`.
`UNSPECIFIED`	Поведение кэширования для юнита персистентности не определено. Будет использовано поведение кэширования по умолчанию для persistence provider.

Одним из следствий использования кэша второго уровня в приложении является то, что данные могли измениться в таблицах базы данных, а значение в кэше — нет. Эта ситуация называется устаревшим чтением. Чтобы избежать устаревшего чтения, используются следующие стратегии:

- Изменение кэша второго уровня одним из параметров режима кэша
- Управление тем, какие объекты могут быть кэшированы (см. Контроль кэширования сущностей)

- Изменение режима извлечения или сохранения из кэша (см. Установка режимов извлечения и сохранения в кэш)

Какая из этих стратегий будет наиболее подходящей и позволит избежать устаревшего чтения, зависит от приложения.

Контроль кэширования сущностей

Аннотация `jakarta.persistence.Cacheable` используется для указания, что класс сущности и все подклассы могут кэшироваться с использованием режимов кэша `ENABLE_SELECTIVE` или `DISABLE_SELECTIVE`. Дочерние классы могут переопределить настройку `@Cacheable`, добавив аннотацию `@Cacheable` с изменённым значением.

Чтобы указать, что объект может быть кэширован, добавьте аннотацию `@Cacheable` на уровне класса:

JAVA

```
@Cacheable
@Entity
public class Person { ... }
```

По умолчанию значение аннотации `@Cacheable` установлено в `true`. Следующий пример полностью аналогичен предыдущему:

JAVA

```
@Cacheable(true)
@Entity
public class Person{ ... }
```

Чтобы указать, что объект не должен кэшироваться, добавьте аннотацию `@Cacheable` и установите для неё значение `false`:

JAVA

```
@Cacheable(false)
@Entity
public class OrderStatus { ... }
```

Когда установлен режим кэширования `ENABLE_SELECTIVE`, persistence provider будет кэшировать все объекты сущностей с аннотацией `@Cacheable(true)` и его дочерних классов, которые не были переопределены. Persistence provider не будет кэшировать объекты сущностей, помеченных `@Cacheable(false)` или не имеющих аннотации `@Cacheable`. Таким образом, в режиме `ENABLE_SELECTIVE` будут кэшироваться только те объекты, которые были явно помечены для кэширования с использованием аннотации `@Cacheable`.

Когда установлен режим кэширования `DISABLE_SELECTIVE`, persistence provider будет кэшировать объекты сущностей, которые не имеют аннотации `@Cacheable(false)`. Объекты, у которых нет аннотаций `@Cacheable`, и объекты с аннотацией `@Cacheable(true)` будут кэшироваться. Таким образом, режим `DISABLE_SELECTIVE` будет кэшировать объекты всех сущностей, для которых кэширование не запрещено явно.

Если режим кэширования установлен на `UNDEFINED` или оставлен неустановленным, поведение сущностей, аннотированных `@Cacheable`, не определено. Если для режима кэширования установлено значение `ALL` или `NONE`, значение аннотации `@Cacheable` игнорируется persistence provider-ом.

Задание настроек режима кэширования для повышения производительности

Чтобы настроить параметры режима кэширования для юнита персистентности, укажите один из режимов кэширования в качестве значения элемента `shared-cache-mode` в дескрипторе развёртывания `persistence.xml`:

XML

```
<persistence-unit name="examplePU" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>java:comp/DefaultDataSource</jta-data-source>
    <shared-cache-mode>DISABLE_SELECTIVE</shared-cache-mode>
</persistence-unit>
```

> Поскольку поддержка кэша второго уровня не требуется спецификацией Jakarta Persistence, установка режима кэширования второго уровня в `persistence.xml` не будет иметь никакого эффекта при использовании провайдера персистентности, который не реализует кэш второго уровня.

Кроме того, можно указать режим общего кэша, установив свойство `jakarta.persistence.sharedCache.mode` в одну из настроек режима общего кэша:

JAVA

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory(
        "myExamplePU", new Properties().add(
            "jakarta.persistence.sharedCache.mode", "ENABLE_SELECTIVE"));
```

Установка режимов извлечения и сохранения в кэш

Если вы включили кэш второго уровня для юнита персистентности, установив режим общего кэша, можете дополнительно изменить поведение кэша второго уровня, установив свойства `jakarta.persistence.cache.retrieveMode` и `jakarta.persistence.cache.storeMode`. Эти свойства могут быть установлены на уровне контекста персистентности передачей имени и значения свойства методу `EntityManager.setProperty` или установкой их для каждой операции `EntityManager` (`EntityManager.find` или `EntityManager.refresh`) или на уровне запроса.

Режим извлечения из кэша

Режим извлечения кэша, заданный в `jakarta.persistence.retrieveMode`, управляет тем, как данные считываются из кэша для вызовов метода `EntityManager.find` и из запросов.

Вы можете установить `retrieveMode` в одну из констант, определённых свойством `jakarta.persistence.CacheRetrieveMode` перечислимого типа: `USE` (по умолчанию) или `BYPASS`.

Когда для свойства установлено значение `USE`, данные извлекаются из кэша второго уровня, если он доступен. Если данные не находятся в кэше, persistence provider будет читать их из базы данных.

Когда для свойства установлено значение `BYPASS`, кэш второго уровня не используется, а для извлечения данных выполняется обращение к базе данных.

Режим сохранения в кэше

Режим хранилища кэша, заданный `jakarta.persistence.storeMode` управляет тем, как данные хранятся в кэше.

Для свойства `storeMode` можно задать одну из констант, определённых перечислимым типом `jakarta.persistence.CacheStoreMode`: `USE` (по умолчанию), `BYPASS` или `REFRESH`.

Если для свойства установлено значение `USE`, данные кэша создаются или обновляются, когда данные считываются или передаются в базу данных. Если данные уже находятся в кэше, установка режима сохранения `USE` не приведёт к принудительному обновлению при чтении данных из базы данных.

Если для свойства установлено значение `BYPASS`, данные, считанные или переданные в базу данных, не вставляются и не обновляются в кэше. То есть кэш неизменен.

Если для свойства установлено значение `REFRESH`, данные кэша создаются или обновляются, когда данные считываются или передаются в базу данных, и при чтении базы данных данные в кэше обновляются принудительно.

Установка режима извлечения из кэша или режима сохранения

Чтобы установить режим извлечения или сохранения в кэш для контекста персистентности, вызовите метод `EntityManager.setProperty`, передав имя и значение свойства:

JAVA

```
EntityManager em = ...;
em.setProperty("jakarta.persistence.cache.storeMode", "BYPASS");
```

Чтобы установить режим извлечения или сохранения в кэш при вызове методов `EntityManager.find` или `EntityManager.refresh`, сначала создайте `Map<String, Object>` и добавьте пару имя-значение следующим образом:

JAVA

```
EntityManager em = ...;
Map<String, Object> props = new HashMap<String, Object>();
props.put("jakarta.persistence.cache.retrieveMode", "BYPASS");
String personPK = ...;
Person person = em.find(Person.class, personPK, props);
```

Режим извлечения из кэша игнорируется при вызове метода `EntityManager.refresh`, так как вызовы `refresh` всегда приводят к чтению данных из базы данных, а не из кэша.

Чтобы установить режим извлечения или сохранения при использовании запросов, вызовите методы `Query.setHint` или `TypedQuery.setHint` в зависимости от типа запроса:

JAVA

```
EntityManager em = ...;
CriteriaQuery<Person> cq = ...;
TypedQuery<Person> q = em.createQuery(cq);
q.setHint("jakarta.persistence.cache.storeMode", "REFRESH");
...
```

Установка режима сохранения или извлечения в запросе или при вызове метода `EntityManager.find` или `EntityManager.refresh` отменяет настройку entity manager-а.

Программный контроль кэша второго уровня

Интерфейс `jakarta.persistence.Cache` определяет методы программного взаимодействия с кэшем второго уровня.

Обзор интерфейса jakarta.persistence.Cache

Интерфейс `Cache` определяет методы для выполнения следующих действий:

- Проверка наличия данных конкретной сущности в кэше

- Удаление всех объектов заданной сущности из кэша
- Удаление всех объектов сущности (включая объекты дочерних классов) из кэша
- Очистка кэша с удалением данных всех объектов

Если кэш второго уровня отключён, вызовы методов интерфейса `Cache` не принесут никакого эффекта, за исключением `contains`, который всегда вернёт `false`.

Проверка наличия данных объекта сущности в кэше

Чтобы узнать, находится ли данный объект в кэше второго уровня:

1. Вызовите метод `Cache.contains`. Метод `contains` возвращает `true`, если данные объекта находятся в кэше, и `false` в противном случае:

```java
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
String personPK = ...;
if (cache.contains(Person.class, personPK)) {
  // данные кэшированы
} else {
  // данные НЕ кэшированы
}
```

Удаление всех объектов заданной сущности из кэша

Чтобы удалить конкретный объект сущности или все объекты заданной сущности из кэша второго уровня:

1. Вызовите один из методов `Cache.evict`.

   a. Чтобы удалить конкретный объект сущности из кэша, вызовите метод `evict` и передайте класс сущности и первичный ключ объекта:

```java
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
String personPK = ...;
cache.evict(Person.class, personPK);
```

   b. Чтобы удалить все объекты определённой сущности с её дочерними классами, вызовите метод `evict` и укажите класс сущности:

```java
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
cache.evict(Person.class);
```

Все объекты сущности `Person` будут удалены из кэша. Если у сущности `Person` есть дочерний класс `Student`, вызовы вышеуказанного метода также удалят все объекты `Student` из кэша.

Очистка кэша

Чтобы полностью очистить кэш второго уровня, вызовите метод `Cache.evictAll`:

```java
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
cache.evictAll();
```

Часть IX: Обмен сообщениями

Часть IX представляет обмен сообщениями.

Глава 48. Концепции обмена сообщениями в Jakarta

Эта глава представляет собой введение в Jakarta Messaging — Java API, который позволяет приложениям создавать, отправлять, получать и читать сообщения, используя надёжную, асинхронную, слабосвязанную связь.

Обзор Jakarta Messaging

Этот обзор определяет концепцию обмена сообщениями, описывает Jakarta Messaging и где его можно использовать, а также объясняет, как Jakarta Messaging работает на платформе Jakarta EE.

Что такое обмен сообщениями?

Обмен сообщениями — это метод связи между программными компонентами или приложениями. Система обмена сообщениями является одноранговым средством: клиент обмена сообщениями может отправлять сообщения и получать сообщения от любого другого клиента. Каждый клиент подключается к агенту обмена сообщениями, который предоставляет средства для создания, отправки, получения и чтения сообщений.

Обмен сообщениями обеспечивает распределённое слабосвязное взаимодействие. Компонент отправляет сообщение в пункт назначения, и получатель может извлечь сообщение из пункта назначения. Слабосвязным взаимодействие делает то обстоятельство, что пункт назначения — это единственное, что есть общего между отправителем и получателем. Отправитель и получатель не обязаны быть доступны для связи одновременно. Фактически, отправитель может ничего не знать о получателе. Получатель также ничего не обязан знать об отправителе. Отправитель и получатель должны знать, какой формат сообщения и пункт назначения использовать. В этом отношении обмен сообщениями отличается от сильносвязных технологий, таких как Remote Method Invocation (RMI), которые требуют от приложения знания методов удалённого приложения.

Обмен сообщениями также отличается от электронной почты, которая является способом общения между людьми или между приложениями и людьми. Обмен сообщениями используется для связи между приложениями или программными компонентами.

Что такое Jakarta Messaging?

Jakarta Messaging — это API Java, позволяющий приложениям создавать, отправлять, получать и читать сообщения. Jakarta Messaging определяет общий набор интерфейсов и связанной с ними семантики, которые позволяют программам Java взаимодействовать с другими реализациями обмена сообщениями.

Jakarta Messaging минимизирует набор понятий, которые программист должен научиться использовать в продуктах обмена сообщениями, но предоставляет достаточно возможностей для поддержки и сложных приложений. Она также стремится к максимальной переносимости приложений обмена сообщениями между провайдерами.

Jakarta Messaging обеспечивает взаимодействие, которое не только слабосвязно, но и

- асинхронно: принимающий клиент не обязательно получает сообщения одновременно с их отправкой отправителем. Отправляющий клиент может отправить их и перейти к другим задачам. Принимающий клиент может получить их намного позже.
- надёжно: провайдер сообщений, который реализует Jakarta Messaging, может гарантировать, что сообщение будет доставлено один и только один раз. Более низкие уровни надёжности доступны для приложений, которые могут позволить пропустить сообщения или получать дубликаты сообщений.

Текущая версия спецификации Jakarta Messaging — версия 3.0.

Когда можно использовать Jakarta Messaging?

Поставщик корпоративных приложений может выбрать API обмена сообщениями вместо сильносвязного API, такого как удалённый вызов процедур (RPC), при следующих обстоятельствах.

- Поставщик хочет, чтобы компоненты не зависели от информации об интерфейсах других компонентов, то есть чтобы компоненты были легко заменимы.
- Поставщик хочет, чтобы приложение работало независимо от того, запущены ли все компоненты одновременно.
- Бизнес-модель приложения позволяет компоненту отправлять информацию другому и продолжать работу без получения немедленного ответа.

Например, компоненты корпоративного приложения для производителя автомобилей могут использовать Jakarta Messaging в следующих ситуациях.

- Компонент инвентаризации может отправлять сообщение производственному компоненту, когда уровень запасов продукта ниже определённого уровня, чтобы завод мог производить больше автомобилей.
- Компонент фабрики может отправить сообщение компонентам деталей, чтобы завод мог скомпоновать необходимые детали.
- Компоненты деталей, в свою очередь, могут отправлять сообщения в свои компоненты инвентаризации и заказа, чтобы тем самым пополнять свои запасы и заказывать материалы у поставщиков.
- Компоненты учёта и деталей могут отправлять сообщения в компонент учёта для обновления номеров счетов бухучёта.
- Компания может публиковать обновлённые каталоги продукции в своих отделах продаж.

Использование обмена сообщениями в этих задачах позволяет различным компонентам эффективно взаимодействовать друг с другом без привязки к сети и другим ресурсам. Рисунок 48-1 иллюстрирует, как может работать этот простой пример.

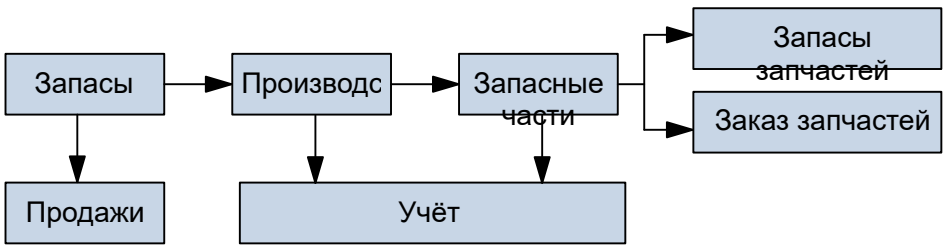

Рис. 48-1. Обмен сообщениями в корпоративном приложении

Производство — это только один пример того, как предприятие может использовать Jakarta Messaging API. Обмен сообщениями может быть успешно использован в приложениях розничной торговли, финансовых услуг, служб здравоохранения и многих других.

Как Jakarta Messaging работает с платформой Jakarta EE?

Когда JMS была впервые представлена, её первоочередной целью было предоставить Java-приложениям доступ к существующим системам, ориентированным на сообщения (MOM — messaging-oriented middleware). С тех пор многие разработчики ПО адаптировали и дополнили JMS, поэтому Jakarta Messaging сообщениями теперь может обеспечить все возможности обмена сообщениями для предприятия.

Jakarta Messaging является неотъемлемой частью платформы Jakarta EE, и разработчики приложений могут использовать обмен сообщениями с компонентами Jakarta EE. Jakarta Messaging 2.0 входит в состав Jakarta EE 8.

Jakarta Messaging на платформе Jakarta EE имеет следующие возможности.

- Клиенты приложений, компоненты Jakarta Enterprise Beans и веб-компоненты могут отправлять или синхронно получать сообщения Jakarta Messaging. Кроме того, клиентские приложения могут установить слушатель сообщений, который позволяет асинхронно доставлять сообщения JMS, уведомляя о доступности сообщения.
- Компоненты, управляемые сообщениями, являются EJB-компонентами особого рода и обеспечивают асинхронную обработку сообщений EJB-контейнером. Сервер приложений обычно объединяет управляемые сообщениями бины в пулы для совместной обработки ими сообщений.
- Операции отправки и получения сообщений могут участвовать в транзакциях Jakarta, что позволяет осуществлять операции Jakarta Messaging и доступ к базе данных в рамках одной транзакции.

Jakarta Messaging расширяет другие части платформы Jakarta EE, упрощая корпоративную разработку, обеспечивая слабосвязанное, надёжное, асинхронное взаимодействие между компонентами Jakarta EE и устаревшими системами, способными передавать сообщения. Разработчик может легко добавить новое поведение в приложение Jakarta EE с существующими бизнес-событиями, добавив новый управляемый событиями компонент для работы с конкретными бизнес-событиями. Кроме того, платформа Jakarta EE расширяет возможности Jakarta Messaging, оказывая поддержку Jakarta Transactions и обеспечивая параллельную обработку сообщений. Для получения дополнительной информации см. спецификацию Jakarta Enterprise Beans, v4.0.

Провайдер Jakarta Messaging может быть интегрирован с сервером приложений с помощью Jakarta Connectors. Доступ к провайдеру сообщений через адаптер ресурсов. Эта возможность позволяет разработчикам ПО создавать провайдеров сообщений, которые могут быть подключены к нескольким серверам приложений, а также позволяет серверам приложений поддерживать нескольких провайдеров сообщений. Дополнительная информация приведена в спецификации Jakarta Connectors, v2.0.

Основные концепции Jakarta Messaging

В этом разделе представлены основные понятия Jakarta Messaging — необходимые, чтобы начать писать простые клиенты приложений, использующие Jakarta Messaging.

В следующем разделе представлена модель программирования Jakarta Messaging. В последующих разделах рассматриваются более сложные понятия, включая необходимые для написания приложений, использующих управляемые сообщениями бины.

Архитектура Jakarta Messaging

Приложение Jakarta Messaging состоит из следующих частей.

- Провайдер Jakarta Messaging — это система обмена сообщениями, которая реализует интерфейсы обмена сообщениями и предоставляет административные и контрольные функции. Реализация платформы Jakarta EE полного профиля включает в себя провайдера сообщений.
- JMS-клиенты — это программы или компоненты Java, отправляющие и получающие сообщения. Любой компонент приложения Jakarta EE может выступать в качестве клиента при обмене сообщениями.

Приложения Java SE могут также выступать в качестве клиентов Jakarta Messaging. Руководство разработчика очереди сообщений для клиентов Java в документации по GlassFish Server (https://glassfish.org/documentation) объясняет, как это работает.

- Сообщения — это объекты, которые передают информацию между клиентами JMS.
- Администрируемые объекты — это объекты JMS, настроенные для использования клиентами. Два вида администрируемых объектов JMS — пункты назначения и фабрики соединений — описаны в Администрируемые объекты JMS. Администратор может создавать объекты, доступные всем приложениям, использующим конкретную установку GlassFish Server. Альтернативно, разработчик может использовать аннотации для создания объектов, специфичных для конкретного приложения.

Рисунок 48-2 иллюстрирует взаимодействие этих частей. Административные инструменты или аннотации позволяют связывать пункты назначения и фабрики соединений с пространством имён JNDI. Затем клиент обмена сообщений может использовать инъецирование ресурсов для доступа к администрируемым объектам в пространстве имён, а затем установить логическое подключение к тем же объектам через провайдера Jakarta Messaging.

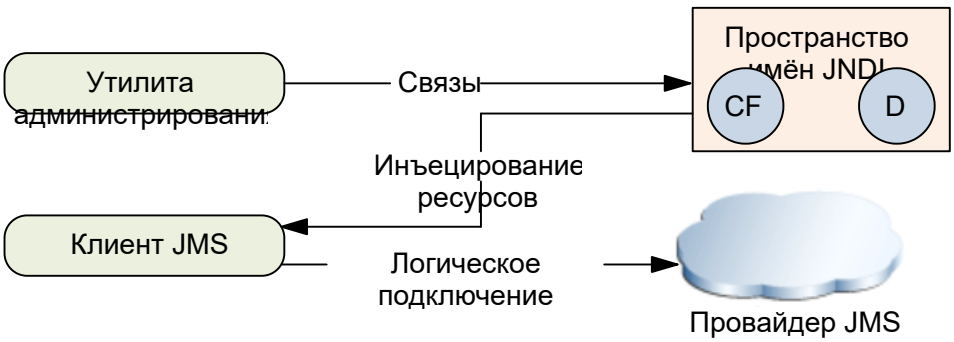

Рисунок 48-2 Архитектура Jakarta Messaging

Стили сообщений

До появления Jakarta Messaging большинство продуктов обмена сообщениями поддерживали стиль обмена сообщениями «точка-точка» или «публикация/подписка». Спецификация Jakarta Messaging определяет соответствие для каждого стиля. Поставщик обмена сообщениями должен реализовать оба стиля, а Jakarta Messaging предоставляет интерфейсы, специфичные для каждого из них. В следующих подразделах описываются эти стили сообщений.

Jakarta Messaging, однако, делает ненужным использование только одного из двух стилей. Он позволяет использовать один и тот же код для отправки и получения сообщений, используя стиль "точка-точка" или "публикация-подписка". Используемые пункты назначения остаются специфичными для каждого стиля, и поведение приложения отчасти будет зависеть от того, используется ли очередь или тема (topic). Однако сам код может быть общим для обоих стилей, что делает ваши приложения гибкими и пригодными для повторного использования. В этом руководстве описывается и иллюстрируется этот подход к кодированию с использованием значительно упрощённого API, предоставляемого Jakarta Messaging 2.0.

Стиль обмена сообщениями "Точка-точка"

Продукт или приложение "точка-точка" основаны на концепции очередей сообщений, отправителей и получателей. Каждое сообщение адресовано определённой очереди, и получающие клиенты извлекают сообщения из очередей, созданных для хранения сообщений. Все отправленные сообщения сохраняются в очереди до тех пор, пока не будут извлечены или не истечёт время их хранения.

Обмен сообщениями "точка-точка", показанный на рисунке 48-3, имеет следующие характеристики.

- Каждое сообщение имеет только одного потребителя.

- Получатель извлекает сообщение независимо от того, был ли он запущен в момент отправки сообщения клиентом.

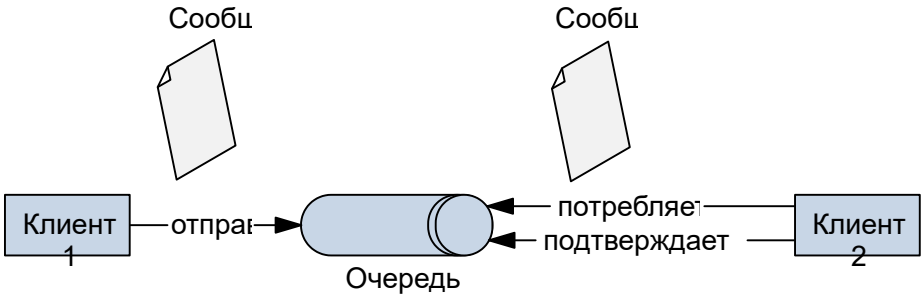

Рис. 48-3. Обмен сообщениями «точка-точка»

Используйте сообщения "точка-точка" в том случае, когда каждое отправленное сообщение предназначено только одному потребителю.

Стиль обмена сообщениями "публикация-подписка"

В продукте или приложении "публикации-подписки" клиенты направляют сообщения в тему, которая работает как доска объявлений. Издатели и подписчики могут динамически публиковать или подписываться на тему. Система заботится о распределении сообщений, поступающих от нескольких издателей темы, к нескольким её подписчикам. Темы сохраняют сообщения ровно столько времени, сколько требуется для их распространения среди подписчиков.

При обмене сообщениями "публикация-подписка" важно различать потребителя, который подписывается на тему (подписчик), и создаваемую подписку. Потребитель — это объект Jakarta Messaging в приложении, а подписка — это объект в провайдере JMS. Обычно тема может иметь много потребителей, но подписка имеет только одного подписчика. Однако можно создавать общие подписки. Смотрите Создание общих подписок для подробностей. См. Использование сообщений из тем для получения подробных сведений о семантике сообщений "публикация-подписка".

Обмен сообщениями "публикация-подписка" имеет следующие характеристики.

- Каждое сообщение может иметь несколько потребителей.
- Клиент, подписывающийся на тему, может получать только сообщения, отправленные после создания подписки, и потребитель должен продолжать оставаться активным, чтобы он мог принимать сообщения.

  Jakarta Messaging в некоторой степени ослабляет это требование, позволяя приложениям создавать долговременные подписки, которые получают сообщения, отправляемые, пока потребители не активны. Долговременные подписки обеспечивают гибкость и надёжность очередей и позволяют клиентам отправлять сообщения многим получателям. Для получения дополнительной информации о долговременных подписках см. Создание долговременных подписок.

Используйте сообщения "публикация-подписка", если каждое сообщение может быть обработано неопределённым количеством потребителей. Рисунок 48-4 иллюстрирует сообщения "публикация-подписка".

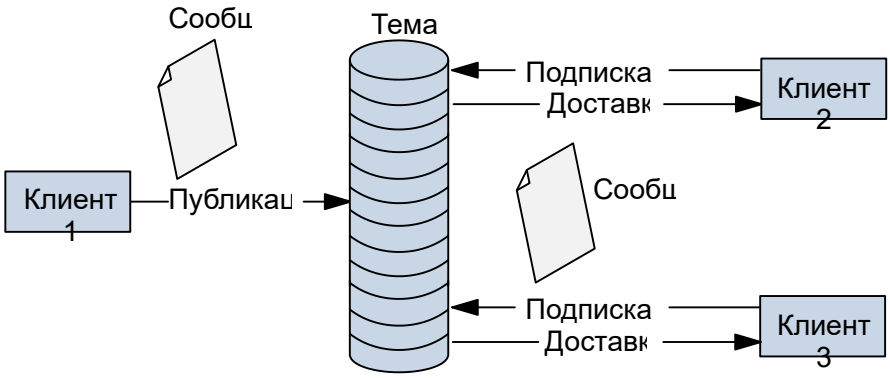

Рисунок 48-4 Обмен сообщениями публикации/подписки

Получение сообщений

Продукты для обмена сообщениями по своей сути асинхронны: нет фундаментальной временно́й зависимости между отправкой и получением сообщения. Однако спецификация Jakarta Messaging использует более точный смысл этого термина. Сообщения могут быть использованы одним из двух способов.

- Синхронно: потребитель явно извлекает сообщение из пункта назначения, вызывая метод `receive`. Метод `receive` может блокировать до тех пор, пока не прибудет сообщение или не истечёт время ожидания, если сообщение не поступило в течение указанного периода времени.
- Асинхронно: клиентское приложение или клиент Java SE могут зарегистрировать слушатель сообщений у потребителя. Слушатель сообщений похож на слушатель событий. Всякий раз, когда сообщение поступает в пункт назначения, провайдер JMS доставляет сообщение, вызывая метод `onMessage` слушателя, который обрабатывает содержимое сообщения. В приложении Jakarta EE компонент, управляемый сообщениями, служит в качестве приёмника сообщений (он также имеет метод `onMessage`), но клиенту не нужно регистрировать его у потребителя.

Модель программирования Jakarta Messaging

Основные строительные блоки приложения JMS:

- Администрируемые объекты: фабрики и направления связи
- Соединения
- Сессии
- Объекты `JMSContext`, объединяющие соединение и сессию в одном объекте
- Производители сообщений
- Потребители сообщений
- Сообщения

Нарис. 48-5 показаны все эти объекты клиентской программы Jakarta Messaging.

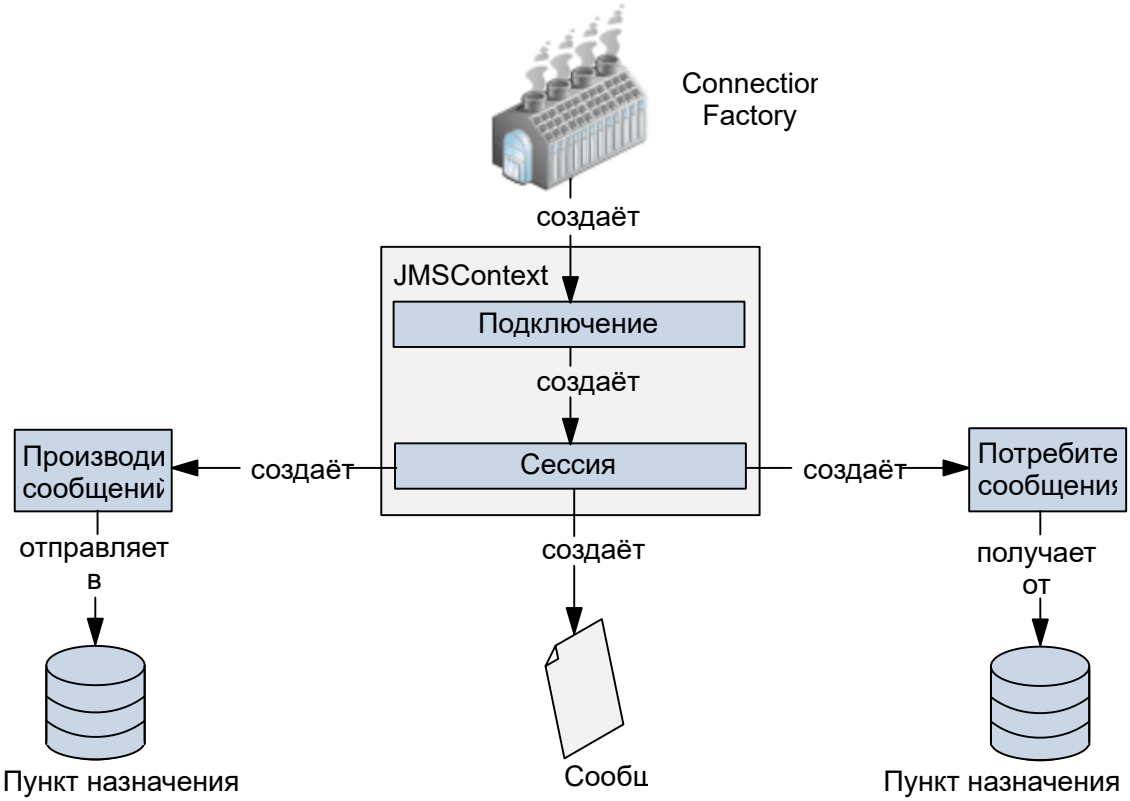

Рис. 48-5 Программная модель Jakarta Messaging

Jakarta Messaging также предоставляет браузеры очереди — объекты, которые позволяют приложению просматривать сообщения в очереди.

В этом разделе кратко описаны все эти объекты и приведены примеры команд и фрагменты кода, которые показывают, как создавать и использовать объекты. В последнем подразделе кратко описывается обработка исключений Jakarta Messaging API.

Примеры, показывающие, как объединить все эти объекты в приложении, приведены в главе 49 *Примеры Jakarta Messaging*, начиная с раздела Простые приложения Jakarta Messaging. Более подробную информацию см. в документации по Jakarta Messaging, входящей в Jakarta EE API.

Администрируемые объекты JMS

Две части приложения JMS — пункты назначения и фабрики соединений — обычно обслуживаются административно, а не программно. Лежащая в основе этих объектов технология, вероятно, сильно отличается от той, что используется Jakarta Messaging. Следовательно, управление этими объектами относится к административным задачам, которые варьируются от поставщика к поставщику.

Клиенты обмена сообщениями получают доступ к администрируемым объектам через переносимые интерфейсы, поэтому клиентское приложение может работать с небольшими изменениями или вообще без изменений с несколькими реализациями Jakarta Messaging. Обычно администратор настраивает администрируемые объекты в пространстве имён JNDI, а клиенты обмена сообщениями затем получают к ним доступ с помощью инъецирования ресурсов.

Используя GlassFish Server, вы можете использовать команду `asadmin create-jms-resource` или Консоль администрирования для создания администрируемых объектов Jakarta Messaging в форме ресурсов коннектора. Вы также можете указать ресурсы в файле с именем `glassfish-resources.xml` , который можно связать с приложением.

В среде IDE NetBeans имеется мастер, позволяющий создавать ресурсы Jakarta Messaging для GlassFish Server. Подробности смотрите в Административное создание объектов JMS.

Спецификация платформы Jakarta EE позволяет разработчику создавать администрируемые объекты с использованием аннотаций или элементов дескриптора развёртывания. Объекты, созданные таким образом, специфичны для приложения, для которого они созданы. Подробности смотрите в Создание ресурсов для приложений Jakarta EE. Определения в дескрипторе развёртывания переопределяют указанные в аннотациях.

Фабрики соединений JMS

Фабрика соединений — это объект, который клиент использует для создания соединения с провайдером. Фабрика соединений содержит набор параметров конфигурации соединения, которые были определены администратором. Каждая фабрика соединений является объектом интерфейса `ConnectionFactory`, `QueueConnectionFactory` или `TopicConnectionFactory`. Чтобы узнать, как создавать фабрики соединений, см. Административное создание объектов JMS.

В начале клиентской программы обмена сообщениями обычно инъецируется ресурс фабрики соединений `ConnectionFactory`. Сервер Jakarta EE должен предоставить фабрике соединений Jakarta Messaging логическое имя JNDI `java:comp/DefaultJMSConnectionFactory`. Фактическое имя JNDI будет зависеть от реализации.

Например, следующий фрагмент кода ищет фабрику соединений Jakarta Messaging по умолчанию и назначает её объекту `ConnectionFactory`:

```java
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
private static ConnectionFactory connectionFactory;
```

Пункты назначения JMS

Пункт назначения — это объект, который клиент использует для указания цели отправляемых сообщений и источника получаемых сообщений. При обмене сообщениями типа "точка-точка" пункты назначения называются очередями. При обмене "публикация-подписка" — темами. Приложение Jakarta Messaging может использовать несколько очередей или тем (или и то и другое). Чтобы узнать, как создавать ресурсы пунктов назначения, см. Административное создание объектов JMS.

Чтобы создать пункт назначения с помощью GlassFish Server, создайте ресурс пункта назначения Jakarta Messaging и укажите для него имя JNDI.

В реализации GlassFish Server Jakarta Messaging каждый целевой ресурс сопоставляется с физическим пунктом назначения. Физический пункт назначения может быть создан явным образом. Если это не сделано, сервер приложений создаст его, когда это необходимо, и удалит его вместе с соответствующим ресурсом.

Помимо инъецирования ресурса фабрики соединений в клиентскую программу, обычно инъецируется и пункт назначения. В отличие от фабрик соединений, пункты назначения отличаются для способов обмена сообщениями "точка-точка" и "публикация-подписка". Чтобы создать приложение, позволяющее использовать один и тот же код как для тем, так и для очередей, вы назначаете пункт назначения для объекта `Destination`.

Следующий код указывает два ресурса: очередь и тему. Имена ресурсов отображаются на целевые ресурсы, созданные в пространстве имён JNDI:

JAVA

```java
@Resource(lookup = "jms/MyQueue")
private static Queue queue;

@Resource(lookup = "jms/MyTopic")
private static Topic topic;
```

В приложении Jakarta EE администрируемые объекты Jakarta Messaging обычно помещаются в субконтекст именования `jms`.

С помощью общих интерфейсов вы можете смешивать или сопоставлять фабрики соединений и пункты назначения. То есть, в дополнение к использованию интерфейса `ConnectionFactory`, вы можете инъецировать ресурс `QueueConnectionFactory` и использовать его с `Topic`, а также можете добавить `TopicConnectionFactory` и использовать его с `Queue`. Поведение приложения будет зависеть от типа используемого пункта назначения, а не от типа используемой фабрики соединений.

Соединения

Соединение инкапсулирует виртуальное соединение с провайдером сообщений. Например, соединение может представлять собой открытый сокет TCP/IP между клиентом и демоном службы провайдера. Соединение используется для создания одной или нескольких сессий.

> В платформе Jakarta EE возможность создания нескольких сессий из одного подключения ограничена клиентскими приложениями. В веб-компонентах и компонентах Enterprise-бина соединение может создавать не более одной сессии.

Обычно соединение создаётся при создании объекта `JMSContext`. Смотрите Объекты JMSContext для подробностей.

Сессии

Сессия — это однопоточный контекст для отправки и получения сообщений.

Обычно сессия (а также соединение) создаётся вместе с созданием объекта `JMSContext`. Смотрите Объекты JMSContext для подробностей. Сессии используются для создания производителей сообщений, потребителей сообщений, собственно сообщений, браузеров очередей и вре́менных пунктов назначения.

Сессии сериализуют выполнение слушателей сообщений. Подробности см. в разделе Слушатели сообщений JMS.

Сессия обеспечивает транзакционный контекст, с помощью которого можно сгруппировать набор отправлений и приёмов сообщений в атомарную операцию. Подробнее см. Использование локальных транзакций JMS.

Объекты JMSContext

Объект `JMSContext` содержит в себе соединение и сессию. То есть он обеспечивает как активное соединение с провайдером сообщений, так и однопоточный контекст для отправки и получения сообщений.

`JMSContext` используется для создания следующих объектов:

- Производители сообщений
- Потребители сообщений
- Сообщения

- Браузеры очереди
- Вре́менные очереди и темы (см. Создание вре́менных пунктов назначения)

Можно создать `JMSContext` в блоке `try` -with-resources.

Чтобы создать `JMSContext` , вызовите метод `createContext` фабрики соединений:

```
JMSContext context = connectionFactory.createContext();
```
JAVA

При вызове без аргументов из клиента приложения или клиента Java SE, либо с веб-сайта Jakarta EE или EJB-контейнера, когда не выполняется активная транзакция Jakarta Transactions, метод `createContext` создаёт не-транзакционную сессию с режимом подтверждения `JMSContext.AUTO_ACKNOWLEDGE` . При вызове без аргументов из веб-контейнера или EJB-контейнера, когда выполняется активная транзакция JTA, метод `createContext` создаёт транзакцию сессии. Для получения информации о том, как транзакции Jakarta Messaging работают в приложениях Jakarta EE, см. Использование Jakarta Messaging в приложениях Jakarta EE.

Чтобы создать транзакционную сессию в клиентском приложении или клиенте Java SE, также можно вызвать метод `createContext` с аргументом `JMSContext.SESSION_TRANSACTED` :

```
JMSContext context =
        connectionFactory.createContext(JMSContext.SESSION_TRANSACTED);
```
JAVA

Сессия использует локальные транзакции. Подробнее смотрите Использование локальных транзакций JMS.

Кроме того, можно указать режим подтверждения не по умолчанию. Смотрите Управление подтверждением сообщения для получения дополнительной информации.

При использовании `JMSContext` доставка сообщений начинается при создании потребителя. Смотрите Потребители сообщений JMS для получения дополнительной информации.

Если `JMSContext` создаётся в блоке `try` -with-resources, не требуется закрывать его явно. Он будет закрыт, когда закончится блок `try` . Убедитесь, что приложение выполняет все действия Jakarta Messaging в блоке `try` -with-resources. Если блок `try` -with-resources не используется, нужно вызвать метод `close` в `JMSContext` , чтобы закрыть соединение, когда приложение закончит свою работу.

Производители сообщений JMS

Источник сообщений — это объект, который создаётся `JMSContext` -ом или сессией и используется для отправки сообщений в пункт назначения. Источник сообщений, созданный `JMSContext` , реализует интерфейс `JMSProducer` . Вы можете создать его следующим образом:

```
try (JMSContext context = connectionFactory.createContext();) {
    JMSProducer producer = context.createProducer();
    ...
}
```
JAVA

Однако `JMSProducer` — это легковесный объект, который не потребляет значительных ресурсов. По этой причине не нужно сохранять `JMSProducer` в переменной. Его можно создавать новый каждый раз, когда возникает необходимость отправить сообщение. Отправляйте сообщения в определённый пункт назначения с помощью метода `send` . Например:

```java
context.createProducer().send(dest, message);
```

Вы можете создать сообщение в переменной перед отправкой, как показано здесь, или создать его с помощью вызова `send`. Смотрите Сообщения JMS для получения дополнительной информации.

Потребители сообщений JMS

Потребитель сообщений — это объект, который создаётся с помощью `JMSContext` или сессией и используется для получения сообщений, отправленных в пункт назначения. Источник сообщений, созданный `JMSContext`, реализует интерфейс `JMSConsumer`. Самый простой способ создать получателя сообщения — использовать метод `JMSContext.createConsumer`:

```java
try (JMSContext context = connectionFactory.createContext();) {
    JMSConsumer consumer = context.createConsumer(dest);
    ...
}
```

Потребитель сообщений позволяет клиенту обмена сообщениями зарегистрироваться в провайдере сообщений на сообщения к пункту назначения. Провайдер Jakarta Messaging управляет доставкой сообщений из пункта назначения зарегистрированным в этом пункте пользователям.

Когда для создания получателя сообщения используется `JMSContext`, доставка сообщения начинается сразу после его создания. Это поведение может быть отключено вызовом `setAutoStart(false)` при создании `JMSContext`, а затем явным вызовом метода `start` для начала доставки сообщений. Если нужно временно остановить доставку сообщений без закрытия соединения, можно вызвать метод `stop`. Для возобновления доставки сообщений вызовите `start`.

Метод `receive` используется для синхронного получения сообщения. Этот метод может быть использован в любое время после создания потребителя.

Если не указать аргументов или указать аргумент `0`, метод блокируется на неопределённый срок до прибытия сообщения:

```java
Message m = consumer.receive();
Message m = consumer.receive(0);
```

Для простого клиента это может не иметь значения. Но если возможно, что сообщение может быть недоступно, используйте синхронный приём с тайм-аутом: вызовите метод `receive` с аргументом тайм-аута, большим, чем `0`. Одна секунда является рекомендуемым значением времени ожидания:

```java
Message m = consumer.receive(1000); // тайм-аут по истечение 1 секунды
```

Чтобы включить асинхронную доставку сообщений от клиентского приложения или клиента Java SE, используйте слушатель сообщений, как описано в следующем разделе.

Для создания долговременной подписки на темы может использоваться метод `JMSContext.createDurableConsumer`. Этот метод применим только к темам. Для получения дополнительной информации см. Создание долговременных подписок. Для тем также могут создаваться общие потребители. См. Создание общих подписок.

Слушатели сообщений JMS

Слушатель сообщений — это объект, который действует как асинхронный обработчик событий для сообщений. Этот объект реализует интерфейс `MessageListener`, содержащий метод `onMessage`. В методе `onMessage` определяются действия, которые должны быть выполнены при получении сообщения.

Используя метод `setMessageListener`, в клиентском приложении или клиенте Java SE регистрируется слушатель сообщений с определённым потребителем сообщений. Например, класс с именем `Listener`, реализующий интерфейс `MessageListener`, может быть зарегистрирован как слушатель сообщений следующим образом:

```java
Listener myListener = new Listener();
consumer.setMessageListener(myListener);
```

Когда начинается доставка сообщения, провайдер сообщений автоматически вызывает метод `onMessage` слушателя сообщений всякий раз, когда доставляется сообщение. Метод `onMessage` принимает один аргумент типа `Message`, который ваша реализация метода может при необходимости привести к другому подтипу сообщения (см. Тело сообщения),

В веб-сайте Jakarta EE или EJB-контейнере используются управляемые сообщениями компоненты для асинхронной доставки сообщений. Управляемый сообщениями компонент также реализует интерфейс `MessageListener` и содержит метод `onMessage`. Для получения дополнительной информации см. Использование управляемых сообщениями бинов для асинхронного получения сообщений.

Ваш метод `onMessage` должен обрабатывать все исключения. Выбрасывание `RuntimeException` считается ошибкой программирования.

Простой пример использования слушателя сообщений см. в разделе Использование слушателя сообщений для асинхронной доставки сообщений. Глава 49 *Примеры Jakarta Messaging* содержит ещё несколько примеров слушателей сообщений и компонентов, управляемых сообщениями.

Селекторы сообщений JMS

Если приложению для обмена сообщениями необходимо фильтровать полученные сообщения, может использоваться селектор сообщений JMS, позволяющий получателю сообщений указать интересующий его пункт назначения. Селекторы сообщений возлагают работу по фильтрации сообщений на провайдер сообщений, а не на приложение. Приложение, использующее селектор сообщений, см. в разделе Отправка сообщений из сессионного компонента в MDB.

Селектор сообщений — выражение типа `String`. Синтаксис этого выражения основан на подмножестве синтаксиса условного выражения SQL92. Селектор сообщений в этом примере выбирает любое сообщение со свойством `NewsType`, для которого установлено значение `'Sports'` или `'Opinion'`:

```java
NewsType = 'Sports' OR NewsType = 'Opinion'
```

Методы `createConsumer` и `createDurableConsumer`, а также методы для создания общих потребителей позволяют указывать селектор сообщений в качестве аргумента при создании получателя сообщения.

Получатель сообщений получает только те сообщения, заголовки и свойства которых соответствуют селектору. (См. Заголовки сообщений и Свойства сообщения.) Селектор сообщений не может выбирать сообщения на основе содержимого тела сообщения.

Получение сообщений из тем

Семантика получения сообщений из тем более сложна, чем из очередей.

Приложение получает сообщения из темы, создавая подписку на эту тему и получателя на эту подписку. Подписки могут быть или не быть долговременными (durable), быть или не быть общими (shared).

Подписка может рассматриваться как объект внутри провайдера сообщений, тогда как потребитель — это объект Jakarta Messaging в приложении.

Подписка будет получать копию каждого сообщения, отправляемого в тему после создания подписки, если не указан селектор сообщений. Если селектор сообщений указан, в подписку будут добавлены только те сообщения, свойства которых соответствуют селектору сообщений.

Подписки, не являющиеся общими, ограничены одним потребителем. В этом случае все сообщения в подписке доставляются этому потребителю. Общие подписки позволяют доставку нескольким потребителям. В этом случае каждое сообщение в подписке доставляется только одному потребителю. Jakarta Messaging не определяет, как сообщения распределяются между несколькими пользователями по одной и той же подписке.

Подписки могут быть или не быть долговременными.

Подписка, не являющаяся долговременной, существует только до тех пор, пока в ней есть активный потребитель. Это означает, что любые сообщения, отправленные в тему, будут добавлены в подписку, только если потребитель существует и не закрыт.

Подписка, не являющаяся долговременной, может быть как быть общей, так и не быть ею.

- Подписки, не являющиеся долговременными и общими, не имеют имени и с ними может быть связан только один объект-потребитель. Они создаются автоматически при создании объекта-потребителя. Они не сохраняются и автоматически удаляются при закрытии объекта-потребителя.

  Метод `JMSContext.createConsumer` создаёт получателя для подписки, не являющейся долговременной общей, если в качестве пункта назначения указана тема.

- Общая подписка, не являющаяся долговременной, идентифицируется по имени и необязательному идентификатору клиента и может иметь несколько объектов-потребителей, получающих от неё сообщения. Она создаётся автоматически при создании первого объекта-потребителя. Она не сохраняется и удаляется автоматически при закрытии последнего объекта-потребителя. Смотрите Создание общих подписок для получения дополнительной информации.

За счёт более высоких накладных расходов подписка может быть долговременной. Долговременная подписка сохраняется и продолжает накапливать сообщения до тех пор, пока не будет явно удалена, даже если нет объектов-потребителей, получающих сообщения от неё. Подробности смотрите в Создание долговременных подписок.

Создание долговременных подписок

Чтобы гарантировать, что приложение "публикация-подписка" получает все отправленные сообщения, используйте долговременные подписки для потребителей этой темы.

Как и подписка, не являющаяся долговременной, долговременная подписка может быть или не быть общей.

- Долговременная подписка, не являющаяся общей, идентифицируется по имени и идентификатору клиента (который должен быть установлен) и может иметь только один объект-потребитель, связанный с ней.
- Общая долговременная подписка идентифицируется по имени и необязательному идентификатору клиента и может иметь несколько объектов-потребителей, получающих от неё сообщения.

Долговременная подписка будет называться неактивной, если она существует, но с ней в настоящее время не связан ни один незакрытый потребитель.

Вы можете использовать метод `JMSContext.createDurableConsumer` , чтобы создать получателя для долговременной подписки, не являющейся общей. У такой подписки одновременно может быть только один активный потребитель.

Потребитель определяет долговременную подписку, из которой он использует сообщения, указывая уникальный идентификатор, которое хранится провайдером сообщений. Последующие объекты-потребители, имеющие одинаковые идентификационные данные, возобновляют подписку в том состоянии, в котором она была оставлена предыдущим потребителем. Если долговременная подписка не имеет активного потребителя, провайдер сообщений сохраняет сообщения подписки до тех пор, пока они не будут получены подпиской или пока срок их действия не истечёт.

Уникальность долговременной подписки, не являющейся общей, достигается сочетанием следующих частей:

- Идентификатор клиента для соединения
- Тема и название подписки

Вы можете установить идентификатор клиента административно для конкретной фабрики соединений, используя либо командную строку, либо Консоль администрирования. (В клиентском приложении или клиенте Java SE вместо этого вы можете вызвать `JMSContext.setClientID` .)

После использования этой фабрики соединений для создания `JMSContext` вызывается метод `createDurableConsumer` с двумя аргументами: темой и именем подписки:

```
JAVA
String subName = "MySub";
JMSConsumer consumer = context.createDurableConsumer(myTopic, subName);
```

Подписка становится активной после создания потребителя. Позже потребитель может быть закрыт:

```
JAVA
consumer.close();
```

Поставщик сообщений хранит сообщения, отправленные в тему, так же как и сообщения, отправленные в очередь. Если программа или другое приложение вызывает `createDurableConsumer` , используя ту же фабрику соединений и её идентификатор клиента, ту же тему и то же имя подписки, то подписка повторно активируется, и провайдер сообщений доставляет сообщения. отправленные в период неактивности подписки.

Для удаления долговременной подписки, сначала закройте получателя, затем вызовите метод `unsubscribe` с именем подписки в качестве аргумента:

```
JAVA
consumer.close();
context.unsubscribe(subName);
```

Метод `unsubscribe` удаляет состояние, которое провайдер поддерживает для подписки.

На рис. 48-7 показана разница между недолговременной и долговременной подписками. При обычной (не долговременной) подписке потребитель и подписка начинаются и заканчиваются в одной точке и фактически идентичны. Когда потребитель закрыт, подписка также заканчивается. Здесь `create` означает

вызов `JMSContext.createConsumer` с аргументом `Topic` , а `close` — вызов `JMSConsumer.close` . Любые сообщения, отправленные в тему между временем первого `close` и временем второго `create` , не попадут ни в одну из подписок. На рис. 48-6 получатели сообщений получат M1, M2, M5 и M6, но не получат M3 и M4.

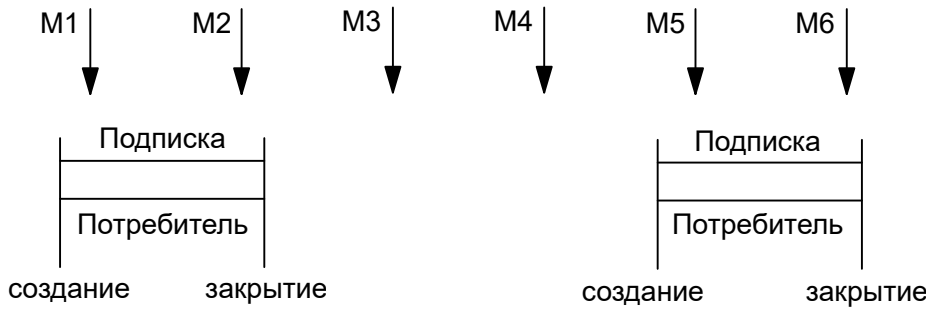

Рис. 48-6. Подписки, не являющиеся длительными, и их потребители

При наличии долговременной подписки потребитель может быть закрыт и повторно создан, но подписка продолжает существовать и хранить сообщения до тех пор, пока приложение не вызовет метод `unsubscribe` . На рис. 48-7 `create` обозначает вызов `JMSContext.createDurableConsumer` , `close` обозначает вызов `JMSConsumer.close` и `unsubscribe` означает вызов `JMSContext.unsubscribe` . Сообщения, отправленные после закрытия первого потребителя, принимаются при создании второго потребителя (по той же долговременной подписке), поэтому даже если сообщения M2, M4 и M5 поступают, когда нет потребителя, они не теряются.

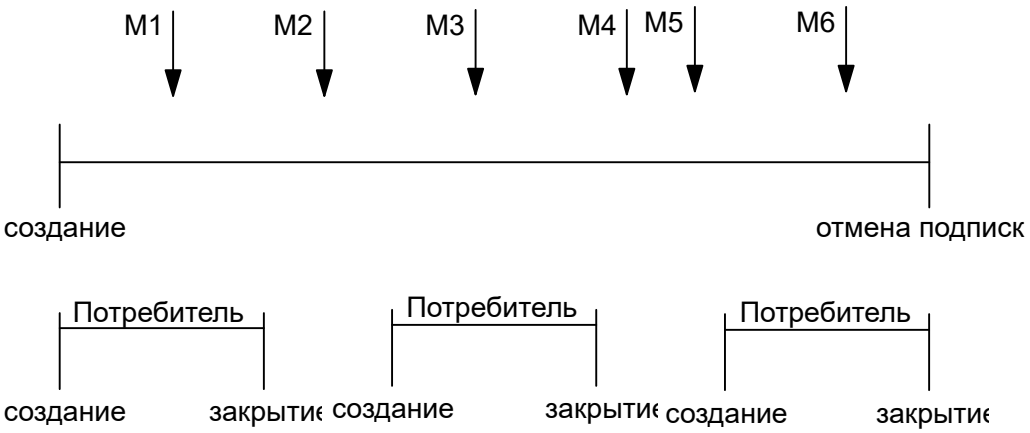

Рис. 48-7. Потребители долговременной подписки

Общая долговременная подписка позволяет использовать несколько потребителей для получения сообщений из долговременной подписки. Если вы используете общую долговременную подписку, используемой фабрике соединений не требуется идентификатор клиента. Для создания общей долговременной подписки вызовите метод `JMSContext.createSharedDurableConsumer` , указав тему и имя подписки:

JAVA

```
JMSConsumer consumer =
        context.createSharedDurableConsumer(topic, "MakeItLast");
```

Смотрите Подтверждение сообщений, Использование долговременных подписок, Использование общих долговременных подписок и Отправка сообщений из сессионного компонента в MDB для примеров приложений Jakarta EE, которые используют долговременные подписки.

Создание общих подписок

Подписка на тему, созданная методом `createConsumer` или `createDurableConsumer` , может иметь только одного потребителя (хотя тема может иметь много). Несколько клиентов, использующих одну и ту же тему, по определению имеют несколько подписок на эту тему, и все клиенты получают все сообщения,

отправленные в тему (если только они не фильтруют их с помощью селекторов сообщений).

Однако можно создать общую подписку, не являющуюся долговременной, на тему, используя метод `createSharedConsumer` и указав не только пункт назначения, но и имя подписки:

```
consumer = context.createSharedConsumer(topicName, "SubName");
```
JAVA

При общей подписке сообщения будут распределяться между несколькими клиентами, использующими одни и те же тему и имя подписки. Каждое сообщение, отправленное в тему, будет добавлено в каждую подписку (с учётом любых селекторов сообщений), но каждое сообщение, добавленное в подписку, будет доставлено только одному из потребителей этой подписки, поэтому оно будет получено только одним из клиентов. Общая подписка может быть полезна, если вы хотите разделить загрузку сообщений между несколькими потребителями в подписке, а не для того, чтобы только один потребитель в подписке получал каждое сообщение. Эта функция может улучшить масштабируемость клиентских приложений приложений Jakarta EE и Java SE. (Бины, управляемые сообщениями, разделяют работу по обработке сообщений из темы среди нескольких потоков.)

См. Использование общих подписок, не являющихся долговременными для простого примера использования потребителей общих подписок, не являющихся долговременными.

Вы также можете создавать общие долговременные подписки с помощью метода `JMSContext.createSharedDurableConsumer` . Для получения дополнительной информации см. Создание долговременных подписок.

Сообщения JMS

Конечная цель приложения JMS — отправлять и получать сообщения, которые затем могут использоваться другими программными приложениями. Сообщения Jakarta Messaging имеют простой, но очень гибкий формат, позволяющий создавать сообщения, соответствующие форматам, используемым не-JMS-приложениями на гетерогенных платформах.

Сообщение Jakarta Messaging может состоять из трёх частей: заголовка, свойств и тела. Обязательным является только заголовок. В следующих разделах описываются эти части.

Полную документацию по заголовку, свойствам и телу сообщений, см. документацию по API интерфейса `Message` . Список возможных типов сообщений см. в разделе Тело сообщения.

Заголовок сообщения

Заголовок сообщения Jakarta Messaging включает несколько предопределённых полей, которые содержат значения, используемые как клиентами, так и поставщиками для идентификации и маршрутизации сообщений. Таблица 48-1 перечисляет и описывает поля заголовка сообщения Jakarta Messaging и значения этих полей. Например, каждое сообщение имеет уникальный идентификатор, который представлен в поле заголовка `JMSMessageID` . Значение другого поля заголовка, `JMSDestination` , представляет очередь или тему, в которую отправляется сообщение. Другие поля включают отметку времени и приоритет.

Каждое поле заголовка имеет get- и set- методы, которые описаны в интерфейсе `Message` . Некоторые поля заголовка предназначены для установки клиентом, но многие устанавливаются автоматически с помощью метода `send` , который переопределяет значения, установленные клиентом.

Таблица 48-1 Как устанавливаются значения полей заголовка сообщения JMS

Поле заголовка	Описание	Установлен

Поле заголовка	Описание	Установлен
`JMSDestination`	Пункт назначения, на который отправляется сообщение	Метод `send` провайдера JMS
`JMSDeliveryMode`	Режим доставки, указанный при отправке сообщения (см. Указание персистентности сообщения)	Метод `sendки` провайдера сообщений
`JMSDeliveryTime`	Время отправки сообщения плюс задержка доставки, указанная при отправке сообщения (см. Указание задержки доставки	Метод `send` провайдера JMS
`JMSExpiration`	Время истечения срока действия сообщения (см. Установка срока действия сообщения)	Метод `send` провайдера JMS
`JMSPriority`	Приоритет сообщения (см. Установка приоритета сообщений)	Метод JMS-провайдера `send`
`JMSMessageID`	Значение, которое однозначно идентифицирует каждое сообщение, отправленное провайдером	Метод `sendки` провайдера сообщений
`JMSTimestamp`	Время передачи сообщения провайдеру для отправки	Метод `sendки` провайдера сообщений
`JMSCorrelationID`	Значение, которое связывает одно сообщение с другим. Обычно используется значение `JMSMessageID`	Клиентское приложение
`JMSReplyTo`	Пункт назначения, куда следует отправлять ответы на сообщение	Клиентское приложение
`JMSType`	Идентификатор типа, предоставленный клиентским приложением	Клиентское приложение
`JMSRedelivered`	Является ли сообщение повторно доставленным	JMS-провайдер перед доставкой

Свойства сообщения

Если сообщению нужны свойства в дополнение к тем, которые указаны в полях заголовка, они могут быть созданы и установлены. Вы можете использовать свойства для обеспечения совместимости с другими системами обмена сообщениями или использовать их для создания селекторов сообщений (см. Селекторы сообщений JMS). Пример установки свойства, которое будет использоваться в качестве селектора сообщений, см. в разделе Отправка сообщений из сессионного компонента в MDB.

Jakarta Messaging предоставляет некоторые предопределённые имена свойств, начинающие с `JMSX`. Провайдер сообщений должен реализовывать только одно из них — `JMSXDeliveryCount` (в котором указывается количество доставленных сообщений), остальные необязательны. Использование этих предопределённых свойств, как и кастомных свойств в приложениях не обязательно.

Тело сообщения

Jakarta Messaging определяет шесть различных типов сообщений. Каждый тип сообщения имеет свойственное ему тело сообщения. Эти типы сообщений позволяют отправлять и получать данные в различных формах. Таблица 48-2 описывает эти типы сообщений.

Таблица 48-2 Типы сообщений JMS

Тип сообщения	**Тело содержит**
`TextMessage`	Объект `java.lang.String` (например, содержимое файла XML).
`MapMessage`	Набор пар имя-значение с ключами в виде объектов `String` и значениями в виде примитивных типов Java. Доступ к записям может осуществляться последовательным обходом или произвольно по имени. Порядок записей не определён.
`BytesMessage`	Поток неинтерпретируемых байтов. Этот тип сообщения предназначен для кодирования тела в соответствии с существующим форматом сообщения.
`StreamMessage`	Поток примитивных значений Java, заполняемый и читаемый последовательно.
`ObjectMessage`	Объект `Serializable` Java.
`Message`	Отсутствует. Состоит только из заголовка и свойств. Этот тип сообщения полезен, когда тело сообщения не требуется.

Jakarta Messaging предоставляет методы для создания сообщений каждого типа и заполнения их содержимого. Например, для создания и отправки `TextMessage` могут использоваться следующие операторы:

```java
TextMessage message = context.createTextMessage();
message.setText(msg_text);     // msg_text имеет тип String
context.createProducer().send(message);
```

На стороне потребителя сообщение приходит как обобщённый (generic) объект `Message`. Затем можно привести объект к соответствующему типу сообщения и использовать специфичные методы для доступа к телу и извлечения содержимого сообщения (а также его заголовков и свойств, если необходимо). Например, могут быть использованы потоковые методы чтения `BytesMessage`. Для получения тела `StreamMessage` всегда нужно приводить сообщение к соответствующему типу.

Вместо приведения сообщения к конкретному типу может быть вызван метод `getBody` у `Message` с указанием типа сообщения в качестве аргумента. Например, можно извлечь `TextMessage` как `String`. В следующем фрагменте кода используется метод `getBody`:

```java
Message m = consumer.receive();
if (m instanceof TextMessage) {
    String message = m.getBody(String.class);
    System.out.println("Reading message: " + message);
} else {
    // Обработка ошибки или другого типа сообщения
}
```

Jakarta Messaging предоставляет ярлыки для создания и приёма сообщений `TextMessage`, `BytesMessage`, `MapMessage` или `ObjectMessage`. Например, не нужно переносить строку в `TextMessage`. Вместо этого можно отправить и получить строку напрямую. Например, можно отправить строку следующим образом:

JAVA

```
String message = "This is a message";
context.createProducer().send(dest, message);
```

Можно получить сообщение, используя метод `receiveBody`:

JAVA

```
String message = receiver.receiveBody(String.class);
```

Можно использовать метод `receiveBody` для получения сообщений любого типа, кроме `StreamMessage` и `Message`, при условии, что телу сообщения может быть назначен определённый тип.

Пустой `Message` может быть полезен, если нужно отправить сообщение, которое является сигналом для приложения. Некоторые из примеров в главе 49 *Примеры Jakarta Messaging* отправляют пустое сообщение после отправки серии текстовых сообщений. Например:

JAVA

```
context.createProducer().send(dest, context.createMessage());
```

Затем код потребителя может интерпретировать нетекстовое сообщение как сигнал о том, что все отправленные сообщения уже получены.

Примеры в главе 49 *Примеры Jakarta Messaging* используют сообщения типа `TextMessage`, `MapMessage` и `Message`.

Браузеры очереди JMS

Сообщения, отправленные в очередь, остаются в очереди до их получения получателем сообщений для этой очереди. Jakarta Messaging предоставляет объект `QueueBrowser`, который позволяет просматривать сообщения в очереди и отображать значения заголовков для каждого сообщения. Чтобы создать объект `QueueBrowser`, используйте метод `JMSContext.createBrowser`.

Например:

JAVA

```
QueueBrowser browser = context.createBrowser(queue);
```

См. Просмотр сообщений в очереди для примера использования объекта `QueueBrowser`.

Метод `createBrowser` позволяет указать селектор сообщений вторым аргументом при создании `QueueBrowser`. Для получения информации о селекторах сообщений см. Селекторы сообщений JMS.

В Jakarta Messaging нет механизма для просмотра темы. Сообщения обычно исчезают из темы, сразу после добавления: если нет потребителей сообщений для их использования, провайдер сообщений удаляет их. Хотя долговременные подписки позволяют сообщениям оставаться в теме, пока потребитель сообщений не активен, Jakarta Messaging не определяет никаких средств для их проверки.

Обработка исключений JMS

Корневым классом для всех проверенных исключений в Jakarta Messaging является `JMSException`. Родительским классом всех непроверяемых исключений в Jakarta Messaging API является `JMSRuntimeException`.

Перехват `JMSException` и `JMSRuntimeException` предоставляет универсальный способ обработки всех исключений, связанных с сообщениями в Jakarta Messaging.

Классы `JMSException` и `JMSRuntimeException` включают следующие дочерние классы, описанные в документации API:

- `IllegalStateException`, `IllegalStateRuntimeException`
- `InvalidClientIDException`, `InvalidClientIDRuntimeException`
- `InvalidDestinationException`, `InvalidDestinationRuntimeException`
- `InvalidSelectorException`, `InvalidSelectorRuntimeException`
- `JMSSecurityException`, `JMSSecurityRuntimeException`
- `MessageEOFException`
- `MessageFormatException`, `MessageFormatRuntimeException`
- `MessageNotReadableException`
- `MessageNotWriteableException`, `MessageNotWriteableRuntimeException`
- `ResourceAllocationException`, `ResourceAllocationRuntimeException`
- `TransactionInProgressException`, `TransactionInProgressRuntimeException`
- `TransactionRolledBackException`, `TransactionRolledBackRuntimeException`

Все примеры учебника перехватывают и обрабатывают `JMSException` или `JMSRuntimeException`, когда это уместно.

Дополнительные функции JMS

В этом разделе объясняется, как использовать функции Jakarta Messaging для достижения необходимого уровня надёжности и производительности. Многие люди используют Jakarta Messaging в своих приложениях, потому что их приложения не переносят потерянных или дублирующихся сообщений и требуют, чтобы каждое сообщение принималось один и только один раз. Эту функциональность обеспечивает Jakarta Messaging.

Самый надёжный способ создания сообщения — отправить `PERSISTENT` сообщение и сделать это в транзакции.

Сообщения Jakarta Messaging по умолчанию `PERSISTENT`. Такие сообщения не будут потеряны в случае сбоя провайдера сообщений. Для получения дополнительной информации см. Указание персистентности сообщения.

Транзакции позволяют отправлять или получать несколько сообщений в атомарной операции. В платформе Jakarta EE они также позволяют комбинировать отправку и получение сообщений со считыванием и записью базы данных в рамках одной транзакции. Транзакция — это атомарное действие, которое может объединять ряд операций, таких как отправка и получение сообщений, так что либо все операции выполняются успешно, либо все завершаются неудачно. Подробнее см. Использование локальных транзакций JMS.

Самый надёжный способ получения сообщений — делать это в транзакции, неважно из очереди или из долговременной подписки на тему. Для получения дополнительной информации см. Создание долговременных подписок, Создание вре́менных пунктов назначения и Использование локальных транзакций JMS.

Некоторые из функций позволяют приложению повысить производительность. Например, может быть указан срок действия сообщений (см. Установка срока действия сообщения), чтобы потребители не получали ненужную устаревшую информацию. Сообщения могут отправляться асинхронно. Смотрите Асинхронная отправка сообщений.

Также могут быть указаны различные уровни контроля над подтверждением сообщения. Смотрите Управление подтверждением сообщения.

Другие функции могут предоставить полезные возможности, не связанные с надёжностью. Например, могут быть созданы вре́менные пункты назначения, которые будут действовать только на время создавшего их соединения. Смотрите Создание вре́менных пунктов назначения для получения подробной информации.

В следующих разделах описываются эти функции применительно к клиентским приложениям или клиентам Java SE. Некоторые из функций работают по-разному в Web- и EJB-контейнере Jakarta EE. В этих случаях различия отмечаются здесь и подробно объясняются в разделе Использование Jakarta Messaging в приложениях Jakarta EE.

Управление подтверждением сообщения

Пока сообщение Jakarta Messaging не будет подтверждено, оно не считается успешно доставленным. Успешная доставка сообщения обычно состоит из трёх этапов.

1. Клиент получает сообщение.
2. Клиент обрабатывает сообщение.
3. Сообщение подтверждено.
   Подтверждение инициируется либо провайдером сообщений, либо клиентом в зависимости от режима подтверждения сессии.

В локальных сессиях (см. Использование локальных транзакций JMS), сообщение подтверждается, когда сессия фиксируется. Если транзакция откатывается, все сообщения будут доставляться повторно.

В транзакции Jakarta (в web- или EJB-контейнере Jakarta EE) сообщение подтверждается при фиксации транзакции.

В нетранзакционных сессиях время и способ подтверждения сообщения зависят от значения, которое может быть указано в аргументе метода `createContext`. Возможные значения аргумента следующие.

- `JMSContext.AUTO_ACKNOWLEDGE`: этот параметр используется по умолчанию для клиентских приложений и клиентов Java SE. `JMSContext` автоматически подтверждает получение клиентом сообщения, либо когда клиент успешно возвратил вызов `receive`, либо когда вызов `MessageListener` для обработки сообщения возвращается успешно.

  Синхронный приём в `JMSContext`, настроенный на использование автоматического подтверждения, является единственным исключением из правила, согласно которому получение сообщений является трёхэтапным процессом, как описано ранее. В этом случае получение и подтверждение происходит за один шаг, после чего следует обработка сообщения.
- `JMSContext.CLIENT_ACKNOWLEDGE`: клиент подтверждает сообщение, вызывая метод `acknowledge` сообщения. В этом режиме подтверждение происходит на уровне сессии: подтверждение полученного сообщения автоматически подтверждает получение всех сообщений, которые были получены его сессией. Например, если потребитель сообщения получает десять сообщений, а затем подтверждает пятое доставленное сообщение, все десять сообщений подтверждаются.

В платформе Jakarta EE параметр `JMSContext.CLIENT_ACKNOWLEDGE` можно использовать только в клиентском приложении, но не в веб-компоненте или Enterprise-бине.

- `JMSContext.DUPS_OK_ACKNOWLEDGE` : эта опция указывает `JMSContext` отложенно (lazy) подтверждать доставку сообщений. Вероятнее всего, это приведёт к дублированию доставки некоторых сообщений в случае сбоя провайдера сообщений, поэтому его следует использовать только потребителями, которые терпимы к дублированию сообщений. (Если провайдер сообщений повторно доставляет сообщение, он должен установить заголовку `JMSRedelivered` сообщения значение `true` .) Этот параметр может снизить накладные расходы сессии за счёт минимизации работы, выполняемой сессией для предотвращения дублирования.

Если сообщения были получены из очереди, но не подтверждены при закрытии `JMSContext` , провайдер обмена сообщениями сохраняет их и повторно доставляет при следующем обращении потребителя к очереди. Поставщик также сохраняет неподтверждённые сообщения, если приложение закрывает `JMSContext` , который получал сообщения из длительной подписки. (См. Создание долговременных подписок.) Неподтверждённые сообщения, полученные из подписки, не являющейся долговременной, будут потеряны при закрытии `JMSContext` .

Если используется очередь или долговременная подписка, можно использовать метод `JMSContext.recover` , чтобы остановить нетранзакционный `JMSContext` и перезапустить его с его первым неподтверждённым сообщением. Фактически, серия доставленных сообщений `JMSContext` сбрасывается в точку после последнего подтверждённого сообщения. Сообщения, которые он теперь доставляет, могут отличаться от первоначально доставленных, если срок действия сообщений истёк или были получены сообщения с более высоким приоритетом. Для потребителя с подпиской, не являющейся долговременной, поставщик может отбрасывать неподтверждённые сообщения при вызове метода `JMSContext.recover` .

Пример программы в Подтверждение сообщений демонстрирует два способа гарантировать, что сообщение не будет подтверждено до тех пор, пока обработка сообщения не будет завершена.

Указание параметров для отправки сообщений

Вы можете установить ряд параметров при отправке сообщения. Эти параметры позволяют выполнять задачи, описанные в следующих разделах:

- Указание персистентности сообщений — укажите, что сообщения являются персистентными, то есть они не должны быть потеряны в случае сбоя провайдера.
- Установка приоритета сообщений — установите сообщениям приоритет, который может повлиять на порядок доставки сообщений.
- Установка срока действия сообщения — укажите срок действия сообщений, чтобы устаревшие сообщения не доставлялись.
- Указание задержки доставки — укажите задержку доставки для сообщений, чтобы они не доставлялись до истечения заданного времени.
- Использование цепочки методов JMSProducer — цепочка методов позволяет указать более одной из этих опций при создании производителя и вызове метода `send` .

Указание персистентности сообщения

Jakarta Messaging поддерживает два режима доставки, указывающих, будут ли сообщения потеряны в случае сбоя провайдера сообщений. Эти режимы доставки являются полями интерфейса `DeliveryMode` .

- Режим доставки по умолчанию, `PERSISTENT`, предписывает провайдеру сообщений проявлять особую осторожность, чтобы гарантировать, что сообщение не будет потеряно при передаче в случае сбоя провайдера сообщений. Сообщение, отправленное в этом режиме доставки, записывается в стабильное хранилище при его отправке.
- Режим доставки `NON_PERSISTING` не требует, чтобы провайдер сообщений хранил сообщение или иным образом гарантировал, что оно не будет потеряно в случае сбоя провайдера.

Для указания режима доставки используйте метод `setDeliveryMode` интерфейса `JMSProducer`, чтобы установить режим доставки для всех сообщений, отправленных этим производителем.

Для установки режима доставки при создании производителя и отправке сообщения может использоваться цепочка методов. Следующий вызов создаёт производителя с режимом доставки `NON_PERSISTENT` и использует его для отправки сообщения:

JAVA

```
context.createProducer()
       .setDeliveryMode(DeliveryMode.NON_PERSISTENT).send(dest, msg);
```

Если не указан режим доставки, по умолчанию используется значение `PERSISTENT`. Режим доставки `NON_PERSISTENT` может повысить производительность и снизить накладные расходы на хранение, но должен использоваться только в том случае, если приложение терпимо к потере сообщений.

Установка приоритета сообщений

Можно использовать уровни приоритета сообщений, чтобы указать провайдеру сообщений сначала доставлять срочные сообщения. Используйте метод `setPriority` интерфейса `JMSProducer` для установки приоритета всем сообщений, отправленных этим производителем.

Используйте цепочку методов для установки приоритета при создании производителя и отправке сообщения. Например, следующий вызов устанавливает приоритет уровня 7 для производителя и затем отправляет сообщение:

JAVA

```
context.createProducer().setPriority(7).send(dest, msg);
```

Десять уровней приоритета варьируются от 0 (самый низкий) до 9 (самый высокий). Если не указано, по умолчанию используется приоритет уровня 4. Провайдер сообщений пытается доставлять сообщения с более высоким приоритетом перед сообщениями с более низким приоритетом, но не обязан доставлять сообщения в точном порядке приоритета.

Установка срока действия сообщения

По умолчанию сообщение никогда не истекает. Если сообщение через некоторое время устареет, ему может быть установлен срок действия. Для установки срока действия по умолчанию для всех сообщений, отправленных этим производителем, используется метод `setTimeToLive` интерфейса `JMSProducer`.

Например, сообщение, содержащее быстро изменяющиеся данные, такие как цена акций, устареет через несколько минут, поэтому можно настроить срок действия сообщений в эти несколько минут.

Для установки времени действия при создании производителя и отправке сообщения может быть использована цепочка методов. Например, следующий вызов устанавливает пять минут для производителя, а затем отправляет сообщение:

JAVA

```
context.createProducer().setTimeToLive(300000).send(dest, msg);
```

Если указанное значение `timeToLive` равно `0`, срок действия сообщения никогда не истечёт.

Когда сообщение отправлено, указанный `timeToLive` добавляется к текущему времени, чтобы указать время действия. Любое сообщение, не доставленное до истечения указанного срока, уничтожается. Уничтожение устаревших сообщений экономит ресурсы хранения и вычислительные ресурсы.

Указание задержки доставки

Можно указать время, которое должно пройти после отправки сообщения до того, как провайдер сообщений доставит сообщение. Используйте метод `setDeliveryDelay` интерфейса `JMSProducer`, чтобы установить задержку доставки для всех сообщений, отправленных этим производителем.

Для установки задержки доставки при создании производителя и отправке сообщения может использоваться цепочка методов. Например, следующий вызов устанавливает задержку доставки в 3 секунды для производителя, а затем отправляет сообщение:

JAVA

```
context.createProducer().setDeliveryDelay(3000).send(dest, msg);
```

Использование цепочки методов JMSProducer

Set-методы в интерфейсе `JMSProducer` возвращают объекты `JMSProducer`, поэтому может быть использована цепочка методов для создания производителя, установки нескольких свойств и отправки сообщения. Например, следующие вызовы связанных методов создают производителя, устанавливают пользовательское свойство, срок действия, режим доставки и приоритет сообщения, а затем отправляют сообщение в очередь:

JAVA

```
context.createProducer()
        .setProperty("MyProperty", "MyValue")
        .setTimeToLive(10000)
        .setDeliveryMode(NON_PERSISTENT)
        .setPriority(2)
        .send(queue, body);
```

Также могут быть вызваны методы `JMSProducer` для установки свойств сообщения, а затем отправлено сообщение отдельным вызовом метода `send`. Свойства сообщения можно установить непосредственно в сообщении.

Создание вре́менных пунктов назначения

Как правило, назначения JMS (очереди и темы) создаются административно, а не программно. Ваш провайдер обмена сообщениями включает в себя утилиту для создания и удаления пунктов назначения, и, как правило, пункты назначения являются долгосрочными.

Jakarta Messaging также позволяет создавать пункты назначения (объекты `TemporaryQueue` и `TemporaryTopic`), которые длятся только в течение всего соединения, в котором они созданы. Эти пункты назначения создаются динамически методами `JMSContext.createTemporaryQueue` и `JMSContext.createTemporaryTopic`, как в следующем примере:

JAVA

```
TemporaryTopic replyTopic = context.createTemporaryTopic();
```

Единственные потребители сообщений, которые могут получать из вре́менного пункта назначения, — это пользователи, созданные тем же соединением, которое создало пункт назначения. Производитель может отправить любое сообщение во вре́менный пункт назначения. Если закрывается соединение, которому принадлежит вре́менный пункт назначения, пункт назначения также закрывается и его содержимое теряется.

Вре́менные пункты назначения могут использоваться для реализации простого механизма запроса/ответа. При создании вре́менного пункта назначения и указании его в качестве значения поля заголовка сообщения `JMSReplyTo` при отправке сообщения получатель сообщения может использовать значение поля `JMSReplyTo` в качестве пункта назначения, на которое он отправляет ответ. Потребитель также может ссылаться на исходный запрос, задав в поле заголовка `JMSCorrelationID` ответного сообщения значение поля заголовка `JMSMessageID` запроса. Например, метод `onMessage` может создать `JMSContext`, чтобы он мог отправить ответ на полученное сообщение. Он может использовать такой код:

JAVA

```
replyMsg = context.createTextMessage("Consumer processed message: "
        + msg.getText());
replyMsg.setJMSCorrelationID(msg.getJMSMessageID());
context.createProducer().send((Topic) msg.getJMSReplyTo(), replyMsg);
```

Пример см. в разделе Использование сущности для объединения сообщений из двух MDB.

Использование локальных транзакций JMS

Транзакция выполняет ряд операций атомарно. Если какая-либо из операций завершается неудачно, транзакцию можно откатить, а операции повторить с самого начала. Если все операции выполнены успешно, транзакция может быть зафиксирована.

В клиентском приложении или клиенте Java SE можно использовать локальные транзакции для группировки отправки и получения сообщений. Метод `JMSContext.commit` используется для совершения транзакции. Можно отправить несколько сообщений в транзакции, и сообщения не будут добавлены в очередь или тему, пока транзакция не будет зафиксирована. При получении нескольких сообщений в транзакции они не будут подтверждены до тех пор, пока транзакция не будет зафиксирована.

Метод `JMSContext.rollback` может использоваться для отката транзакции. Откат транзакции означает, что все созданные сообщения уничтожаются, а все использованные сообщения восстанавливаются и доставляются, если срок их действия не истёк (см. Установка срока действия сообщения).

Транзакционная сессия всегда участвует в транзакции. Чтобы создать транзакционную сессию, вызовите метод `createContext` следующим образом:

JAVA

```
JMSContext context =
        connectionFactory.createContext(JMSContext.SESSION_TRANSACTED);
```

Как только вызывается метод `commit` или `rollback`, одна транзакция заканчивается и начинается другая. Закрытие транзакции отменяет текущую транзакцию, включая все ожидающие отправки и получения.

В приложении, запускаемом в веб- или EJB-контейнере Jakarta EE, нельзя использовать локальные транзакции. Вместо этого используйте транзакции Jakarta, описанные в разделе Использование Jakarta Messaging в приложениях Jakarta EE.

Несколько отправок и получений могут быть объединены в одной локальной транзакции JMS, если они все выполняются с использованием одного и того же `JMSContext`.

Не используйте одну транзакцию с механизмом запроса/ответа, в котором отправляется сообщение, а затем получается ответ на это сообщение. Если попытаться это сделать, программа зависнет, потому что отправка не может быть выполнена до тех пор, пока транзакция не будет зафиксирована. Следующий фрагмент кода иллюстрирует проблему:

```java
// Не делайте так!
outMsg.setJMSReplyTo(replyQueue);
context.createProducer().send(outQueue, outMsg);
consumer = context.createConsumer(replyQueue);
inMsg = consumer.receive();
context.commit();
```

Поскольку сообщение, отправленное в рамках транзакции, фактически не отправляется, пока транзакция не будет зафиксирована, транзакция не может содержать никаких получений сообщений, зависящих от того, что это сообщение было отправлено.

Отправка и получение сообщения не могут быть частью одной транзакции. Причина в том, что транзакции происходят между клиентами и провайдером сообщений, который находится между производителем и потребителем сообщения. Рисунок 48-8 иллюстрирует это взаимодействие.

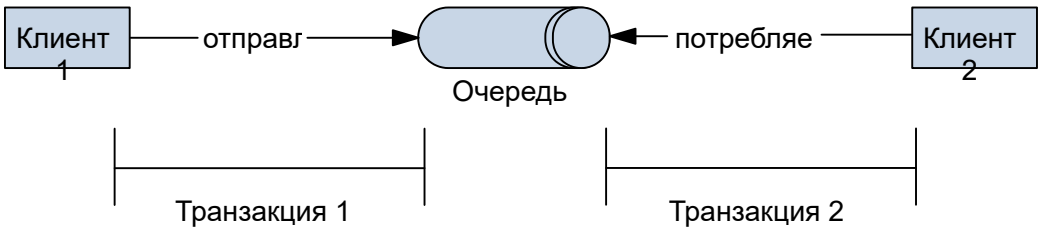

Рис. 48-8. Использование локальных транзакций JMS

Отправка одного или нескольких сообщений одному или нескольким адресатам Клиентом 1 может формировать единую транзакцию, поскольку она формирует единый набор взаимодействий с провайдером сообщений с использованием единственного `JMSContext`. Аналогично, получение одного или нескольких сообщений от одного или нескольких пунктов назначения клиентом 2 также формирует одну транзакцию с использованием одного `JMSContext`. Но поскольку оба клиента не имеют прямого взаимодействия и используют два разных объекта `JMSContext`, единая транзакция для них невозможна.

Другой способ заключается в том, что транзакция — это договор между клиентом и провайдером сообщений, который определяет, отправляется ли сообщение в пункт назначения или поступает сообщение от пункта назначения. Это не контракт между отправляющим и получающим клиентами.

В этом принципиальное отличие обмена сообщениями от синхронной обработки. Вместо сильной связности отправителя и получателя сообщения, JMS связывает отправителя сообщения с пунктом назначения и отдельно пункт назначения с получателем сообщения. Следовательно, хотя отправка и получение имеют сильную связь с провайдером сообщений, они не связаны друг с другом.

При создании `JMSContext` можно указать, будет ли он транзакционным, передавая аргумент `JMSContext.SESSION_TRANSACTED` методу `createContext`. Например:

```java
try (JMSContext context = connectionFactory.createContext(
        JMSContext.SESSION_TRANSACTED);) {
    ...
}
```

Методы `commit` и `rollback` для локальных транзакций связаны с сессией, которая лежит в основе `JMSContext`. Операции над несколькими очередями или темами или комбинациями очередей и тем могут быть объединены в одной транзакции, если для выполнения операций используется одна и та же сессия.

Например, можно использовать тот же `JMSContext`, чтобы получить сообщение из очереди и отправить сообщение в тему в той же транзакции.

Пример в Использование локальных транзакций показывает, как использовать локальные транзакции JMS.

Асинхронная отправка сообщений

Обычно при отправке персистентного сообщения метод `send` блокируется до тех пор, пока провайдер сообщений не подтвердит, что сообщение было успешно отправлено. Механизм асинхронной отправки позволяет приложению отправить сообщение и продолжить работу в ожидании информации о завершении отправки.

Эта функция в настоящее время доступна только в клиентских приложениях и клиентах Java SE.

Асинхронная отправка сообщения включает предоставление объекта Callback-вызова. Укажите `CompletionListener` с помощью метода `onCompletion`. Например, следующий код создаёт объект `CompletionListener` с именем `SendListener`. Затем он вызывает метод `setAsync` для указания, что отправка от этого производителя должна быть асинхронной и использован указанный слушатель:

JAVA

```
CompletionListener listener = new SendListener();
context.createProducer().setAsync(listener).send(dest, message);
```

Класс `CompletionListener` должен реализовывать два метода: `onCompletion` и `onException`. Метод `onCompletion` вызывается в случае успешной отправки, а метод `onException` вызывается в случае сбоя. Простая реализация этих методов может выглядеть так:

JAVA

```
@Override
public void onCompletion(Message message) {
    System.out.println("onCompletion method: Send has completed.");
}

@Override
public void onException(Message message, Exception e) {
    System.out.println("onException method: send failed: " + e.toString());
    System.out.println("Unsent message is: \n" + message);
}
```

Использование Jakarta Messaging в приложениях Jakarta EE

В этом разделе описывается, чем использование Jakarta Messaging в EJB-приложениях или веб-приложениях отличается от использования в консольных приложениях.

Обзор использования Jakarta Messaging

Общее правило в спецификации платформы Jakarta EE применяется ко всем компонентам Jakarta EE, которые используют JMS API в EJB или веб-контейнерах: прикладные компоненты в веб- и EJB-контейнерах не должны пытаться создавать более одного активного (не закрытого) объекта `Session` для каждого соединения. Однако допускается использование нескольких объектов `JMSContext`, поскольку они объединяют одно соединение и одну сессию.

Это правило не распространяется на клиентские приложения. Клиентский контейнер приложения поддерживает создание нескольких сессий для каждого соединения.

Создание ресурсов для приложений Jakarta EE

Можно использовать аннотации для создания специфичных для приложения фабрик соединений и пунктов назначения для Enterprise-бина Jakarta EE или веб-компонентов. Ресурсы, созданные таким образом, видны только приложению, для которого их создают.

Можно использовать элементы дескриптора развёртывания для создания этих ресурсов. Элементы, указанные в дескрипторе развёртывания, переопределяют элементы, указанные в аннотациях. Смотрите Упаковка приложений для получения базовой информации о дескрипторах развёртывания. Для создания специфичных для приложения ресурсов для клиентских приложений нужно использовать дескриптор развёртывания.

Чтобы создать пункт назначения, используйте аннотацию `@JMSDestinationDefinition`, как показано ниже:

JAVA

```
@JMSDestinationDefinition(
    name = "java:app/jms/myappTopic",
    interfaceName = "jakarta.jms.Topic",
    destinationName = "MyPhysicalAppTopic"
  )
```

Элементы `name`, `interfaceName` и `destinationName` обязательны. Указание элемента `description` опционально. Чтобы создать несколько пунктов назначения, заключите их в аннотацию `@JMSDestinationDefinitions`, разделив запятыми.

Чтобы создать фабрику соединений, используйте аннотацию `@JMSConnectionFactoryDefinition`, как показано ниже для класса:

JAVA

```
@JMSConnectionFactoryDefinition(
    name="java:app/jms/MyConnectionFactory"
)
```

Элемент `name` обязателен. При желании могут быть указаны ряд других элементов, таких как `clientId`, если будет использоваться фабрика соединений для долговременных подписок, или `description`. Если не указать элемент `interfaceName`, интерфейсом по умолчанию является `jakarta.jms.ConnectionFactory`. Чтобы создать несколько фабрик соединений, заключите их в аннотацию `@JMSConnectionFactoryDefinitions`, разделив запятыми.

Аннотация должна быть указана только один раз для данного приложения, в любом из компонентов.

> Если приложение содержит один или несколько компонентов, управляемых сообщениями, вы можете разместить аннотацию в одном из компонентов, управляемых сообщениями. Если аннотация размещена в отправляющем компоненте, таком как клиентское приложение, должен быть указан элемент `mappedName` для поиска темы вместо использования свойства `destinationLookup` объекта конфигурации активации.

Когда инъецируется ресурс в компонент, нужно использовать значение элемента `name` в аннотации определения в качестве значения элемента `lookup` аннотации `@Resource`:

JAVA

```
@Resource(lookup = "java:app/jms/myappTopic")
private Topic topic;
```

Доступны следующие переносимые пространства имён JNDI. Какие из них использовать, зависит способа упаковки приложения.

- `java:global` : делает ресурс доступным для всех развёрнутых приложений
- `java:app` : делает ресурс доступным для всех компонентов во всех модулях в одном приложении
- `java:module` : делает ресурс доступным для всех компонентов в данном модуле (например, все EJB-компоненты в модуле Jakarta Enterprise Beans)
- `java:comp` : делает ресурс доступным только для одного компонента (кроме веб-приложения, где он эквивалентен `java:module`)

См. документацию API о деталях этих аннотаций. Все примеры в Отправка и получение сообщений с использованием простого веб-приложения, Отправка сообщений из сессионного компонента в MDB и Использование сущности для объединения сообщений из двух MDB используют аннотацию `@JMSDestinationDefinition` . Другие примеры JMS не используют эти аннотации. Примеры, состоящие только из клиентских приложений, не развёртываются на сервере приложений и поэтому должны обмениваться данными друг с другом с помощью административно созданных ресурсов, которые существуют за пределами отдельных приложений.

Использование инъецирования ресурсов в Enterprise-бине и веб-компонентах

Инъецирование ресурсов может использоваться в приложениях Jakarta EE для инъецирования как администрируемых объектов, так и объектов `JMSContext` .

Инъецирование фабрики соединений, очереди и темы

Обычно для инъецирования `ConnectionFactory` , `Queue` или `Topic` в приложение Jakarta EE используется аннотация `@Resource` . Эти объекты должны быть созданы административно перед развёртыванием приложения. Возможно, вы захотите использовать фабрику соединений по умолчанию с именем JNDI `java:comp/DefaultJMSConnectionFactory` .

При использовании инъецирования ресурсов в компоненте клиента приложения обычно объявляют статические ресурсы сервиса сообщений:

JAVA

```
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
private static ConnectionFactory connectionFactory;

@Resource(lookup = "jms/MyQueue")
private static Queue queue;
```

Однако, при использовании этой аннотации в сессионном компоненте, управляемом сообщениями компоненте или веб-компоненте, ресурс не должен объявляться статически:

JAVA

```
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
private ConnectionFactory connectionFactory;

@Resource(lookup = "jms/MyTopic")
private Topic topic;
```

Объявление ресурса статическим в этих компонентах приведёт к ошибкам во время выполнения.

Инъецирование объекта JMSContext

Чтобы получить доступ к объекту `JMSContext` в Enterprise-бине или веб-компоненте, вместо инъецирования ресурса `ConnectionFactory` и создания `JMSContext` могут использоваться аннотации `@Inject` и `@JMSConnectionFactory` для инъецирования `JMSContext` . Чтобы использовать фабрику соединений по умолчанию, используется код, подобный следующему:

JAVA

```
@Inject
private JMSContext context1;
```

Чтобы использовать собственную фабрику соединений, используется код, подобный следующему:

JAVA

```
@Inject
@JMSConnectionFactory("jms/MyConnectionFactory")
private JMSContext context2;
```

Использование компонентов Jakarta EE для отправки и синхронного получения сообщений

Приложение, которое создаёт сообщения или синхронно получает их, может использовать веб-компонент Jakarta EE или Jakarta Enterprise Beans, такой как Managed-бин, сервлет или сессионный компонент, для выполнения этих операций. В примере Отправка сообщений из сессионного компонента в MDB используется сессионный компонент без сохранения состояния для отправки сообщений в тему. В примере Отправка и получение сообщений с использованием простого веб-приложения используются Managed-бины для отправки и получения сообщений.

Поскольку синхронный приём без заданного тайм-аута связывает ресурсы сервера, этот механизм обычно не лучшее решение для веб-компонента или компонента Jakarta Enterprise Beans. Вместо этого используется синхронное получение с указанием тайм-аута или управляемый сообщениями компонент с асинхронным получением сообщений. Для подробной информации о синхронном получении см. Потребители сообщений JMS.

Использование Jakarta Messaging в компоненте Jakarta EE во многом аналогично использованию его в клиенте приложения. Основными отличиями являются области управления ресурсами и транзакции.

Управление ресурсами Jakarta Messaging в веб-компонентах и компонентах Jakarta Enterprise Beans

Ресурсы JMS — это соединение и сессия, которые обычно объединяются в объект `JMSContext`. В общем, важно освободить ресурсы обмена сообщений, когда они больше не используются. Вот несколько полезных практик.

- Если вы хотите поддерживать ресурс обмена сообщениями только в течение жизненного цикла бизнес-метода, используйте оператор `try`-with-resources для создания `JMSContext`, чтобы он был закрыт автоматически в конце блока `try`.
- Чтобы поддерживать ресурс обмена сообщениями в течение транзакции или запроса, инъецируйте `JMSContext`, как описано в разделе Инъецирование объекта JMSContext. Это также приведёт к освобождению ресурса, когда он больше не нужен.
- Если вы хотите поддерживать ресурс Messaging в течение всего срока службы объекта EJB-компонента, можно использовать вызов Callback-метода `@PostConstruct` для создания ресурса и вызов Callback-метода `@PreDestroy` для закрытия ресурса. Однако обычно в этом нет необходимости, поскольку серверы приложений, как правило, поддерживают пул соединений. Если используется сессионный компонент с состоянием и нужно поддерживать ресурс обмена сообщениями в кэшированном состоянии, нужно закрыть ресурс в Callback-методе `@PrePassivate` и установить для него значение `null`, и снова создать его Callback-методе `@PostActivate`.

Управление транзакциями в сессионных бинах

Вместо использования локальных транзакций используйте транзакции Jakarta. Можно использовать как транзакции, управляемые контейнером, так и транзакции, управляемые компонентом. Как правило, транзакции, управляемые контейнером, используются для методов компонента, которые выполняют

отправку или получение, позволяя EJB-контейнеру обрабатывать разграничение транзакций. Поскольку по умолчанию используются управляемые контейнером транзакции, указывать их не нужно.

Можно использовать транзакции, управляемые компонентом, и `jakarta.transaction.UserTransaction`, но это стоит делать только в том случае, если приложение имеет особые требования, и вы имеете экспертные знания в использовании транзакций. Обычно управляемые контейнером транзакции реализуют наиболее эффективное и правильное поведение. В этом руководстве нет примеров транзакций, управляемых компонентом.

Использование управляемых сообщениями бинов для асинхронного получения сообщений

Разделы Что такое компонент, управляемый сообщениями? и Как Jakarta Messaging работает с платформой Jakarta EE? описывают, как платформа Jakarta EE поддерживает управляемый сообщениями компонент, который позволяет приложениям Jakarta EE обрабатывать сообщения Jakarta Messaging асинхронно. Другие веб- и EJB-компоненты Jakarta EE позволяют отправлять сообщения и получать их синхронно, но не асинхронно.

Управляемый сообщениями компонент — это слушатель сообщений, которому сообщения могут быть доставлены из очереди или из темы. Сообщения могут отправляться любым компонентом Jakarta EE (из клиентского приложения, другого Enterprise-бина или веб-компонента) или из приложения или системы, которая не использует технологию Jakarta EE.

К классу бина, управляемого сообщениями, предъявляются следующие требования.

- Он должен быть аннотирован `@MessageDriven`, если не использует дескриптор развёртывания.
- Класс должен быть определён как `public`, но не как `abstract` или `final`.
- Он должен содержать публичный конструктор без аргументов.

Рекомендуется, но не обязательно, чтобы класс управляемого сообщениями компонента реализовывал интерфейс слушателя сообщений для поддерживаемого им типа сообщения. Компонент, поддерживающий Jakarta Messaging, реализует интерфейс `jakarta.jms.MessageListener`, то есть он должен предоставить метод `onMessage` со следующей сигнатурой:

```
void onMessage(Message inMessage)
```
JAVA

Метод `onMessage` вызывается контейнером, когда для компонента поступает сообщение. Этот метод содержит бизнес-логику, которая обрабатывает сообщения. Бин, управляемый сообщениями, ответствен за анализ сообщения и выполнение необходимой бизнес-логики.

Компонент, управляемый сообщениями, отличается от приёмника сообщений клиентского приложения следующими способами.

- В клиентском приложении необходимо создать `JMSContext`, затем `JMSConsumer`, а затем вызвать `setMessageListener` для активации слушателя. Для компонента, управляемого сообщениями, нужно только определить класс и аннотировать его, а EJB-контейнер создаст его.
- Класс бина использует аннотацию `@MessageDriven`, которая обычно включает элемент `activationConfig`, содержащий аннотации `@ActivationConfigProperty`, определяющие свойства, используемые бином или фабрикой соединений. Эти свойства могут включать фабрику соединений, тип пункта назначения, долговременную подписку, селектор сообщений или режим подтверждения. Некоторые из примеров в главе 49 *Примеры Jakarta Messaging* задают эти свойства. Также можно установить свойства в дескрипторе развёртывания.

- Контейнер клиентского приложения имеет только один объект `MessageListener`, который вызывается одновременно для одного потока. Однако управляемый сообщениями компонент может иметь несколько объектов, сконфигурированных контейнером, которые могут вызываться одновременно несколькими потоками (хотя каждый объект вызывается только одним потоком одновременно). Поэтому бины, управляемые сообщениями, могут обрабатывать сообщения намного быстрее, чем слушатели сообщений.
- Не нужно указывать режим подтверждения сообщения, если не используются транзакции, управляемые компонентом. Сообщение будет получено в транзакции, в которой вызывается метод `onMessage`.

Таблица 48-3 перечисляет свойства конфигурации активации, определённые в спецификации JMS.

Таблица 48-3. Параметры @ActivationConfigProperty для управляемых сообщениями компонентов

Имя свойства	**Описание**
`acknowledgeMode`	Режим подтверждения, используемый только для управляемых компонентом транзакций. По умолчанию используется `Auto-acknowledge` (также разрешено `Dups-ok-acknowledge`)
`destinationLookup`	Имя для поиска очереди или темы, из которой бин будет получать сообщения
`destinationType`	`jakarta.jms.Queue` или `jakarta.jms.Topic`
`subscriptionDurability`	Для долговременных подписок нужно установить значение `Durable`. Смотрите Создание долговременных подписок для получения дополнительной информации
`clientId`	Для долговременных подписок — идентификатор клиента для подключения (необязательно)
`subscriptionName`	Для долговременных подписок — название подписки
`messageSelector`	Строка, фильтрующая сообщения. Смотрите Селекторы сообщений JMS для информации
`connectionFactoryLookup`	Имя фабрики соединений, которая будет использоваться для подключения к провайдеру сообщений, от которого компонент будет получать сообщения.

Вот пример компонента, управляемого сообщениями, используемый в Асинхронный приём сообщений с использованием бина, управляемого сообщениями:

JAVA

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup",
            propertyValue = "jms/MyQueue"),
    @ActivationConfigProperty(propertyName = "destinationType",
            propertyValue = "jakarta.jms.Queue")
})
public class SimpleMessageBean implements MessageListener {

    @Resource
    private MessageDrivenContext mdc;
    static final Logger logger = Logger.getLogger("SimpleMessageBean");

    public SimpleMessageBean() {
    }

    @Override
    public void onMessage(Message inMessage) {

        try {
            if (inMessage instanceof TextMessage) {
                logger.log(Level.INFO,
                        "MESSAGE BEAN: Message received: {0}",
                        inMessage.getBody(String.class));
            } else {
                logger.log(Level.WARNING,
                        "Message of wrong type: {0}",
                        inMessage.getClass().getName());
            }
        } catch (JMSException e) {
            logger.log(Level.SEVERE,
                    "SimpleMessageBean.onMessage: JMSException: {0}",
                    e.toString());
            mdc.setRollbackOnly();
        }
    }
}
```

Если Jakarta Messaging интегрирован с сервером приложений с помощью адаптера ресурсов, адаптер ресурсов Messaging обрабатывает эти задачи для EJB-контейнера.

Класс бина обычно инъецирует ресурс `MessageDrivenContext` , предоставляющий некоторые дополнительные методы, которые можно использовать для управления транзакциями (например, `setRollbackOnly`):

JAVA

```
    @Resource
    private MessageDrivenContext mdc;
```

Бин, управляемый сообщениями, никогда не имеет локального или удалённого интерфейса. Вместо этого у него есть только класс бина.

Бин, управляемый сообщениями, в некотором смысле похож на сессионный бин без состояния: его объекты относительно недолговечны и не сохраняют состояния для конкретного клиента. Переменные объекта бина, управляемого сообщениями, могут содержать некоторое состояние при обработке клиентских сообщений: например, соединение с открытой базой данных или ссылка на объект Enterprise-бина.

Как и сессионный компонент без сохранения состояния, управляемый сообщениями компонент может иметь много взаимозаменяемых объектов, работающих одновременно. Контейнер может объединять эти объекты в пулы для одновременной обработки потоков сообщений. Контейнер пытается доставить сообщения в хронологическом порядке, когда это не повлияет на параллельную обработку сообщений, но нет никаких

гарантий относительно точного порядка, в котором сообщения доставляются бинам, управляемым сообщениями. Если для приложения важен порядок сообщений, следует настроить сервер приложений на использование только одного объекта компонента, управляемого сообщениями.

Подробнее о жизненном цикле бина, управляемого сообщениями, см. Жизненный цикл бина, управляемого сообщениями.

Управление транзакциями в Jakarta

Клиенты приложений Jakarta EE и клиенты Java SE используют локальные транзакции JMS (описанные в разделе Использование локальных транзакций Jakarta Messaging), которые позволяют группировать отправляемые и получаемые сообщения в рамках определённой сессии обмена сообщениями. Приложения Jakarta EE, которые работают в веб- или EJB-контейнере, обычно используют Jakarta Transactions для обеспечения целостности доступа к внешним ресурсам. Ключевое различие между транзакцией Jakarta и локальной транзакцией Jakarta Messaging заключается в том, что транзакция Jakarta контролируется менеджерами транзакций сервера приложений. Транзакции Jakarta могут быть распределёнными, что означает, что они могут охватывать несколько ресурсов в одной транзакции, таких как провайдер сообщений и база данных.

Например, распределённые транзакции позволяют нескольким приложениям атомарно выполнять обновления для одной и той же базы данных, а также они позволяют одному приложению атомарно выполнять обновления для нескольких баз данных.

В приложении Jakarta EE, использующем Jakarta Messaging, можно использовать транзакции для объединения отправки или получения сообщений с обновлениями базы данных и другими операциями диспетчера ресурсов. Можно получить доступ к ресурсам из нескольких компонентов приложения в рамках одной транзакции. Например, сервлет может начать транзакцию, получить доступ к нескольким базам данных, вызвать Enterprise-бин, который отправляет сообщение Jakarta Messaging, вызвать другой Enterprise-бин, который обращается в систему ИС с помощью коннекторов, и, наконец, зафиксировать транзакцию. Однако приложение не может отправить сообщение Jakarta Messaging и получить ответ на него в рамках одной транзакции.

Операции Jakarta в пределах EJB- и веб-контейнеров могут быть двух типов.

- Управляемые контейнером транзакции: контейнер контролирует целостность транзакций без необходимости вызывать `commit` или `rollback`. Управляемые контейнером транзакции проще в использовании, чем управляемые бином транзакции. Можно указать соответствующие атрибуты транзакции для методов Enterprise-бина.

  Используйте атрибут транзакции `Required` (по умолчанию), чтобы гарантировать, что метод всегда является частью транзакции. Если транзакция выполняется при вызове метода, метод будет частью этой транзакции. Если нет, новая транзакция будет запущена до вызова метода и будет зафиксирована после его возврата. Смотрите Атрибуты транзакции для получения дополнительной информации.

- Транзакции, управляемые компонентами: могут использоваться вместе с интерфейсом `jakarta.transaction.UserTransaction`, который предоставляет собственные методы `commit` и `rollback`, которые можно использовать для разграничения транзакции. Управляемые бином транзакции рекомендуются только тем, кто имеет опыт программирования транзакций.

С компонентами, управляемыми сообщениями, могут использоваться транзакции как управляемые контейнером, так и управляемые компонентом. Чтобы обеспечить получение и обработку всех сообщений в контексте транзакции, следует использовать транзакции, управляемые контейнером, и атрибут транзакции `Required` (по умолчанию) для метода `onMessage`.

Когда используются транзакции, управляемые контейнером, можно вызывать следующие методы `MessageDrivenContext`.

- `setRollbackOnly`: используйте этот метод для обработки ошибок. Если возникает исключение, `setRollbackOnly` отмечает текущую транзакцию, так что единственным возможным результатом транзакции является откат.
- `getRollbackOnly`: используйте этот метод, чтобы проверить, была ли текущая транзакция помечена для отката.

Если используются транзакции, управляемые компонентом, доставка сообщения методу `onMessage` происходит вне контекста транзакции Jakarta. Транзакция начинается, когда вызывается метод `UserTransaction.begin` в методе `onMessage`, и заканчивается с вызовом `UserTransaction.commit` или `UserTransaction.rollback`. Любой вызов метода `Connection.createSession` должен происходить внутри транзакции.

Использование транзакций, управляемых компонентом, позволяет обрабатывать сообщение с помощью более чем одной транзакции или выполнять некоторые части обработки сообщения вне контекста транзакции. Однако если используются транзакции, управляемые контейнером, сообщение принимается MDB и обрабатывается методом `onMessage` в той же транзакции. Невозможно добиться такого поведения с помощью управляемых компонентом транзакций.

Когда создаётся `JMSContext` в транзакции Jakarta (в веб- или EJB-контейнере), контейнер игнорирует любые указанные вами аргументы, поскольку он управляет всеми свойствами транзакции. Когда создаётся `JMSContext` в веб- или EJB-контейнере и нет транзакции Jakarta, значение (если есть), переданное методу `CreateContext`, должно быть `JMSContext.AUTO_ACKNOWLEDGE` или `JMSContext.DUPS_OK_ACKNOWLEDGE`.

Когда используются управляемые контейнером транзакции, обычно используется атрибут транзакции `Required` (по умолчанию) для бизнес-методов Enterprise-бина.

Не указывайте свойство конфигурации активации `acknowledgeMode` при создании бина, управляемого сообщениями, который использует транзакции, управляемые контейнером. Контейнер автоматически подтверждает сообщение при совершении транзакции.

Если управляемый сообщениями компонент использует транзакции, управляемые компонентом, получение сообщения не может быть частью транзакции, управляемой компонентом. Можно установить для свойства конфигурации активации `acknowledgeMode` значение `Auto-acknowledge` или `Dups-ok-acknowledge`, чтобы указать, как именно сообщение будет подтверждено бином, управляемым сообщениями.

Если метод `onMessage` выдает `RuntimeException`, контейнер не подтверждает обработку сообщения. В этом случае провайдер сообщений повторно доставит неподтвержденное сообщение в будущем.

Дополнительная информация о Jakarta Messaging

Дополнительные сведения о Jakarta Messaging см.

- Веб-сайт Jakarta Messaging:
  https://projects.eclipse.org/projects/ee4j.jms
- Спецификация Jakarta Messaging, версия 3.0, доступна по адресу:
  https://jakarta.ee/specifications/messaging/3.0/

Глава 49. Примеры обмена сообщениями Jakarta

В этой главе приведены примеры, которые показывают, как использовать Jakarta Messaging в различных видах приложений Jakarta EE.

Примеры сборки и запуска Jakarta Messaging

Примеры лежат в каталоге *tut-install*`/examples/jms/` .

Чтобы собрать и запустить каждый пример:

1. Используйте IDE NetBeans или Maven для компиляции, упаковки и в некоторых случаях развёртывания примера.
2. Используйте IDE NetBeans, Maven или команду `appclient` для запуска клиентского приложения или используйте браузер для запуска примеров веб-приложений.

Прежде чем развёртывать и запускать примеры, необходимо создать для них ресурсы. В некоторых примерах есть файл `glassfish-resources.xml` , который используется, чтобы создать ресурсы для этого примера и других. Вы можете использовать команду `asadmin` для создания ресурсов.

Чтобы использовать команды `asadmin` и `appclient` , нужно поместить каталог `bin` GlassFish Server в системную переменную PATH, как описано в разделе Рекомендации по установке GlassFish Server.

Обзор примеров JMS

В следующих таблицах перечислены примеры, используемые в этой главе, описаны их действия и ссылки на раздел, в котором дан подробный разбор каждого из них. Каталог для каждого примера лежит в каталоге *tut-install*`/examples/jms/` .

Таблица 49-1. Примеры JMS, демонстрирующие использование клиентских приложений Jakarta EE

Каталог примера	Описание
`simple/producer`	Использование клиентского приложения для отправки сообщений. Смотрите Отправка сообщений
`simple/synchconsumer`	Использование клиентского приложения для синхронного получения сообщений. Смотрите Синхронное получение сообщений
`simple/asynchconsumer`	Использование клиентского приложения для асинхронного получения сообщений. Смотрите Использование слушателя сообщений для асинхронной доставки сообщений
`simple/messagebrowser`	Как в клиентском приложении использовать `QueueBrowser` для просмотра очереди смотрите Просмотр сообщений в очереди

Каталог примера	Описание
`simple/clientackconsumer`	Использование клиентского приложения для подтверждения сообщений, полученных синхронно. Смотрите Подтверждение сообщений
`durablesubscriptionexample`	Использование клиентского приложения для создания долговременной подписки на тему. Смотрите Использование долговременных подписок
`transactedexample`	Использование клиентского приложения для отправки и получения сообщений в локальных транзакциях (также используется обмен запросами и ответами). Смотрите Использование локальных транзакций
`shared/sharedconsumer`	Использование клиентского приложения для создания общих подписок, не являющихся долговременными, на темы. Смотрите Использование общих подписок, не являющихся долговременными
`shared/shareddurableconsumer`	Использование клиентского приложения для создания общих долговременных подписок. Смотрите Использование общих долговременных подписок

Таблица 49-2 Примеры Jakarta Messaging, показывающие использование в веб- и EJB-компонентах Jakarta EE

Каталог примера	Описание
`websimplemessage`	Использование Managed-бинов для отправки сообщений и синхронного получения сообщений. См. Отправка и получение сообщений в простом веб-приложении
`simplemessage`	Использование приложения-клиента для отправки сообщений и использование бина, управляемого сообщениями, для асинхронного получения сообщений. Смотрите Асинхронный приём сообщений с использованием бина, управляемого сообщениями
`clientsessionmdb`	Использование сессионного компонента для отправки сообщений и использование управляемого сообщениями компонента для получения сообщений. Смотрите Отправка сообщений из сессионного компонента в MDB
`clientmdbentity`	Использование клиентского приложения, двух управляемых сообщениями бинов и устойчивости JPA для создания простого приложения HR. Смотрите Использование сущности для объединения сообщений из двух MDB

Простые приложения JMS

В этом разделе показано, как создавать, упаковывать и запускать простые клиенты сообщений, упакованные как клиенты приложений.

Обзор простого приложения JMS

Клиенты демонстрируют основные задачи, которые должно выполнять приложение JMS:

- Создание `JMSContext`
- Создание сообщений производителей и потребителей
- Отправка и получение сообщений

В каждом примере используются два клиента: один отправляет сообщения, а другой получает их. Вы можете запустить клиентов в двух терминальных окнах.

При создании клиента обмена сообщениями для запуска в EJB-приложении, используются многие из тех же методов в той же последовательности, что и для клиента приложения. Тем не менее, есть некоторые существенные различия. Использование Jakarta Messaging в приложениях Jakarta EE описывает эти различия, и в этой главе приведены примеры, иллюстрирующие их.

Примеры для этого раздела находятся в каталоге *`tut-install`*`/examples/jms/simple/` в следующих подкаталогах:

```
producer/
synchconsumer/
asynchconsumer/
messagebrowser/
clientackconsumer/
```

Перед запуском примеров необходимо запустить GlassFish Server и административно создать некоторые объекты.

Запуск провайдера JMS

При использовании GlassFish Server сервер сервиса обмена сообщениями является GlassFish Server. Запустите сервер, как описано в Запуск и остановка GlassFish Server.

Административное создание объектов JMS

В этом примере используются следующие администрируемые объекты JMS:

- Фабрика соединений
- Два ресурса пунктов назначения: тема и очередь

Перед запуском приложений можно использовать команду `asadmin add-resources` для создания необходимых ресурсов обмена сообщениями, указав в качестве аргумента файл с именем `glassfish-resources.xml` . Этот файл может быть создан в любом проекте с использованием IDE NetBeans, хотя он также может быть создан вручную. Файл для необходимых ресурсов находится в каталоге `jms/simple/seller/src/main/setup/` .

В примерах Jakarta Messaging используется фабрика соединений с именем поиска JNDI `java:comp/DefaultJMSConnectionFactory` , предустановленная в GlassFish Server.

Также может быть использована команда `asadmin create-jms-resource` для создания ресурсов, команда `asadmin list-jms-resources` для отображения их имён и команда `asadmin delete-jms-resource` для их удаления.

Создание ресурсов для простых примеров

Файл `glassfish-resources.xml` в одном из проектов Maven может создать все ресурсы, необходимые для простых примеров.

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В командном окне перейдите к примеру `Producer`.

```
cd tut-install/jms/simple/producer
```
SHELL

3. Создайте ресурсы командой `asadmin add-resources`:

```
asadmin add-resources src/main/setup/glassfish-resources.xml
```
SHELL

4. Проверьте создание ресурсов:

```
asadmin list-jms-resources
```
SHELL

   Команда перечисляет два пункта назначения и фабрику соединений, указанные в файле `glassfish-resources.xml` в дополнение к фабрике соединений платформы по умолчанию:

```
jms/MyQueue
jms/MyTopic
jms/__defaultConnectionFactory
Command list-jms-resources executed successfully.
```
SHELL

   В GlassFish Server ресурсу Jakarta EE `java:comp/DefaultJMSConnectionFactory` соответствует фабрика соединений `jms/__defaultConnectionFactory`.

Сборка всех простых примеров

Чтобы запустить простые примеры с использованием GlassFish Server, упакуйте каждый пример в файл JAR клиентского приложения. JAR-файл клиентского приложения требует файл манифеста, расположенный в каталоге `src/main/java/META-INF/` каждого примера вместе с файлом `.class`.

Файл `pom.xml` каждого примера определяет плагин, который создаёт JAR-файл приложения-клиента. Вы можете создавать примеры, используя IDE NetBeans или Maven.

Сборка всех простых примеров в IDE NetBeans

1. В меню **Файл** выберите **Открыть проект**.
2. В диалоговом окне **Открыть проект** перейдите к:

```
tut-install/examples/jms
```

3. Разверните узел `jms` и выберите каталог `simple`.
4. Кликните **Открыть проект**, чтобы открыть примеры.
5. На вкладке **Проектц** кликните правой кнопкой мыши проект `simple` и выберите **Сборка** для сборки примеров.

Эта команда помещает файлы JAR клиентских приложений в каталоги `target` примеров.

Сборка всех простых примеров, используя Maven

1. В окне терминала перейдите в каталог `simple` :

```
cd tut-install/jms/simple/
```
SHELL

2. Введите следующую команду, чтобы собрать все проекты:

```
mvn install
```
SHELL

Эта команда помещает файлы JAR клиентских приложений в каталоги `target` примеров.

Отправка сообщений

В этом разделе описывается, как использовать клиент для отправки сообщений. Клиент `Producer.java` будет отправлять сообщения во всех этих примерах.

Основные шаги, выполненные в примере

Основные шаги, выполняемые в этом примере, следующие.

1. Инъецируйте ресурсы для административных объектов, используемых в примере.
2. Примените и проверьте аргументы командной строки. Этот пример может использоваться для отправки любого количества сообщений в очередь или тему, поэтому в командной строке при запуске программы нужно указывать тип пункта назначения и количество сообщений.
3. Создайте `JMSContext` , а затем отправьте указанное количество текстовых сообщений в виде строк, как описано в Тело сообщения.
4. Последним отправьте сообщение типа `Message` , чтобы указать, что потребитель не должен ожидать дальнейших сообщений.
5. Отловите и обработайте все возникшие исключения.

Клиент Producer.java

Отправляющий клиент `Producer.java` выполняет следующие шаги.

1. Инъецирует ресурсы фабрики соединений, очереди и темы:

```
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
private static ConnectionFactory connectionFactory;
@Resource(lookup = "jms/MyQueue")
private static Queue queue;
@Resource(lookup = "jms/MyTopic")
private static Topic topic;
```
JAVA

2. Извлекает и проверяет аргументы командной строки, которые определяют тип назначения и количество аргументов:

JAVA

```
final int NUM_MSGS;
String destType = args[0];
System.out.println("Destination type is " + destType);
if ( ! ( destType.equals("queue") || destType.equals("topic") ) ) {
    System.err.println("Argument must be \"queue\" or " + "\"topic\"");
    System.exit(1);
}
if (args.length == 2){
    NUM_MSGS = (new Integer(args[1])).intValue();
} else {
    NUM_MSGS = 1;
}
```

3. Назначает очередь или тему пунктом назначения в зависимости от указанного типа пункта назначения:

JAVA

```
Destination dest = null;
try {
    if (destType.equals("queue")) {
        dest = (Destination) queue;
    } else {
        dest = (Destination) topic;
    }
} catch (Exception e) {
    System.err.println("Error setting destination: " + e.toString());
    System.exit(1);
}
```

4. В блоке `try-with-resources` создаётся `JMSContext`:

JAVA

```
try (JMSContext context = connectionFactory.createContext();) { ... }
```

5. Устанавливает счётчик сообщений равным нулю, затем создаёт `JMSProducer`, отправляет одно или несколько сообщений в пункт назначения и увеличивает счётчик. Сообщения в виде строк имеют тип сообщения `TextMessage`:

JAVA

```
    int count = 0;
    for (int i = 0; i < NUM_MSGS; i++) {
        String message = "This is message " + (i + 1)
                + " from producer";
        // Раскомментируйте следующую строку чтобы отправить сообщения
        System.out.println("Sending message: " + message);
        context.createProducer().send(dest, message);
        count += 1;
    }
    System.out.println("Text messages sent: " + count);
```

6. Посылает пустое сообщение-сигнал для обозначения завершения потока сообщений:

JAVA

```
    context.createProducer().send(dest, context.createMessage());
```

Отправка пустого сообщения неуказанного типа является удобным способом для приложения указать потребителю, что заключительное сообщение получено.

7. Ловит и обрабатывает все возможные исключения. В конце блока `try-with-resources` автоматически вызывается закрытие `JMSContext`:

JAVA

```
} catch (Exception e) {
    System.err.println("Exception occurred: " + e.toString());
    System.exit(1);
}
System.exit(0);
```

Запуск клиента Producer

Вы можете запустить клиент с помощью команды `appclient`. Клиент `Producer` принимает один или два аргумента командной строки: тип пункта назначения и, необязательно, количество сообщений. Если вы не укажете количество сообщений, клиент отправит одно сообщение.

В этом примере клиент используется для отправки трёх сообщений в очередь.

1. Убедитесь, что GlassFish Server был запущен (см. Запуск и остановка GlassFish Server) и созданы ресурсы и простые примеры Jakarta Messaging (см. Административное создание объектов JMS и Создание всех простых примеров).
2. В окне терминала перейдите в каталог `producer`:

   SHELL

   ```
   cd producer
   ```

3. Запустите программу `Producer`, отправив три сообщения в очередь:

   SHELL

   ```
   appclient -client target/producer.jar queue 3
   ```

   Вывод программы выглядит следующим образом (вместе с дополнительным выводом):

   SHELL

   ```
   Destination type is queue
   Sending message: This is message 1 from producer
   Sending message: This is message 2 from producer
   Sending message: This is message 3 from producer
   Text messages sent: 3
   ```

   Сообщения теперь находятся в очереди, ожидая получения.

Запуск клиентских приложений и выполнение команды может занять много времени.

Синхронное получение сообщений

В этом разделе описывается принимающий клиент, который использует метод `receive` для синхронного получения сообщений. Затем в этом разделе объясняется, как запускать клиенты с помощью GlassFish Server.

Клиент SynchConsumer.java

Принимающий клиент `SynchConsumer.java` выполняет следующие шаги.

1. Инъецирует ресурсы фабрики соединений, очереди и темы.
2. Устанавливает очередь или тему пунктом назначения в зависимости от указанного типа пункта назначения.
3. Внутри `try-with-resources`, создаёт `JMSContext`.
4. Создаёт `JMSConsumer` и начинает доставку сообщения:

   JAVA

   ```
   consumer = context.createConsumer(dest);
   ```

5. Получает сообщения, отправленные в пункт назначения до тех пор, пока не будет получено управляющее сообщение конца потока сообщений:

```java
int count = 0;
while (true) {
    Message m = consumer.receive(1000);
    if (m != null) {
        if (m instanceof TextMessage) {
            System.out.println(
                    "Reading message: " + m.getBody(String.class));
            count += 1;
        } else {
            break;
        }
    }
}
System.out.println("Messages received: " + count);
```

   Поскольку сообщение-сигнал не является `TextMessage`, принимающий клиент завершает цикл `while` и прекращает приём сообщений после поступления сообщения-сигнала.

6. Ловит и обрабатывает все возможные исключения. В конце `try-with-resources` автоматически закрывается `JMSContext`.

Клиент `SynchConsumer` использует цикл `while` для получения сообщений, вызывая `receive` с аргументом timeout.

Запуск клиентов SynchConsumer и Producer

Вы можете запустить клиент с помощью команды `appclient`. Клиент `SynchConsumer` принимает один аргумент командной строки — тип пункта назначения.

Эти шаги показывают, как получать и отправлять сообщения синхронно, используя очередь и тему. В этих шагах предполагается, что вы уже запустили клиент `Producer` и в очереди ожидают три сообщения.

1. В том же окне терминала, где вы запустили `Producer`, перейдите в каталог `synchconsumer`:

```shell
cd ../synchconsumer
```

2. Запустите клиент `SynchConsumer` с аргументом queue:

```shell
appclient -client target/synchconsumer.jar queue
```

   Вывод клиента выглядит следующим образом (вместе с дополнительным выводом):

```shell
Destination type is queue
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
Messages received: 3
```

3. Теперь попробуйте запустить клиентов в обратном порядке. Запустите клиент `SynchConsumer`:

```shell
appclient -client target/synchconsumer.jar queue
```

   Клиент отображает тип пункта назначения, а затем ожидает сообщения.

4. Откройте новое окно терминала и запустите клиент `Producer`:

```
cd tut-install/jms/simple/producer
appclient -client target/producer.jar queue 3
```

SHELL

Когда сообщения отправлены, клиент `SynchConsumer` получает их и завершает работу.

5. Теперь запустите клиент `Producer`, используя topic вместо queue:

```
appclient -client target/producer.jar topic 3
```

SHELL

Вывод клиента выглядит следующим образом (вместе с дополнительным выводом):

```
Destination type is topic
Sending message: This is message 1 from producer
Sending message: This is message 2 from producer
Sending message: This is message 3 from producer
Text messages sent: 3
```

SHELL

6. Теперь в другом окне терминала запустите клиент `SynchConsumer`, используя тему:

```
appclient -client target/synchconsumer.jar topic
```

SHELL

Результат, однако, другой. Поскольку используется подписка на тему, сообщения, отправленные до создания подписки на тему, не будут добавлены в подписку и доставлены потребителю. (Подробнее см. Стиль обмена сообщениями "публикация-подписка" и Использование сообщений из тем.) Вместо получения имеющихся в теме сообщений клиент ожидает поступления в тему новых сообщений.

7. Оставьте клиент `SynchConsumer` запущенным и снова запустите клиент `Producer`:

```
appclient -client target/producer.jar topic 3
```

SHELL

Теперь клиент `SynchConsumer` получает сообщения:

```
Destination type is topic
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
Messages received: 3
```

SHELL

Поскольку эти сообщения были отправлены после запуска потребителя, клиент получает их.

Использование слушателя сообщений для асинхронной доставки сообщений

В этом разделе описываются принимающие клиенты в примере, который использует слушатель сообщений для асинхронной доставки сообщений. Затем в этом разделе объясняется, как скомпилировать и запустить клиенты с помощью GlassFish Server.

В платформе Jakarta EE слушатели сообщений могут использоваться только в клиентских приложениях, как в этом примере. Чтобы разрешить асинхронную доставку сообщений в веб-приложении или приложении Enterprise-бина, нужно использовать компонент, управляемый сообщениями, как показано в следующих примерах этой главы.

Написание клиентов AsynchConsumer.java и TextListener.java

Отправляющим клиентом является `Producer.java`, тот же клиент, который используется в Синхронном приёме сообщений.

Асинхронный потребитель обычно работает бесконечно. В этом примере клиент выполняется до тех пор, пока пользователь не введёт символ `q` или `Q`, чтобы остановить его.

1. Клиент `AsynchConsumer.java` выполняет следующие шаги.
   a. Инъецирует ресурсы фабрики соединений, очереди и темы.
   b. Устанавливает очередь или тему пунктом назначения в зависимости от указанного типа пункта назначения.
   c. В блоке `try-with-resources` создаётся `JMSContext`.
   d. Создаёт `JMSConsumer`.
   e. Создаёт объект класса `TextListener` и регистрирует его в качестве слушателя сообщений для `JMSConsumer`:

```
listener = new TextListener();
consumer.setMessageListener(listener);
```
JAVA

   f. Слушает сообщения, отправленные пункту назначения, и завершает работу, когда пользователь вводит символ `q` или `Q` (для этого он использует `java.io.InputStreamReader`).
   g. Ловит и обрабатывает все возможные исключения. В конце `try-with-resources` автоматически закрывается `JMSContext`, тем самым останавливая доставку сообщений слушателю сообщений.
2. Слушатель сообщений `TextListener.java` выполняет следующие действия:
   a. Когда приходит сообщение, метод `onMessage` вызывается автоматически.
   b. Если сообщение имеет тип `TextMessage`, метод `onMessage` отображает его содержимое в виде строкового значения. Если сообщение не является текстовым сообщением, он сообщает об этом:

```
public void onMessage(Message m) {
    try {
        if (m instanceof TextMessage) {
            System.out.println(
                    "Reading message: " + m.getBody(String.class));
        } else {
             System.out.println("Message is not a TextMessage");
        }
    } catch (JMSException | JMSRuntimeException e) {
        System.err.println("JMSException in onMessage(): " + e.toString());
    }
}
```
JAVA

В этом примере будут использоваться те же фабрика соединений и пункт назначения, которые были созданы в Создание ресурсов для простых примеров.

В этих шагах предполагается, что все примеры уже собраны и упакованы, используя IDE NetBeans или Maven.

Запуск клиентов AsynchConsumer и Producer

Вам потребуется два окна терминала, аналогично Синхронное получение сообщений.

1. В окне терминала, где вы запустили клиент `SynchConsumer`, перейдите в каталог примера `asynchconsumer`:

```
cd tut-install/jms/simple/asynchconsumer
```
SHELL

2. Запустите клиент `AsynchConsumer`, указав тип пункта назначения `topic`:

```
appclient -client target/asynchconsumer.jar topic
```
SHELL

Клиент отображает следующие строки (вместе с дополнительным выводом), а затем ожидает сообщения:

```
Destination type is topic
To end program, enter Q or q, then <return>
```
SHELL

3. В окне терминала, где ранее запускался клиент `Producer`, снова запустите клиент, отправив три сообщения:

```
appclient -client target/producer.jar topic 3
```
SHELL

Вывод клиента выглядит следующим образом (вместе с дополнительным выводом):

```
Destination type is topic
Sending message: This is message 1 from producer
Sending message: This is message 2 from producer
Sending message: This is message 3 from producer
Text messages sent: 3
```
SHELL

В другом окне клиент `AsynchConsumer` отображает следующее (вместе с некоторыми дополнительными выходными данными):

```
Destination type is topic
To end program, enter Q or q, then <return>
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
Message is not a TextMessage
```
SHELL

Последняя строка появляется, потому что клиент получил нетекстовое управляющее сообщение, отправленное клиентом `Producer`.

4. Введите `Q` или `q` и нажмите Return, чтобы остановить клиент `AsynchConsumer`.
5. Теперь запустите клиенты, используя очередь.

   В этом случае, как и в синхронном примере, вы можете сначала запустить клиент `Producer`, потому что между отправителем и получателем нет временно́й зависимости:

```
appclient -client target/producer.jar queue 3
```
SHELL

Вывод клиента выглядят так:

```
Destination type is queue
Sending message: This is message 1 from producer
Sending message: This is message 2 from producer
Sending message: This is message 3 from producer
Text messages sent: 3
```
SHELL

6. В другом окне запустите клиент `AsynchConsumer`:

```
appclient -client target/asynchconsumer.jar queue
```
SHELL

Вывод клиента выглядит следующим образом (вместе с дополнительным выводом):

```
Destination type is queue
To end program, enter Q or q, then <return>
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
Message is not a TextMessage
```
SHELL

7. Введите `Q` или `q` и нажмите Return, чтобы остановить клиента.

Просмотр сообщений в очереди

В этом разделе описывается пример, который создаёт объект `QueueBrowser` для проверки сообщений в очереди, как описано в Браузеры очереди JMS. Затем в этом разделе объясняется, как скомпилировать, упаковать и запустить пример с использованием GlassFish Server.

Клиент MessageBrowser.java

Чтобы создать `QueueBrowser` для очереди, вызывается метод `JMSContext.createBrowser` с очередью в качестве аргумента. Затем получаются сообщения в очереди как объект `Enumeration`. Затем можно перебирать объект `Enumeration` и отображать содержимое каждого сообщения.

Клиент `MessageBrowser.java` выполняет следующие шаги.

1. Инъецирует ресурсы для фабрики соединений и очереди.
2. В блоке `try-with-resources` создаётся `JMSContext`.
3. Создаёт `QueueBrowser`:

```
QueueBrowser browser = context.createBrowser(queue);
```
JAVA

4. Получает `Enumeration`, содержащий сообщения:

```
Enumeration msgs = browser.getEnumeration();
```
JAVA

5. Проверяет, что `Enumeration` содержит сообщения, а затем отображает содержимое сообщений:

```
if ( !msgs.hasMoreElements() ) {
    System.out.println("No messages in queue");
} else {
    while (msgs.hasMoreElements()) {
        Message tempMsg = (Message)msgs.nextElement();
        System.out.println("Message: " + tempMsg);
    }
}
```
JAVA

6. Ловит и обрабатывает все возможные исключения. В конце `try-with-resources` автоматически закрывается `JMSContext`.

Вывод содержимого сообщения в стандартный вывод позволяет получить тело и свойства сообщения в формате, который зависит от реализации метода `toString`. В GlassFish Server формат сообщения выглядит примерно так:

```
Text:   This is message 3 from producer
Class:                  com.sun.messaging.jmq.jmsclient.TextMessageImpl
getJMSMessageID():      ID:8-10.152.23.26(bf:27:4:e:e7:ec)-55645-1363100335526
getJMSTimestamp():      1129061034355
getJMSCorrelationID():  null
JMSReplyTo:             null
JMSDestination:         PhysicalQueue
getJMSDeliveryMode():   PERSISTENT
getJMSRedelivered():    false
getJMSType():           null
getJMSExpiration():     0
getJMSPriority():       4
Properties:             {JMSXDeliveryCount=0}
```
SHELL

Вместо того, чтобы отображать содержимое сообщения таким образом, вы можете вызвать некоторые из методов получения интерфейса `Message`, чтобы получить части сообщения, которые хотите увидеть.

В этом примере будет использоваться фабрика соединений и очередь, которая была создана для Синхронного получения сообщений. Предполагается, что все примеры уже собраны и упакованы.

Запуск клиента QueueBrowser

Чтобы запустить пример `MessageBrowser` с помощью команды `appclient`, выполните следующие действия.

Нужен также пример `Producer`, чтобы отправить сообщение в очередь, и один из клиентов-потребителей, чтобы получить сообщения после их проверки.

Для запуска клиентов требуется два окна терминала.

1. В окне терминала перейдите в каталог `producer`:

   ```
   cd tut-install/examples/jms/simple/producer/
   ```
   SHELL

2. Запустите клиент `Producer`, отправив одно сообщение в очередь вместе с нетекстовым сигнальным сообщением:

   ```
   appclient -client target/producer.jar queue
   ```
   SHELL

   Вывод клиента выглядит следующим образом (вместе с дополнительным выводом):

   ```
   Destination type is queue
   Sending message: This is message 1 from producer
   Text messages sent: 1
   ```
   SHELL

3. В другом окне терминала перейдите в каталог `messagebrowser`:

   ```
   cd tut-install/jms/simple/messagebrowser
   ```
   SHELL

4. Запустите клиент `MessageBrowser` следующей командой:

   ```
   appclient -client target/messagebrowser.jar
   ```
   SHELL

   Вывод клиента выглядит примерно так (вместе с дополнительным выводом):

```
Message:
Text:   This is message 1 from producer
Class:                  com.sun.messaging.jmq.jmsclient.TextMessageImpl
getJMSMessageID():      ID:9-10.152.23.26(bf:27:4:e:e7:ec)-55645-1363100335526
getJMSTimestamp():      1363100335526
getJMSCorrelationID():  null
JMSReplyTo:             null
JMSDestination:         PhysicalQueue
getJMSDeliveryMode():   PERSISTENT
getJMSRedelivered():    false
getJMSType():           null
getJMSExpiration():     0
getJMSPriority():       4
Properties:             {JMSXDeliveryCount=0}

Message:
Class:                  com.sun.messaging.jmq.jmsclient.MessageImpl
getJMSMessageID():      ID:10-10.152.23.26(bf:27:4:e:e7:ec)-55645-1363100335526
getJMSTimestamp():      1363100335526
getJMSCorrelationID():  null
JMSReplyTo:             null
JMSDestination:         PhysicalQueue
getJMSDeliveryMode():   PERSISTENT
getJMSRedelivered():    false
getJMSType():           null
getJMSExpiration():     0
getJMSPriority():       4
Properties:             {JMSXDeliveryCount=0}
```

Первое сообщение — это `TextMessage`, а второе — нетекстовое сообщение-сигнал.

5. Перейдите в каталог `synchconsumer`.
6. Запустите клиент `SynchConsumer`, чтобы получить сообщения:

```
appclient -client target/synchconsumer.jar queue
```

Вывод клиента выглядит следующим образом (вместе с дополнительным выводом):

```
Destination type is queue
Reading message: This is message 1 from producer
Messages received: 1
```

Запуск нескольких потребителей для одного пункта назначения

Чтобы дополнительно проиллюстрировать, как работает обмен сообщениями типа «точка-точка» и «публикация-подписка», вы можете использовать примеры `Producer` и `SynchConsumer` для отправки сообщений, которые затем используются двумя клиентами, работающими одновременно.

1. Откройте три командных окна. В одном перейдите в каталог `producer`. В двух других перейдите в каталог `synchconsumer`.
2. В каждом из окон `synchconsumer` запустите клиент, получающий сообщения из очереди:

```
appclient -client target/synchconsumer.jar queue
```

Подождите, пока в обоих окнах не появится сообщение «Destination type is queue».

3. В окне `provider` запустите клиент, отправив в очередь 20 сообщений:

```
appclient -client target/producer.jar queue 20
```
SHELL

4. Посмотрите на вывод в окнах `synchconsumer`. В обмене сообщениями точка-точка каждое сообщение может иметь только одного потребителя. Следовательно, каждый из клиентов получает только часть сообщений. Один из клиентов получает нетекстовое сигнальное сообщение, сообщает количество полученных сообщений и завершает работу.
5. В окне клиента, который не получил нетекстовое сигнальное сообщение, введите Control-C для выхода из программы.
6. Затем запустите клиенты `synchconsumer`, используя тему. В каждом окне выполните следующую команду:

```
appclient -client target/synchconsumer.jar topic
```
SHELL

   Подождите, пока в обоих окнах не появится сообщение «Destination type is topic».

7. В окне `producer` запустите клиент, отправив 20 сообщений в тему:

```
appclient -client target/producer.jar topic 20
```
SHELL

8. Снова посмотрите на вывод в окнах `synchconsumer`. В сообщениях "публикация-подписка" копия каждого сообщения отправляется каждой подписке по теме. Следовательно, каждый из клиентов получает все 20 текстовых сообщений, а также нетекстовое сигнальное сообщение.

Подтверждение сообщений

Jakarta Messaging предоставляет два альтернативных способа для получающего клиента, чтобы гарантировать, что сообщение не будет подтверждено, пока приложение не закончит обработку сообщения:

- Использование синхронного потребителя в `JMSContext`, который был настроен с использованием параметра `CLIENT_ACKNOWLEDGE`
- Использование слушателя сообщений для асинхронной доставки сообщений в `JMSContext`, который был настроен с использованием параметра `AUTO_ACKNOWLEDGE` по умолчанию

> В платформе Jakarta EE сессии `CLIENT_ACKNOWLEDGE` могут использоваться только в клиентских приложениях, как в этом примере.

Пример `clientackconsumer` демонстрирует первый вариант, в котором синхронный потребитель использует подтверждение клиента. Пример `asynchconsumer`, описанный в Использование слушателя сообщений для асинхронной доставки сообщений, демонстрирует второй вариант.

Для получения информации о подтверждении сообщения см. Управление подтверждением сообщения.

В следующей таблице описаны четыре возможных соотношения между типами потребителей и типами подтверждения.

Таблица 49-3. Подтверждение сообщения с синхронными и асинхронными потребителями

Тип потребителя	Тип подтверждения	Поведение

Тип потребителя	Тип подтверждения	Поведение
Синхронный	Клиент	Клиент подтверждает сообщение после завершения обработки
Асинхронный	Клиент	Клиент подтверждает сообщение после завершения обработки
Синхронный	Auto	Подтверждение происходит сразу после вызова `receive`. Сообщение не может быть доставлено в случае сбоя на последующих этапах обработки
Асинхронный	Auto	Сообщение автоматически подтверждается, когда возвращается метод `onMessage`

Пример находится в каталоге *`tut-install`*`/examples/jms/simple/clientackconsumer/`.

Пример клиента `ClientAckConsumer.java` создаёт `JMSContext`, который указывает тип подтверждение Client:

```
try (JMSContext context =
      connectionFactory.createContext(JMSContext.CLIENT_ACKNOWLEDGE);) {
    ...
}
```
JAVA

Клиент использует цикл `while`, практически идентичный циклу `SynchConsumer.java` за тем исключением, что после обработки каждого сообщения он вызывает метод `accept` в `JMSContext`:

```
context.acknowledge();
```
JAVA

В примере используются следующие объекты:

- Ресурс `jms/MyQueue`, который был создан для Синхронного получения сообщений.
- `java:comp/DefaultJMSConnectionFactory` — фабрика соединений платформы по умолчанию, предустановленная в GlassFish Server

Запуск клиента ClientAckConsumer

1. В окне терминала перейдите в следующий каталог:

   ```
   tut-install/examples/jms/simple/producer/
   ```

2. Запустите клиент `Producer`, отправив несколько сообщений в очередь:

   ```
   appclient -client target/producer.jar queue 3
   ```
   SHELL

3. В другом окне терминала перейдите в следующий каталог:

   ```
   tut-install/examples/jms/simple/clientackconsumer/
   ```

4. Чтобы запустить клиент, используйте следующую команду:

SHELL

```
appclient -client target/clientackconsumer.jar
```

Вывод клиента выглядят следующим образом (наряду с некоторыми дополнительными выходными данными):

SHELL

```
Created client-acknowledge JMSContext
Reading message: This is message 1 from producer
Acknowledging TextMessage
Reading message: This is message 2 from producer
Acknowledging TextMessage
Reading message: This is message 3 from producer
Acknowledging TextMessage
Acknowledging non-text control message
```

Клиент подтверждает каждое сообщение явным образом после его обработки, так же как `JMSContext`, настроенный на использование `AUTO_ACKNOWLEDGE`, выполняет это автоматически после успешного завершения `MessageListener` после обработки асинхронно полученного сообщения.

Более сложные приложения JMS

В следующих примерах показано, как использовать некоторые из дополнительных функций Jakarta Messaging: долговременные подписки и транзакции.

Использование долговременных подписок

Пример `durablesubscriptionexample` показывает, как работают долговременные подписки, не являющиеся общими. Это демонстрирует, что долговременная подписка продолжает существовать и накапливать сообщения, даже если у неё нет активного потребителя.

Пример состоит из двух модулей: приложения `durableconsumer`, создающего долговременную подписку и принимающего сообщения, и приложения `unsubscriber`, позволяющего отказаться от подписки на долговременную подписку после завершения работы приложения `durableconsumer`.

Для получения информации о долговременных подписках см. Создание долговременных подписок.

Основной клиент `DurableConsumer.java` находится в каталоге *`tut-install`*`/examples/jms/durablesubscriptionexample/durableconsumer`.

В этом примере используется фабрика соединений `jms/DurableConnectionFactory`, которая имеет идентификатор клиента.

Клиент `DurableConsumer` создаёт `JMSContext`, используя фабрику соединений. Затем он останавливает `JMSContext`, вызывает `createDurableConsumer`, чтобы создать долговременную подписку и получателя по теме, указав имя подписки, регистрирует слушатель сообщений и запускает `JMSContext` ещё раз. Подписка создаётся только в том случае, если она ещё не существует, поэтому пример можно запустить несколько раз:

JAVA

```
try (JMSContext context = durableConnectionFactory.createContext();) {
    context.stop();
    consumer = context.createDurableConsumer(topic, "MakeItLast");
    listener = new TextListener();
    consumer.setMessageListener(listener);
    context.start();
    ...
}
```

Для отправки сообщений в тему запустите клиент `producer`.

Пример `unsubscriber` содержит очень простой клиент `Unsubscriber`, который создаёт `JMSContext` с той же фабрикой соединений и затем вызывает метод `unsubscribe`, определяющий имя подписки:

```java
try (JMSContext context = durableConnectionFactory.createContext();) {
    System.out.println("Unsubscribing from durable subscription");
    context.unsubscribe("MakeItLast");
    ...
}
```

Создание ресурсов для примера долговременной подписки

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В командном окне перейдите к примеру `durableconsumer`.

   ```shell
   cd tut-install/jms/durablesubscriptionexample/durableconsumer
   ```

3. Создайте ресурсы командой `asadmin add-resources`:

   ```shell
   asadmin add-resources src/main/setup/glassfish-resources.xml
   ```

   Вывод команды сообщает о создании пула соединений коннекторов и ресурса коннектора.
4. Проверьте создание ресурсов:

   ```shell
   asadmin list-jms-resources
   ```

   В дополнение к ресурсам, которые были созданы для простых примеров, команда выводит новую фабрику соединений:

   ```shell
   jms/MyQueue
   jms/MyTopic
   jms/__defaultConnectionFactory
   jms/DurableConnectionFactory
   Command list-jms-resources executed successfully.
   ```

Запуск примера долговременной подписки

1. В окне терминала перейдите в следующий каталог:

   ```
   tut-install/examples/jms/durablesubscriptionexample/
   ```

2. Создайте примеры `durableconsumer` и `unsubscriber`:

   ```shell
   mvn install
   ```

3. Перейдите в каталог `durableconsumer`:

   ```shell
   cd durableconsumer
   ```

4. Чтобы запустить клиент, введите следующую команду:

   ```shell
   appclient -client target/durableconsumer.jar
   ```

Клиент создаёт долговременного потребителя и затем ждёт сообщений:

```
Creating consumer for topic
Starting consumer
To end program, enter Q or q, then <return>
```
SHELL

5. В другом окне терминала запустите клиент `Producer`, отправив несколько сообщений в тему:

```
cd tut-install/examples/jms/simple/producer
appclient -client target/producer.jar topic 3
```
SHELL

6. После того как клиент `DurableConsumer` получит сообщения, введите `q` или `Q` для выхода из программы. На этом этапе клиент вёл себя как любой другой асинхронный потребитель.
7. Теперь, пока клиент `DurableConsumer` не запущен, используйте клиент `Producer` для отправки дополнительных сообщений:

```
appclient -client target/producer.jar topic 2
```
SHELL

Если бы долговременная подписка не существовала, эти сообщения были бы потеряны, потому что ни один потребитель в теме в настоящее время не работает. Однако долговременная подписка всё ещё активна и сохраняет сообщения.

8. Запустите клиент `DurableConsumer` ещё раз. Он сразу получает сообщения, которые были отправлены, когда он был неактивен:

```
Creating consumer for topic
Starting consumer
To end program, enter Q or q, then <return>
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Message is not a TextMessage
```
SHELL

9. Введите `q` или `Q`, чтобы выйти из программы.

Запуск unsubscriber

После завершения работы клиента `DurableConsumer` запустите пример `unsubscriber`, чтобы отписаться от долговременной подписки.

1. В окне терминала перейдите в следующий каталог:

```
tut-install/examples/jms/durablesubscriptionexample/unsubscriber
```

2. Чтобы запустить клиент `Unsubscriber`, введите следующую команду:

```
appclient -client target/unsubscriber.jar
```
SHELL

Клиент сообщает, что отписался от долговременной подписки.

Использование локальных транзакций

Пример `transactedexample` демонстрирует использование локальных транзакций в клиентском приложении обмена сообщениями. Он также демонстрирует использование шаблона обмена сообщениями запрос/ответ, описанного в Создание вре́менных пунктов назначения, хотя он использует постоянные, а не вре́менные пункты назначения. Пример состоит из трёх модулей: `genericsupplier`, `retailer` и `vendor`,

которые можно найти в каталоге *tut-install* `/examples/jms/transactedexample/` . Исходный код можно найти в каталоге `src/main/java/ee.jakarta.tutorial` каждого модуля. Каждый из модулей `genericsupplier` и `retailer` содержит по одному классу, `genericsupplier/GenericSupplier.java` и `retailer/Retailer.java` , соответственно. Модуль `vendor` сложнее и содержит четыре класса: `vendor/Vendor.java` , `vendor/VendorMessageListener.java` , `vendor/Order.java` и `vendor/SampleUtilities.java` .

В этом примере показано, как использовать очередь и тему в одной транзакции, а также как передать `JMSContext` в конструктор слушателя сообщений. Пример представляет собой чрезвычайно упрощённое приложение электронной коммерции, в котором выполняются следующие действия.

1. Розничный продавец (`retailer/src/main/java/ee/jakarta/tutorial/retailer/Retailer.java`) отправляет `MapMessage` в очередь заказов поставщика, заказывая некоторое количество компьютеров и ожидая ответа поставщика:

   ```
   outMessage = context.createMapMessage();
   outMessage.setString("Item", "Computer(s)");
   outMessage.setInt("Quantity", quantity);
   outMessage.setJMSReplyTo(retailerConfirmQueue);
   context.createProducer().send(vendorOrderQueue, outMessage);
   System.out.println("Retailer: ordered " + quantity + " computer(s)");
   orderConfirmReceiver = context.createConsumer(retailerConfirmQueue);
   ```
   JAVA

2. Поставщик (`vendor/src/main/java/ee/jakarta/tutorial/retailer/Vendor.java`) получает сообщение о заказе розничного продавца и отправляет сообщение о заказе в тему заказа оптового поставщика в одной транзакции. Эта JMS-транзакция использует одну сессию, можно объединить получение из очереди с отправкой в тему. Вот код, который использует ту же сессию для создания потребителя для очереди:

   ```
   vendorOrderReceiver = session.createConsumer(vendorOrderQueue);
   ```
   JAVA

   Следующий код получает входящее сообщение, отправляет исходящее сообщение и фиксирует `JMSContext` . Обработка сообщений была удалена, чтобы упростить последовательность:

   ```
   inMessage = vendorOrderReceiver.receive();
   // Обработка входящего сообщения и форматирование
   // исходящего
   ...
   context.createProducer().send(supplierOrderTopic, orderMessage);
   ...
   context.commit();
   ```
   JAVA

   Для простоты есть только два производителя, один производит процессоры, другой — жёсткие диски.

3. Каждый оптовый поставщик (`genericsupplier/src/main/java/ee/jakarta/tutorial/retailer/GenericSupplier.java`) получает заказ из темы заказов, проверяет свой склад, а затем отправляет заказанные товары в очередь, указанную в поле `JMSReplyTo` . Если товара недостаточно на складе, производитель отправляет имеющееся количество. Синхронное получение из темы и отправка в очередь происходят в одной транзакции JMS:

```java
receiver = context.createConsumer(SupplierOrderTopic);
...
inMessage = receiver.receive();
if (inMessage instanceof MapMessage) {
    orderMessage = (MapMessage) inMessage;
} ...
// Обработка сообщения
outMessage = context.createMapMessage();
// Добавление содержимого в сообщение
context.createProducer().send(
         (Queue) orderMessage.getJMSReplyTo(),
         outMessage);
// Отображение содержимого сообщения
context.commit();
```

4. Оптовый продавец получает ответы производителя из своей очереди подтверждения и обновляет состояние заказа. Сообщения обрабатываются асинхронным слушателем сообщений `VendorMessageListener`. Этот шаг показывает использование транзакций Jakarta Messaging слушателем сообщения:

```java
MapMessage component = (MapMessage) message;
...
int orderNumber = component.getInt("VendorOrderNumber");
Order order = Order.getOrder(orderNumber).processSubOrder(component);
context.commit();
```

5. Когда все ожидающие ответы обрабатываются для данного заказа, слушатель сообщений оптового продавца отправляет сообщение, уведомляющее розничного продавца, может ли он выполнить заказ:

```java
Queue replyQueue = (Queue) order.order.getJMSReplyTo();
MapMessage retailerConfirmMessage = context.createMapMessage();
// Форматирование сообщения
context.createProducer().send(replyQueue, retailerConfirmMessage);
context.commit();
```

6. Розничный продавец получает сообщение от оптового продавца:

```java
inMessage = (MapMessage) orderConfirmReceiver.receive();
```

   Затем розничный продавец размещает второй заказ на вдвое большее количество компьютеров, чем в первом случае, поэтому эти шаги выполняются дважды.

Рисунок 49-1 иллюстрирует эти шаги.

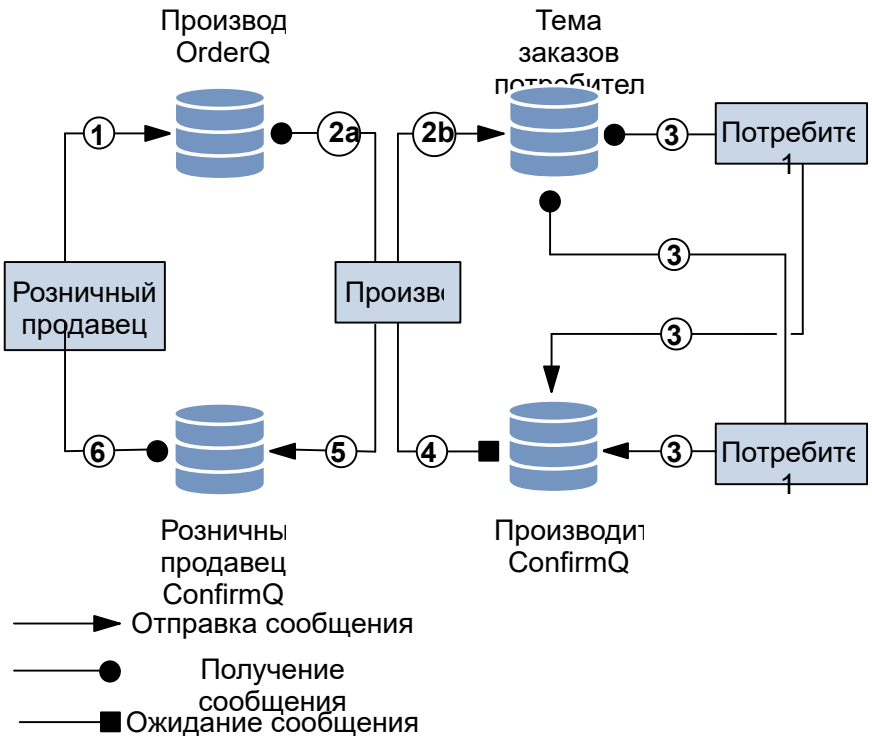

Рис. 49-1 Транзакции: пример клиента обмена сообщениями

Все сообщения имеют тип `MapMessage`. Синхронное получение используется для всех сообщений, кроме тех случаев, когда оптовый продавец обрабатывает ответы производителей. Эти ответы обрабатываются асинхронно и демонстрируют, как использовать транзакции в слушателе сообщений.

Через случайные интервалы клиент `Vendor` генерирует исключение, чтобы симулировать проблему с базой данных и вызвать откат.

Все клиенты, кроме `Retailer`, используют транзакционные контексты.

В этом примере используются три очереди `jms/AQueue`, `jms/BQueue` и `jms/CQueue` и одна тема с именем `jms/OTopic`.

Создание ресурсов для transactedexample

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В командном окне перейдите к примеру `genericsupplier`:

```
cd tut-install/jms/transactedexample/genericsupplier
```
SHELL

3. Создайте ресурсы командой `asadmin add-resources`:

```
asadmin add-resources src/main/setup/glassfish-resources.xml
```
SHELL

4. Проверьте создание ресурсов:

```
asadmin list-jms-resources
```
SHELL

В дополнение к ресурсам, которые вы создали для простых примеров и примера долговременной подписки, команда перечисляет четыре новых пункта назначения:

```
jms/MyQueue
jms/MyTopic
jms/AQueue
jms/BQueue
jms/CQueue
jms/OTopic
jms/__defaultConnectionFactory
jms/DurableConnectionFactory
Command list-jms-resources executed successfully.
```
SHELL

Запуск клиентов для transactedexample

Вам потребуется четыре окна терминала для запуска клиентов. Убедитесь, что вы запускаете клиентов в правильном порядке.

1. В окне терминала перейдите в следующий каталог:

   ```
   tut-install/examples/jms/transactedexample/
   ```

2. Чтобы собрать и упаковать все модули, введите следующую команду:

   ```
   mvn install
   ```
   SHELL

3. Перейдите в каталог `genericsupplier`:

   ```
   cd genericsupplier
   ```
   SHELL

4. Используйте следующую команду для запуска клиента производителя процессоров:

   ```
   appclient -client target/genericsupplier.jar CPU
   ```
   SHELL

   После некоторого начального вывода клиент сообщает следующее:

   ```
   Starting CPU supplier
   ```
   SHELL

5. Во втором окне терминала перейдите в каталог `genericsupplier`:

   ```
   cd tut-install/examples/jms/transactedexample/genericsupplier
   ```
   SHELL

6. Используйте следующую команду для запуска клиента производителя жёстких дисков:

   ```
   appclient -client target/genericsupplier.jar HD
   ```
   SHELL

   После некоторого начального вывода клиент сообщает следующее:

   ```
   Starting Hard Drive supplier
   ```
   SHELL

7. В третьем окне терминала перейдите в каталог `vendor`:

   ```
   cd tut-install/examples/jms/transactedexample/vendor
   ```
   SHELL

8. Используйте следующую команду для запуска клиента `Vendor`:

SHELL

```
appclient -client target/vendor.jar
```

После некоторого начального вывода клиент сообщает следующее:

SHELL

```
Starting vendor
```

9. В другом окне терминала перейдите в каталог `retailer` :

SHELL

```
cd tut-install/examples/jms/transactedexample/retailer
```

10. Используйте команду, подобную следующей, чтобы запустить клиент `Retailer` . Аргумент указывает количество компьютеров для заказа:

SHELL

```
appclient -client target/retailer.jar 4
```

После некоторого начального вывода клиент `Retailer` сообщает что-то вроде следующего. В этом случае первый заказ заполнен, а второй нет:

SHELL

```
Retailer: Quantity to be ordered is 4
Retailer: Ordered 4 computer(s)
Retailer: Order filled
Retailer: Placing another order
Retailer: Ordered 8 computer(s)
Retailer: Order not filled
```

Клиент `Vendor` сообщает что-то вроде следующего, заявляя в этом случае, что он может отправлять все компьютеры по первому заказу, но не по второму:

SHELL

```
Vendor: Retailer ordered 4 Computer(s)
Vendor: Ordered 4 CPU(s) and hard drive(s)
  Vendor: Committed transaction 1
Vendor: Completed processing for order 1
Vendor: Sent 4 computer(s)
  Vendor: committed transaction 2
Vendor: Retailer ordered 8 Computer(s)
Vendor: Ordered 8 CPU(s) and hard drive(s)
  Vendor: Committed transaction 1
Vendor: Completed processing for order 2
Vendor: Unable to send 8 computer(s)
  Vendor: Committed transaction 2
```

Производитель процессоров сообщает что-то вроде следующего. В этом случае он может отправить все процессоры для обоих заказов:

SHELL

```
CPU Supplier: Vendor ordered 4 CPU(s)
CPU Supplier: Sent 4 CPU(s)
  CPU Supplier: Committed transaction
CPU Supplier: Vendor ordered 8 CPU(s)
CPU Supplier: Sent 8 CPU(s)
  CPU Supplier: Committed transaction
```

Производитель жёстких дисков сообщает что-то вроде следующего. В этом случае ему не хватает жёстких дисков для второго заказа:

```
Hard Drive Supplier: Vendor ordered 4 Hard Drive(s)
Hard Drive Supplier: Sent 4 Hard Drive(s)
  Hard Drive Supplier: Committed transaction
Hard Drive Supplier: Vendor ordered 8 Hard Drive(s)
Hard Drive Supplier: Sent 1 Hard Drive(s)
  Hard Drive Supplier: Committed transaction
```
SHELL

11. Повторите Шаг 10 сколько угодно раз. Иногда оптовый продавец сообщает об исключительной ситуации, которая вызывает откат:

```
Vendor: JMSException occurred: jakarta.jms.JMSException: Simulated
database concurrent access exception
  Vendor: Rolled back transaction 1
```
SHELL

12. После завершения работы с клиентами вы можете удалить ресурсы пунктов назначения с помощью следующих команд:

```
asadmin delete-jms-resource jms/AQueue
asadmin delete-jms-resource jms/BQueue
asadmin delete-jms-resource jms/CQueue
asadmin delete-jms-resource jms/OTopic
```
SHELL

Высокопроизводительные и масштабируемые приложений JMS

В этом разделе описывается использование Jakarta Messaging для написания приложений, надёжно обрабатывающих большие объемы сообщений. В этих примерах используются как долговременные, так и не являющиеся таковыми, общие потребители.

Использование общих подписок, не являющихся долговременными

В этом разделе описываются принимающие клиенты на примере, который показывает, как использовать общего потребителя для рассылки сообщений, отправленных в тему, используемую несколькими потребителями. Затем в этом разделе объясняется, как скомпилировать и запустить клиенты с помощью GlassFish Server.

Возможно, вы захотите сравнить этот пример с результатами Запуска нескольких потребителей для одного пункта назначения с использованием общего пользователя. В этом примере сообщения распределяются среди потребителей в очереди, но каждый потребитель в теме получает все сообщения, потому что каждый потребитель в теме использует отдельную подписку на тему.

В этом примере, однако, сообщения распределяются между несколькими потребителями темы, потому что все потребители используют одну и ту же подписку. Каждое сообщение, добавленное в подписку на тему, принимается только одним потребителем, аналогично тому, как каждое сообщение, добавленное в очередь, принимается только одним потребителем.

Тема может иметь несколько подписок. Каждое сообщение, отправленное в тему, будет добавлено в каждую подписку. Однако, если в определённой подписке есть несколько потребителей, каждое сообщение, добавленное этой подписке, будет доставлено только одному из этих потребителей.

Написание клиентов для примера общего потребителя

Отправляющий клиент — `Producer.java` — тот же клиент, который использовался в предыдущих примерах.

Получающим клиентом является `SharedConsumer.java`. Он очень похож на `AsynchConsumer.java`, за тем исключением, что он всегда использует тему. Он выполняет следующие шаги.

1. Инъецирует ресурсы для фабрики соединений и темы.
2. В блоке `try-with-resources` создаётся `JMSContext`.
3. Создаёт получателя для общей подписки, не являющейся долговременной, с указанием имени подписки:

```
consumer = context.createSharedConsumer(topic, "SubName");
```
JAVA

4. Создаёт объект класса `TextListener` и регистрирует его в качестве слушателя сообщений для общего потребителя.
5. Слушает сообщения, опубликованные в пункте назначения, и завершает работу, когда пользователь вводит символ `q` или `Q`.
6. Ловит и обрабатывает все возможные исключения. В конце `try-with-resources` автоматически закрывается `JMSContext`.

Класс `TextListener.java` идентичен классу примера `asynchconsumer`.

В этом примере будет использоваться фабрика соединений по умолчанию и тема, которая была создана в Создание ресурсов для простых примеров.

Запуск клиентов SharedConsumer и Producer

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. Откройте три командных окна. Во-первых, перейдите в каталог `simple/seller/`:

```
cd tut-install/examples/jms/simple/producer/
```
SHELL

3. Во втором и третьем окнах команд перейдите в каталог `shared/sharedconsumer/`:

```
cd tut-install/examples/jms/shared/sharedconsumer/
```
SHELL

4. В одном из окон `sharedconsumer` соберите пример:

```
mvn install
```
SHELL

5. В каждом из двух окон `sharedconsumer` запустите клиент. Не обязательно указывать аргумент `topic`:

```
appclient -client target/sharedconsumer.jar
```
SHELL

   Подождите, пока вы не увидите следующий вывод в обоих окнах:

```
Waiting for messages on topic
To end program, enter Q or q, then <return>
```
SHELL

6. В окне `producer` запустите клиент, указав тему и количество сообщений:

```
appclient -client target/producer.jar topic 20
```
SHELL

   Каждый клиент потребителя получает некоторые из сообщений. Только один из клиентов получает нетекстовое сообщение, которое сигнализирует об окончании потока сообщений.
7. Введите `Q` или `q` и нажмите Return, чтобы остановить каждого клиента и просмотреть отчёт о количестве полученных текстовых сообщений.

Использование общих долговременных подписок

Клиент `shareddurableconsumer` показывает, как использовать общие долговременные подписки. В нём показано, как общие долговременные подписки сочетают в себе преимущества долговременных подписок (подписка остаётся активной, если клиент не активен) с подписками общих потребителей (нагрузка по обработке сообщений может быть разделена между несколькими клиентами).

Этот пример намного больше похож на пример `sharedconsumer`, чем на клиент `DurableConsumer.java`. Он использует два класса, `SharedDurableConsumer.java` и `TextListener.java`, которые можно найти в каталоге *`tut-install`*`/examples/jms/shareddurableconsumer/`.

Клиент использует `java:comp/DefaultJMSConnectionFactory`, фабрику соединений, у которой нет идентификатора клиента, как рекомендуется для общих долговременных подписок. Он использует метод `createSharedDurableConsumer` с именем подписки, чтобы установить подписку:

```
consumer = context.createSharedDurableConsumer(topic, "MakeItLast");
```
JAVA

Вы запускаете пример в сочетании с клиентом `Producer.java`.

Запуск клиентов SharedDurableConsumer и Producer

1. В окне терминала перейдите в следующий каталог:

   ```
   tut-install/examples/jms/shared/shareddurableconsumer
   ```

2. Чтобы скомпилировать и упаковать клиент, введите следующую команду:

   ```
   mvn install
   ```
   SHELL

3. Сначала запустите клиент, чтобы установить долговременную подписку:

   ```
   appclient -client target/shareddurableconsumer.jar
   ```
   SHELL

4. Клиент отображает следующее и делает паузу:

   ```
   Waiting for messages on topic
   To end program, enter Q or q, then <return>
   ```
   SHELL

5. В окне `shareddurableconsumer` введите `q` или `Q` для выхода из программы. Подписка остаётся активной, хотя клиент не работает.
6. Откройте другое окно терминала и перейдите в каталог примера `provider`:

   ```
   cd tut-install/examples/jms/simple/producer
   ```
   SHELL

7. Запустите пример `producer`, отправив несколько сообщений в тему:

   ```
   appclient -client target/producer.jar topic 6
   ```
   SHELL

8. После того, как производитель отправил сообщения, откройте третье окно терминала и перейдите в каталог `shareddurableconsumer`.
9. Запустите клиент в первом и третьем окнах терминала. Клиент, запущенный первым, получит все сообщения, которые были отправлены, когда не было активного подписчика:

```
appclient -client target/shareddurableconsumer.jar
```
SHELL

10. Пока оба клиента `shareddurableconsumer` работают, перейдите в окно `provider` и отправьте большее количество сообщений в тему:

```
appclient -client target/producer.jar topic 25
```
SHELL

Теперь сообщения будут переданы обоим клиентам. Если продолжить отправлять группы сообщений в тему, каждый клиент получит некоторые из сообщений. Если выйти из одного из клиентов и отправить больше сообщений, другой клиент получит все сообщения.

Отправка и получение сообщений в простом веб-приложении

Веб-приложения могут использовать Jakarta Messaging для отправки и получения сообщений, как указано в разделе Использование компонентов Jakarta EE для создания и синхронного приёма сообщений. В этом разделе описаны компоненты простейшего веб-приложения, в котором используется Jakarta Messaging.

В этом разделе предполагается, что вы знакомы с основами Jakarta Faces, описанной в части III, «Веб-слой».

Пример `websimplemessage` находится в каталоге *`tut-install`*`/jms/examples/`. Для отправки и получения он использует страницы Facelets, а также соответствующие вспомогательные бины. Когда пользователь вводит сообщение в текстовое поле отправляющей страницы и кликает кнопку, вспомогательный компонент для страницы отправляет сообщение в очередь и отображает его на странице. Когда пользователь переходит на принимающую страницу и кликает другую кнопку, вспомогательный компонент для этой страницы получает сообщение синхронно и отображает его.

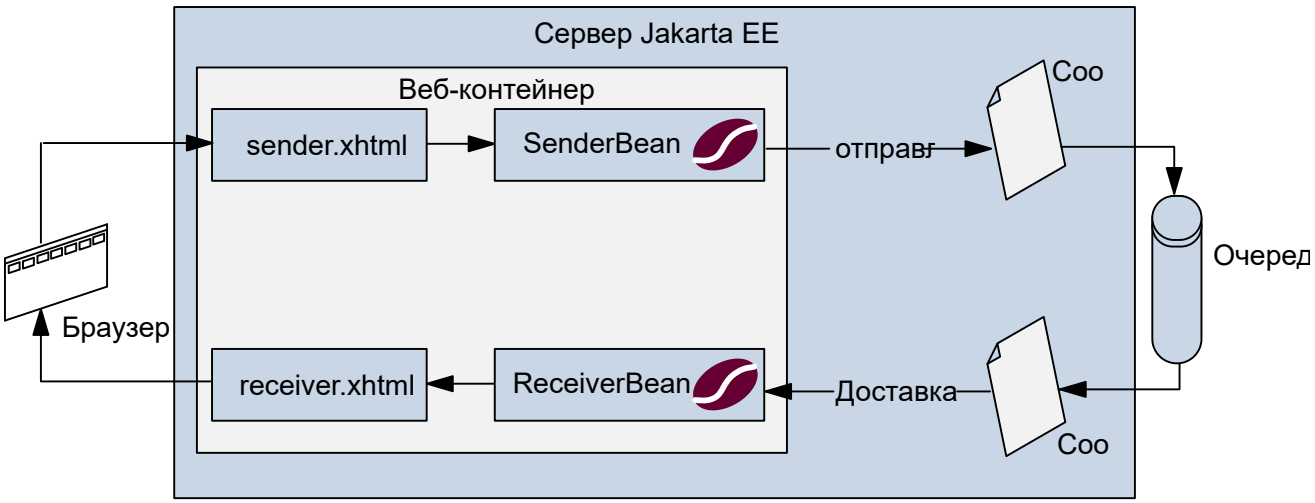

Рис. 49-2. Приложение простого веб-приложения

Страницы Facelets websimplemessage

Страницы Facelets для примера следующие.

- `sender.xhtml`, который предоставляет помеченный тег `h:inputText`, в который пользователь вводит сообщение, вместе с двумя командными кнопками. Когда пользователь кликает кнопку «Отправить сообщение», вызывается метод `senderBean.sendMessage`, чтобы отправить сообщение в очередь и отобразить его содержимое. Когда пользователь кликает кнопку «Перейти к странице получения», появляется страница `receiver.xhtml`.

- `receive.xhtml`, который также предоставляет две кнопки управления. Когда пользователь кликает кнопку «Получить сообщение», вызывается метод `receiveBean.getMessage`, чтобы извлечь сообщение из очереди и отобразить его содержимое. Когда пользователь кликает кнопку «Отправить другое сообщение», страница sender.xhtml появляется снова.

Managed-бины websimplemessage

Два Managed-бина для примера следующие.

- `SenderBean.java`, Managed-бин CDI со свойством `messageText` и бизнес-методом `sendMessage`. Класс аннотируется `@JMSDestinationDefinition` для создания очереди для компонента:

```java
@JMSDestinationDefinition(
        name = "java:comp/jms/webappQueue",
        interfaceName = "jakarta.jms.Queue",
        destinationName = "PhysicalWebappQueue")
@Named
@RequestScoped
public class SenderBean { ... }
```

  Метод `sendMessage` инъецирует `JMSContext` (используя фабрику соединений по умолчанию) и очередь, создаёт производителя, отправляет сообщение, введённое пользователем на странице Facelets и создаёт `FacesMessage` для отображения на странице Facelets:

```java
@Inject
private JMSContext context;
@Resource(lookup = "java:comp/jms/webappQueue")
private Queue queue;
private String messageText;
...
public void sendMessage() {
    try {
        String text = "Message from producer: " + messageText;
        context.createProducer().send(queue, text);

        FacesMessage facesMessage =
                new FacesMessage("Sent message: " + text);
        FacesContext.getCurrentInstance().addMessage(null, facesMessage);
    } catch (Throwable t) {
        logger.log(Level.SEVERE,
                "SenderBean.sendMessage: Exception: {0}",
                t.toString());
    }
}
```

- `ReceiverBean.java`, Managed-бин CDI с бизнес-методом `getMessage`. Метод инъецирует `JMSContext` (используя фабрику соединений по умолчанию) и очередь, которая была определена в `SenderBean`, создаёт получателя, получает сообщение и создаёт `FacesMessage` для отображения на странице Facelets:

JAVA

```
@Inject
private JMSContext context;
@Resource(lookup = "java:comp/jms/webappQueue")
private Queue queue;
...
public void getMessage() {
    try {
        JMSConsumer receiver = context.createConsumer(queue);
        String text = receiver.receiveBody(String.class);

        if (text != null) {
            FacesMessage facesMessage =
                    new FacesMessage("Reading message: " + text);
            FacesContext.getCurrentInstance().addMessage(null, facesMessage);
        } else {
            FacesMessage facesMessage =
                    new FacesMessage("No message received after 1 second");
            FacesContext.getCurrentInstance().addMessage(null, facesMessage);
        }
    } catch (Throwable t) {
        logger.log(Level.SEVERE,
                "ReceiverBean.getMessage: Exception: {0}",
                t.toString());
    }
}
```

Запуск websimplemessage

Вы можете использовать IDE NetBeans или Maven для создания, упаковки, развёртывания и запуска приложения `websimplemessage`.

Создание ресурсов для websimplemessage

В этом примере используется очередь, заданная аннотацией, и предустановленная фабрика соединений по умолчанию `java:comp/DefaultJMSConnectionFactory`.

Упаковка и развёртывание веб-сайта с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/jms
   ```

4. Выберите каталог `websimplemessage`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `websimplemessage` и выберите **Сборка**.

   Эта команда собирает и развёртывает проект.

Упаковка и развёртывание веб-сайта с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/jms/websimplemessage/
   ```

3. Чтобы скомпилировать исходные файлы, упаковать и развернуть приложение, используйте следующую команду:

```
mvn install
```
SHELL

Запуск websimplemessage

1. В веб-браузере введите следующий URL:

```
http://localhost:8080/websimplemessage
```

2. Введите сообщение в текстовое поле и нажмите «Отправить сообщение».

   Например, если вы введёте «Hello, Duke», под кнопками появится следующее:

```
Sent message: Message from producer: Hello, Duke
```

3. Нажмите Перейти на страницу получения.
4. Нажмите Получить сообщение.

   Следующее появляется под кнопками:

```
Reading message: Message from producer: Hello, Duke
```

5. Нажмите «Отправить другое сообщение», чтобы вернуться на страницу отправки.
6. После завершения работы приложения удалите его, используя вкладку «Службы» в среде IDE NetBeans или команду `mvn cargo:undeploy` .

Асинхронный приём сообщений с использованием бина, управляемого сообщениями

Если пишется приложение для запуска в клиентском контейнере приложения Jakarta EE или на платформе Java SE и требуется асинхронное получение сообщений, то нужно определить класс, реализующий интерфейс `MessageListener` , создать `JMSConsumer` и вызвать метод `SetMessageListener` .

Если пишется приложение для запуска в веб- или EJB-контейнере Jakarta EE и нужно получать сообщения асинхронно, также необходимо определить класс, реализующий интерфейс `MessageListener` . Однако вместо того, чтобы создавать `JMSConsumer` и вызывать метод `setMessageListener` , нужно сконфигурировать свой класс приёмника сообщений для бина, управляемого сообщениями. Об остальном позаботится сервер приложений.

Управляемые сообщениями бины могут реализовывать любую технологию обмена сообщениями. Однако чаще всего они реализуют технологию Jakarta Messaging.

В этом разделе описывается простой пример компонента, управляемого сообщениями. Прежде чем продолжить, вы должны ознакомиться с общими понятиями в разделах Что такое бин, управляемый сообщениями? и Использование управляемых сообщениями бинов для асинхронного получения сообщений.

Обзор примера simplemessage

Приложение `simplemessage` состоит из следующих компонентов:

- `SimpleMessageClient` : клиентское приложение, которое отправляет несколько сообщений в очередь

- `SimpleMessageBean` : управляемый сообщениями компонент, который асинхронно обрабатывает сообщения, отправленные в очередь

Рисунок 49-3 иллюстрирует структуру этого приложения. Клиентское приложение отправляет сообщения в очередь, которая была создана административно в Консоли администрирования. Провайдер сообщений (в данном случае GlassFish Server) доставляет сообщения объектам компонента, управляемого сообщениями, который затем обрабатывает сообщения.

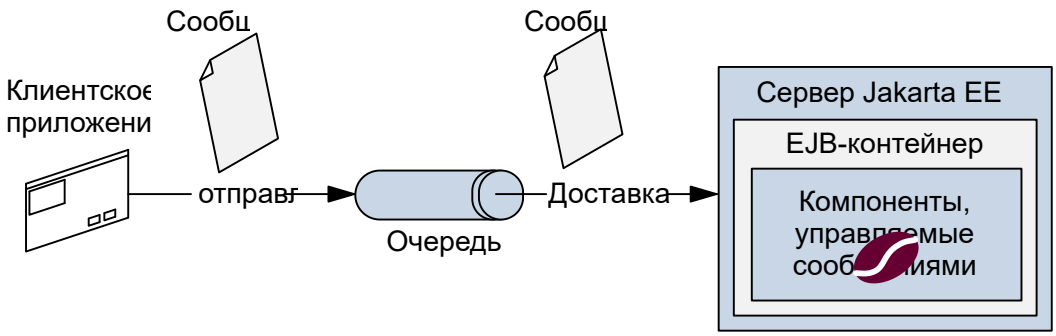

Рис. 49-3. Приложение simplemessage

Исходный код этого приложения находится в каталоге *tut-install*`/examples/jms/simplemessage/` .

Клиент simplemessage

`SimpleMessageClient` отправляет сообщения в очередь, которую слушает `SimpleMessageBean` . В первую очередь клиент инъецирует фабрику соединений и ресурс очереди:

JAVA

```
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
private static ConnectionFactory connectionFactory;

@Resource(lookup = "jms/MyQueue")
private static Queue queue;
```

Затем клиент создаёт `JMSContext` в блоке `try-with-resources` :

JAVA

```
String text;
final int NUM_MSGS = 3;

try (JMSContext context = connectionFactory.createContext();) { ... }
```

Наконец, клиент отправляет несколько текстовых сообщений в очередь:

JAVA

```
for (int i = 0; i < NUM_MSGS; i++) {
    text = "This is message " + (i + 1);
    System.out.println("Sending message: " + text);
    context.createProducer().send(queue, text);
}
```

Класс бина, управляемого сообщениями в simplemessage

Код для класса `SimpleMessageBean` иллюстрирует требования к классам управляемых сообщениями компонентов, описанных в Использование управляемых сообщениями бинов для асинхронного получения сообщений.

Первые несколько строк класса `SimpleMessageBean` используют атрибут `activationConfig` аннотации `@MessageDriven` для указания свойств конфигурации:

JAVA

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup",
            propertyValue = "jms/MyQueue"),
    @ActivationConfigProperty(propertyName = "destinationType",
            propertyValue = "jakarta.jms.Queue")
})
```

Смотрите таблицу 48-3 для списка доступных свойств.

См. Отправка сообщений из сессионного компонента в MDB для примера свойств `subscriptionDurability`, `clientId`, `subscriptionName` и `messageSelector`.

Метод onMessage

Когда очередь получает сообщение, EJB-контейнер вызывает метод или методы обработчика сообщений. Для компонента, использующего Jakarta Messaging, это метод `onMessage` интерфейса `MessageListener`.

В классе `SimpleMessageBean` метод `onMessage` преобразует входящее сообщение в `TextMessage` и отображает текст:

JAVA

```
public void onMessage(Message inMessage) {

    try {
        if (inMessage instanceof TextMessage) {
            logger.log(Level.INFO,
                    "MESSAGE BEAN: Message received: {0}",
                    inMessage.getBody(String.class));
        } else {
            logger.log(Level.WARNING,
                    "Message of wrong type: {0}",
                    inMessage.getClass().getName());
        }
    } catch (JMSException e) {
        logger.log(Level.SEVERE,
                "SimpleMessageBean.onMessage: JMSException: {0}",
                e.toString());
        mdc.setRollbackOnly();
    }
}
```

Запуск simplemessage

Вы можете использовать IDE NetBeans или Maven для сборки, развёртывания и запуска примера `simplemessage`.

Создание ресурсов для simplemessage

В этом примере используется очередь с именем `jms/MyQueue` и предустановленная фабрика соединений по умолчанию `java:comp/DefaultJMSConnectionFactory`.

Если вы запускали простые примеры Jakarta Messaging в Простых приложениях JMS и не удалили ресурсы, у вас уже есть очередь. В противном случае следуйте инструкциям в Создание ресурсов для простых примеров для её создания.

Дополнительные сведения о создании ресурсов обмена сообщениями см. в разделе Создание администрируемых объектов Jakarta Messaging.

Запуск simplemessage в IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).

2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне **Открыть проект** перейдите к:

```
tut-install/examples/jms/simplemessage
```

4. Выберите каталог `simplemessage` .
5. Убедитесь, что выбран чекбокс **Открыть требуемые проекты**, а затем кликните **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `simplemessage` и выберите **Сборка**. (В среде IDE NetBeans требуется установить флажок для запуска предварительной сборки.)

   Эта команда упаковывает клиентское приложение и управляемый сообщениями компонент, затем создаёт файл `simplemessage.ear` в каталоге `simplemessage-ear/target/` . Затем она развёртывает модуль `simplemessage-ear` , извлекает клиентские заглушки и запускает клиентское приложение.

   Вывод в окне вывода выглядит следующим образом (ему предшествует вывод контейнера клиентского приложения):

```
Sending message: This is message 1
Sending message: This is message 2
Sending message: This is message 3
To see if the bean received the messages,
 check <install_dir>/domains/domain1/logs/server.log.
```

   В файле журнала сервера появляются строки, подобные следующим:

```
MESSAGE BEAN: Message received: This is message 1
MESSAGE BEAN: Message received: This is message 2
MESSAGE BEAN: Message received: This is message 3
```

   Полученные сообщения могут появляться не в том порядке, в котором они были отправлены.
7. После завершения работы приложения удалите его с помощью вкладки **Сервисы**.

Запуск simplemessage с помощью Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

```
tut-install/examples/jms/simplemessage/
```

3. Чтобы скомпилировать исходные файлы и упаковать приложение, используйте следующую команду:

```
mvn install
```
SHELL

   Эта команда упаковывает клиентское приложение и управляемый сообщениями компонент, а затем создаёт файл `simplemessage.ear` в каталоге `simplemessage-ear/target/` . Затем она развёртывает модуль `simplemessage-ear` , извлекает клиентские заглушки и запускает клиентское приложение.

   Вывод в окне терминала выглядит следующим образом (ему предшествует вывод контейнера клиентского приложения):

```
Sending message: This is message 1
Sending message: This is message 2
Sending message: This is message 3
To see if the bean received the messages,
 check <install_dir>/domains/domain1/logs/server.log.
```

В файле журнала сервера появляются строки, подобные следующим:

```
MESSAGE BEAN: Message received: This is message 1
MESSAGE BEAN: Message received: This is message 2
MESSAGE BEAN: Message received: This is message 3
```

Полученные сообщения могут появляться не в том порядке, в котором они были отправлены.

4. После того, как вы закончили запуск приложения, удалите его командой `mvn cargo:undeploy`.

Отправка сообщений из сессионного компонента в MDB

В этом разделе объясняется, как писать, компилировать, упаковать, развёртывать и запускать приложение, использующее Jakarta Messaging в сочетании с сессионным компонентом. Приложение содержит следующие компоненты:

- Клиентское приложение, которое вызывает сессионный компонент
- Сессионный компонент, который публикует несколько сообщений в теме
- Компонент, управляемый сообщениями, который получает и обрабатывает сообщения, используя долговременную подписку на тему и селектор сообщений

Исходные файлы для этого раздела находятся в каталоге *`tut-install`*`/examples/jms/clientsessionmdb/`. Пути в этом разделе относятся к этому каталогу.

Компоненты приложения clientsessionmdb

Это приложение демонстрирует, как отправлять сообщения из Enterprise-бина (в данном случае из сессионного компонента), а не из клиентского приложения, как в примере в Асинхронный приём сообщений с использованием бина, управляемого сообщениями. Рисунок 49-4 иллюстрирует структуру этого приложения. Отправка сообщений из Enterprise-бина очень похожа на отправку сообщений из Managed-бина, что было показано в Отправка и получение сообщений в простом веб-приложении.

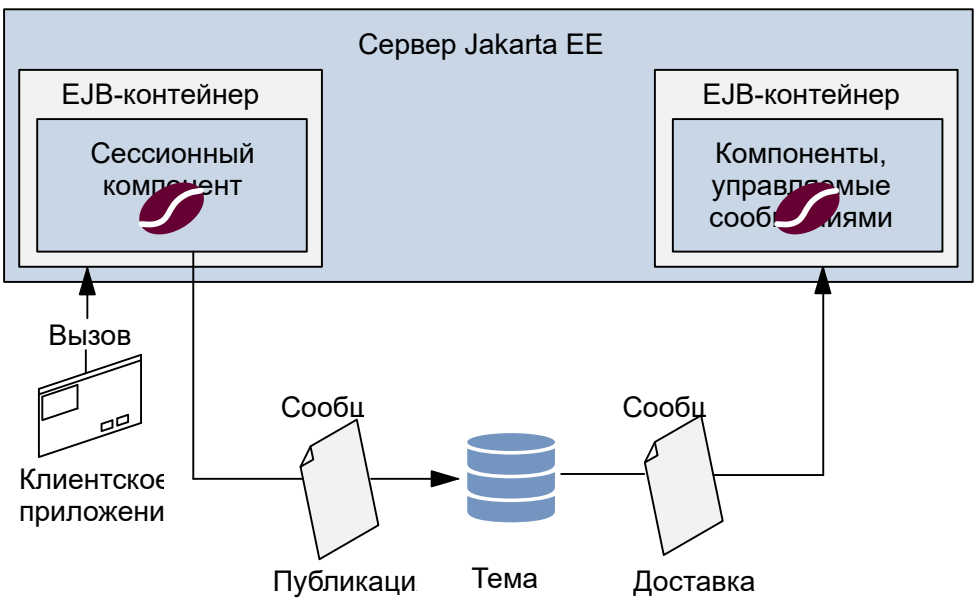

Рис. 49.4. Приложение EJB: клиент -> сессионный компонент -> управляемый сообщениями компонент

В этом примере Enterprise-бин Publisher представляет собой эквивалент корпоративного приложения для новостной ленты сервиса, которая разбивает новостные события на шесть новостных категорий. Компонент, управляемый сообщениями, может представлять новостное отделение, где, например, спортивная колонка будет устанавливать подписку на все новостные события, относящиеся к спорту.

Клиентское приложение в этом примере инъецирует удалённый домашний интерфейс Enterprise-бина Publisher и затем вызывает бизнес-метод компонента. Enterprise-бин создаёт 18 текстовых сообщений. Для каждого сообщения он случайным образом устанавливает свойство `String` категории новостей в одно из шести значений, а затем публикует сообщение в теме. Компонент, управляемый сообщениями, использует селектор сообщений для ограничения количества опубликованных сообщений, которое будет ему доставлено.

Кодирование клиентского приложения: MyAppClient.java

Клиентское приложение `MyAppClient.java`, находящееся в `clientsessionmdb-app-client`, не выполняет операций обмена сообщениями и поэтому устроен проще, чем клиент в Асинхронном получении сообщений с использованием компонента, управляемого сообщениями. Клиент использует инъецирование зависимостей для получения бизнес-интерфейса Enterprise-бина Publisher:

JAVA

```
@EJB(name="PublisherRemote")
private static PublisherRemote publisher;
```

Затем клиент дважды вызывает бизнес-метод компонента.

Кодирование сессионного компонента Publisher

Бин Publisher — это сессионный компонент без сохранения состояния, имеющий один бизнес-метод. Бин Publisher использует удалённый, а не локальный интерфейс, поскольку к нему обращается клиентское приложение.

Удалённый интерфейс `PublisherRemote.java`, расположенный в `clientsessionmdb-ejb`, объявляет один бизнес-метод `publishNews`.

Класс бинов `PublisherBean.java`, также находящийся в `clientsessionmdb-ejb`, реализует метод `publishNews` и его вспомогательный метод `chooseType`. Класс бина инъецирует ресурсы `SessionContext` и `Topic` (тема определена в бине, управляемом сообщениями). Затем он инъецирует `JMSContext`, который использует предустановленную фабрику соединений по умолчанию, если не указано иное. Класс бина начинается следующим образом:

JAVA

```
@Stateless
@Remote({
    PublisherRemote.class
})
public class PublisherBean implements PublisherRemote {

    @Resource
    private SessionContext sc;
    @Resource(lookup = "java:module/jms/newsTopic")
    private Topic topic;
    @Inject
    private JMSContext context;
    ...
}
```

Бизнес-метод `publishNews` создаёт `JMSProducer` и публикует сообщения.

Кодирование бина, управляемого сообщениями: MessageBean.java

Класс управляемого сообщениями бина `MessageBean.java`, найденный в `clientsessionmdb-ejb`, практически идентичен классу в Асинхронном приёме сообщений с использованием бина, управляемого сообщениями. Однако аннотация `@MessageDriven` отличается, потому что вместо очереди компонент использует тему, долговременную подписку и селектор сообщений. Бин определяет тему для использования приложения. Определение использует область видимости `java:module`, поскольку и сессионный компонент, и управляемый сообщениями компонент находятся в одном и том же модуле. Поскольку назначение определяется в бине, управляемом сообщениями, аннотация `@MessageDriven` использует свойство конфигурации активации `destinationLookup`. (См. Создание ресурсов для приложений Jakarta EE для получения дополнительной информации.) Аннотация также устанавливает свойства конфигурации активации `messageSelector`, `subscriptionDurability`, `clientId` и `subscriptionName` следующим образом:

JAVA

```
@JMSDestinationDefinition(
        name = "java:module/jms/newsTopic",
        interfaceName = "jakarta.jms.Topic",
        destinationName = "PhysicalNewsTopic")
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup",
            propertyValue = "java:module/jms/newsTopic"),
    @ActivationConfigProperty(propertyName = "destinationType",
            propertyValue = "jakarta.jms.Topic"),
    @ActivationConfigProperty(propertyName = "messageSelector",
            propertyValue = "NewsType = 'Sports' OR NewsType = 'Opinion'"),
    @ActivationConfigProperty(propertyName = "subscriptionDurability",
            propertyValue = "Durable"),
    @ActivationConfigProperty(propertyName = "clientId",
            propertyValue = "MyID"),
    @ActivationConfigProperty(propertyName = "subscriptionName",
            propertyValue = "MySub")
})
```

Тема определена в `PublisherBean`. Селектор сообщений в этом случае представляет как спортивные, так и аналитические колонки для демонстрации синтаксиса селекторов сообщений.

Адаптер ресурсов Jakarta Messaging использует эти свойства для создания фабрики соединений для бина, управляемого сообщениями, который позволяет бину использовать долговременную подписку.

Запуск clientsessionmdb

Вы можете использовать IDE NetBeans или Maven для сборки, развёртывания и запуска примера `simplemessage`.

В этом примере используется тема, заданная аннотацией, и предустановленная фабрика соединений по умолчанию `java:comp/DefaultJMSConnectionFactory`, поэтому не требуется создавать ресурсы для неё.

Запуск clientsessionmdb с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/jms/clientsessionmdb
   ```

4. Выберите каталог `clientsessionmdb`.

5. Убедитесь, что выбран чекбокс **Открыть требуемые проекты**, а затем кликните **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `clientsessionmdb` и выберите **Сборка**. (В среде IDE NetBeans требуется установить флажок для запуска предварительной сборки.)

   Эта команда создаёт следующее:

   a. JAR-файл клиентского приложения, содержащий файл класса клиента и удалённый интерфейс сессионного компонента, а также файл манифеста, который задаёт основной класс и помещает файл EJB JAR в его classpath.

   b. JAR-файл EJB-компонента, содержащий как сессионный компонент, так и компонент, управляемый сообщениями

   c. EAR-файл приложения, содержащий два файла JAR

   Файл `clientsessionmdb.ear` создаётся в каталоге `clientsessionmdb-ear/target/`.

   Затем команда развёртывает файл EAR, извлекает клиентские заглушки и запускает клиентское приложение.

   Клиент отображает эти строки:

   ```
   To view the bean output,
    check <install_dir>/domains/domain1/logs/server.log.
   ```

   Вывод Enterprise-бинов отображается в файле журнала сервера. Сессионный компонент Publisher отправляет два набора из 18 сообщений, пронумерованные от 0 до 17. Из-за селектора сообщений бин, управляемый сообщениями, получает только те сообщения, у которых свойство `NewsType` имеет значение `Sports` или `Opinion`.
7. Используйте **Сервисы** чтобы удалить приложения после его запуска.

Запуск clientsessionmdb с помощью Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. Перейдите в следующий каталог:

   ```
   tut-install/examples/jms/clientsessionmdb/
   ```

3. Чтобы скомпилировать исходные файлы и пакет, развернуть и запустить приложение, введите следующую команду:

   SHELL

   ```
   mvn install
   ```

   Эта команда создаёт следующее:

   - JAR-файл клиента приложения, содержащий файл класса клиента и удалённый интерфейс сессионного компонента, а также файл манифеста, указывающий основной класс и помещает JAR-файл EJB в Classpath
   - JAR-файл EJB-компонента, содержащий как сессионный компонент, так и компонент, управляемый сообщениями
   - EAR-файл приложения, содержащий два файла JAR

     Файл `clientsessionmdb.ear` создаётся в каталоге `clientsessionmdb-ear/target/`.

Затем команда развёртывает файл EAR, извлекает клиентские заглушки и запускает клиентское приложение.

Клиент отображает эти строки:

```
To view the bean output,
 check <install_dir>/domains/domain1/logs/server.log.
```

Вывод Enterprise-бинов отображается в файле журнала сервера. Сессионный компонент Publisher отправляет два набора из 18 сообщений, пронумерованные от 0 до 17. Из-за селектора сообщений бин, управляемый сообщениями, получает только те сообщения, у которых свойство `NewsType` имеет значение `Sports` или `Opinion`.

4. Удалите приложение после того, как вы закончили его выполнение:

```
mvn cargo:undeploy
```
SHELL

Использование сущности для объединения сообщений из двух MDB

В этом разделе объясняется, как написать, скомпилировать, упаковать, развернуть и запустить приложение, использующее Jakarta Messaging с сущностью. Приложение использует следующие компоненты:

- Клиентское приложение, которое отправляет и получает сообщения
- Два управляемых сообщениями бина
- Класс сущности

Вы найдете исходные файлы для этого раздела в каталоге *`tut-install`*`/examples/jms/clientmdbentity/`. Пути в этом разделе относятся к этому каталогу.

Обзор примера clientmdbentity

Это приложение упрощённо моделирует рабочий процесс отдела кадров компании (HR) при найме нового сотрудника. Это приложение также демонстрирует, как использовать платформу Jakarta EE для выполнения типовой для многих приложений задачи.

Клиент обмена сообщениями часто должен ждать несколько сообщений из разных источников. Он использует содержимое всех этих сообщений для формирования нового сообщения, которое затем отправляет другому пункту назначения. Общим термином для этого шаблона проектирования (который не является специфическим для Jakarta Messaging) является объединение сообщений. Такая задача должна быть транзакционной, со всеми получениями и отправками в рамках одной транзакции. Если часть сообщений не были успешно получены, транзакцию можно откатить. Пример клиентского приложения, который иллюстрирует эту задачу, см. в разделе Использование локальных транзакций.

Компонент, управляемый сообщениями, может обрабатывать за раз только одно сообщение в транзакции. Чтобы обеспечить возможность объединения сообщений, приложение содержит компонент, управляемый сообщениями, и хранит промежуточную информацию в персистентной сущности. Затем сущность может определить, была ли получена вся информация. Если это так, сущность может сообщить об этом обратно одному из управляемых сообщениями компонентов, который создаст и отправит сообщение в другой пункт назначения. После выполнения задачи сущность может быть удалена.

Основные этапы применения следующие.

1. Клиентское приложение отдела кадров генерирует идентификатор каждого нового сотрудника, а затем публикует сообщение (M1), содержащее имя нового сотрудника, идентификатор и должность. Он публикует сообщение в теме, потому что сообщение должно быть использовано двумя компонентами, управляемыми сообщениями. Затем клиент создаёт вре́менную очередь `ReplyQueue` со слушателем сообщений, который ожидает ответа на сообщение. (См. Создание вре́менных пунктов назначения для получения дополнительной информации.)
2. Два управляемых сообщениями бина обрабатывают каждое сообщение: один бин, `OfficeMDB` , назначает номер офиса новому сотруднику, а другой бин, `EquipmentMDB` , назначает ему оборудование. Первый компонент, обрабатывающий сообщение, создаёт и сохраняет объект с именем `SetupOffice` , а затем вызывает бизнес-метод объекта для сохранения созданной им информации. Второй компонент находит существующую сущность и вызывает другой бизнес-метод для изменения её информации.
3. Когда и офис, и оборудование были назначены, бизнес-метод объекта возвращает значение `true` бину, управляемому сообщениями, который вызвал метод. Затем управляемый сообщениями компонент отправляет в очередь ответов сообщение (M2), описывающее назначения. Затем он удаляет сущность. Слушатель сообщений клиентского приложения получает информацию.

Рисунок 49-5 иллюстрирует структуру этого приложения. Конечно, реальное приложение HR будет иметь больше компонентов. Другие компоненты могут настраивать учёт заработной платы и премий, расписание и т. д.

Рисунок 49-5 предполагает, что `OfficeMDB` является первым управляемым сообщениями компонентом, который принимает сообщение от клиента. Затем `OfficeMDB` создаёт и сохраняет объект `SetupOffice` и устанавливает информацию об офисе. Затем `EquipmentMDB` извлекает объект, устанавливает информацию об оборудовании и определяет, что объект закончил сбор информации. Затем `EquipmentMDB` отправляет сообщение в очередь ответов и удаляет объект.

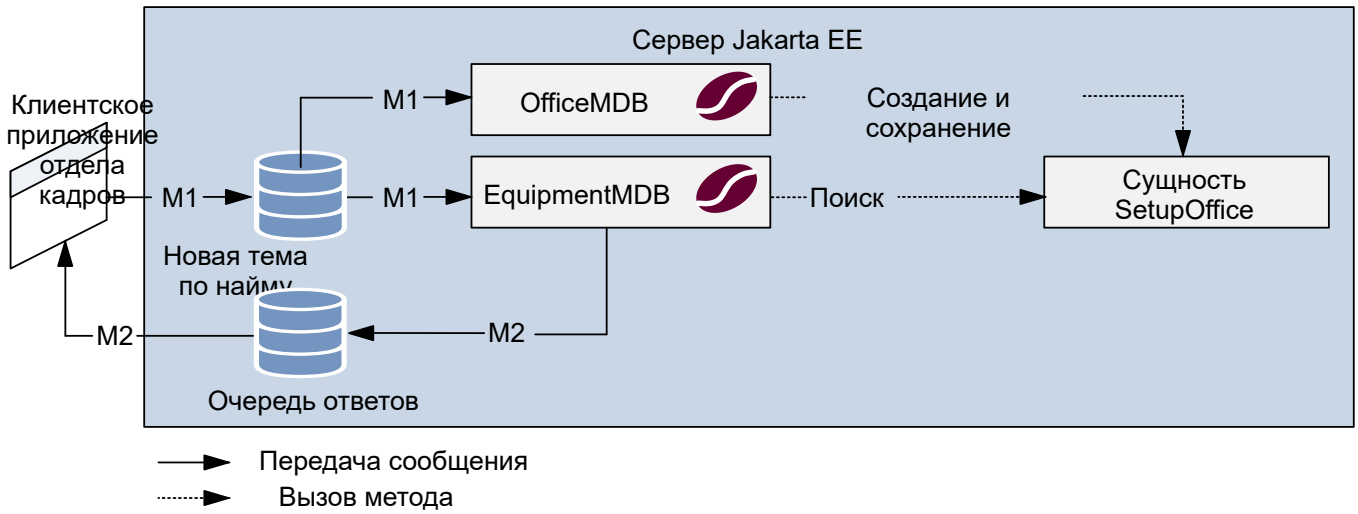

Рис. 49-5. Приложение EJB: клиент -> управляемый сообщениями компонент -> сущность

Компоненты приложения clientmdbentity

Пример включает в себя клиентское приложение, управляемые сообщениями компоненты и класс сущности.

Клиентское приложение: HumanResourceClient.java

Для создания клиентского приложения `HumanResourceClient.java` из `clientmdbentity-app-client` необходимо выполнить следующие шаги:

1. Определяет тему для приложения, используя пространство имён `java:app` , поскольку тема используется как в клиенте приложения, так и в модуле Jakarta Enterprise Beans

2. Инъецировать ресурсы `ConnectionFactory` и `Topic`
3. Создать `TemporaryQueue` для получения уведомления о происходящей обработке на основе опубликованных им событий нового найма.
4. Создать `JMSConsumer` для `TemporaryQueue`, установить слушатель сообщений `JMSConsumer` и запустить соединение
5. Создать `MapMessage`
6. Создать пять новых сотрудников со случайно сгенерированными именами, должностями и идентификационными номерами (последовательно) и опубликовать пять сообщений с информацией об этих сотрудниках

Слушатель сообщений `HRListener` ожидает сообщений, которые содержат назначенные офис и оборудование для каждого сотрудника. Когда приходит сообщение, слушатель сообщений отображает полученную информацию и определяет, все ли пять сообщений поступили. Если да, слушатель сообщения уведомляет метод `main` и тот завершается.

Управляемые сообщениями компоненты примера clientmdbentity

В этом примере используются два управляемых сообщениями бина, оба в `clientmdbentity-ejb`:

- `EquipmentMDB.java`
- `OfficeMDB.java`

Бины предпринимают следующие шаги.

1. Они инъецируют ресурсы `MessageDrivenContext`, `EntityManager` и `JMSContext`.
2. Метод `onMessage` извлекает информацию из сообщения. Метод `onMessage` у `EquipmentMDB` выбирает оборудование в зависимости от должности нового сотрудника. Метод `onMessage` у `OfficeMDB` случайным образом генерирует номер офиса.
3. После небольшой задержки (для моделирования реальных проблем обработки) метод `onMessage` вызывает вспомогательный метод `compose`.
4. Метод `compose` выполняет следующие шаги.
   a. Создаёт и сохраняет объект сущности `SetupOffice`, либо выбирает его по первичному ключу.
   b. Использует объект для хранения оборудования или служебной информации в базе данных, вызывая либо бизнес-метод `doEquipmentList`, либо `doOfficeNumber`.
   c. Если бизнес-метод возвращает `true`, что означает, что вся информация была сохранена, он извлекает информацию о пункте назначения для ответа из сообщения, создаёт `JMSProducer` и отправляет ответное сообщение с информацией, которая содержится в объекте.
   d. Удаляет сущность.

Класс сущности примера clientmdbentity

Класс сущности `SetupOffice.java` находится также в `clientmdbentity-ejb`. Сущность и компоненты, управляемые сообщениями, упаковываются вместе в JAR-файл EJB. Класс сущности объявлен следующим образом:

```java
@Entity
public class SetupOffice implements Serializable { ... }
```

Класс содержит конструктор без аргументов и конструктор, который принимает два аргумента: идентификатор сотрудника и его имя. Он также содержит get- и set- методы для идентификатора сотрудника, его имени, номера офиса и списка оборудования. Метод получения идентификатора сотрудника имеет аннотацию `@Id` для указания, что это поле является первичным ключом:

JAVA

```
@Id
public String getEmployeeId() {
    return id;
}
```

Класс также реализует два бизнес-метода `doEquipmentList` и `doOfficeNumber` и их вспомогательный метод `checkIfSetupComplete`.

Бины, управляемые сообщениями, вызывают бизнес-методы и методы получения.

В файле `persistence.xml` для сущности указаны основные настройки:

XML

```
<persistence version="3.0"
             xmlns="https://jakarta.ee/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
               https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd">
  <persistence-unit name="clientmdbentity-ejbPU" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>java:comp/DefaultDataSource</jta-data-source>
    <properties>
      <property name="eclipselink.ddl-generation"
                value="drop-and-create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

Запуск примера clientmdbentity

Вы можете использовать IDE NetBeans или Maven для сборки, развёртывания и запуска примера `clientmdbentity`.

Поскольку в примере определяется собственная тема приложения и используется предустановленная фабрика соединений по умолчанию `java:comp/DefaultJMSConnectionFactory` и предустановленный ресурс JDBC по умолчанию `java:comp/DefaultDataSource`, не нужно создавать дополнительных ресурсов.

Запуск clientmdbentity в IDE NetBeans

1. Убедитесь, что GlassFish Server (см. Запуск и остановка GlassFish Server), а также сервер базы данных (см. Запуск и остановка Apache Derby) запущены.
2. В меню **Файл** выберите **Открыть проект**.
3. В диалоговом окне **Открыть проект** перейдите к:

   ```
   tut-install/examples/jms/clientmdbentity
   ```

4. Выберите каталог `clientmdbentity`.
5. Нажмите **Открыть проект**.
6. На вкладке **Проекты** кликните правой кнопкой мыши проект `clientmdbentity` и выберите **Сборка**.

Эта команда создаёт следующее:

- JAR-файл приложения-клиента, содержащий файлы класса клиента и класса слушателя, а также файл манифеста, в котором указан основной класс
- JAR-файл EJB, содержащий компоненты, управляемые сообщениями, и класс сущности вместе с файлом `persistence.xml`
- Файл EAR приложения, который содержит два JAR-файла вместе с файлом `application.xml`

  Файл `clientmdbentity.ear` создаётся в каталоге `clientmdbentity-ear/target/`.

  Затем команда развёртывает файл EAR, извлекает клиентские заглушки и запускает клиентское приложение.

Запуск clientmdbentity с помощью Maven

1. Убедитесь, что GlassFish Server (см. Запуск и остановка GlassFish Server), а также сервер базы данных (см. Запуск и остановка Apache Derby) запущены.
2. Перейдите в следующий каталог:

   ```
   tut-install/examples/jms/clientmdbentity/
   ```

3. Чтобы скомпилировать исходные файлы и пакет, развернуть и запустить приложение, введите следующую команду:

   ```
   mvn install
   ```
   SHELL

   Эта команда создаёт следующее:

   - JAR-файл приложения-клиента, содержащий файлы класса клиента и класса слушателя, а также файл манифеста, в котором указан основной класс
   - JAR-файл EJB, содержащий компоненты, управляемые сообщениями, и класс сущности вместе с файлом `persistence.xml`
   - Файл EAR приложения, который содержит два JAR-файла вместе с файлом `application.xml`

     Затем команда развёртывает приложение, получает клиентские заглушки и запускает клиентское приложение.

Просмотр вывода приложения

Вывод в окне вывода IDE NetBeans или в окне терминала выглядит примерно так (ему предшествует вывод контейнера клиентского приложения и вывод Maven):

```
SENDER: Setting hire ID to 50, name Bill Tudor, position Programmer
SENDER: Setting hire ID to 51, name Carol Jones, position Senior Programmer
SENDER: Setting hire ID to 52, name Mark Wilson, position Manager
SENDER: Setting hire ID to 53, name Polly Wren, position Senior Programmer
SENDER: Setting hire ID to 54, name Joe Lawrence, position Director
Waiting for 5 message(s)
New hire event processed:
  Employee ID: 52
  Name: Mark Wilson
  Equipment: Tablet
  Office number: 294
Waiting for 4 message(s)
New hire event processed:
  Employee ID: 53
  Name: Polly Wren
  Equipment: Laptop
  Office number: 186
Waiting for 3 message(s)
New hire event processed:
  Employee ID: 54
  Name: Joe Lawrence
  Equipment: Mobile Phone
  Office number: 135
Waiting for 2 message(s)
New hire event processed:
  Employee ID: 50
  Name: Bill Tudor
  Equipment: Desktop System
  Office number: 200
Waiting for 1 message(s)
New hire event processed:
  Employee ID: 51
  Name: Carol Jones
  Equipment: Laptop
  Office number: 262
```

Вывод из бинов, управляемых сообщениями, и класса сущностей появляется в журнале сервера.

Для каждого сотрудника приложение сначала создаёт объект сущности, а затем извлекает его из хранилища. Вы можете увидеть ошибки времени выполнения в журнале сервера и произошедшие откаты соответствующих транзакций. Ошибки возникают, если оба управляемых сообщениями бина одновременно обнаруживают, что сущность ещё не существует, поэтому они оба пытаются его создать. Первая попытка успешна, но вторая — неудачна, потому что сущность уже существует. После отката и на последующей повторной попытке второй управляемый сообщениями компонент успешно извлекает сущность из хранилища. Управляемые контейнером транзакции позволяют приложению работать корректно, несмотря на эти ошибки, без специального программирования.

Чтобы удалить приложение после его завершения, перейдите на вкладку «Службы» или введите команду `mvn cargo:undeploy`.

Создание ресурсов Jakarta Messaging в IDE NetBeans

Когда пишутся собственные приложения обмена сообщениями, нужно создавать ресурсы для них. В этом разделе объясняется, как использовать IDE NetBeans для создания файлов `src/main/setup/glassfish-resources.xml`, аналогичных использующимся в примерах этой главы. Также объясняется, как использовать IDE NetBeans для удаления ресурсов.

Вы также можете создавать, просматривать и удалять ресурсы Jakarta Messaging из Консоли администрирования или команд `asadmin create-jms-resource`, `asadmin list-jms-resources` и `asadmin delete-jms-resources`. Для получения информации обратитесь к документации GlassFish Server или введите команду `asadmin help` и имя команды.

Создание ресурсов Jakarta Messaging в IDE NetBeans

Выполните следующие действия, чтобы создать ресурс Jakarta Messaging в GlassFish Server в IDE NetBeans. Повторите эти шаги для каждого требующегося ресурса.

1. Кликните правой кнопкой мыши проект, для которого требуется создать ресурсы, и выберите **Создать**, затем выберите **Другое**.
2. В мастере **Создать файл** в разделе **Категории** выберите **GlassFish**.
3. В разделе **Типы файлов** выберите **Ресурс JMS**.
4. На странице Общие атрибуты — Ресурс JMS в поле Имя JNDI введите имя ресурса.

   По соглашению имена ресурсов обмена сообщениями начинаются с `jms/`.
5. Выберите опцию для типа ресурса.

   Обычно это либо `jakarta.jms.Queue`, `jakarta.jms.Topic`, либо `jakarta.jms.ConnectionFactory`.
6. Кликните **Следующий**.
7. На странице свойств JMS для очереди или темы введите имя физической очереди в поле «Значение» свойства «Имя».

   Вы можете ввести любое значение для этого обязательного поля.

   Фабрики соединений не имеют обязательных свойств. В некоторых ситуациях может потребоваться указать свойство.
8. Нажмите **Готово**.

   Файл с именем `glassfish-resources.xml` создаётся в вашем проекте Maven в каталоге `src/main/setup/`. На вкладке «Проекты» вы можете найти его в разделе «Другие источники». Вам нужно будет выполнить команду `asadmin add-resources`, чтобы создать ресурсы в GlassFish Server.

Удаление ресурсов Jakarta Messaging в IDE NetBeans

1. На вкладке **Сервисы** разверните узел **Серверы**, затем узел **GlassFish Server**.
2. Разверните узел **Ресурсы**, а затем узел **Ресурсы соединения**.
3. Разверните узел **Объекты администрируемых ресурсы**.
4. Кликните правой кнопкой мыши любой пункт назначения, который хотите удалить, и выберите **Отменить регистрацию**.
5. Разверните узел **Пулы подключений коннекторов**.
6. Кликните правой кнопкой мыши пул подключений, соответствующий удалённой фабрике подключений, и выберите **Отменить регистрацию**.

   При удалении пула коннекторов соединений соответствующий ресурс коннектора также удаляется. Это действие удаляет фабрику соединений.

Часть X: Безопасность

В части X рассматриваются концепции безопасности и примеры их использования.

Глава 50. Введение в безопасность Jakarta EE Platform

В этой главе представлены основные концепции и механизмы безопасности. Более подробную информацию об этих концепциях и механизмах можно найти в главе, посвящённой безопасности, содержащейся в спецификации Jakarta EE 9.

Обзор Jakarta Security

На каждом предприятии имеются конфиденциальные ресурсы, к которым могут обращаться многие пользователи, или ресурсы, передающиеся по незащищённым открытым сетям, и которые должны быть защищены.

Приложения слоя предприятия и веб-слоя состоят из компонентов, которые развёрнуты в различных контейнерах. Эти компоненты объединяются для создания многослойного корпоративного приложения. Безопасность компонентов обеспечивается их контейнерами. Контейнер обеспечивает два вида безопасности: декларативный и программный.

- Декларативная безопасность выражает требования безопасности компонента приложения, используя дескрипторы развёртывания или аннотации.

  Дескриптор развёртывания — это XML-файл, который является внешним по отношению к приложению и описывает структуру безопасности приложения, включая роли безопасности, управление доступом и требования к аутентификации. Для получения дополнительной информации о дескрипторах развёртывания прочитайте Использование дескрипторов развёртывания для декларативной безопасности.

  Аннотации (иногда называемые также метаданными) используются для указания информации о безопасности в файле класса. Когда приложение развёрнуто, эта информация может быть использована или переопределена дескриптором развёртывания приложения. Аннотации избавляют от необходимости записывать декларативную информацию внутри XML-дескрипторов. Вместо этого вы просто помещаете аннотации в код, и необходимая информация генерируется автоматически. В этом уроке аннотации используются для защиты приложений везде, где это возможно. Для получения дополнительной информации об аннотациях см. Использование аннотаций для указания информации о безопасности.
- Программная безопасность встроена в приложение и используется для принятия решений по доступу. Программная безопасность полезна, когда одной декларативной безопасности недостаточно для реализации модели безопасности приложения. Для получения дополнительной информации о программной безопасности прочитайте Использование программной безопасности.

Jakarta EE 9 содержит спецификацию API безопасности, которая определяет переносимые подключаемые интерфейсы для аутентификации и хранилищ идентификаторов, а также новый инъецируемый тип `SecurityContext` , который обеспечивает точку доступа для программной безопасности. Вы можете использовать встроенные реализации этих API или определить кастомные реализации.

Более подробную информацию об этих концепциях и механизмах можно найти в главе, посвящённой безопасности, содержащейся в спецификации Jakarta EE 9.

В других главах этой части рассматриваются требования к безопасности в приложениях веб-уровня и уровня бизнес-логики, а также Jakarta Security.

- Глава 51 *Начало работы по защите веб-приложений* объясняет, как защитить веб-компоненты, такие как сервлеты.

- Глава 52 *Начало работы по защите EJB-приложений* объясняет, как защитить компоненты Jakarta EE, такие как Enterprise-бин и клиентские приложения.
- Глава 53 *Использование Jakarta Security* описывает функциональные возможности аутентификации и валидации учётных данных, предоставляемые Jakarta Security, и примеры их использования.

Простое руководство по безопасности приложений

Поведение безопасности среды Jakarta EE может стать более понятным при изучении того, что происходит в простом приложении с веб-клиентом, пользовательским интерфейсом и бизнес-логикой Enterprise-бина.

В следующем примере, который взят из спецификации Jakarta EE, веб-клиент использует веб-сервер в качестве своего посредника аутентификации, собирает данные аутентификации пользователя с клиента и используя их для установления аутентифицированной сессии.

Шаг 1: Первоначальный запрос

На первом шаге этого примера веб-клиент запрашивает основной URL приложения. Это действие показано на рисунке 50-1.

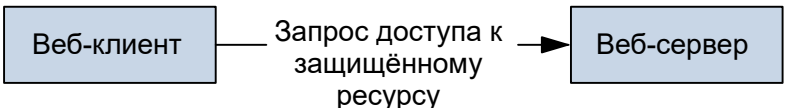

Рисунок 50-1 Первоначальный запрос

Поскольку клиент ещё не аутентифицировался в среде приложения, сервер, ответственный за доставку веб-части приложения (веб-сервер), обнаруживает это и вызывает соответствующий механизм аутентификации для этого ресурса. Для получения дополнительной информации об этих механизмах см. Механизмы безопасности.

Шаг 2: Начальная аутентификация

Веб-сервер возвращает форму, которую веб-клиент использует для сбора данных аутентификации, таких как имя пользователя и пароль, от пользователя. Веб-клиент пересылает данные аутентификации на веб-сервер, где они проверяются веб-сервером, как показано на рис. 50-2. Механизм проверки может быть локальным для сервера или использовать сервисы безопасности. По итогам проверки веб-сервер устанавливает учётные данные для пользователя.

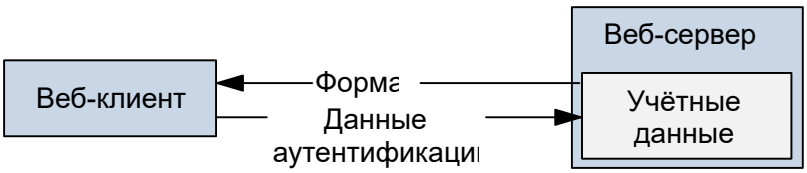

Рисунок 50-2 Начальная аутентификация

Шаг 3: Авторизация URL

Учётные данные используются в будущем для определения того, авторизован ли пользователь для доступа к ограниченным ресурсам, которые он может запросить. Веб-сервер обращается к политике безопасности, связанной с веб-ресурсом, чтобы определить роли безопасности, которым разрешён доступ к ресурсу. Политика безопасности определяется на основе аннотаций или дескриптора развёртывания. Затем веб-контейнер проверяет учётные данные пользователя для каждой роли, чтобы определить, может ли он соотнести пользователя с этой ролью. На рисунке 50-3 показан этот процесс.

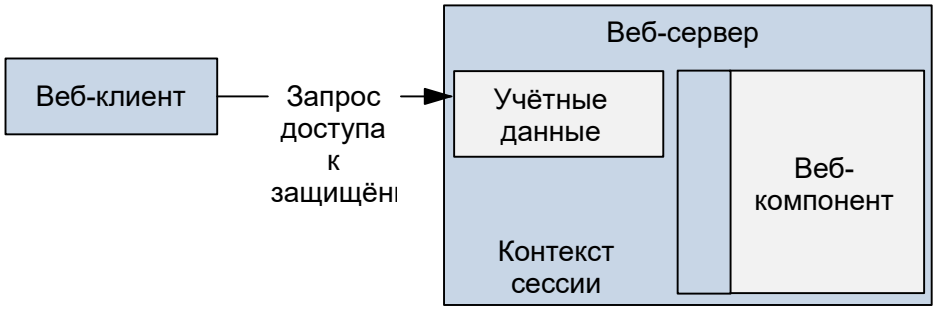

Рис. 50-3. Авторизация URL

Оценка веб-сервера заканчивается «авторизованным» результатом, если веб-сервер может соотнести пользователя с ролью. «Несанкционированный» результат достигается, если веб-сервер не может соотнести пользователя с какой-либо из разрешённых ролей.

Шаг 4: Выполнение исходного запроса

Если пользователь авторизован, веб-сервер возвращает результат исходного запроса URL, как показано на рис. 50-4.

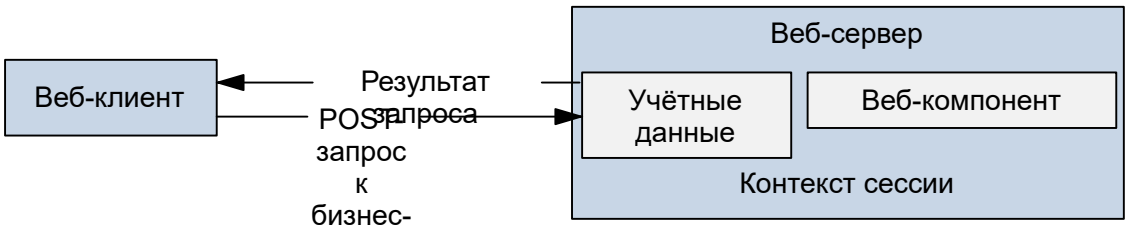

Рисунок 50-4 Выполнение исходного запроса

В нашем примере в ответ возвращается URL веб-страницы, что позволяет пользователю размещать данные формы, которые должны обрабатываться компонентом бизнес-логики приложения. См. главу 51 *Начало работы по защите веб-приложений* для получения дополнительной информации о защите веб-приложений.

Шаг 5: Вызов бизнес-методов Enterprise-бина

Веб-страница выполняет удалённый вызов метода для Enterprise-бина, используя учётные данные пользователя для установления безопасной связи между веб-страницей и Enterprise-бином, как показано на рис. 50-5. Связь реализуется как два связанных Security контекста: один на веб-сервере и один в EJB-контейнере.

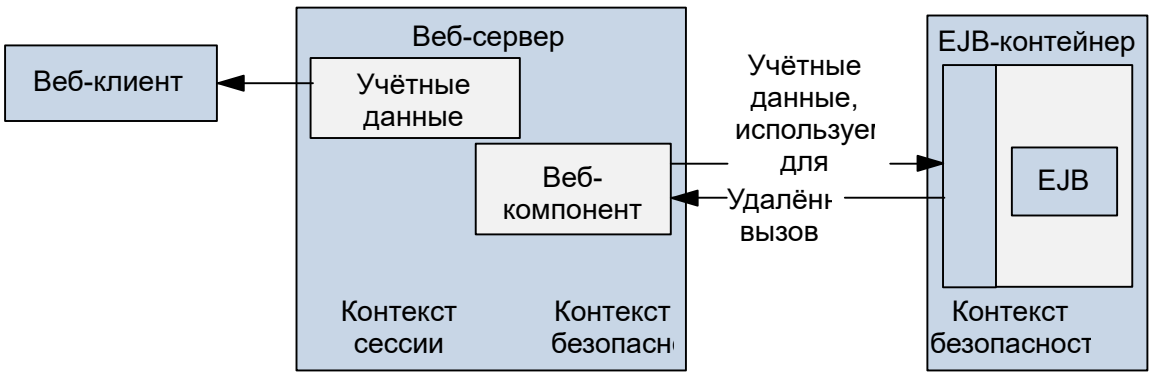

Рис. 50-5. Вызов бизнес-метода Enterprise-бина

EJB-контейнер отвечает за контроль доступа к методам EJB-компонента. Контейнер обращается к политике безопасности, связанной с Enterprise-бином, чтобы определить роли безопасности, которым разрешён доступ к методу. Политика безопасности определяется на основе аннотаций или дескриптора развёртывания. Для

каждой роли EJB-контейнер определяет, может ли он определить роль вызывающего субъекта, используя контекст безопасности, связанный с вызовом.

Оценка контейнера заканчивается «авторизованным» результатом, когда контейнер может соотнести учётные данные вызывающего субъекта с ролью. «Несанкционированный» результат достигается, если контейнер не смог соотнести вызывающего субъекта с какой-либо из разрешённых ролей. «Несанкционированный» результат вызывает исключение, выбрасываемое контейнером и передающееся обратно на вызывающую веб-страницу.

Если вызов авторизован, контейнер отправляет управление методу Enterprise-бина. Результат выполнения вызова бином возвращается на веб-страницу и, в конечном счёте, пользователю веб-сервером и веб-клиентом.

Особенности механизма безопасности

Правильно реализованный механизм безопасности обеспечивает следующие функциональные возможности:

- Предотвращение несанкционированного доступа к функциям приложения и коммерческим или персональным данным (аутентификация)
- Привлечение пользователей системы к ответственности за операции, которые они выполняют (безотзывность)
- Защита системы от простоев в обслуживании и других нарушений, влияющих на качество обслуживания.

В идеале, корректно реализованные механизмы безопасности также будут обеспечивать

- Лёгкость администрирования
- Прозрачность для пользователей системы
- Сквозная совместимость между приложениями и предприятиями

Характеристики безопасности приложений

Приложения Jakarta EE состоят из компонентов, которые могут содержать как защищённые, так и незащищённые ресурсы. Зачастую требуется защитить ресурсы, чтобы обеспечить доступ только авторизованным пользователям. Авторизация обеспечивает контролируемый доступ к защищённым ресурсам. Авторизация основана на идентификации и аутентификации. Идентификация — это процесс, позволяющий системе распознавать субъект, а аутентификация — это процесс, проверяющий личность пользователя, устройство или другого субъекта в информационной системе, обычно в качестве предварительного условия для предоставления доступа к ресурсам в системе.

Авторизация и аутентификация не требуются для доступа субъекта к незащищённым ресурсам. Доступ к ресурсу без аутентификации называется неаутентифицированным, или анонимным доступом.

Характеристики безопасности приложений, которые при правильном обращении помогают минимизировать угрозы безопасности, с которыми сталкивается предприятие:

- Аутентификация: средство, с помощью которого взаимодействующие объекты, такие как клиент и сервер, доказывают друг другу, что они действуют от имени определённых идентичностей, которым разрешён доступ. Это гарантирует, что пользователи являются теми, за кого они себя выдают.
- Авторизация, или контроль доступа: средства, с помощью которых взаимодействия с ресурсами ограничиваются заданным набором пользователей или программ с целью обеспечения соблюдения целостности, конфиденциальности или доступности. Это гарантирует, что пользователи имеют

разрешение на выполнение операций или доступ к данным.

- Целостность данных: средство, используемое для доказательства того, что информация не была изменена третьей стороной — субъектом, не являющимся источником информации. Например, получатель данных, отправленных по открытой сети, должен иметь возможность обнаруживать и отбрасывать сообщения, которые были изменены после их отправки. Это гарантирует, что только авторизованные пользователи могут изменять данные.
- Конфиденциальность, или приватность данных. Средства, используемые для обеспечения того, чтобы информация была доступна только пользователям, имеющим доступ к ней. Это гарантирует, что только авторизованные пользователи могут просматривать конфиденциальные данные.
- Безотзывность: средство, используемое для доказательства того, что пользователь, совершивший какое-либо действие, не может отрицать, что он это сделал. Это гарантирует, что транзакции могут быть подтверждены.
- Качество обслуживания: средства, используемые для обеспечения лучшего обслуживания выбранного сетевого трафика по различным технологиям.
- Аудит: средство, используемое для сбора защищённой от несанкционированного доступа записи событий, связанных с безопасностью, с целью оценки эффективности политик и механизмов безопасности. Для этого система ведёт учёт транзакций и информацию о безопасности.

Механизмы безопасности

Характеристики приложения должны учитываться при определении уровня и типа защиты, которая должна быть предоставлена для приложений. В следующих разделах обсуждаются характеристики общих механизмов, которые можно использовать для защиты приложений Jakarta EE. Каждый из этих механизмов может использоваться индивидуально или совместно с другими для обеспечения уровней защиты в зависимости от потребностей конкретного приложения.

Механизмы безопасности Java SE

Java SE обеспечивает поддержку различных функций и механизмов безопасности.

- Сервис аутентификации и авторизации Java (Java Authentication and Authorization Service — JAAS) — это набор API, которые позволяют службам аутентифицировать и обеспечивать контроль доступа для пользователей. JAAS предоставляет подключаемую и расширяемую среду для программной аутентификации и авторизации пользователей. JAAS — это базовый API Java SE и технология, лежащая в основе механизмов безопасности Jakarta EE.
- Java Generic Security Services (Java GSS-API) — это API на основе токенов, используемый для безопасного обмена сообщениями между взаимодействующими приложениями. GSS-API предлагает разработчикам приложений унифицированный доступ к сервисам безопасности на основе множества базовых механизмов безопасности, включая Kerberos.
- Расширение Java Cryptography Extension (JCE) предоставляет структуру и реализацию для шифрования, генерации ключей и согласования ключей, а также алгоритмов кода аутентификации сообщений (MAC). Поддержка шифрования включает в себя симметричные, асимметричные, блочные и потоковые шифры. Блочные шифры работают с группами байтов. Потоковые шифры обрабатывают по одному байту за раз. Программное обеспечение также поддерживает безопасные потоки и запечатанные (sealed) объекты.
- Расширение Java Secure Sockets (JSSE) предоставляет платформу и реализацию в Java протоколов Secure Sockets Layer (SSL) и Transport Layer Security (TLS) и включает в себя функциональность для шифрования данных, серверной аутентификации, целостности сообщений и, опционально, аутентификации клиента по цифровому сертификату для обеспечения безопасного интернет-соединения.

- Простой слой аутентификации и безопасности (Simple Authentication and Security Layer — SASL) — это стандарт Интернета (RFC 2222), который определяет протокол для аутентификации и необязательного установления слоя безопасности между клиентскими и серверными приложениями. SASL определяет способ обмена данными аутентификации, но сам по себе не определяет содержимое этих данных. SASL — это фреймворк, в который могут вписываться конкретные механизмы аутентификации, которые определяют содержимое и семантику данных аутентификации.

Java SE также предоставляет набор инструментов для управления хранилищами ключей, сертификатами и файлами политик, генерирования и проверки подписей JAR, получения, распечатки и управления тикетами Kerberos.

Дополнительная информация о защите Java SE находится на веб-сайте https://docs.oracle.com/javase/8/docs/technotes/guides/security/.

Механизмы безопасности Jakarta EE

Сервисы безопасности Jakarta EE предоставляются контейнером компонентов и могут быть реализованы с использованием декларативных или программных методов (см. Защита контейнеров). Сервисы безопасности Jakarta EE предоставляют надёжный и легко настраиваемый механизм безопасности для аутентификации пользователей и авторизации доступа к функциям приложения и связанным данным на различных уровнях. Сервисы безопасности Jakarta EE отделены от механизмов безопасности операционной системы.

Безопасность на уровне приложений

В Jakarta EE контейнеры компонентов отвечают за обеспечение безопасности на уровне приложений, сервисов безопасности для определённого типа приложений, адаптированных к потребностям приложения. На уровне приложения брандмауэры приложений могут использоваться для усиления защиты приложений путём защиты коммуникационного потока и всех связанных с ним ресурсов приложения от атак.

Jakarta Security легко реализуется и настраивается и может предложить детальный контроль доступа к функциям и данным приложений. Однако, как присуще безопасности, применяемой на уровне приложений, свойства безопасности не могут передаваться приложениям, работающим в других средах, и защищают данные только тогда, когда они находятся в среде приложений. В контексте традиционного корпоративного приложения это не всегда будет проблемой, но применительно к приложению веб-сервисов, в котором данные часто передаются через посредников, необходимо будет использовать механизмы безопасности Jakarta EE наряду с безопасностью транспортного уровня и безопасностью на уровне сообщений для более полного решения вопроса.

Преимущества использования безопасности на уровне приложений.

- Безопасность однозначно соответствует потребностям приложения.
- Безопасность детальная, с настройками приложения.

Недостатки использования безопасности на уровне приложений:

- Приложение зависит от атрибутов безопасности, которые нельзя передавать между типами приложений.
- Поддержка нескольких протоколов делает этот тип безопасности уязвимым.
- Данные близки или находятся в пределах уязвимости.

Для получения дополнительной информации об обеспечении безопасности на уровне приложений см. Защита контейнеров.

Безопасность транспортного уровня

Безопасность транспортного уровня обеспечивается транспортными механизмами, используемыми для передачи информации по проводам между клиентами и поставщиками. Таким образом, безопасность транспортного уровня опирается на безопасный транспорт HTTP (HTTPS) с использованием Secure Sockets Layer (SSL). Транспортная безопасность — это механизм безопасности «точка-точка», который можно использовать для проверки подлинности, целостности сообщений и конфиденциальности. При использовании защищённой SSL сессии сервер и клиент могут аутентифицировать друг друга и согласовывать алгоритм шифрования и криптографические ключи, прежде чем протокол приложения передаст или получит свой первый байт данных. Защита активна с момента, когда данные покидают клиента, до их прибытия в пункт назначения, и в обратную сторону, даже с учётом посредников. Проблема в том, что данные не защищены, когда они попадают в пункт назначения. Одним из решений является шифрование сообщения перед отправкой.

Безопасность транспортного уровня выполняется в несколько этапов следующим образом.

- Клиент и сервер согласовывают подходящий алгоритм.
- Обмен ключами осуществляется с использованием шифрования с открытым ключом и аутентификации на основе сертификатов.
- При обмене информацией используется симметричный шифр.

Цифровые сертификаты необходимы при запуске HTTPS с использованием SSL. Служба HTTPS большинства веб-серверов не будет работать, если не установлен цифровой сертификат. Цифровые сертификаты уже созданы для GlassFish Server.

Преимущества использования безопасности транспортного уровня:

- Это относительно простая, понятная, стандартная технология.
- Она применима как к телу сообщения, так и к его вложениям.

К недостаткам использования безопасности транспортного уровня относятся:

- Она тесно связана с протоколом транспортного уровня.
- Она представляет собой подход к безопасности «всё или ничего». Это подразумевает, что механизм безопасности не знает о содержимом сообщения, поэтому вы не можете выборочно применять защиту к частям сообщения, как это возможно с безопасностью на уровне сообщений.
- Защита вре́менная. Сообщение защищено только при передаче. Защита снимается автоматически конечной точкой при получении сообщения.
- Это решение точка-точка, но не сквозное решение.

Для получения дополнительной информации о безопасности транспортного уровня см. Установление безопасного соединения с использованием SSL.

Безопасность на уровне сообщений

При защите на уровне сообщений информация о безопасности содержится во вложении сообщения SOAP и/или сообщения SOAP, что позволяет информации о безопасности перемещаться вместе с сообщением или вложением. Например, часть сообщения может быть подписана отправителем и зашифрована для конкретного получателя. При отправке от первоначального отправителя сообщение может проходить через промежуточные узлы, прежде чем достигнет своего предполагаемого получателя. В этом сценарии

зашифрованные части остаются непрозрачными для любых промежуточных узлов и могут быть расшифрованы только предполагаемым получателем. По этой причине безопасность на уровне сообщений также иногда называют сквозной защитой.

Преимущества безопасности на уровне сообщений:

- Сообщение остаётся защищённым на всех транзитных участках и после того, как сообщение прибывает в пункт назначения.
- Защита может быть выборочно применена к различным частям сообщения и, если используется безопасность XML веб-сервисов, к вложениям.
- Защита сообщений может использоваться при наличии нескольких транзитных посредников.
- Безопасность сообщений не зависит от окружения, в котором работает приложение, и транспортного протокола.

Недостаток использования безопасности на уровне сообщений заключается в том, что она является относительно сложной и увеличивает накладные расходы на обработку.

GlassFish Server поддерживает безопасность сообщений с помощью Metro, стека веб-сервисов, который использует безопасность веб-сервисов (WSS) для защиты сообщений. Поскольку эта защита сообщений специфична для Metro и не является частью платформы Jakarta EE, в этом руководстве не обсуждается использование WSS для защиты сообщений. Смотрите публикацию Metro User's Guide по ссылке: https://eclipse-ee4j.github.io/metro-jax-ws/.

Использование подключаемых провайдеров

Jakarta EE включает две спецификации, определяющие интерфейсы SPI для подключаемых провайдеров безопасности: Jakarta Authentication и Jakarta Security. Эти спецификации описаны более подробно в следующих разделах:

- Jakarta Authentication
- Jakarta Security

Jakarta Authentication

Jakarta Authentication определяет модель защиты сообщений, отправляемых между клиентом и сервером, в которой отправитель сообщения «защищает» его, а получатель его «валидирует». Сведения о том, как сообщения защищаются и валидируются, не определены моделью. Поддержка защиты и валидации определённых типов сообщений обеспечивается «модулями auth» — реализациями аутентификации `ClientAuthModule` и `ServerAuthModule` — которые поддерживают определённые протоколы или типы сообщений и подключаются к инфраструктуре аутентификации. (Обратите внимание, что клиенту и серверу не обязательно использовать аутентификацию, если обе стороны правильно обрабатывают сообщения для данного протокола.)

Аутентификация определяет два «профиля» для интеграции модулей проверки подлинности в контейнерах Jakarta EE: профиль контейнера сервлетов и профиль SOAP. Каждый указывает, как обработка сообщений аутентификации должна быть интегрирована в поток обработки запросов данного контейнера для валидации входящих запросов и защиты исходящих ответов.

В случае профиля контейнера сервлета, если `ServerAuthModule` настроен/доступен для данного контекста приложения, то метод `validateRequest()` модулей должен быть вызван (и успешно выполнен) перед авторизацией доступа и вызовом целевого сервлета, перед вызовом необходимо вызвать метод

`secureResponse()` модуля. Как правило, `ServerAuthModule`, записанный для профиля контейнера сервлета, ищет учётные данные пользователя или токены во входящем запросе, а затем использует их для аутентификации вызывающего субъекта и установления его личности. `ServerAuthModule` может также участвовать в протоколе вызова/ответа с клиентом или взаимодействовать с третьей стороной для установления/проверки личности клиента.

Как и в случае с профилем контейнера сервлета, профиль SOAP требует, чтобы `validateRequest()` был вызван и успешно завершён, прежде чем продолжить авторизацию доступа и любую дальнейшую обработку входящего сообщения, и что `secureResponse()` вызывается для ответа перед его отправкой. В отличие от профиля контейнера сервлета, обработка `validateRequest()` для сообщений SOAP обычно включает проверку подписей на подписанных элементах, дешифрование зашифрованных элементов и/или установление личности субъекта SOAP на основе токена, включённого в сообщение, в то время как `secureResponse()` обычно включает в себя подписывание и/или шифрование элементов исходящего сообщения.

Аутентификация не определяет никаких стандартных или встроенных модулей ServerAuthModules. Они должны предоставляться либо приложением, использующим модуль, либо как нестандартное расширение продукта Jakarta EE от конкретного поставщика. Иногда ServerAuthModules могут быть напрямую сконфигурированы для приложения в зависимости от поставщика, но стандартный механизм обеспечения доступности `ServerAuthModule` для приложения заключается в регистрации соответствующего `AuthConfigProvider` в глобальный `AuthConfigFactory`. `AuthConfigProvider` делает `ServerAuthModule` доступным для контейнера через серию промежуточных объектов для обработки сообщений во время выполнения.

Jakarta Security

Jakarta Security определяет следующий модуль SPI, связанный с идентификацией:

- `HttpAuthenticationMechanism` — интерфейс для модулей, которые аутентифицируют вызывающих субъектов в веб-приложении. Он определяет три метода, которые соответствуют методам проверки подлинности `ServerAuthModule`, хотя и с немного отличающимися сигнатурами. `HttpAuthenticationMechanism` предоставляет функциональность, аналогичную `ServerAuthModule`, а контейнер сервлетов использует специальный `ServerAuthModule` для вызова методов HttpAuthenticationMechanism, но HttpAuthenticationMechanism проще для написания и развёртывания, чем ServerAuthModule.
- `IdentityStore` — этот интерфейс определяет методы для проверки учётных данных вызывающего субъекта (таких как имя пользователя и пароль) и возврата информации о членстве в группе. Объекты IdentityStore вызываются под управлением `IdentityStoreHandler`, который, если присутствует несколько объектов IdentityStore, вызывает доступные объекты IdentityStore в определённом порядке и агрегирует результаты.
- `RememberMeIdentityStore` — этот интерфейс представляет собой вариант интерфейса `IdentityStore`, предназначенный специально для решения случаев, когда идентификация аутентифицированного пользователя должна запоминаться в течение продолжительного периода времени, чтобы вызывающий субъект мог периодически возвращаться к приложению без необходимости каждый раз представлять учётные данные первичной аутентификации.

Реализации этих интерфейсов SPI являются компонентами CDI, и, как таковые, приложения могут предоставлять реализации, которые поддерживают механизмы аутентификации конкретного приложения, или проверять учётные данные пользователя по хранилищам идентификаторов конкретного приложения, просто включив их в архив компонентов, который является частью развёрнутого приложения. Существует

также несколько стандартных встроенных реализаций `HttpAuthenticationMechanism` и `IdentityStore`, которые обеспечивают настраиваемую поддержку общих случаев использования аутентификации и проверки учётных данных без необходимости написания пользовательских реализаций.

Поскольку эти SPI, связанные аннотации и механизм развёртывания CDI являются частью стандартного Jakarta EE, реализации являются полностью переносимыми (в той степени, в которой они не зависят внутренне от API или библиотек, специфичных для платформы) и могут быть развёрнуты путём переноса на любой сервер Jakarta EE.

Защита контейнеров

В Jakarta EE контейнеры компонентов отвечают за обеспечение безопасности приложений. Контейнер обеспечивает два типа безопасности: декларативную и программную.

Использование аннотаций для указания информации о безопасности

Аннотации обеспечивают декларативный стиль программирования и охватывают как декларативную, так и программную концепции безопасности. Пользователи могут указать информацию о безопасности аннотациями в файле класса. GlassFish Server использует эту информацию при развёртывании приложения. Однако не всю информацию о безопасности можно указать в аннотациях. Некоторая информация должна быть указана в дескрипторах развёртывания приложения.

Конкретные аннотации, которые можно использовать для указания информации о безопасности в файле класса Enterprise-бина, описаны в Защита Enterprise-бинов с использованием декларативной безопасности. Глава 51 *Начало работы по защите веб-приложений* описывает, как использовать аннотации для защиты веб-приложений там, где это возможно. Дескрипторы развёртывания описаны только там, где это необходимо.

Для получения дополнительной информации об аннотациях см. Дополнительная информация о безопасности.

Использование дескрипторов развёртывания для декларативной безопасности

Декларативная безопасность может выражать требования безопасности компонента приложения с помощью дескрипторов развёртывания. Поскольку информация дескриптора развёртывания является декларативной, она может быть изменена без необходимости изменения исходного кода. Во время выполнения сервер Jakarta EE считывает дескриптор развёртывания и воздействует на соответствующее приложение, модуль или компонент. Дескрипторы развёртывания должны предоставлять определённую структурную информацию для каждого компонента, если эта информация не была предоставлена в аннотациях или не должна использоваться по умолчанию.

Эта часть руководства не описывает, как создавать дескрипторы развёртывания. Она описывает только элементы дескриптора развёртывания, относящиеся к безопасности. IDE NetBeans предоставляет инструменты для создания и изменения дескрипторов развёртывания.

Различные типы компонентов используют разные форматы или схемы для своих дескрипторов развёртывания. Элементы безопасности дескрипторов развёртывания, обсуждаемые в этом руководстве:

- Веб-компоненты могут использовать дескриптор развёртывания веб-приложения `web.xml`.

  Схема дескрипторов развёртывания веб-компонентов приведена в главе 14 спецификации Jakarta Servlet 5.0, которую можно загрузить с https://jakarta.ee/specifications/servlet/5.0/.

- Компоненты Jakarta Enterprise Beans могут использовать дескриптор развёртывания EJB-компонента с именем `META-INF/ejb-jar.xml`, содержащийся в JAR-файле EJB.

Схема дескрипторов развёртывания Enterprise-бинов приведена в главе 13 спецификации основных контрактов и требований Jakarta Enterprise Beans 4.0, которую можно загрузить с https://jakarta.ee/specifications/enterprise-beans/4.0/.

Использование программной безопасности

Программная безопасность встроена в приложение и используется для принятия решений по доступу. Программная безопасность полезна, когда одной декларативной безопасности недостаточно для реализации модели безопасности приложения. API программной безопасности состоит из методов интерфейса Jakarta Security `SecurityContext` и методов интерфейса `EJBContext` EJB-компонента и интерфейса сервлета `HttpServletRequest`. Эти методы позволяют компонентам принимать решения бизнес-логики на основе роли вызывающего субъекта или удалённого пользователя.

Программная безопасность обсуждается более подробно в следующих разделах:

- Использование программной безопасности в веб-приложениях
- Защита Enterprise-бинов с использованием программной безопасности

Обеспечение безопасности GlassFish Server

В этом руководстве описывается развёртывание в GlassFish Server, который обеспечивает высокую надёжность, совместимость и распределённость компонентов на основе модели безопасности Jakarta EE. GlassFish Server поддерживает модель безопасности Jakarta EE 9. Вы можете настроить GlassFish Server для следующих целей.

- Добавление, удаление или изменение авторизованных пользователей. Для получения дополнительной информации по этой теме см. Работа с областями безопасности, пользователями, группами и ролями.
- Настройка защищённых слушателей HTTP и Internet Inter-Orb Protocol (IIOP).
- Конфигурирование безопасных коннекторов Java Management Extensions (JMX).
- Добавление, удаление или изменение существующих или кастомных областей безопасности (realms).
- Определение интерфейса для подключаемых провайдеров авторизации с использованием Jakarta Authorization. Jakarta Authorization определяет контракты безопасности между GlassFish Server и модулями политик авторизации. В этих контрактах указывается, как поставщики авторизации устанавливаются, настраиваются и используются при принятии решений о доступе.
- Использование подключаемых модулей аудита.
- Настройка механизмов аутентификации. Для всех реализаций совместимых с Jakarta EE 9 веб-контейнеров требуется поддержка Servlet Profile от Jakarta Authentication, который предлагает возможность кастомизации механизма аутентификации, применяемого веб-контейнером от имени одного или нескольких приложений.
- Установка и изменение политики разрешений для приложения.

Работа с хранилищами идентификаторов

Хранилище идентификаторов — это база данных или каталог, в котором содержится идентификационная информация о группе субъектов, работающих с приложением. Хранилище идентификаторов содержит имена вызывающих субъектов, информацию о членстве их в группах и информацию, достаточную для того, чтобы проверить учётные данные вызывающего субъекта. Хранилище идентификаторов также может содержать другую информацию, такую как глобально уникальные идентификаторы или другие атрибуты вызывающих субъектов.

Как указано в API безопасности Jakarta EE, интерфейс `IdentityStore` предоставляет абстракцию хранилища идентификаторов. Реализации интерфейса `IdentityStore` взаимодействуют с хранилищами идентификаторов для аутентификации пользователей и получения информации о группе вызывающих субъектов. Чаще всего реализация интерфейса `IdentityStore` взаимодействует с внешним хранилищем идентификаторов, таким как сервер LDAP, но также может управлять данными учётной записи пользователя.

Интерфейс `IdentityStore` предназначен главным образом для использования `HttpAuthenticationMechanism` (также указан в Jakarta Security), но может использоваться другими реализациями, такими как JASPIC `ServerAuthModule` или встроенными механизмами проверки подлинности контейнера. С помощью `HttpAuthenticationMechanism` и `IdentityStore`, как встроенные, так и кастомные, обеспечивают значительное преимущество перед механизмами BASIC и FORM, определёнными Servlet 5.0 (и предыдущими версиями) и декларативно настраиваемыми с помощью `<login-config>` в `web.xml`, потому что они позволяют приложению управлять хранилищами идентификаторов, с которым будут аутентифицироваться стандартным переносимым способом.

Приложение может предоставить свой собственный `IdentityStore` или использовать встроенные реализации интерфейса LDAP или база данных идентификаторов. Для получения подробной информации об интерфейсах `IdentityStore` и примерах их использования см. Обзор интерфейсов хранилища идентификаторов.

Работа с областями безопасности, пользователями, группами и ролями

Часто требуется защита ресурсов с предоставлением доступа только авторизованным пользователям. См. Характеристики безопасности приложений для ознакомления с понятиями аутентификации, идентификации и авторизации.

В этом разделе обсуждается настройка пользователей таким образом, чтобы их можно было идентифицировать, а также предоставить им доступ к защищённым ресурсам или запретить доступ, если они не авторизованы для доступа к защищённым ресурсам. Для аутентификации пользователя необходимо выполнить следующие основные шаги.

1. Разработчик приложения пишет код для запроса имени пользователя и пароля. Различные методы аутентификации обсуждаются в Указание механизмов аутентификации.
2. Разработчик приложения сообщает, как настроить безопасность для развёрнутого приложения, используя аннотации метаданных или дескриптор развёртывания. Этот шаг обсуждается в Настройка ролей безопасности.
3. Администратор сервера устанавливает авторизованных пользователей и группы в GlassFish Server. Это обсуждается в Управление пользователями и группами в GlassFish Server.
4. Разработчик приложения назначает роли безопасности приложения пользователям, группам и принципалам, определённым в GlassFish Server. Эта тема обсуждается в Назначение ролей пользователям и группам.

По умолчанию имена участников группы назначаются одноименным ролям.

Что такое области безопасности, пользователи, группы и роли?

Область безопасности — это домен политики безопасности, определённый для веб-сервера или сервера приложений. Область содержит список пользователей, которым может быть назначена (или не может быть назначена) группа. Управление пользователями в GlassFish Server обсуждается в разделе Управление пользователями и группами в GlassFish Server.

Приложение часто запрашивает имя пользователя и пароль, прежде чем разрешить доступ к защищённому ресурсу. После ввода имени пользователя и пароля эта информация передаётся на сервер, который либо аутентифицирует пользователя и отправляет защищённый ресурс, либо не аутентифицирует пользователя, и в этом случае доступ к защищённому ресурсу запрещается. Этот тип аутентификации пользователя обсуждается в Указание механизма аутентификации в дескрипторе развёртывания.

В некоторых приложениях авторизованным пользователям назначаются роли. В этом случае роль, назначенная пользователю в приложении, должна быть назначена принципалу или группе, определённым на сервере приложений. Рисунок 50-6 показывает это. Дополнительную информацию о назначении ролей пользователям и группам можно найти в Настройка ролей безопасности.

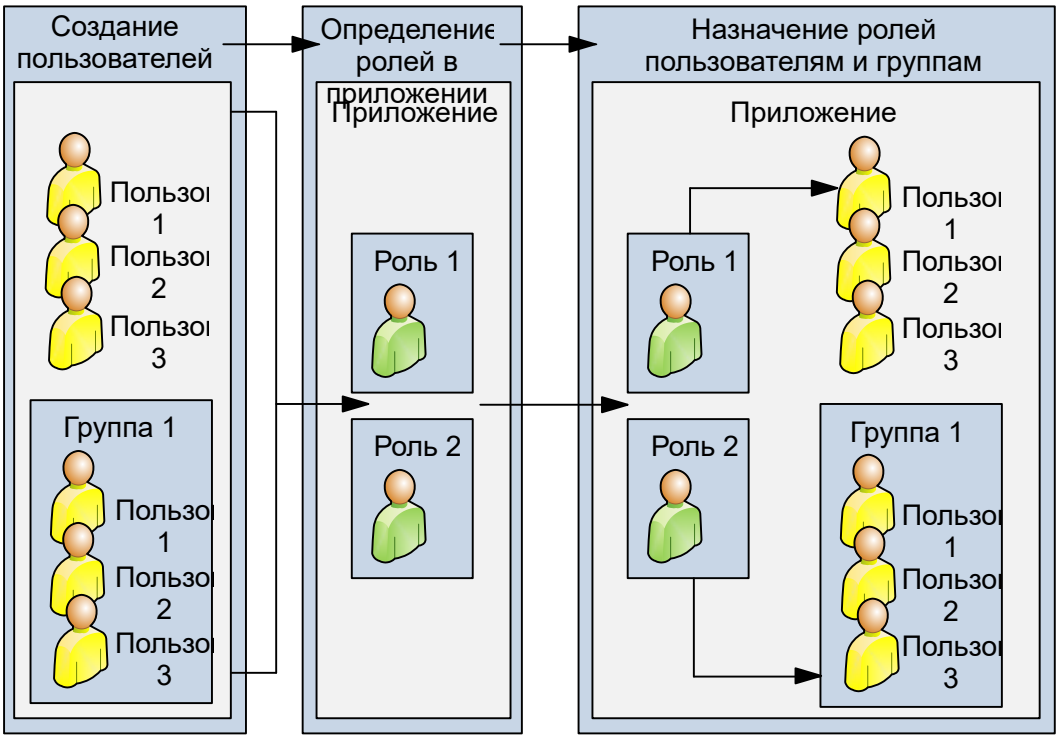

Рисунок 50-6. Назначение ролей пользователям и группам

В следующих разделах представлена дополнительная информация об областях безопасности, пользователях, группах и ролях.

Что такое область безопасности?

Защищённые ресурсы на сервере могут быть сгруппированы в наборы защищённых пространств, каждое со своей собственной схемой аутентификации и/или базой данных авторизации, содержащей коллекцию пользователей и групп. Область безопасности — это полная база данных пользователей и групп, определённых как корректные пользователи одного или нескольких приложений и управляемые одной и той же политикой аутентификации.

Сервис аутентификации сервера Jakarta EE может управлять пользователями нескольких областей. Области `file`, `admin-realm` и `attribute` предустановлены для GlassFish Server.

В области `file` сервер хранит учётные данные пользователя локально в файле с именем `keyfile`. Вы можете использовать Консоль администрирования для управления пользователями в области `file`. При использовании области `file` служба проверки подлинности сервера проверяет подлинность пользователя, проверяя область `file`. Эта область используется для аутентификации всех клиентов, кроме клиентов веб-браузера, которые используют HTTPS и сертификаты.

В области `certificate` сервер хранит учётные данные пользователя в базе данных сертификатов. При использовании области `certificate` сервер использует сертификаты с HTTPS для аутентификации веб-клиентов. Чтобы проверить личность пользователя в области `certificate`, сервис аутентификации проверяет сертификат X.509. Пошаговые инструкции по созданию сертификата этого типа см. в разделе Работа с цифровыми сертификатами. Поле общего имени сертификата X.509 используется в качестве основного имени.

`admin-realm` также является областью `file` и хранит учётные данные пользователя администратора локально в файле с именем `admin-keyfile`. Вы можете использовать Консоль администрирования для управления пользователями в этой области так же, как вы управляете пользователями в области `file`. Для получения дополнительной информации см. Управление пользователями и группами в GlassFish Server.

Что такое пользователь?

Пользователь — это личность или идентификационная информация прикладной программы, определённая в GlassFish Server. В веб-приложении пользователь может связать с этим идентификатором набор ролей, которые дают пользователю доступ ко всем ресурсам, защищённым этими ролями. Пользователи могут быть связаны с группой.

Пользователь Jakarta EE похож на пользователя операционной системы. Как правило, оба типа пользователей представляют людей. Однако эти два типа пользователей не одинаковы. Сервис аутентификации сервера Jakarta EE не знает имени пользователя и пароля, которые вы указываете при входе в операционную систему. Сервис аутентификации сервера Jakarta EE не подключен к механизму безопасности операционной системы. Эти сервисы безопасности управляют пользователями, принадлежащими к разным областям безопасности.

Что такое группа?

Группа — это набор аутентифицированных пользователей, классифицированных по общим признакам, определённым в GlassFish Server. Пользователь Jakarta EE из области `file` может входить в группу GlassFish Server. (Пользователь в области `certificate` не может.) Группа в GlassFish Server — это категория пользователей, классифицированных по общим признакам, таким как должность или профиль клиента. Например, большинство клиентских приложений электронной коммерции могут принадлежать к группе `CUSTOMER`, но крупные клиенты будут принадлежать к группе `PREFERRED`. Распределение пользователей по группам упрощает контроль доступа большого количества пользователей.

Группа в GlassFish Server отличается от области видимости. Группа предназначена для всего GlassFish Server, тогда как роль связана только с конкретным приложением в GlassFish Server.

Что такое роль?

Роль — это абстрактное имя для разрешения доступа к определённому набору ресурсов в приложении. Роль можно сравнить с ключом, который может открыть замок. У многих людей может быть копия ключа. Замок не заботится о том, кто вы, только о том, что у вас есть правильный ключ.

Некоторая другая терминология

Следующая терминология также используется для описания требований безопасности платформы Jakarta EE.

- Принципал — это субъект, который может быть аутентифицирован с помощью протокола аутентификации в сервисе безопасности, развёрнутом на предприятии. Принципал идентифицируется по имени субъекта и аутентифицируется с использованием данных аутентификации.

- Домен политики безопасности, также известный как домен безопасности или область безопасности, является областью, в которой общая политика безопасности определяется и применяется администратором сервиса безопасности.
- Атрибуты безопасности — это набор атрибутов, связанных с каждым принципалом. Атрибуты безопасности имеют много применений: например, доступ к защищённым ресурсам и аудит пользователей. Атрибуты безопасности могут быть связаны с принципалом по протоколу аутентификации.
- Учётные данные — это объект, который содержит или ссылается на атрибуты безопасности, используемые для аутентификации принципала для сервисов Jakarta EE. Принципал получает учётные данные после аутентификации или от другого принципала, который позволяет использовать его учётные данные.

Управление пользователями и группами в GlassFish Server

Выполните следующие шаги для управления пользователями, прежде чем запускать учебные примеры.

Добавление пользователей в GlassFish Server

1. Запустите GlassFish Server, если вы этого ещё не сделали.

   Информация о запуске GlassFish Server доступна в разделе Запуск и остановка GlassFish Server.

2. Запустите Консоль администрирования, если вы этого ещё не сделали.

   Чтобы запустить Консоль администрирования, откройте веб-браузер по URL http://localhost:4848/. Если порт администратора по умолчанию был изменён во время установки, введите корректный номер порта вместо `4848`.

3. В дереве навигации разверните узел Конфигурации, затем разверните узел server-config.
4. Разверните узел безопасности.
5. Разверните узел Realms.
6. Выберите область, в которую вы добавляете пользователей.
   - Выберите область `file`, чтобы добавить пользователей, которым вы хотите предоставить доступ к приложениям, выполняющимся в этой области.

     В качестве примера приложений безопасности выберите область `file`.
   - Выберите `admin-realm`, чтобы добавить пользователей, которых вы хотите дабавить в качестве системных администраторов GlassFish Server.

     Вы не можете добавлять пользователей в область `certificate` из Консоли администрирования. В область `certificate` вы можете добавлять только сертификаты. Информацию о добавлении (импорте) сертификатов в область `certificate` см. в разделе Добавление пользователей в область безопасности certificate.
7. На странице «Изменить область» нажмите «Управление пользователями».
8. На странице «Пользователи File» или «Администраторы» нажмите «Создать», чтобы добавить нового пользователя в область.
9. На странице «Новый пользователь области File» введите значения в поля «ID пользователя», «Список групп», «Новый пароль» и «Подтверждение нового пароля».

   Для области администратора поле списка групп доступно только для чтения, а имя группы — `asadmin`. Перезапустите GlassFish Server и консоль администрирования после добавления пользователя в область администратора.

Для получения дополнительной информации об этих свойствах см. Работа с сферами, пользователями, группами и ролями.

Для примеров приложений безопасности укажите пользователя с любым понравившимся именем и паролем, но убедитесь, что пользователь добавлен в группу `TutorialUser`. Имя пользователя и пароль чувствительны к регистру. Сохраните запись имени пользователя и пароля для работы с примерами, приведёнными далее в этом руководстве.

10. Нажмите OK, чтобы добавить этого пользователя в область, или нажмите Отмена, чтобы выйти без сохранения.

Настройка ролей безопасности

При разработке Enterprise-бина или веб-компонента всегда нужно думать о том, какие пользователи будут иметь доступ к компоненту. Например, веб-приложение для отдела кадров может иметь разные URL запроса для людей с ролями `DEPT_ADMIN` и `DIRECTOR`. Роль `DEPT-ADMIN` может позволять просматривать данные о сотрудниках, но роль `DIRECTOR` позволяет изменять данные о сотрудниках, включая данные о зарплатах. Каждая из этих ролей безопасности представляет собой абстрактную логическую группу пользователей, которая определяется человеком, компонующим приложение. При развёртывании приложения роли назначаются пользователям и группам, как показано на рис. 50-6.

Для компонентов Jakarta EE роли безопасности определяются аннотациями `@DeclareRoles` и `@RolesAllowed`.

Ниже приведён пример приложения, в котором роль `DEPT-ADMIN` авторизована для методов, которые проверяют данные заработной платы сотрудников, а роль `DIRECTOR` авторизована для методов, которые изменяют данные заработной платы сотрудников.

Enterprise-бин будет аннотирован, как показано в следующем коде:

JAVA

```
import jakarta.annotation.security.DeclareRoles;
import jakarta.annotation.security.RolesAllowed;
...
@DeclareRoles({"DEPT-ADMIN", "DIRECTOR"})
@Stateless public class PayrollBean implements Payroll {
    @Resource SessionContext ctx;


    @RolesAllowed("DEPT-ADMIN")
    public void reviewEmployeeInfo(EmplInfo info) {

        oldInfo = ... read from database;

        // ...
    }

    @RolesAllowed("DIRECTOR")
    public void updateEmployeeInfo(EmplInfo info) {

        newInfo = ... update database;

        // ...
    }
    ...
}
```

Для сервлета вы можете использовать аннотацию `@HttpConstraint` в аннотации `@ServletSecurity`, чтобы указать роли, которым разрешён доступ к сервлету. Например, сервлет может быть аннотирован следующим образом:

```java
@WebServlet(name = "PayrollServlet", urlPatterns = {"/payroll"})
@ServletSecurity(
@HttpConstraint(transportGuarantee = TransportGuarantee.CONFIDENTIAL,
    rolesAllowed = {"DEPT-ADMIN", "DIRECTOR"}))
public class GreetingServlet extends HttpServlet { ... }
```

Эти аннотации более подробно обсуждаются в Указание безопасности для базовой аутентификации с использованием аннотаций и Защита Enterprise-бинов с использованием декларативной безопасности.

После того, как пользователи предоставили свои регистрационные данные и в приложении было объявлено, какие роли авторизованы для доступа к защищённым частям приложения, следующим шагом будет назначение роли безопасности пользователю или принципалу.

Назначение ролей пользователям и группам

При разработке приложения Jakarta EE не нужно знать, какие категории пользователей были определены в области безопасности, в которой будет запускаться приложение. В платформе Jakarta EE архитектура безопасности предоставляет механизм назначения ролей, определённых в приложении, пользователям или группам, определённым в области безопасности среды выполнения.

Имена ролей, используемые в приложении, часто совпадают с именами групп, определённых в GlassFish Server. Jakarta Security требует, чтобы имена участников групп по умолчанию сопоставлялись с одноимёнными ролями. Соответственно, параметр **Назначение по умолчанию для ролей** включён по умолчанию на странице безопасности консоли администрирования GlassFish Server. Все учебные примеры безопасности рассчитывают, что эта настройка включена. Если этот параметр включён, если имя группы, определённое в GlassFish Server, совпадает с именем роли, определённым в приложении, нет необходимости использовать для назначения дескриптор развёртывания среды выполнения. Сервер приложений будет неявно выполнять это назначение для групп и ролей с совпадающими именами.

Если имена ролей, используемые в приложении, не совпадают с именами групп, определёнными на сервере, используйте дескриптор развёртывания среды выполнения, чтобы указать назначение. В следующем примере показано, как сделать назначение в файле `glassfish-web.xml`, который используется для веб-приложений:

```xml
<glassfish-web-app>
    ...
    <security-role-mapping>
        <role-name>Mascot</role-name>
        <principal-name>Duke</principal-name>
    </security-role-mapping>

    <security-role-mapping>
        <role-name>Admin</role-name>
        <group-name>Director</group-name>
    </security-role-mapping>
    ...
</glassfish-web-app>
```

Роль может быть назначена определённым принципалам и/или группам. Имена участников или групп должны быть корректными участниками или группами в текущей области безопасности по умолчанию или в области, указанной в элементе `login-config`. В этом примере роль `Mascot`, используемая в приложении, назначается принципалу `Duke`, который существует на сервере приложений. Назначение роли конкретному принципалу полезно, когда лицо, занимающее эту роль, может измениться. Для этого приложения нужно изменить только дескриптор развёртывания во время выполнения, а не искать и заменять во всём приложении ссылки на этого принципала.

Также в этом примере роль `Admin` назначена группе `Director`. Это полезно, потому что группа людей, которым разрешён доступ к административным данным на уровне директора, должна поддерживаться только в GlassFish Server. Разработчику приложения не нужно знать, кто эти люди, а нужно лишь определить группу людей, которым будет предоставлен доступ к информации.

`role-name` должно соответствовать `role-name` в элементе `security-role` соответствующего дескриптора развёртывания или имени роли, определённой в аннотации `@DeclareRoles`.

Установление безопасного соединения с использованием SSL

Технология Secure Sockets Layer (SSL) — это безопасность, которая реализована на транспортном уровне (для получения дополнительной информации о безопасности транспортного уровня см. Безопасность транспортного уровня). SSL позволяет веб-браузерам и веб-серверам общаться через безопасное соединение. В этом безопасном соединении данные шифруются перед отправкой, а затем дешифруются при получении перед обработкой. И браузер, и сервер шифруют весь трафик перед отправкой каких-либо данных.

SSL отвечает следующим важным соображениям безопасности.

- Аутентификация: при первоначальной попытке установить связь с веб-сервером по безопасному соединению этот сервер предоставит веб-браузеру набор учётных данных в виде сертификата сервера (также называемого сертификатом открытого ключа). Целью сертификата является подтверждение того, что сайт является тем, за который он себя выдаёт. В некоторых случаях сервер может запросить сертификат, подтверждающий, что клиент является тем, за кого он себя выдаёт. Этот механизм известен как аутентификация клиента по цифровому сертификату.
- Конфиденциальность: при передаче данных между клиентом и сервером по сети третьи лица имеют возможность просматривать и перехватывать эти данные. Ответы SSL зашифрованы, так что данные не могут быть расшифрованы третьей стороной, и данные остаются конфиденциальными.
- Целостность: при передаче данных между клиентом и сервером по сети третьи лица могут просматривать и перехватывать эти данные. SSL гарантирует, что данные не будут изменены этой третьей стороной при передаче.

Протокол SSL разработан так, чтобы быть максимально эффективным и безопасным. Однако шифрование и дешифрование — вычислительно дорогостоящие процессы. Не является строго обязательным запускать всё веб-приложение через SSL, и разработчик обычно решает, какие страницы требуют безопасного соединения, а какие — нет. Страницы, для которых может потребоваться безопасное соединение, включают страницы для входа в систему, персональные данные, проверки корзины покупок или передачу информации о кредитной карте. Любая страница приложения может быть запрошена через защищённый сокет простым добавлением в префикс URL `https:` вместо `http:`. Любые страницы, для которых абсолютно необходимо безопасное соединение, должны проверить тип протокола, связанный с запросом страницы, и предпринять соответствующие действия, если `https:` не указан.

Использование виртуальных хостов на основе доменных имён с защищённым соединением может быть проблематичным. Это ограничение дизайна самого протокола SSL. Установление соединения SSL, при котором клиентский браузер принимает сертификат сервера, должно произойти до обращения к HTTP-запросу. В результате информация запроса, содержащая имя виртуального хоста, не может быть определена до аутентификации, и поэтому невозможно назначить несколько сертификатов одному IP-адресу. Если все виртуальные хосты на одном IP-адресе должны проходить аутентификацию с одним и тем же сертификатом, добавление нескольких виртуальных хостов не должно мешать нормальной работе SSL на сервере. Имейте в виду, однако, что большинство клиентских браузеров сравнивают доменное имя сервера с доменным именем, указанным в сертификате, если таковые имеются. Это применимо прежде всего к официальным

сертификатам, подписанным центром сертификации (CA). Если доменные имена не совпадают, эти браузеры будут отображать предупреждение для клиента. Как правило, только виртуальные хосты на основе IP-адресов обычно используются с SSL в производственной среде.

Верификация и настройка поддержки SSL

Как правило, вы должны решить следующие проблемы, чтобы включить SSL для сервера.

- В дескрипторе развёртывания сервера должен быть элемент `Connector` для SSL.
- Должно быть указано корректное хранилище ключей и файлов сертификатов.
- Расположение файла хранилища ключей и его пароль должны быть указаны в дескрипторе развёртывания сервера.

Коннектор SSL HTTPS уже встроен в GlassFish Server.

Для тестирования и проверки корректности настройки SSL загрузите страницу входа по умолчанию с URL, который подключается к порту, определённому в дескрипторе развёртывания сервера:

```
https://localhost:8181/
```

`https` в этом URL указывает, что браузер должен использовать протокол SSL. `localhost` предполагает, что пример запускается на локальном компьютере. `8181` — это безопасный порт, который был указан там, где был создан коннектор SSL. Если используется другой сервер или порт, нужно изменить это значение соответствующим образом.

При первой загрузке этого приложения открывается диалоговое окно «Новый сертификат сайта или предупреждение системы безопасности». Нажмите «Далее», чтобы перейти через ряд диалоговых окон, и нажмите «Готово», когда вы дойдёте до последнего диалогового окна. Сертификаты появятся только в первый раз. После принятия сертификатов, все последующие посещения этого сайта будут предполагать, что вы всё ещё доверяете контенту.

Дополнительная информация о безопасности

Для получения дополнительной информации о безопасности в приложениях Jakarta EE см.

- Спецификация Jakarta EE 9:
  https://jakarta.ee/specifications/platform/9/
- Jakarta Security 2.0:
  https://jakarta.ee/specifications/security/2.0/
- Jakarta Authentication 2.0:
  https://jakarta.ee/specifications/authentication/2.0/
- Спецификация Jakarta Enterprise Beans 4.0:
  https://jakarta.ee/specifications/enterprise-beans/4.0/
- Спецификация Jakarta Enterprise Web Services 2.0:
  https://jakarta.ee/specifications/enterprise-ws/2.0/
- Сведения о безопасности Java SE:
  https://docs.oracle.com/javase/8/docs/technotes/guides/security/
- Спецификация Jakarta Servlet 5.0:
  https://jakarta.ee/specifications/servlet/5.0/

- Спецификация Jakarta Authorization 2.0: https://jakarta.ee/specifications/authorization/2.0/

Глава 51. Начало работы по защите веб-приложений

В этой главе более подробно описываются способы реализации безопасности для веб-приложений Jakarta EE, которые в общих чертах обсуждались в Защита контейнеров. Детали и примеры в этой главе исследуют эти сервисы безопасности, связанные с веб-компонентами.

Обзор безопасности веб-приложений

Доступ к веб-приложению осуществляется через веб-браузер по сети, например Интернет или интранет. Как обсуждалось в Распределённые многослойные приложения, платформа Jakarta EE использует модель распределённых многослойных приложений, а веб-приложения выполняются на веб-уровне.

Веб-приложения содержат ресурсы, к которым могут обращаться многие пользователи. Эти ресурсы часто проходят по незащищённым открытым сетям, таким как Интернет. В такой среде для значительного числа веб-приложений требуется определённый тип безопасности.

Защита приложений и их клиентов на уровне бизнес-логики и уровне ИС обсуждается в главе 52 *Начало работы по защите EJB-приложений*.

В платформе Jakarta EE веб-компоненты предоставляют возможности динамического расширения веб-сервера. Веб-компоненты могут быть сервлетами Jakarta или страницами Jakarta Faces.

Определённые аспекты безопасности веб-приложения можно настроить, когда приложение развёрнуто в веб-контейнере. Аннотации и/или дескрипторы развёртывания используются для передачи установщику информации о безопасности и других аспектах приложения. Указание этой информации в аннотациях или в дескрипторе развёртывания помогает установщику настроить соответствующую политику безопасности для веб-приложения. Любые значения, явно указанные в дескрипторе развёртывания, переопределяют любые значения, указанные в аннотациях.

Безопасность для веб-приложений Jakarta EE может быть реализована следующими способами.

- Декларативная безопасность может быть реализована с использованием аннотаций или дескриптора развёртывания приложения. Дополнительные сведения см. в разделе Обзор Jakarta Security .

  Декларативная безопасность для веб-приложений описана в Защита веб-приложений.

- Программная безопасность встроена в приложение и может использоваться для принятия решений по вопросам безопасности, когда одной декларативной безопасности недостаточно для реализации модели безопасности приложения. Одной декларативной безопасности может быть недостаточно, если в середине приложения требуется условный вход в систему в конкретном рабочем процессе, а не во всех случаях. Дополнительные сведения см. в разделе Обзор Jakarta Security .

  Servlet 5.0 предоставляет методы `authenticate`, `login` и `logout` интерфейса `HTTPServletRequest`. С добавлением методов `authenticate`, `login` и `logout` к спецификации сервлета дескриптор развёртывания приложения больше не требуется для веб-приложений, но всё ещё может использоваться для дальнейшего определения требований безопасности, выходящих за рамки базовых значений по умолчанию.

  Программная безопасность обсуждается в Использование программной безопасности в веб-приложениях.

- Защита сообщений работает с веб-сервисами и включает функции безопасности, такие как цифровые подписи и шифрование, в заголовок сообщения SOAP, работая на уровне приложений, обеспечивая сквозную безопасность. Безопасность сообщений не является компонентом Jakarta EE и упоминается здесь только в ознакомительных целях.

Некоторые материалы этой главы основаны на материалах, представленных ранее в этом руководстве. В частности, в этой главе предполагается, что вы знакомы с информацией из следующих глав:

- Глава 6 *Начало работы с веб-приложениями*
- Раздел 1.8.3 «Технология Jakarta Faces»
- Глава 18 *Технология Jakarta Servlet*
- Глава 50 *Введение в безопасность на платформе Jakarta EE*

Защита веб-приложений

Веб-приложения создаются разработчиками приложений, которые передают, продают или иным образом отдают приложение для установки в среде выполнения.

Обзор безопасности веб-приложений

Разработчики приложений сообщают, как настроить безопасность для развёрнутого приложения, используя аннотации или дескрипторы развёртывания. Эта информация передаётся установщику, который использует её для задания разрешений методов для ролей безопасности, настройки проверки подлинности пользователя и настройки соответствующего транспортного механизма. Если разработчик приложения не определяет требования безопасности, установщик должен будет самостоятельно установить требования безопасности.

Некоторые элементы, необходимые для безопасности веб-приложения, нельзя указать в качестве аннотаций для всех типов веб-приложений. В этой главе объясняется, как защитить веб-приложения аннотациями, где это возможно. В нём объясняется, как использовать дескрипторы развёртывания, в которых нельзя использовать аннотации.

Указание ограничений безопасности

Ограничение безопасности используется для определения прав доступа к коллекции ресурсов путём их сопоставления с URL.

Если веб-приложение использует сервлеты, можно выразить информацию об ограничениях безопасности в аннотациях. В частности, используются аннотации `@HttpConstraint` и, необязательно, аннотации `@HttpMethodConstraint` в аннотации `@ServletSecurity`, чтобы указать ограничение безопасности.

Однако, если веб-приложение не использует сервлет, нужно указать элемент `security-constraint` в файле дескриптора развёртывания. Механизм аутентификации не может быть выражен аннотациями, поэтому, если используется любой метод аутентификации, кроме `BASIC` (по умолчанию), требуется дескриптор развёртывания.

Следующие подэлементы могут быть частью `security-constraint`.

- Коллекция веб-ресурсов (`web-resource-collection`): список шаблонов URL (часть URL после имени хоста и порта, который хотите защитить) и операций HTTP (методы в файлах, которые соответствовать шаблону URL, который хотите защитить), который описывает набор ресурсов, которые должны быть защищены. Коллекции веб-ресурсов обсуждаются в Указание коллекции веб-ресурсов.
- Ограничение авторизации (`auth-constraint`): указывает, должна ли использоваться аутентификация, и называет роли, авторизованные для выполнения запросов. Для получения дополнительной информации об ограничениях авторизации см. Указание ограничения авторизации.
- Защита пользовательских данных (`user-data-constraint`): указывает, как данные защищены при передаче между клиентом и сервером. Защита пользовательских данных обсуждается в Указание безопасного соединения.

Указание коллекции веб-ресурсов

Коллекция веб-ресурсов состоит из следующих подэлементов.

- `web-resource-name` — это имя, которое вы используете для этого ресурса. Его использование не является обязательным.
- `url-pattern` используется для перечисления URI запросов, которые нужно защитить. Многие приложения имеют как незащищённые, так и защищённые ресурсы. Чтобы обеспечить неограниченный доступ к ресурсу, не настраивайте ограничение безопасности для этого конкретного URI запроса.

  URI запроса является частью URL после имени хоста и порта. Для примера предположим, что у вас есть сайт электронной коммерции с каталогом, и вы хотите, чтобы любой мог иметь доступ на просмотр, но область корзины покупок доступна только для клиентов. Вы можете настроить пути для своего веб-приложения, чтобы шаблон `/cart/*` был защищён, но всё остальное оставалось незащищённым. Предполагая, что приложение установлено по контекстному пути `/myapp` , верно следующее.

  - `http://localhost:8080/myapp/index.xhtml` не защищён.
  - `http://localhost:8080/myapp/cart/index.xhtml` защищён.

    Пользователю будет предложено войти в первый раз, когда он/она получит доступ к ресурсу в подкаталоге `cart/` .
- `http-method` используется, чтобы указать, какие методы должны быть защищены, а `http-method-omission` — исключены из защиты. HTTP-метод защищён `web-resource-collection` при любом из следующих обстоятельств:
  - Если в коллекции не указаны методы HTTP (это означает, что все они защищены)
  - Если коллекция специально указывает метод HTTP в подэлементе `http-method`
  - Если коллекция содержит один или несколько элементов `http-method-omission` , ни один из которых не указывает метод HTTP

Указание ограничений авторизации

Ограничение авторизации (`auth-constraint`) содержит элемент `role-name` . Вы можете использовать столько элементов `role-name` , сколько требуется.

Ограничение авторизации устанавливает требование для аутентификации и указывает роли, авторизованные для доступа к шаблонам URL и методам HTTP, объявленным этим ограничением безопасности. Если нет ограничений авторизации, контейнер должен принять запрос, не требуя аутентификации пользователя. Если есть ограничение авторизации, но в нём не указаны роли, контейнер не разрешит доступ к запросам ни при каких обстоятельствах. Каждое имя роли, указанное здесь, должно либо соответствовать имени роли одного из элементов `security-role` , определённых для этого веб-приложения, либо быть специально зарезервированным именем роли `*` , которое указывает все роли веб-приложения. Имена ролей чувствительны к регистру. Роли, определённые для приложения, должны быть назначены пользователям и группам, определённым на сервере, за исключением случаев, когда используется назначение принципалу и роли по умолчанию.

Для получения дополнительной информации о ролях безопасности см. Объявление ролей безопасности. Для получения информации о назначении ролей безопасности см. Назначение ролей пользователям и группам.

Для сервлета аннотации `@HttpConstraint` и `@HttpMethodConstraint` принимают элемент `roleAllowed` , который определяет авторизованные роли.

Указание безопасного соединения

Защита пользовательских данных (`user-data-constraint` в дескрипторе развёртывания) содержит подэлемент `transport-guarantee` . Защита пользовательских данных может использоваться для удовлетворения требования, чтобы защищённое соединение транспортного уровня, такое как HTTPS, использовалось для всех защищаемых шаблонов URL и методов HTTP, указанных в ограничении безопасности. Варианты транспортных гарантий: `CONFIDENTIAL` , `INTEGRAL` и `NONE` . Если ограничением безопасности указан `CONFIDENTIAL` или `INTEGRAL` , это обычно означает, что использование SSL является обязательным и применяется ко всем запросам, которые соответствуют шаблонам URL в коллекции веб-ресурсов, а не только в диалоговом окне входа в систему.

Уровень требуемой защиты определяется значением транспортной гарантии следующим образом.

- `CONFIDENTIAL` , когда приложение требует передачи данных, чтобы другие объекты не могли наблюдать за содержимым передачи.
- `INTEGRAL` , когда приложение требует, чтобы данные передавались между клиентом и сервером таким образом, чтобы их нельзя было изменить при передаче.
- `NONE` , когда контейнер должен принимать запросы для любого соединения, включая незащищённое.

На практике серверы Jakarta EE обрабатывают значения транспортных гарантий `CONFIDENTIAL` и `INTEGRAL` одинаково.

Защиту пользовательских данных удобно использовать в сочетании с аутентификациями пользователя: базовой и на основе форм. Если для метода аутентификации входа установлено значение `BASIC` или `FORM` , пароли не защищены, что означает, что пароли, отправленные между клиентом и сервером в незащищённой сессии, могут быть просмотрены и перехвачены третьими лицами. Использование защиты пользовательских данных с механизмом аутентификации пользователя может облегчить эту проблему. Настройка механизма аутентификации пользователя описана в Указание механизма аутентификации в дескрипторе развёртывания.

Чтобы гарантировать передачу данных по безопасному соединению, убедитесь, что поддержка SSL настроена для вашего сервера. Для GlassFish Server поддержка SSL уже настроена.

После переключения на SSL для сессии, ни в коем случае нельзя принимать не-SSL запросы на оставшуюся часть этой сессии. Например, сайт покупок может не использовать SSL до страницы оформления заказа, а затем он может переключиться на использование SSL, чтобы принять номер вашей карты. После переключения на SSL вы должны прекратить слушать не-SSL запросы для этой сессии. Причиной такой практики является то, что сам идентификатор сессии не был зашифрован в предыдущих сообщениях. Это не так уж плохо, когда вы только делаете покупки, но после того, как информация о кредитной карте будет сохранена в сессии, нельзя допустить, чтобы кто-либо использовал эту информацию для подделки транзакции покупки с вашей кредитной карты. Эта практика может быть легко реализована с помощью фильтра.

Указание ограничений безопасности для ресурсов

Вы можете создать ограничения безопасности для ресурсов в вашем приложении. Например, вы можете разрешить пользователям с ролью `PARTNER` полный доступ ко всем ресурсам с URL-шаблоном `/acme/wholesale/*` и разрешить пользователям с ролью `CLIENT` полный доступ ко всем ресурсам по URL-шаблону `/acme/retail/*` . Это рекомендуемый способ защиты ресурсов, если вы хотите защитить только некоторые методы HTTP, оставляя другие методы HTTP незащищёнными. Пример дескриптора развёртывания, который продемонстрировал бы эту функциональность, следующий:

XML

```xml
<!-- SECURITY CONSTRAINT #1 -->
<security-constraint>
    <web-resource-collection>
        <web-resource-name>wholesale</web-resource-name>
        <url-pattern>/acme/wholesale/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>PARTNER</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>

<!-- SECURITY CONSTRAINT #2 -->
<security-constraint>
    <web-resource-collection>
        <web-resource-name>retail</web-resource-name>
        <url-pattern>/acme/retail/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>CLIENT</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

Указание механизмов аутентификации

В этом разделе описываются встроенные механизмы аутентификации, определённые в спецификации Servlet.

> Альтернативный способ выполнить аутентификацию пользователя, включая аутентификации BASIC и FORM, — использовать `HttpAuthenticationMechanism`, указанный Jakarta Security и задокументированный в Using Jakarta Security.

Механизм аутентификации пользователя определяет:

- Способ, которым пользователь получает доступ к веб-контенту
- При базовой аутентификации область, в которой пользователь будет аутентифицирован
- С аутентификацией на основе форм, дополнительные атрибуты

Когда указан механизм аутентификации, пользователь должен пройти аутентификацию, прежде чем ему будет предоставлен доступ к любому ресурсу, который защищён ограничением безопасности. Может быть несколько ограничений безопасности, применяемых к нескольким ресурсам, но один и тот же метод аутентификации будет применяться ко всем защищённым ресурсам в приложении.

Прежде чем вы сможете аутентифицировать пользователя, у вас должна быть база данных имён пользователей, паролей и ролей, настроенных на вашем веб-сервере или сервере приложений. Информацию о настройке базы данных пользователей смотрите в разделе Управление пользователями и группами в GlassFish Server.

Платформа Jakarta EE поддерживает следующие механизмы аутентификации:

- Базовая аутентификация
- Аутентификация на основе форм

- Дайджест аутентификации
- Аутентификация клиента по цифровому сертификату
- Взаимная аутентификация

В этом разделе рассматриваются базовая, основанная на формах и дайджест аутентификации. Клиентская и взаимная аутентификация обсуждаются в главе 54 *Jakarta EE Security: дополнительные темы*.

Базовая аутентификация HTTP и аутентификация на основе форм не очень безопасные механизмы аутентификации. Обычная проверка подлинности отправляет имена пользователей и пароли через Интернет в виде текста в кодировке Base64. Проверка подлинности на основе форм отправляет эти данные в виде простого текста. В обоих случаях целевой сервер не аутентифицирован. Следовательно, эти формы аутентификации делают данные пользователя открытыми и уязвимыми. Если кто-то перехватит передачу, имя пользователя и пароль могут быть легко декодированы.

Тем не менее, когда безопасный транспортный механизм, такой как SSL, или безопасность на сетевом уровне, таком как протокол Internet Protocol Security (IPsec) или стратегии виртуальной частной сети (VPN), используется в сочетании с базовой или основанной на форме аутентификацией, некоторые из этих проблем могут быть смягчены. Чтобы указать безопасный транспортный механизм, используйте элементы, описанные в Указание безопасного соединения.

Базовая аутентификация HTTP

Указание базовой аутентификации HTTP требует от сервера запрашивал имя пользователя и пароль у веб-клиента и проверять действительность их, сравнивая их с базой данных авторизованных пользователей в указанной области или области по умолчанию.

По умолчанию используется базовая аутентификация, если не указан механизм проверки подлинности.

При использовании базовой аутентификации выполняются следующие действия.

1. Клиент запрашивает доступ к защищённому ресурсу.
2. Веб-сервер возвращает диалоговое окно, которое запрашивает имя пользователя и пароль.
3. Клиент отправляет имя пользователя и пароль на сервер.
4. Сервер аутентифицирует пользователя в указанной области и, в случае успеха, возвращает запрошенный ресурс.

На рисунке 51-1 показано, что происходит при выборе базовой аутентификации HTTP.

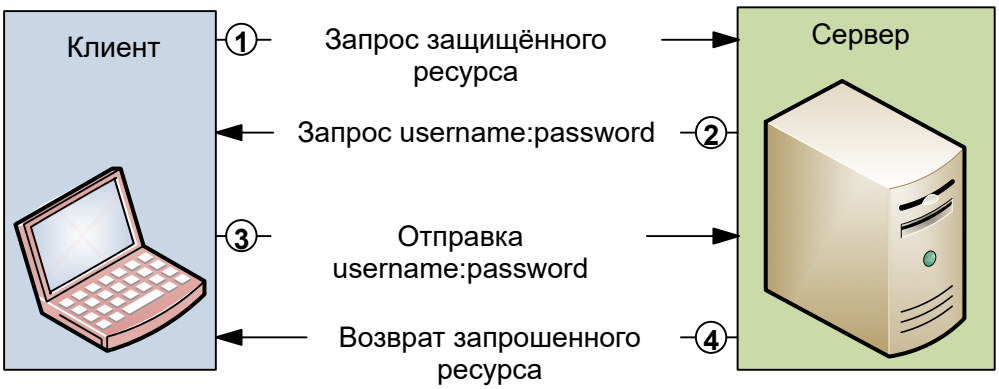

Рисунок 51-1 Базовая аутентификация HTTP

Аутентификация на основе форм

Аутентификация на основе форм позволяет разработчику контролировать внешний вид экранов аутентификации при входе, настраивая экран входа и страницы ошибок, которые HTTP-браузер предоставляет конечному пользователю. При объявлении проверки подлинности на основе форм выполняются следующие действия.

1. Клиент запрашивает доступ к защищённому ресурсу.
2. Если клиент не прошёл проверку подлинности, сервер перенаправляет клиента на страницу входа.
3. Клиент отправляет форму входа на сервер.
4. Сервер пытается аутентифицировать пользователя.
    - Если аутентификация успешна, субъект аутентифицированного пользователя проверяется, чтобы убедиться, что имеет роль, которая авторизована для доступа к ресурсу. Если пользователь авторизован, сервер перенаправляет клиента на ресурс, используя сохранённый путь URL.
    - Если аутентификация не удалась, клиент перенаправляется на страницу ошибки.

На рисунке 51-2 показано, что происходит при выборе аутентификации на основе форм.

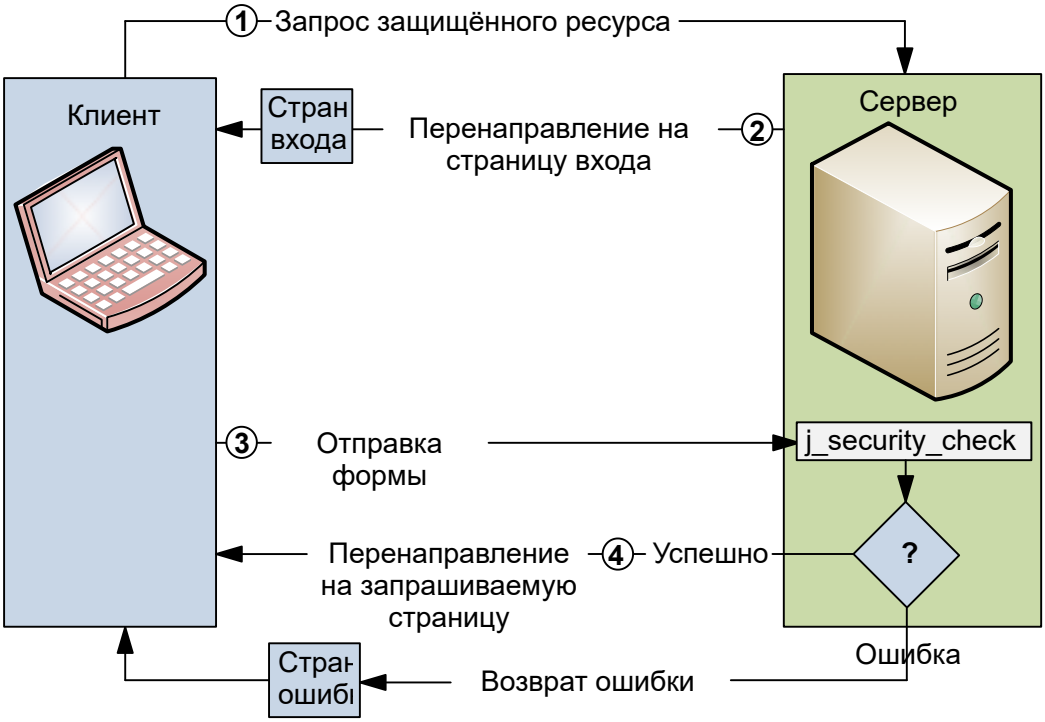

Рисунок 51-2 Аутентификация на основе форм

Раздел Пример hello1-formauth: аутентификация на основе форм с приложением Jakarta Faces — это пример приложения, использующего аутентификацию на основе форм.

При выполнении входа на основе формы обязательно поддерживайте сессии с использованием файлов cookie или данных сессии SSL.

Чтобы аутентификация прошла надлежащим образом, атрибут action формы входа в систему всегда должен быть установлен в `j_security_check` . Это ограничение сделано для того, чтобы форма входа в систему работала независимо от того, для какого ресурса она предназначена, и чтобы сервер не указывал поле действия исходящей формы. Следующий фрагмент кода показывает, какой вид форма должна иметь на HTML-странице:

```xml
<form method="POST" action="j_security_check">
<input type="text" name="j_username">
<input type="password" name="j_password">
</form>
```

Дайджест аутентификация

Как и базовая аутентификация, дайджест аутентификация аутентифицирует пользователя на основе имени пользователя и пароля. Однако, в отличие от базовой аутентификации, дайджест аутентификация не отправляет пароли пользователей по сети. Вместо этого клиент отправляет односторонний криптографический хэш пароля и дополнительные данные. Хотя пароли не передаются по сети, дайджест аутентификация требует, чтобы эквиваленты паролей в открытом виде были доступны для контейнера аутентификации, чтобы он мог валидировать подлинность путём вычисления ожидаемого дайджеста.

Указание механизма аутентификации в дескрипторе развёртывания

Чтобы указать механизм аутентификации, используется элемент `login-config`. Он может содержать следующие подэлементы.

- Подэлемент `auth-method` настраивает механизм аутентификации для веб-приложения. Содержимое элемента должно быть `NONE`, `BASIC`, `DIGEST`, `FORM` или `CLIENT-CERT`.
- Подэлемент `realm-name` указывает имя области, которое следует использовать при выборе базовой аутентификации для веб-приложения.
- Подэлемент `form-login-config` определяет страницы входа и ошибки, которые следует использовать, если указан вход на основе формы.

> Другой способ указать аутентификацию на основе форм — использовать методы `authenticate`, `login` и `logout` для `HttpServletRequest`, как обсуждалось в Программная аутентификация пользователей.

При попытке получить доступ к веб-ресурсу, который защищён элементом `security-constraint`, веб-контейнер активирует механизм аутентификации, настроенный для этого ресурса. Механизм аутентификации определяет, как пользователю будет предложено войти в систему. Если элемент `login-config` присутствует, а элемент `auth-method` содержит значение, отличное от `NONE`, пользователь должен пройти аутентификацию для доступа к ресурсу. Если не указан механизм аутентификации, аутентификация пользователя не требуется.

В следующем примере показано, как объявить аутентификацию на основе форм в дескрипторе развёртывания:

```xml
<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>file</realm-name>
    <form-login-config>
        <form-login-page>/login.xhtml</form-login-page>
        <form-error-page>/error.xhtml</form-error-page>
    </form-login-config>
</login-config>
```

Расположение страницы входа в систему и ошибок указывается относительно расположения дескриптора развёртывания. Примеры страниц входа в систему и ошибок показаны в Создание формы входа в систему и страницы ошибок.

В следующем примере показано, как объявить дайджест аутентификацию в дескрипторе развёртывания:

```xml
<login-config>
    <auth-method>DIGEST</auth-method>
</login-config>
```

Объявление ролей безопасности

Вы можете объявить названия ролей безопасности, используемые в веб-приложениях, используя элемент `security-role` дескриптора развёртывания. Используйте этот элемент, чтобы получить список всех ролей безопасности, на которые вы ссылаетесь в своём приложении.

В следующем фрагменте дескриптора развёртывания объявляются роли, которые будут использоваться в приложении с использованием элемента `security-role`, и указывается, какая из этих ролей авторизована для доступа к защищённым ресурсам с использованием элемента `auth-constraint`:

```xml
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Protected Area</web-resource-name>
        <url-pattern>/security/protected/*</url-pattern>
        <http-method>PUT</http-method>
        <http-method>DELETE</http-method>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>manager</role-name>
    </auth-constraint>
</security-constraint>

 <!-- Security roles used by this web application -->
<security-role>
    <role-name>manager</role-name>
</security-role>
<security-role>
    <role-name>employee</role-name>
</security-role>
```

В этом примере элемент `security-role` перечисляет все роли безопасности, используемые в приложении: `manager` и `employee`. Это позволяет установщику назначить все роли, определённые в приложении, пользователям и группам, определённым в GlassFish Server.

Элемент `auth-constraint` определяет роль `manager`, которая может обращаться к HTTP методам PUT, DELETE, GET и POST, расположенным в каталоге, указанном в элементе `url-pattern` (`/security/protected/*`).

В этой ситуации нельзя использовать аннотацию `@ServletSecurity`, поскольку её ограничения применяются ко всем шаблонам URL, указанным в аннотации `@WebServlet`.

Использование программной безопасности в веб-приложениях

Программная безопасность используется в приложениях, когда одной декларативной безопасности недостаточно для реализации модели безопасности приложения.

Программная аутентификация пользователей

Вы можете использовать интерфейсы `SecurityContext` и `HttpServletRequest` для программной аутентификации пользователей веб-приложения.

SecurityContext

Интерфейс `SecurityContext`, как указано в спецификации Jakarta Security, определяет следующий метод для программного запуска процесса аутентификации:

- `authenticate()` позволяет приложению сигнализировать контейнеру, что оно должно начать процесс аутентификации вызывающего субъекта.

Программный запуск означает, что контейнер отвечает, как если бы вызывающий субъект пытался получить доступ к защищённому ресурсу. Это заставляет контейнер вызывать механизм аутентификации, настроенный для приложения. Если настроен механизм аутентификации `HttpAuthenticationMechanism`, то аргумент `AuthenticationParameters` имеет смысл и доступны расширенные возможности `HttpAuthenticationMechanism`. Если нет, поведение и результат будут такими, как если бы были вызваны `HttpServletRequest.authenticate()`.

HttpServletRequest

Интерфейс `HttpServletRequest` определяет следующие методы, которые позволяют программно аутентифицировать пользователей веб-приложения.

- `authenticate` позволяет приложению инициировать аутентификацию вызывающего субъекта контейнером внутри контекста незащищённого запроса. Отображается диалоговое окно входа в систему для ввода имени пользователя и пароля в целях аутентификации.
- `login` позволяет приложению получить имя пользователя и пароль как альтернатива указанию аутентификации на основе форм в дескрипторе развёртывания приложения.
- `logout` позволяет приложению "забыть" вызывающего субъекта.

В следующем примере кода показано, как использовать методы `login` и `logout`:

JAVA

```java
package test;

import java.io.IOException;
import java.io.PrintWriter;
import java.math.BigDecimal;
import jakarta.ejb.EJB;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

@WebServlet(name="TutorialServlet", urlPatterns={"/TutorialServlet"})
public class TutorialServlet extends HttpServlet {
    @EJB
    private ConverterBean converterBean;

    /**
     * Обрабатывает HTTP-запросы как для <code>GET</code>
     *    так и <code>POST</code>.
     * @param request запрос сервлета
     * @param response ответ сервлета
     * @throws ServletException в случае специфичной сервлетам ошибки
     * @throws IOException в случае ошибки ввода/вывода
     */
    protected void processRequest(HttpServletRequest request,
            HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {

            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet TutorialServlet</title>");
            out.println("</head>");
            out.println("<body>");
            request.login("TutorialUser", "TutorialUser");
            BigDecimal result =
                converterBean.dollarToYen(new BigDecimal("1.0"));
            out.println("<h1>Servlet TutorialServlet result of dollarToYen= "
                + result + "</h1>");
            out.println("</body>");
            out.println("</html>");
        } catch (Exception e) {
            throw new ServletException(e);
        } finally {
            request.logout();
            out.close();
        }
    }
}
```

В следующем примере кода показано, как использовать метод `authenticate`:

JAVA

```java
package com.example.test;

import java.io.*;
import jakarta.servlet.*;
import jakarta.servlet.http.*;

public class TestServlet extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            request.authenticate(response);
            out.println("Authenticate Successful");
        } finally {
            out.close();
        }
    }
}
```

Программная проверка вызывающего субъекта

В общем случае контейнер обеспечивает прозрачное для веб-компонента управление безопасностью. Используйте API безопасности, описанные в этом разделе, только в тех редких ситуациях, когда методам веб-компонента требуется доступ к информации из контекста безопасности.

Спецификация Jakarta Security определяет следующие методы интерфейса `SecurityContext`, позволяющие приложению тестировать аспекты данных вызывающего субъекта:

- `getCallerPrincipal()` извлекает `Principal`, представляющий вызывающего субъекта. Это специфичное для контейнера представление вызывающего принципала, и тип может отличаться от типа вызывающего принципала, первоначально установленного `HttpAuthenticationMechanism`. Этот метод возвращает null для неаутентифицированного пользователя.
- `getPrincipalsByType()` извлекает всех принципалов данного типа. Этот метод может использоваться для извлечения специфичного для данного приложения вызывающего принципала, определённого при аутентификации. Этот метод в первую очередь полезен в том случае, если вызывающий принципал контейнера отличается от вызывающего принципала приложения, и приложению требуется особая информация о поведении, доступная только из субъекта приложения. Этот метод возвращает пустой `Set`, если вызывающий субъект не прошёл проверку подлинности или если запрошенный тип не найден.

  Если присутствуют и вызывающий принципал контейнера, и вызывающий принципал приложения, значение, возвращаемое `getName()`, одинаково для обоих принципалов.
- `isCallerInRole()` принимает аргумент String, представляющий роль, подлежащую проверке. Спецификация не определяет способ определения роли, но результат должен быть таким же, как если бы был сделан соответствующий вызов, специфичный для контейнера (например, `HttpServletRequest.isUserInRole()`, `EJBContext.isCallerInRole()`) и должен соответствовать результату, подразумеваемому спецификациями, предписывающими поведение отображения ролей.

Servlet 5.0 указывает следующие методы, которые позволяют получить доступ к сведениям о безопасности вызывающего субъекта.

- `getRemoteUser` определяет имя пользователя, с которым аутентифицировался клиент. Метод `getRemoteUser` возвращает имя удалённого пользователя (вызывающего субъекта), связанного контейнером с запросом. Если ни один пользователь не был аутентифицирован, этот метод возвращает

`null`.

- `isUserInRole` определяет, находится ли удалённый пользователь в определённой роли безопасности. Если ни один пользователь не был аутентифицирован, этот метод возвращает `false`. Этот метод ожидает в параметре объект типа `String`, содержащий название роли из `role-name`.

  Элемент `security-role-ref` должен быть объявлен в дескрипторе развёртывания с подэлементом `role-name`, содержащим имя роли, которое будет передано методу. Использование ссылок на роли безопасности обсуждается в Объявление и связывание ссылок на роли.

- `getUserPrincipal` определяет название принципала текущего пользователя и возвращает объект `java.security.Principal`. Если ни один пользователь не был аутентифицирован, этот метод возвращает `null`. Вызов метода `getName` у `Principal`, возвращённого `getUserPrincipal`, возвращает имя удалённого пользователя.

Приложение может принимать решения бизнес-логики на основе информации, полученной с помощью этих API.

Программное тестирование доступа к ресурсу

Интерфейс `SecurityContext`, как указано в спецификации Jakarta Security API, определяет следующий метод для программного тестирования доступа к ресурсу:

- Метод `hasAccessToWebResource()` определяет, имеет ли вызывающий субъект доступ к указанному веб-ресурсу для указанных методов HTTP, что определяется ограничениями безопасности, настроенными для приложения.

  Параметр ресурса — это `URLPatternSpec`, как определено Jakarta Authorization (https://jakarta.ee/specifications/authorization/2.0/), который идентифицирует специфичный для приложения веб-ресурс.

  Этот метод может использоваться для проверки доступа к ресурсам только в текущем приложении — его нельзя использовать для проверки доступа к ресурсам в других приложениях.

Например, рассмотрим следующее определение сервлета:

```java
@WebServlet("/protectedServlet")
@ServletSecurity(@HttpConstraint(rolesAllowed = "foo"))
public class ProtectedServlet extends HttpServlet { ... }
```

И следующий вызов `hasAccessToWebResource()`:

```java
securityContext.hasAccessToWebResource("/protectedServlet", GET)
```

Вышеупомянутый вызов `hasAccessToWebResource()` возвращает true только в том случае, когда вызывающий субъект имеет роль "foo".

Пример кода для программной безопасности

Следующий код демонстрирует использование программной защиты в целях программного входа в систему. Этот сервлет делает следующее.

1. Отображает информацию о текущем пользователе.
2. Предлагает пользователю войти в систему.
3. Снова печатает информацию для демонстрации эффекта метода `login`.

4. Осуществляет выход пользователя из системы.

5. Снова выводит информацию для демонстрации эффекта метода `logout`.

JAVA

```java
package enterprise.programmatic_login;

import java.io.*;
import java.net.*;
import jakarta.annotation.security.DeclareRoles;
import jakarta.servlet.*;
import jakarta.servlet.http.*;

@DeclareRoles("jakartaeeuser")
public class LoginServlet extends HttpServlet {

    /**
     * Обрабатывает HTTP-запросы как для GET, так и для POST.
     * @param request запрос сервлета
     * @param response ответ сервлета
     */
    protected void processRequest(HttpServletRequest request,
                HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            String userName = request.getParameter("txtUserName");
            String password = request.getParameter("txtPassword");

            out.println("Before Login" + "<br><br>");
            out.println("IsUserInRole?.."
                        + request.isUserInRole("jakartaeeuser")+"<br>");
            out.println("getRemoteUser?.." + request.getRemoteUser()+"<br>");
            out.println("getUserPrincipal?.."
                        + request.getUserPrincipal()+"<br>");
            out.println("getAuthType?.." + request.getAuthType()+"<br><br>");

            try {
                request.login(userName, password);
            } catch(ServletException ex) {
                out.println("Login Failed with a ServletException.."
                    + ex.getMessage());
                return;
            }
            out.println("After Login..."+"<br><br>");
            out.println("IsUserInRole?.."
                        + request.isUserInRole("jakartaeeuser")+"<br>");
            out.println("getRemoteUser?.." + request.getRemoteUser()+"<br>");
            out.println("getUserPrincipal?.."
                        + request.getUserPrincipal()+"<br>");
            out.println("getAuthType?.." + request.getAuthType()+"<br><br>");

            request.logout();
            out.println("After Logout..."+"<br><br>");
            out.println("IsUserInRole?.."
                        + request.isUserInRole("jakartaeeuser")+"<br>");
            out.println("getRemoteUser?.." + request.getRemoteUser()+"<br>");
            out.println("getUserPrincipal?.."
                        + request.getUserPrincipal()+"<br>");
            out.println("getAuthType?.." + request.getAuthType()+"<br>");
        } finally {
            out.close();
        }
    }
    ...
}
```

Объявление и связывание ссылок на роли

Ссылка на роль безопасности — это сопоставление имени роли, которая вызывается из веб-компонента с помощью `isUserInRole(String role)`, и имени роли безопасности, определённой для приложения. Если в дескрипторе развёртывания не объявлено никакого элемента `security-role-ref` и вызван метод `isUserInRole`, контейнер по умолчанию проверяет указанное имя роли по списку всех ролей, определённых для веб-приложения. Использование метода по умолчанию вместо использования элемента `security-role-ref` ограничивает вашу гибкость в изменении имён ролей в приложении без перекомпиляции сервлета, выполняющего вызов.

Элемент `security-role-ref` используется, когда приложение использует `HttpServletRequest.isUserInRole(String role)`. Значение, переданное методу `isUserInRole`, является объект типа `String`, представляющий имя роли пользователя. Значением элемента `role-name` должен быть объект типа `String`, используемый в качестве параметра для `HttpServletRequest.isUserInRole(String role)`. `role-link` должна содержать имя одной из ролей безопасности, определённых в элементах `security-role`. Контейнер использует отображение `security-role-ref` на `security-role` при определении возвращаемого значения вызова.

Например, чтобы отобразить ссылку роли `cust` на роль `bankCustomer`, элементы должны выглядеть следующим образом:

XML

```
<servlet>
...
    <security-role-ref>
        <role-name>cust</role-name>
        <role-link>bankCustomer</role-link>
    </security-role-ref>
...
</servlet>
```

Если метод сервлета вызывается пользователем с ролью безопасности `bankCustomer`, `isUserInRole("cust")` возвращает `true`.

Элемент `role-link` в элементе `security-role-ref` должен соответствовать `role-name`, определённому в элементе `security-role` того же дескриптора развёртывания `web.xml`, как показано здесь:

XML

```
<security-role>
    <role-name>bankCustomer</role-name>
</security-role>
```

Ссылка на роль безопасности, включая имя, определённое ссылкой, распространяется на компонент, дескриптор развёртывания которого содержит элемент дескриптора развёртывания `security-role-ref`.

Примеры: защита веб-приложений

Для корректной работы любого примера требуется некоторая предварительная настройка.

Обзор примеров защиты веб-приложений

В примерах используются аннотации, программная и/или декларативная безопасность для демонстрации добавления безопасности в существующие веб-приложения.

Вот некоторые другие места, где вы найдёте примеры защиты различных типов приложений:

- Пример cart-secure: декларативная защита Enterprise-бина

- Пример converter-secure: программная защита Enterprise-бина
- Примеры Eclipse GlassFish для Jakarta EE 9: https://github.com/eclipse-ee4j/glassfish-samples

Настройка системы для запуска примеров безопасности

Чтобы настроить систему для запуска примеров безопасности, необходимо настроить базу данных пользователей, которую приложение может использовать для аутентификации пользователей. Прежде чем продолжить, выполните следующие действия.

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. Добавьте авторизованного пользователя в GlassFish Server. Для примеров, приведённых в этой главе и в Начало работы по защите EJB-приложений, добавьте пользователя в область безопасности `file` GlassFish Server и в группу `TutorialUser` .
   a. В Консоли администрирования разверните узел Конфигурации, затем разверните узел серверной конфигурации.
   b. Разверните узел безопасности.
   c. Разверните узел Realms.
   d. Выберите узел File.
   e. На странице «Изменить область» нажмите «Управление пользователями».
   f. На странице «Пользователи File» нажмите «Создать».
   g. В поле Идентификатор пользователя введите идентификатор пользователя.
   h. В поле списка групп введите `TutorialUser` .
   i. В полях Новый пароль и Подтверждение нового пароля введите пароль.
   j. Нажмите ОК.

   Обязательно запишите имя пользователя и пароль для пользователя, которого создаёте, чтобы вы могли использовать его для тестирования примеров приложений. Аутентификация чувствительна к регистру имени пользователя и пароля, поэтому пишите имя пользователя и пароль тщательно. Эта тема обсуждается более подробно в Управление пользователями и группами в GlassFish Server.

Jakarta Security требует, чтобы имена участников групп по умолчанию сопоставлялись с одноимёнными ролями. Следовательно, параметр **Назначение по умолчанию для ролей** включён по умолчанию на странице безопасности консоли.

Пример hello2-basicauth: базовая аутентификация с сервлетом

В этом примере объясняется, как использовать базовую аутентификацию с сервлетом. При базовой аутентификации сервлета веб-браузер предоставляет стандартное диалоговое окно входа в систему, которое нельзя настроить. Когда пользователь отправляет своё имя и пароль, сервер определяет, являются ли указанные имя пользователя и пароль реквизитами авторизованного пользователя, и отправляет запрошенный веб-ресурс, если пользователь авторизован для его просмотра.

В общем, следующие шаги необходимы для добавления базовой аутентификации к незащищенному сервлету, например, описанные в главе 6 *Начало работы с веб-приложениями*. В примере приложения из этого учебника многие из этих шагов были выполнены и перечислены здесь просто для того, чтобы показать, что нужно сделать, если вы захотите создать подобное приложение. Это приложение можно найти в каталоге `tut-install/examples/security/hello2-basicauth/` .

1. Следуйте инструкциям в Настройка системы для запуска примеров безопасности.
2. Создайте веб-модуль для примера `hello2` как описано в главе 6 *Начало работы с веб-приложениями*.
3. Добавьте соответствующие аннотации безопасности к сервлету. Аннотации безопасности описаны в Указание безопасности для базовой аутентификации с использованием аннотаций.
4. Соберите, упакуйте и разверните веб-приложение, выполнив шаги из Сборка, упаковка и развёртывание примера hello2-basicauth с использованием IDE NetBeans или Сборка, упаковка и развёртывание примера hello2-basicauth с использованием Maven.
5. Запустите веб-приложение, выполнив шаги, описанные в Запуск hello2-basicauth.

Указание безопасности для базовой аутентификации с использованием аннотаций

Механизм аутентификации по умолчанию, используемый GlassFish Server, — это базовая аутентификация. При базовой аутентификации GlassFish Server создаёт стандартное диалоговое окно входа в систему для получения данных об имени пользователя и пароле для доступа к защищённому ресурсу. После аутентификации пользователя доступ к защищённому ресурсу разрешается.

Чтобы указать безопасность для сервлета, используйте аннотацию `@ServletSecurity`. Эта аннотация позволяет указывать как конкретные ограничения для методов HTTP, так и более общие ограничения, которые применяются ко всем методам HTTP, для которых не указано никаких конкретных ограничений. В аннотации `@ServletSecurity` можно указать следующие аннотации:

- `@HttpMethodConstraint`, которая применяется к определённому методу HTTP
- Более общая `@HttpConstraint`, которая применяется ко всем методам HTTP, для которых нет соответствующей аннотации `@HttpMethodConstraint`

Обе аннотации `@HttpMethodConstraint` и `@HttpConstraint` в аннотации `@ServletSecurity` могут указывать следующее:

- Элемент `transportGuarantee` определяет требования к защите данных (то есть, требуется ли SSL/TLS), которые должны выполняться соединениями, по которым поступают запросы. Допустимые значения для этого элемента: `NONE` и `CONFIDENTIAL`.
- Элемент `roleAllowed`, который определяет имена авторизованных ролей.

Для приложения `hello2-basicauth`, `GreetingServlet` имеет следующие аннотации:

JAVA

```
@WebServlet(name = "GreetingServlet", urlPatterns = {"/greeting"})
@ServletSecurity(
@HttpConstraint(transportGuarantee = TransportGuarantee.CONFIDENTIAL,
    rolesAllowed = {"TutorialUser"}))
```

Эти аннотации указывают, что URI запроса `/greeting` может быть доступен только пользователям, которым был разрешён доступ к этому URL, поскольку было подтверждено, что они имеют роль `TutorialUser`. Данные будут отправлены через защищённый транспорт, чтобы данные пользователя и пароль не могли быть прочитаны при передаче.

Если используется аннотация `@ServletSecurity`, указывать параметры безопасности в дескрипторе развёртывания не обязательно. Используйте дескриптор развёртывания, чтобы указать параметры для механизмов проверки подлинности по умолчанию, для которых нельзя использовать аннотацию `@ServletSecurity`.

Сборка, упаковка и развёртывание примера hello2-basicauth с использованием IDE NetBeans

1. Следуйте инструкциям в Настройка системы для запуска примеров безопасности.
2. В меню «Файл» выберите «Открыть проект».
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/security
   ```

4. Выберите каталог `hello2-basicauth` .
5. Нажмите Открыть проект.
6. На вкладке «Проекты» кликните правой кнопкой мыши проект `hello2-basicauth` и выберите «Сборка».

   Эта команда собирает и развёртывает пример приложения в GlassFish Server.

Сборка, упаковка и развёртывание примера hello2-basicauth с использованием Maven

1. Следуйте инструкциям в Настройка системы для запуска примеров безопасности.
2. В окне терминала перейдите в:

   ```
   tut-install/examples/security/hello2-basicauth/
   ```

3. Введите следующую команду:

   ```
   mvn install
   ```
   SHELL

   Эта команда собирает и упаковывает приложение в WAR-файл `hello2-basicauth.war` , расположенный в каталоге `target` , а затем развёртывает WAR-файл.

Запуск hello2-basicauth

1. В веб-браузере введите следующий URL:

   ```
   https://localhost:8181/hello2-basicauth/greeting
   ```

   Вам может быть предложено принять сертификат безопасности для сервера. Если это так, примите сертификат безопасности. Если браузер предупреждает, что сертификат недействителен, потому что он самоподписан, добавьте исключение безопасности для приложения.

   Откроется диалоговое окно «Требуется аутентификация». Его внешний вид зависит от браузера, который вы используете.

2. Введите комбинацию имени пользователя и пароля, соответствующие пользователю, который уже был создан в области безопасности `file` GlassFish Server и был добавлен в группу `TutorialUser` , и нажмите ОК.

   Базовая аутентификация чувствительна к регистру как для имени пользователя, так и для пароля, поэтому введите имя пользователя и пароль в точности так, как это определено для GlassFish Server.

   Сервер возвращает запрошенный ресурс, если выполнены все следующие условия:

   - Пользователь с указанным именем определён для GlassFish Server.
   - Пользователь с указанным именем пользователя имеет введённый вами пароль.
   - Введённая комбинация имени пользователя и пароля связана с группой `TutorialUser` в GlassFish Server.

- Роль `TutorialUser` , определённая в приложении, соответствует группе `TutorialUser` , определённой в GlassFish Server.

3. Введите имя в поле и нажмите «Отправить».

   Поскольку вы уже авторизованы, вводимое на этом шаге имя не имеет никаких ограничений. Сейчас у вас есть неограниченный доступ к приложению.

   Приложение отвечает «Hello» на имя, которое вы ввели.

Пример hello1-formauth: аутентификация на основе форм в приложении Jakarta Faces

В этом примере объясняется, как использовать аутентификацию на основе форм в приложении Jakarta Faces. С помощью аутентификации на основе форм вы можете настроить экран входа в систему и страницы ошибок, которые отображаются веб-клиентом для аутентификации имени пользователя и пароля. Когда пользователь отправляет своё имя и пароль, сервер определяет, принадлежат ли введённые реквизиты авторизованному пользователю, и, если да, отправляет запрошенный веб-ресурс.

В этом примере `hello1-formauth` добавлена защита для приложения Jakarta Faces, показанного в веб-модуле с использованием Jakarta Faces: пример hello1.

В общем, шаги, необходимые для добавления аутентификации на основе форм в незащищённое приложение Jakarta Faces, аналогичны тем, которые описаны в Пример hello2-basicauth: базовая аутентификация с сервлетом. Основное отличие состоит в том, что вы должны использовать дескриптор развёртывания, чтобы указать использование аутентификации на основе форм, как описано в Указание безопасности для примера аутентификации на основе форм. Кроме того, вы должны создать страницу формы входа и страницы ошибок, как описано в Создание формы входа и страницы ошибок.

Это приложение можно найти в каталоге *`tut-install`*`/examples/security/hello1-formauth/` .

Создание формы входа и страницы ошибок

При использовании механизмов входа на основе форм необходимо указать страницу, содержащую форму, которую вы хотите использовать для получения имени пользователя и пароля, а также страницу, отображаемую в случае сбоя аутентификации при входе. В этом разделе обсуждаются форма входа и страница ошибки, использованная в этом примере. Указание безопасности для примера проверки подлинности на основе форм показывает, как эти страницы указываются в дескрипторе развёртывания.

Страница входа может быть HTML-страницей или сервлетом и она должна возвращать HTML-страницу, содержащую форму, соответствующую определённым соглашениям об именовании (для получения дополнительной информации об этих требованиях см. спецификацию Jakarta Servlet 5.0). Для этого включите элементы, которые принимают имя пользователя и пароль, между тегами `<form></form>` на странице входа. Содержимое HTML-страницы или сервлета для страницы входа в систему должно быть закодировано следующим образом:

HTML

```
<form method="post" action="j_security_check">
    <input type="text" name="j_username">
    <input type="password" name= "j_password">
</form>
```

Полный код страницы login.html входа в систему, используемый в этом примере, можно найти в каталоге *`tut-install`*`/examples/security/hello1-formauth/src/main/webapp/` . Вот код для этой страницы:

```HTML
<html lang="en">
    <head>
        <title>Login Form</title>
    </head>
    <body>
        <h2>Hello, please log in:</h2>
        <form method="post" action="j_security_check">
            <table role="presentation">
                <tr>
                    <td>Please type your user name: </td>
                    <td><input type="text" name="j_username"
                               size="20"/></td>
                </tr>
                <tr>
                    <td>Please type your password: </td>
                    <td><input type="password" name="j_password"
                               size="20"/></td>
                </tr>
            </table>
            <p></p>
            <input type="submit" value="Submit"/>
             
            <input type="reset" value="Reset"/>
        </form>
    </body>
</html>
```

Страница ошибки входа отображается, если пользователь вводит комбинацию имени пользователя и пароля, которая не авторизована для доступа к защищённому URI. В этом примере страницу error.html ошибки входа можно найти в каталоге *tut-install*/examples/security/hello1-formauth/src/main/webapp/ . В этом примере страница ошибки входа в систему объясняет причину получения страницы ошибки и предоставляет ссылку, которая позволит пользователю повторить попытку. Вот код для этой страницы:

```HTML
<html lang="en">
    <head>
        <title>Login Error</title>
    </head>
    <body>
        <h2>Invalid user name or password.</h2>

        <p>Please enter a user name or password that is authorized to access
           this application. For this application, this means a user that
           has been created in the <code>file</code> realm and has been
           assigned to the <em>group</em> of <code>TutorialUser</code>.</p>
        <p><a href="login.html">Return to login page</a></p>
    </body>
</html>
```

Указание безопасности для примера проверки подлинности на основе форм

В этом примере показано очень простое веб-приложение на основе сервлетов и добавлена аутентификация на основе форм. Чтобы использовать аутентификацию на основе формы вместо базовой аутентификации для примера Jakarta Faces, необходимо использовать дескриптор развёртывания.

В следующем примере кода показаны элементы безопасности, добавленные в дескриптор развёртывания для этого примера, который можно найти в *tut-install*/examples/security/hello1-formauth/src/main/webapp/WEB-INF/web.xml :

XML

```
<security-constraint>
    <display-name>Constraint1</display-name>
    <web-resource-collection>
        <web-resource-name>wrcoll</web-resource-name>
        <description/>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <description/>
        <role-name>TutorialUser</role-name>
    </auth-constraint>
</security-constraint>

<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>file</realm-name>
    <form-login-config>
        <form-login-page>/login.xhtml</form-login-page>
        <form-error-page>/error.xhtml</form-error-page>
    </form-login-config>
</login-config>

<security-role>
    <description/>
    <role-name>TutorialUser</role-name>
</security-role>
```

Сборка, упаковка и развёртывание примера hello1-formauth с использованием IDE NetBeans

1. Следуйте инструкциям в Настройка системы для запуска примеров безопасности.
2. В меню «Файл» выберите «Открыть проект».
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/security
   ```

4. Выберите каталог `hello1-formauth`.
5. Нажмите Открыть проект.
6. На вкладке «Проекты» кликните правой кнопкой мыши проект `hello1-formauth` и выберите «Выполнить».

   Эта команда собирает и развёртывает пример приложения в GlassFish Server, а затем открывает его в браузере.

Сборка, упаковка и развёртывание примера hello1-formauth с помощью Maven и команды asadmin

1. Следуйте инструкциям в Настройка системы для запуска примеров безопасности.
2. В окне терминала перейдите в:

   ```
   tut-install/examples/security/hello1-formauth/
   ```

3. Введите следующую команду в окне терминала или командной строке:

   SHELL

   ```
   mvn install
   ```

   Эта команда собирает и упаковывает приложение в WAR-файл `hello1-formauth.war`, расположенный в каталоге `target`, а затем развёртывает WAR-файл в GlassFish Server.

Запуск примера hello1-formauth

Чтобы запустить веб-клиент для `hello1-formauth`, выполните следующие действия.

1. Откройте веб-браузер по следующему URL:

   ```
   http://localhost:8080/hello1-formauth/
   ```

2. В форме входа введите комбинацию имени пользователя и пароля, соответствующие пользователю, который уже был создан в области безопасности `file` GlassFish Server и был включён в группу `TutorialUser`.

   Аутентификация на основе форм чувствительна к регистру как имени пользователя, так и пароля, поэтому введите имя пользователя и пароль в точности так, как это определено для GlassFish Server.

3. Нажмите Отправить.

   Если вы ввели `My_Name` в качестве имени и `My_Pwd` в качестве пароля, сервер вернёт запрошенный ресурс, если выполнены все следующие условия.

   - В GlassFish Server определён пользователь с именем `My_Name`.
   - Пользователь с именем `My_Name` имеет пароль `My_Pwd`, определённый в GlassFish Server.
   - Пользователь `My_Name` с паролем `My_Pwd` является членом группы `TutorialUser` в GlassFish Server.
   - Роль `TutorialUser`, определённая в приложении, назначена группе `TutorialUser`, определённой в GlassFish Server.

   Когда эти условия выполнены и сервер аутентифицировал пользователя, приложение становится доступным.

4. Введите своё имя и нажмите «Отправить».

   Поскольку вы уже авторизованы, вводимое на этом шаге имя не имеет никаких ограничений. Сейчас у вас есть неограниченный доступ к приложению.

   Приложение отвечает «Hello».

Дальнейшие шаги

Для дополнительного тестирования и просмотра созданной страницы ошибки входа в систему закройте и снова откройте браузер, введите URL приложения и введите имя пользователя и пароль, которые не авторизованы.

Глава 52. Начало работы по защите EJB-приложений

В этой главе описывается, как администрировать безопасность для EJB-приложений.

Основные задачи безопасности для EJB-приложений

Системные администраторы, разработчики приложений, провайдеры компонентов и установщики отвечают за администрирование безопасности EJB-приложений. Основные задачи безопасности:

- Настройка базы данных пользователей и назначение им соответствующей группы
- Настройка распространения идентификационной информации
- Настройка свойств GlassFish Server, которые позволяют приложениям работать правильно. Обратите внимание, что по умолчанию в консоли администрирования GlassFish Server включён параметр **Назначение по умолчанию для ролей**.
- Аннотирование классов и методов корпоративного приложения для предоставления информации о том, какие методы должны иметь ограниченный доступ

Разделы примеров безопасности в этой и предыдущей главе объясняют, как выполнять эти задачи.

Защита Enterprise-бинов

EJB-компоненты — это компоненты Jakarta EE, реализующие технологию EJB. EJB-компоненты работают в EJB-контейнере — среде выполнения в GlassFish Server. Несмотря на то, что EJB-контейнер прозрачен для разработчика, он предоставляет сервисы системного уровня, такие как транзакции и безопасность, для EJB-компонентов, которые составляют ядро транзакционных приложений Jakarta EE.

Методы Enterprise-бина могут быть защищены одним из следующих способов.

- Декларативная безопасность (предпочтительно). Выражает требования безопасности компонента приложения, используя дескрипторы развёртывания или аннотации. Наличие аннотации в бизнес-методе класса Enterprise-бина, в которой указываются разрешения метода, — это всё, что необходимо для защиты метода и проверки подлинности в некоторых ситуациях. В этом разделе обсуждается этот простой и эффективный метод защиты Enterprise-бинов.

  Из-за некоторых ограничений, связанных с упрощённым методом защиты Enterprise-бинов, в некоторых случаях может потребоваться использовать дескриптор развёртывания для указания информации о безопасности. Для работы решения на сервере необходимо настроить механизм проверки подлинности. Метод аутентификации GlassFish Server по умолчанию — базовая аутентификация.

  В этом руководстве объясняется, как вызвать аутентификацию по имени пользователя и паролю путём аннотирования бизнес-методов корпоративного приложения с указанием разрешений для методов.

  Чтобы упростить задачу развёртывания, разработчик приложения может определить роли безопасности. Роль безопасности — это группа разрешений, которые должны иметь пользователи приложения определённого типа для успешного использования приложения. Например, в приложении для расчёта заработной платы некоторые пользователи захотят просматривать свою собственную информацию о заработной плате (сотрудник), некоторым необходимо будет просматривать информацию о заработной плате других (менеджер), а некоторые должны будут иметь возможность изменять информацию о заработной плате других (отдел заработной платы). Разработчик приложения будет определять потенциальных пользователей приложения и какие методы будут им доступны. Затем разработчик приложения будет аннотировать классы или методы Enterprise-бинов, указывая типы пользователей,

которым разрешён доступ к этим методам. Использование аннотаций для указания авторизованных пользователей описано в Указание авторизованных пользователей путём объявления ролей безопасности.

Когда одна из аннотаций используется для определения разрешений метода, установщик автоматически потребует аутентификацию по имени пользователя и паролю. При таком типе аутентификации пользователю предлагается ввести имя пользователя и пароль, которые будут сравниваться с базой данных известных пользователей. Если пользователь найден и пароль совпадает, назначенные ему роли будут сравниваться с ролями, которым разрешён доступ к методу. Если пользователь прошёл проверку подлинности и обнаружил, что у него есть роль, которой разрешён доступ к этому методу, данные будут возвращены пользователю.

Использование декларативной безопасности обсуждается в Защита Enterprise-бинов с использованием декларативной безопасности.

- Программная безопасность: для Enterprise-бина — код, встроенный в бизнес-метод, который используется для программного доступа к идентификационной информации вызывающего субъекта и который использует эту информацию для принятия решений по вопросам безопасности. Программная безопасность полезна, когда одной декларативной безопасности недостаточно для реализации модели безопасности приложения.

  В общем случае управление безопасностью должно осуществляться контейнером таким образом, чтобы это было прозрачно для бизнес-методов Enterprise-бинов. Программные API безопасности, описанные в этой главе, следует использовать только в тех редких ситуациях, когда бизнес-методам Enterprise-бина требуется доступ к информации о контексте безопасности, например, когда вы хотите предоставить доступ на основе времени суток или других нетривиальных проверка состояния для конкретной роли.

  Программная безопасность обсуждается в Защита Enterprise-бинов с использованием программной безопасности.

Некоторые материалы в этой главе предполагают, что вы уже прочитали главу 35 *Enterprise-бины*, главу 36 *Начало работы с Enterprise-бинами* и главу 50 *Введение в безопасность на платформе Jakarta EE*.

В этом разделе обсуждается защита приложения Jakarta EE, в котором один или несколько модулей, таких как JAR-файлы EJB, упакованы в файл EAR — файл архива, содержащего приложение. Аннотации безопасности будут использоваться в файлах классов программирования Java для указания авторизованных пользователей и базовой аутентификации или имени пользователя/пароля.

Enterprise-бины часто предоставляют бизнес-логику веб-приложения. В этих случаях упаковка Enterprise-бина в модуль WAR веб-приложения упрощает развёртывание и организацию приложения. Enterprise-бины могут быть упакованы в модуль WAR как файлы классов Java или в файл JAR, который входит в состав модуля WAR. Когда веб-интерфейс построен на сервлетах или страницах Jakarta Faces и приложение упаковано в WAR-файл, защита приложения может быть описана в файле `web.xml` приложения. EJB-компонент в WAR-файле может иметь собственный дескриптор развёртывания, `ejb-jar.xml`, если он требуется. Защита веб-приложений с помощью `web.xml` обсуждается в Начало работы по защите веб-приложений.

В следующих разделах описываются декларативные и программные механизмы безопасности, которые можно использовать для защиты ресурсов Enterprise-бина. К защищённым ресурсам относятся методы Enterprise-бинов, которые вызываются из клиентских приложений, веб-компонентов или других Enterprise-бинов.

Дополнительные сведения по этой теме см. в спецификации Jakarta Enterprise Beans 4.0. Этот документ можно загрузить с веб-сайта https://jakarta.ee/specifications/enterprise-beans/4.0/. В главе 11 спецификации «Управление безопасностью» обсуждается управление безопасностью для Enterprise-бинов.

Защита Enterprise-бинов с использованием декларативной безопасности

Декларативная безопасность позволяет разработчику приложения указывать, каким пользователям разрешён доступ и к каким методам Enterprise-бинов, аутентифицировать этих пользователей с помощью базовой аутентификации. Зачастую человек, разрабатывающий корпоративное приложение, и человек, отвечающий за развёртывание приложения, — разные люди. Разработчик приложения, который использует декларативную безопасность для определения разрешений методов и механизмов аутентификации, передает разработчику представление безопасности EJB-компонентов, содержащихся в JAR EJB. При передаче представления безопасности установщику, он/она использует эту информацию для задания разрешений метода для ролей безопасности. Если не передать представление бевзопасности, установщику придётся самостоятельно разобраться, что делает каждый бизнес-метод, чтобы задать разрешения на их выполнение конкретным пользователям.

Представление безопасности состоит из набора ролей безопасности, семантической группировки разрешений, которые должны иметь пользователи определённого типа для доступа к приложению. Роли безопасности должны быть логическими ролями, представляющими тип пользователя. Вы можете определить разрешения метода для каждой роли безопасности. Разрешение метода — это разрешение для вызова определённой группы методов бизнес-интерфейса Enterprise-бина, домашнего интерфейса, интерфейса компонента и/или конечных точек веб-сервиса. После определения разрешений метода для проверки личности пользователя будет использоваться аутентификация по имени пользователя и паролю.

Важно помнить, что роли безопасности используются для определения логического представления безопасности приложения. Их не следует путать с группами пользователей, пользователями, принципалами и другими понятиями, существующими в GlassFish Server. Обратите внимание, что Jakarta Security требует, чтобы по умолчанию имена участников групп сопоставлялись с одноимёнными ролями, но реализация стандарта может позволить настроить и другое значение по умолчанию. В GlassFish Server не нужно выполнять никаких дополнительных шагов для назначения ролей, определённых в приложении, пользователям, группам и принципалам, которые являются компонентами пользовательской базы данных в области безопасности `file`. Это отображение по умолчанию устанавливается в Консоли администрирования GlassFish Server, как описано в Назначение ролей пользователям и группам.

В следующих разделах показано, как разработчик приложения использует декларативную безопасность для защиты приложения или для создания представления безопасности для передачи установщику.

Указание авторизованных пользователей путём объявления ролей безопасности

В этом разделе обсуждается, как использовать аннотации для указания разрешений для методов класса EJB. Для получения дополнительной информации об этих аннотациях см. спецификацию Jakarta Annotations по ссылке https://jakarta.ee/specifications/annotations/2.0/.

Разрешения методов могут быть указаны для класса, бизнес-методов класса или для обоих. Права доступа к методу могут быть указаны для метода класса компонента для переопределения значения полномочий метода, указанного для всего класса компонента. Следующие аннотации используются для указания разрешений метода.

- `@DeclareRoles` : указывает все роли, которые будет использовать приложение, включая роли, которые не указаны в аннотации `@RolesAllowed`. Набор ролей безопасности, используемых приложением, представляет собой сумму ролей безопасности, определённых в аннотациях `@DeclareRoles` и `@RolesAllowed`.

  Аннотация `@DeclareRoles` указывается в классе компонента, где она служит для объявления ролей, которые можно проверить (например, путём вызова `isCallerInRole`) из методов аннотированного класса. При объявлении названия роли, используемой в качестве параметра для метода

`isCallerInRole(String roleName)`, объявленное имя должно совпадать со значением параметра.

В следующем примере кода демонстрируется использование аннотации `@DeclareRoles`:

JAVA

```
@DeclareRoles("BusinessAdmin")
public class Calculator {
    ...
}
```

Синтаксис объявления нескольких ролей показан в следующем примере:

JAVA

```
@DeclareRoles({"Administrator", "Manager", "Employee"})
```

- `@RolesAllowed("список-ролей")`: определяет роли безопасности, которым разрешён доступ к методам в приложении. Эта аннотация может быть указана для класса или для одного или нескольких методов. При указании на уровне класса аннотация применяется ко всем методам в классе. При указании в методе аннотация применяется только к этому методу и переопределяет любые значения, указанные на уровне класса.

  Чтобы указать, что никакие роли не авторизованы для доступа к методам в приложении, используйте аннотацию `@DenyAll`. Чтобы указать, что пользователь в любой роли авторизован для доступа к приложению, используйте аннотацию `@PermitAll`.

  При использовании вместе с аннотацией `@DeclareRoles` комбинированный набор ролей безопасности используется приложением.

  В следующем примере кода демонстрируется использование аннотации `@RolesAllowed`:

  JAVA

  ```
  @DeclareRoles({"Administrator", "Manager", "Employee"})
  public class Calculator {

      @RolesAllowed("Administrator")
      public void setNewRate(int rate) {
          ...
      }
  }
  ```

- `@PermitAll`: указывает, что всем ролям безопасности разрешено выполнять указанный метод или методы. Пользователь не проверяется в базе данных, чтобы убедиться, что он/она имеет право доступа к этому приложению.

  Эта аннотация может быть указана для класса или для одного или нескольких методов. Указание этой аннотации для класса означает, что она применяется ко всем методам класса. Указание его на уровне метода означает, что оно применяется только к этому методу.

  В следующем примере кода демонстрируется использование аннотации `@PermitAll`:

JAVA

```
import jakarta.annotation.security.*;
@RolesAllowed("RestrictedUsers")
public class Calculator {

    @RolesAllowed("Administrator")
    public void setNewRate(int rate) {
        //...
    }
    @PermitAll
    public long convertCurrency(long amount) {
        //...
    }
}
```

- `@DenyAll` : указывает, что никакие роли безопасности не могут выполнять указанный метод или методы. Это означает, что эти методы исключены из выполнения в контейнере Jakarta EE.

  В следующем примере кода демонстрируется использование аннотации `@DenyAll` :

JAVA

```
import jakarta.annotation.security.*;
@RolesAllowed("Users")
public class Calculator {
    @RolesAllowed("Administrator")
    public void setNewRate(int rate) {
        //...
    }
    @DenyAll
    public long convertCurrency(long amount) {
        //...
    }
}
```

В следующем фрагменте кода демонстрируется использование аннотации `@DeclareRoles` с методом `isCallerInRole` . В этом примере аннотация `@DeclareRoles` объявляет роль, которую Enterprise-бин `PayrollBean` использует для проверки безопасности с помощью `isCallerInRole("payroll")` чтобы убедиться, что вызывающий субъект имеет право изменять данные о зарплате:

JAVA

```
@DeclareRoles("payroll")
@Stateless public class PayrollBean implements Payroll {
    @Resource SessionContext ctx;

    public void updateEmployeeInfo(EmplInfo info) {

        oldInfo = ... read from database;

        // Поле salary может быть изменено только
        // пользователями с ролью "payroll"
        Principal callerPrincipal = ctx.getCallerPrincipal();
        if (info.salary != oldInfo.salary && !ctx.isCallerInRole("payroll")) {
            throw new SecurityException(...);
        }
        ...
    }
    ...
}
```

В следующем примере кода показано использование аннотации `@RolesAllowed` :

JAVA

```java
@RolesAllowed("admin")
public class SomeClass {
    public void aMethod () {...}
    public void bMethod () {...}
    ...
}

@Stateless
public class MyBean extends SomeClass implements A {

    @RolesAllowed("HR")
    public void aMethod () {...}

    public void cMethod () {...}
    ...
}
```

В этом примере, предполагая, что `aMethod`, `bMethod` и `cMethod` являются методами бизнес-интерфейса `A`, значениями разрешений для методов `aMethod` и `bMethod` являются `@RolesAllowed("HR")` и `@RolesAllowed("admin")` соответственно. Разрешения для метода `cMethod` не указаны.

Для ясности: аннотации не наследуются самим дочерним классом. Вместо этого аннотации применяются к методам родительского класса, которые наследуются дочерним классом.

Указание механизма аутентификации и безопасного соединения

Когда указаны разрешения метода, GlassFish Server будет использовать базовую аутентификацию по имени пользователя и паролю.

Чтобы использовать другой тип аутентификации или требовать защищённого соединения с использованием SSL, укажите эту информацию в дескрипторе развёртывания приложения.

Защита Enterprise-бинов с использованием программной безопасности

Программная безопасность — код, встроенный в бизнес-метод, используется для программного контроля доступа к идентификационной информации вызывающего субъекта и использует эту информацию для принятия решений по вопросам безопасности в самом методе.

В общем случае управление безопасностью должно осуществляться контейнером таким образом, чтобы это было прозрачно для бизнес-методов Enterprise-бина. В этом разделе описывается API `SecurityContext` и связанные с безопасностью методы API `EJBContext`. Новый API `SecurityContext` дублирует некоторые функции API `EJBContext`, поскольку он предназначен для обеспечения согласованного API для всех контейнеров. Эти API безопасности следует использовать только в тех редких ситуациях, когда бизнес-методам Enterprise-бинов требуется доступ к информации о контексте безопасности.

Интерфейс `SecurityContext`, указанный в спецификации Jakarta Security, определяет три метода, которые позволяют провайдеру компонентов обращаться к информации о безопасности вызывающего EJB-компонента:

- `getCallerPrincipal()` извлекает `Principal`, который представляет имя аутентифицированного вызывающего субъекта. Это специфичное для контейнера представление вызывающего принципала, и тип может отличаться от типа вызывающего принципала, первоначально установленного `HttpAuthenticationMechanism`. Этот метод возвращает null для неаутентифицированного пользователя. Обратите внимание, что это поведение отличается от поведения метода `EJBContext.getCallerPrincipal()`, который возвращает определённого разработчиком принципала для анонимного субъекта.

- Метод `getPrincipalsByType()` извлекает всех участников данного типа из аутентифицированного субъекта. Этот метод возвращает пустой `Set`, если вызывающий субъект не прошёл проверку подлинности или если запрошенный тип не найден.

  Если присутствуют и вызывающий принципал контейнера, и вызывающий принципал приложения, значение, возвращаемое `getName()`, одинаково для обоих принципалов.

- `isCallerInRole()` принимает аргумент String, представляющий роль, подлежащую проверке. Спецификация не определяет способ определения роли, но результат должен быть таким же, как если бы был сделан соответствующий вызов, специфичный для контейнера (например, `EJBContext.isCallerInRole()`), и должен быть в соответствии с результатом, подразумеваемым спецификациями, которые предписывают поведение отображения ролей.

Интерфейс `jakarta.ejb.EJBContext` предоставляет два метода, которые позволяют провайдеру компонента получать доступ к информации о безопасности вызывающего Enterprise-бина.

- `getCallerPrincipal` позволяет методам Enterprise-бина получать имя текущего вызывающего принципала. Методы могут, например, использовать имя в качестве ключа к информации в базе данных. Этот метод никогда не возвращает null. Вместо этого он возвращает (специфичный для поставщика) принципала со специальным именем пользователя, чтобы указать анонимного/неаутентифицированного пользователя. Обратите внимание, что это поведение отличается от поведения метода `SecurityContext.getCallerPrincipal()`, который возвращает значение null для неаутентифицированного вызывающего субъекта.

  В следующем примере кода показано использование метода `getCallerPrincipal`:

  JAVA

```java
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    @Resource
    SessionContext ctx;

    @PersistenceContext
    EntityManager em;

    public void changePhoneNumber(...) {
        ...
        // получение вызывающего принципала
        callerPrincipal = ctx.getCallerPrincipal();

        // получение имени вызывающего принципала
        callerKey = callerPrincipal.getName();

        // использованием callerKey как первичного ключа для поиска EmployeeRecord
        EmployeeRecord myEmployeeRecord =
            em.find(EmployeeRecord.class, callerKey);

        // обновление номера телефона
        myEmployeeRecord.setPhoneNumber(...);

        ...
    }
}
```

  В этом примере Enterprise-бин получает имя текущего вызывающего принципала и использует его в качестве первичного ключа для получения объекта `EmployeeRecord`. В этом примере предполагается, что приложение было развёрнуто так, что текущий вызывающий принципал содержит первичный ключ, используемый для идентификации сотрудников (например, номер сотрудника).

- `isCallerInRole` позволяет разработчику кодировать проверки безопасности, которые нельзя легко определить с помощью разрешений метода. Такая проверка может ограничить выполнение запроса заданной ролью или зависеть от информации, хранящейся в базе данных.

  Код Enterprise-бина может использовать метод `isCallerInRole` , чтобы проверить, назначена ли текущему вызывающему субъекту данная роль безопасности. Роли безопасности определяются разработчиком компонентов или компоновщиком приложений и присваиваются администратором развёртывания субъектам или основным группам, существующим в операционной среде.

  Следующий пример кода иллюстрирует использование метода `isCallerInRole` :

```java
@Stateless
public class PayrollBean implements Payroll {
    @Resource
    SessionContext ctx;

    public void updateEmployeeInfo(EmplInfo info) {

        oldInfo = ... read from database;

        // Поле salary может изменяться только вызывающим субъектом
        // имеющим роль "payroll"
        if (info.salary != oldInfo.salary &&
            !ctx.isCallerInRole("payroll")) {
                throw new SecurityException(...);
        }
        ...
    }
    ...
}
```

Таким образом, программная защита может быть использована для динамического управления доступом к методу, например, чтобы ограничить доступ в определённое время суток. Приложение, в котором используются методы `getCallerPrincipal` и `isCallerInRole` , описан в Пример converter-secure: программная защита Enterprise-бина.

Передача идентификационной информации (Run-As)

Вы можете указать, следует ли использовать выполнения методов Enterprise-бина от имени вызывающего субъекта или от имени какого-то другого субъекта. Рисунок 52-1 иллюстрирует эту концепцию.

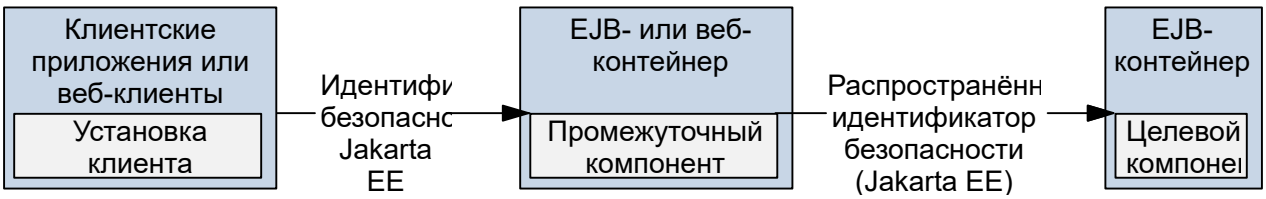

Рисунок 52-1 Передача идентификационной информации

На этом рисунке клиентское приложение вызывает метод EJB-компонента в одном EJB-контейнере. Этот метод Enterprise-бина, в свою очередь, вызывает метод Enterprise-бина в целевом контейнере. Идентификационная информация во время первого вызова является идентификатором вызывающего субъекта. Идентификатором субъекта для второго вызова может быть один из следующих.

- По умолчанию идентификатор вызывающего промежуточный компонент субъекта передаётся целевому Enterprise-бину. Эта техника используется, когда целевой контейнер доверяет промежуточному контейнеру.

- Другой, специфичный идентификатор передаётся целевому Enterprise-бину. Эта техника используется, когда целевой контейнер ожидает доступ с использованием определённого идентификатора.

Чтобы передать идентификатор на целевой Enterprise-бин, настройте передаваемый идентификатор для компонента, как описано в Настройке передачи идентификационной информации. Запуск Enterprise-бина на выполнение от имени другого субъекта не влияет на идентификаторы текущих вызывающих субъектов, то есть на идентификаторы субъектов, которым разрешён доступ к методам текущего Enterprise-бина. Указание запуска от имени другого субъекта устанавливает идентификатор субъекта, который будет использоваться Enterprise-бином при выполнении вызовов.

Указание запуска от имени другого субъекта применяется ко всему Enterprise-бину, включая все методы бизнес-интерфейса Enterprise-бина, локальные и удалённые интерфейсы, интерфейс компонента и интерфейсы конечных точек веб-сервиса, методы слушателя сообщений бина, управляемого сообщениями, методы EJB-таймера и все внутренние методы компонента, которые могут вызываться.

Настройка передачи идентификационной информации

Вы можете настроить выполнение Enterprise-бина от имени другого субъекта с помощью аннотации `@RunAs`, которая определяет роль приложения во время выполнения в контейнере Jakarta EE. Аннотация может быть указана для класса, что позволяет разработчикам выполнять приложение под определённой ролью. Для роли должно быть установлено соответствие с пользователем/группой в области безопасности контейнера. Аннотация `@RunAs` указывает имя роли в параметре.

Следующий код демонстрирует использование аннотации `@RunAs`:

JAVA

```java
@RunAs("Admin")
public class Calculator {
    //....
}
```

Если указанные роли связаны с несколькими пользователями в GlassFish Server, требуется установить для роли принципала.

Доверие между контейнерами

Если Enterprise-бин разработан так, что для вызова целевого компонента используется либо оригинальный вызывающий субъект, либо другой, предустановленный субъект, то целевой компонент будет получать только передаваемую ему информацию субъекта. Целевой компонент не будет получать никаких данных аутентификации.

Целевой контейнер не может аутентифицировать переданного субъекта. Однако, поскольку переданный субъект используется при проверках авторизации (например, разрешение на доступ к методу или вызов метода `isCallerInRole`), необходимо иметь уверенность, что подлинность субъекта подтверждена. Поскольку данных для аутентификации передаваемого субъекта нет, целевой компонент должен доверять вызывающему контейнеру в том, что тот передаёт ему информацию корректно аутентифицированного субъекта.

По умолчанию GlassFish Server настроен на доверие к идентификационной информации, которая передаётся из разных контейнеров. Поэтому не требуется предпринимать каких-либо особых шаги для установления доверительных отношений.

Развёртывание защищённых Enterprise-бинов

Установщик отвечает за обеспечение безопасности скомпонованного приложения после его развёртывания в целевой среде выполнения. Если представление безопасности было предоставлено установщику с использованием аннотаций безопасности и/или дескриптора развёртывания, представление безопасности сопоставляется с механизмами и политиками, используемыми доменом безопасности в целевой среде выполнения, в данном случае GlassFish Server. Если представление безопасности не предоставлено, установщик должен установить соответствующую политику безопасности для приложения Enterprise-бина.

Информация о развёртывании зависит от веб-сервера или сервера приложений.

Примеры: защита Enterprise-бинов

В следующих примерах показано, как защитить Enterprise-бины с помощью декларативной и программной безопасности.

Пример cart-secure: декларативная защита Enterprise-бина

В этом разделе обсуждается, как настроить Enterprise-бин для базовой аутентификации. Когда запрашивается бин, который защищён таким образом, сервер запрашивает имя пользователя и пароль у клиента и проверяет, являются ли имя пользователя и пароль действующими, сравнивая их с базой данных авторизованных пользователей в GlassFish Server.

Если тема аутентификации является новой для вас, см. Указание механизмов аутентификации.

В этом примере демонстрируется безопасность для уже имеющегося незащищённого приложения EJB `cart`, которое находится в *`tut-install`*`/examples/ejb/cart/` и обсуждалось в Примере cart.

Как правило, для добавления аутентификации по имени пользователя/паролю в существующее приложение, содержащее Enterprise-бин, необходимы следующие шаги. В примере приложения, включённого в этот учебник, эти шаги были выполнены и перечислены здесь просто для того, чтобы показать, что нужно сделать, если вы захотите создать подобное приложение.

1. Создайте приложение наподобие Пример cart. Начнём с этого примера и продемонстрируем добавление базовой аутентификации клиента в приложение. Приложение, обсуждаемого в этом разделе, можно найти в каталоге *`tut-install`*`/examples/security/cart-secure/`.
2. Если вы ещё этого не сделали, выполните действия, описанные в Настройка системы для запуска примеров безопасности.
3. Измените исходный код Enterprise-бина `CartBean.java`, чтобы указать, каким ролям и к каким защищённым методам разрешён доступ. Этот шаг обсуждается в Аннотируем бин.
4. Соберите, упакуйте и разверните Enterprise-бин. Затем создайте и запустите клиентское приложение, следуя инструкциям в Запуск примера cart-secure с IDE NetBeans или Запуск примера cart-secure с помощью Maven.

Аннотируем бин

Исходный код приложения `cart` был изменён, как показано в следующем фрагменте кода (изменения выделены жирным шрифтом). Полученный файл можно найти в *`tut-install`*`/examples/security/cart-secure/cart-secure-ejb/src/main/java/ee/jakarta/tutorial/cart/ejb/CartBean.java`.

Фрагмент кода выглядит следующим образом:

JAVA

```java
package ee.jakarta.tutorial.cartsecure.ejb;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import ee.jakarta.tutorial.cart.util.BookException;
import ee.jakarta.tutorial.cart.util.IdVerifier;
import jakarta.ejb.Remove;
import jakarta.ejb.Stateful;
import jakarta.annotation.security.DeclareRoles;
import jakarta.annotation.security.RolesAllowed;

@Stateful
@DeclareRoles("TutorialUser")
public class CartBean implements Cart, Serializable {
    List<String> contents;
    String customerId;
    String customerName;

    @Override
    public void initialize(String person) throws BookException {
        if (person == null) {
            throw new BookException("Null person not allowed.");
        } else {
            customerName = person;
        }

        customerId = "0";
        contents = new ArrayList<>();
    }

    @Override
    public void initialize(String person, String id) throws BookException {
        if (person == null) {
            throw new BookException("Null person not allowed.");
        } else {
            customerName = person;
        }

        IdVerifier idChecker = new IdVerifier();

        if (idChecker.validate(id)) {
            customerId = id;
        } else {
            throw new BookException("Invalid id: " + id);
        }

        contents = new ArrayList<>();
    }

    @Override
    @RolesAllowed("TutorialUser")
    public void addBook(String title) {
        contents.add(title);
    }

    @Override
    @RolesAllowed("TutorialUser")
    public void removeBook(String title) throws BookException {
        boolean result = contents.remove(title);

        if (result == false) {
            throw new BookException("\"" + title + "\" not in cart.");
        }
    }

    @Override
```

```
    @RolesAllowed("TutorialUser")
    public List<String> getContents() {
        return contents;
    }

    @Override
    @Remove()
    @RolesAllowed("TutorialUser")
    public void remove() {
        contents = null;
    }
}
```

Аннотация `@RolesAllowed` указывается на методах, доступ к которым вы хотите защитить. В этом примере только пользователям роли `TutorialUser` будет разрешено добавлять и удалять книги из корзины и выводить содержимое корзины. Аннотация `@RolesAllowed` неявно объявляет роль, на которую будет ссылаться в приложении. Следовательно, аннотация `@DeclareRoles` не требуется. Наличие аннотации `@RolesAllowed` также неявно заявляет, что для доступа пользователей к этим методам потребуется аутентификация. Если в дескрипторе развёртывания не указан метод проверки подлинности, будет проводиться аутентификация по имени пользователя/паролю.

Запуск примера cart-secure с IDE NetBeans

1. Следуйте инструкциям в Настройка системы для запуска примеров безопасности.
2. В меню «Файл» выберите «Открыть проект».
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/security
   ```

4. Выберите каталог `cart-secure`.
5. Установите флажок "Открыть требуемые проекты".
6. Нажмите Открыть проект.
7. На вкладке «Проекты» кликните правой кнопкой мыши проект `cart-secure` и выберите «Сборка».

   На этом этапе выполняется сборка и упаковка приложения в `cart-secure.ear`, расположенном в каталоге `cart-secure-ear/target/`, и развёртывание этого файла EAR в GlassFish Server, извлечение клиентских заглушек и запуск клиентского приложения.
8. В диалоговом окне «Вход для пользователя:» введите имя и пароль пользователя области `file`, созданного в GlassFish Server и назначенного группе `TutorialUser`. Затем нажмите ОК.

   Если введённые имя пользователя и пароль аутентифицированы, выходные данные клиентского приложения появятся на вкладке Вывод:

   ```
   ...
   Retrieving book title from cart: Infinite Jest
   Retrieving book title from cart: Bel Canto
   Retrieving book title from cart: Kafka on the Shore
   Removing "Gravity's Rainbow" from cart.
   Caught a BookException: "Gravity's Rainbow" not in cart.
   Java Result: 1
   ...
   ```

   Если имя пользователя и пароль не аутентифицированы, диалоговое окно появляется снова, пока не будут введены корректные значения.

Запуск примера cart-secure с помощью Maven

1. Следуйте инструкциям в Настройка системы для запуска примеров безопасности.
2. В окне терминала перейдите в:

   ```
   tut-install/examples/security/cart-secure/
   ```

3. Чтобы собрать приложение, упаковать его в файл EAR в подкаталоге `cart-secure-ear/target`, развернуть его и запустить, введите следующую команду в окне терминала или командной строке:

   ```
   mvn install
   ```
   SHELL

4. В диалоговом окне «Вход для пользователя:» введите имя и пароль пользователя области `file`, созданного в GlassFish Server и назначенного группе `TutorialUser`. Затем нажмите ОК.

   Если введённые имя пользователя и пароль аутентифицированы, выходные данные клиентского приложения появятся на вкладке Вывод:

   ```
   ...
   Retrieving book title from cart: Infinite Jest
   Retrieving book title from cart: Bel Canto
   Retrieving book title from cart: Kafka on the Shore
   Removing "Gravity's Rainbow" from cart.
   Caught a BookException: "Gravity's Rainbow" not in cart.
   Java Result: 1
   ...
   ```

   Если имя пользователя и пароль не аутентифицированы, диалоговое окно появляется снова, пока не будут введены корректные значения.

Пример converter-secure: программная защита Enterprise-бина

В этом примере показано, как использовать методы `getCallerPrincipal` и `isCallerInRole` с Enterprise-бином. Этот пример начинается с простейшего приложения EJB-компонента `converter` и изменяет методы `ConverterBean` так, чтобы конвертация валюты происходила только тогда, когда запрашивающая сторона имеет роль `TutorialUser`.

Этот пример можно найти в каталоге *`tut-install`*`/examples/security/converter-secure`. Этот пример основан на незащищенном приложении Enterprise-бина `converter`, которое обсуждается в главе 36 *Начало работы с Enterprise-бинами* и находится в каталоге *`tut-install`*`/examples/ejb/converter/`. Этот раздел основан на примере, добавляя необходимые элементы для защиты приложения с помощью методов `getCallerPrincipal` и `isCallerInRole`, которые более подробно рассматриваются в разделе Программная защита Enterprise-бинов.

В общем, следующие шаги необходимы при использовании методов `getCallerPrincipal` и `isCallerInRole` с Enterprise-бином. В примере приложения из этого учебника многие из этих шагов были выполнены и перечислены здесь просто для того, чтобы показать, что нужно сделать, если вы захотите создать подобное приложение.

1. Создайте простое приложение Enterprise-бина.
2. Настройте пользователя в GlassFish Server в области `file`, в группе `TutorialUser` и установите настройку "Назначение по умолчанию для ролей". Для этого выполните шаги, описанные в Настройка системы для запуска примеров безопасности.
3. Измените бин, добавив методы `getCallerPrincipal` и `isCallerInRole`.

4. Если приложение содержит веб-клиент, который является сервлетом, укажите защиту для сервлета, как описано в Указание безопасности для базовой аутентификации с использованием аннотаций.

5. Соберите, упакуйте, разверните и запустите приложение.

Модификация ConverterBean

Исходный код для исходного класса `ConverterBean` был изменён, чтобы добавить предложение `if..else`, которое проверяет, имеет ли вызывающий субъект роль `TutorialUser`. Если пользователь имеет указанную роль, выполняется конвертация валюты и отображается рассчитанное значение. Если пользователь не имеет указанной роли, расчёт не выполняется, и приложение отображает `0`. Пример кода можно найти в каталоге *`tut-install`*`/examples/security/converter-secure/converter-secure-ejb/src/main/java/ee/jakarta/tutorial/converter/ejb/ConverterBean.java`.

Фрагмент кода выглядит следующим образом:

JAVA

```
package ee.jakarta.tutorial.convertersecure.ejb;

import java.math.BigDecimal;
import java.security.Principal;

import jakarta.annotation.Resource;
import jakarta.annotation.security.DeclareRoles;
import jakarta.annotation.security.RolesAllowed;
import jakarta.ejb.SessionContext;
import jakarta.ejb.Stateless;

@Stateless()
@DeclareRoles("TutorialUser")
public class ConverterBean{

    @Resource SessionContext ctx;
    private final BigDecimal yenRate = new BigDecimal("104.34");
    private final BigDecimal euroRate = new BigDecimal("0.007");

    @RolesAllowed("TutorialUser")
     public BigDecimal dollarToYen(BigDecimal dollars) {
        BigDecimal result = new BigDecimal("0.0");
        Principal callerPrincipal = ctx.getCallerPrincipal();
        if (ctx.isCallerInRole("TutorialUser")) {
            result = dollars.multiply(yenRate);
            return result.setScale(2, BigDecimal.ROUND_UP);
        } else {
            return result.setScale(2, BigDecimal.ROUND_UP);
        }
    }

    @RolesAllowed("TutorialUser")
    public BigDecimal yenToEuro(BigDecimal yen) {
        BigDecimal result = new BigDecimal("0.0");
        Principal callerPrincipal = ctx.getCallerPrincipal();
        if (ctx.isCallerInRole("TutorialUser")) {
             result = yen.multiply(euroRate);
             return result.setScale(2, BigDecimal.ROUND_UP);
        } else {
             return result.setScale(2, BigDecimal.ROUND_UP);
        }
    }
}
```

Модификация ConverterServlet

Следующие аннотации определяют настройки безопасности для `ConverterServlet` — веб-клиента `converter`:

JAVA

```
@WebServlet(urlPatterns = {"/"})
@ServletSecurity(
@HttpConstraint(transportGuarantee = TransportGuarantee.CONFIDENTIAL,
    rolesAllowed = {"TutorialUser"}))
```

Запуск converter-secure с IDE NetBeans

1. Следуйте инструкциям в Настройка системы для запуска примеров безопасности.
2. В меню «Файл» выберите «Открыть проект».
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/security
   ```

4. Выберите каталог `converter-secure`.
5. Нажмите Открыть проект.
6. Кликните правой кнопкой мыши проект `converter-secure` и выберите Сборка.

   Эта команда собирает и развёртывает пример приложения в GlassFish Server.

Запуск converter-secure с использованием Maven

1. Следуйте инструкциям в Настройка системы для запуска примеров безопасности.
2. В окне терминала перейдите в:

   ```
   tut-install/examples/security/converter-secure/
   ```

3. Введите следующую команду:

   SHELL

   ```
   mvn install
   ```

   Эта команда собирает и упаковывает приложение в WAR-файл, `converter-secure.war`, который находится в каталоге `target`, и развёртывает WAR-файл.

Запуск converter-secure

1. Откройте веб-браузер по следующему URL:

   ```
   http://localhost:8080/converter-secure
   ```

   Откроется диалоговое окно «Требуется аутентификация».
2. Введите комбинацию имени пользователя и пароля, соответствующие пользователю, который уже был создан в области безопасности `file` GlassFish Server и был добавлен в группу `TutorialUser`, и нажмите ОК.
3. Введите `100` в поле ввода и нажмите «Отправить».

   Появится вторая страница, показывающая преобразованные значения.

Глава 53. Использование Jakarta Security

В этой главе описаны функции проверки подлинности и проверки учётных данных, предоставляемые Jakarta Security. API также определяет точку доступа `SecurityContext` для программной безопасности.

О Jakarta Security

Jakarta EE включает в себя поддержку Jakarta Security, которая определяет переносимые, подключаемые интерфейсы для идентификации и хранилища идентификационных данных, а также новый инъецируемый интерфейс SecurityContext, предоставляющий точку доступа для программной защиты. Вы можете использовать встроенные реализации этих API или определить кастомные реализации.

Jakarta Security содержит следующие пакеты:

- Пакет `jakarta.security.enterprise` является основным пакетом безопасности Jakarta и содержит классы и интерфейсы, которые охватывают вопросы аутентификации, авторизации и идентификации. Таблица 53-1 перечисляет основные класс и интерфейсы этого пакета.
- Пакет `jakarta.security.enterprise.authentication.mechanism.http` содержит классы и интерфейсы, связанные с механизмами аутентификации на основе HTTP, которые могут взаимодействовать с вызывающим субъектом или третьими сторонами в рамках протокола аутентификации. Таблица 53-2 перечисляет основные классы и интерфейсы в этом пакете.
- Пакет `jakarta.security.enterprise.credential` содержит классы и интерфейсы для представления учётных данных пользователя. Таблица 53-3 перечисляет основные классы и интерфейсы в этом пакете.
- Пакет `jakarta.security.enterprise.identitystore` содержит классы и интерфейсы, связанные с хранилищами идентификаторов, которые валидируют учётные данные вызывающего субъекта и определяют группы вызывающего субъекта. В таблице 53-4 перечислены основные классы и интерфейсы в этом пакете.

Таблица 53-1 Основные классы и интерфейсы пакета `jakarta.security.enterprise`

Класс или интерфейс	Описание
`SecurityContext`	Инъецирумый интерфейс, обеспечивающий точку доступа к программной безопасности, предназначенную для использования кодом приложения для запроса и взаимодействия с Jakarta Security.
`CallerPrincipal`	Основной тип, который может представлять идентификационную информацию вызывающего приложение субъекта.
`AuthenticationStatus`	Enum используется для указания возвращаемого значения из механизма аутентификации.
`AuthenticationException`	Указывает, что в процессе аутентификации возникла проблема.

Таблица 53-2 Основные классы и интерфейсы в `jakarta.security.enterprise.authentication.mechanism.http`

Класс или интерфейс	Описание

Класс или интерфейс	Описание
`HttpAuthenticationMechanism`	Интерфейс, представляющий механизм аутентификации HTTP. Разработчики могут предоставить собственную реализацию этого интерфейса или использовать один из нескольких встроенных механизмов HTTP-аутентификации.
`HttpMessageContext`	Интерфейс, представляющий параметры, передаваемые в/из методов `HttpAuthenticationMechanism` во время выполнения.
`AuthenticationParameters`	Класс, который содержит параметры, переданные методу `SecurityContext.authenticate()`.
`HttpMessageContextWrapper`	Разработчики абстрактных классов могут расширить возможности для настройки поведения `HttpMessageContext`.

Пакет `jakarta.security.enterprise.authentication.mechanism.http` также включает ряд классов аннотаций, которые используются для настройки/включения встроенных механизмов аутентификации или для изменения поведения механизма аутентификации.

Таблица 53-3 Основные классы и интерфейсы в `jakarta.security.enterprise.credential`

Класс или интерфейс	Описание
`Учётные данные`	Интерфейс, который представляет общие учётные данные и определяет несколько методов для работы с учётными данными. Все остальные классы в этом пакете являются реализациями интерфейса Credential.
`AbstractClearableCredential`	Абстрактный класс, реализующий поведение, общее для всех типов учётных данных, значение которых поддерживает очистку.
`BasicAuthenticationCredential`	Класс, расширяющий `UsernamePasswordCredential` для представления учётных данных, используемых базовой аутентификацией HTTP.
`CallerOnlyCredential`	Учётные данные, которые содержат только имя вызывающего субъекта. Может использоваться для подтверждения личности, но не для аутентификации пользователя из-за отсутствия каких-либо секретных или других учётных данных, которые могут использоваться для проверки.
`Пароль`	Класс, представляющий текстовый пароль.
`RememberMeCredential`	Класс, представляющий учётные данные, представленные в виде токена, для явного использования с функцией remember me из Jakarta Security.
`UsernamePasswordCredential`	Класс, представляющий учётные данные, обычно используемые аутентификацией по имени пользователя и его паролю.

Таблица 53-4 Основные классы и интерфейсы в `jakarta.security.enterprise.identitystore`

Класс или интерфейс	Описание
`IdentityStore`	Интерфейс, представляющий хранилище идентификаторов. Разработчики могут предоставить собственную реализацию этого интерфейса или использовать одно из встроенных хранилищ идентификаторов.
`IdentityStoreHandler`	Интерфейс, который определяет методы, используемые приложениями для взаимодействия с хранилищами идентификаторов. Приложения могут использовать встроенный IdentityStoreHandler или предоставлять собственную реализацию, если требуется поведение, отличающееся от поведения встроенного.
`PasswordHash`	Интерфейс, определяющий методы для генерации и проверки хэшей паролей, необходимых для безопасной проверки паролей при использовании встроенного хранилища идентификаторов базы данных. Разработчики могут реализовать этот интерфейс для генерации/проверки хэшей паролей с использованием любого желаемого алгоритма.
`Pbkdf2PasswordHash`	Интерфейс маркера реализуется во встроенной реализации PBKDF2 PasswordHash. Разработчики могут использовать этот интерфейс для выбора встроенного алгоритма PBKDF2 при настройке хранилища идентификаторов базы данных.
`RememberMeIdentityStore`	Интерфейс, определяющий специальный тип хранилища идентификаторов, используемый в сочетании с аннотацией RememberMe и обеспечивающий поведение RememberMe в приложении.
`CredentialValidationResult`	Класс, представляющий результат попытки проверить учётные данные.
`IdentityStorePermission`	Требуется разрешение для вызова метода `getGroups` в IdentityStore, когда настроен SecurityManager.

Обзор интерфейса механизма аутентификации HTTP

Интерфейс `HttpAuthenticationMechanism` определяет SPI для записи механизмов аутентификации, которые могут быть предоставлены приложением и развёрнуты с использованием CDI. Разработчики могут написать свои собственные реализации `HttpAuthenticationMechanism` для поддержки определённых типов токенов аутентификации или протоколов. Существует также несколько встроенных механизмов аутентификации, которые выполняют аутентификацию BASIC, FORM и Custom FORM.

Встроенные механизмы аутентификации включаются и настраиваются с помощью одной из следующих аннотаций:

- `BasicAuthenticationMechanismDefinition` — реализует BASIC аутентификацию, которая соответствует поведению контейнера сервлета, когда BASIC <auth-method> объявлен в web.xml.
- `FormAuthenticationMechanismDefinition` — реализует проверку подлинности в форме, соответствующей поведению контейнера сервлетов при объявлении формы <auth-method> в файле web.xml.
- `CustomFormAuthenticationMechanismDefinition` — реализует модифицированную версию аутентификации FORM, в которой пользовательская обработка заменяет POST на j_security_check.

Реализация HttpAuthenticationMechanism должна быть компонентом CDI, который должен находиться контейнером и развёртывается во время выполнения, и предполагается, что он имеет нормальную область видимости. Для обнаружения бина контейнер сервлета ищет бин, который реализует `HttpAuthenticationMechanism` — он должен быть только один для каждого приложения — и, если он находится, организует его развёртывание для аутентификации вызывающих приложение субъектов.

Контейнер сервлетов использует аутентификацию Jakarta для развёртывания механизмов аутентификации. Контейнер предоставляет модуль проверки подлинности сервера аутентификации (SAM — Server Auth Module) Jakarta, который может делегировать `HttpAuthenticationMechanism`, и организует регистрацию этого «моста» SAM в Jakarta `Authentication AuthConfigFactory`. Во время выполнения обычная обработка Jakarta Authentication вызывает мост SAM, который затем делегирует `HttpAuthenticationMechanism` для выполнения аутентификации и управления любым необходимым диалогом с вызывающим субъектом или с третьими лицами, участвующими в потоке протокола аутентификации.

Интерфейс HttpAuthenticationMechanism определяет следующие три метода, которые соответствуют трём методам, определённым интерфейсом Jakarta Authentication ServerAuth. Когда один из методов Jakarta Authentication вызывается у моста SAM, он делегирует соответствующий метод `HttpAuthenticationMechanism`. Хотя имена методов идентичны, сигнатуры методов не совпадают. SAM мост сопоставляет параметры, переданные ему инфраструктурой Jakarta Authentication, и параметры, ожидаемые `HTTPAuthenticationMechanism`.

- `validateRequest()` — проверяет входящий запрос и аутентифицирует вызывающего субъекта.
- `secureResponse()` — (необязательно, если реализация по умолчанию устраивает) защищает сообщение ответа.
- `cleanSubject()` — (необязательно, если реализация по умолчанию устраивает) очищает объекты принципала и учётных данных.

Только метод `validateRequest()` должен быть реализован с помощью `HttpAuthenticationMechanism`. Интерфейс включает в себя реализации по умолчанию для `secureResponse()` и `cleanSubject()`, реализация которых обычно устраивает.

Следующие аннотации можно использовать для добавления дополнительных поведений в `HttpAuthenticationMechanism`:

- `AutoApplySession` — указывает, что функция Jakarta Authentication `registerSession` должна быть включена таким образом, чтобы аутентифицированная личность вызывающего субъекта сохранялась в его сессии сервлета.
- `LoginToContinue` — механизм указания свойств для входа в систему FORM — страницы входа, страницы ошибок и т. д. Встроенные механизмы аутентификации FORM используют LoginToContinue для настройки необходимых параметров.
- `RememberMe` — указывает, что хранилище идентификаторов `RememberMe` должно использоваться для включения функциональности `RememberMe` в механизм аутентификации.

Обзор интерфейсов хранилища идентификаторов

Интерфейсы хранилища идентификаторов описаны в следующих разделах:

- Интерфейс IdentityStore
- Интерфейс RememberMeIdentityStore

Интерфейс IdentityStore

Интерфейс `IdentityStore` определяет SPI для взаимодействия с хранилищами идентификаторов, которые являются каталогами или базами данных, содержащими информацию об учётных записях пользователей. Реализация интерфейса `IdentityStore` может проверять учётные данные пользователей, предоставлять информацию о группах, к которым они принадлежат, или обоих. Чаще всего реализация `IdentityStore` будет взаимодействовать с внешним хранилищем идентификаторов — например, с сервером LDAP — для выполнения фактической проверки учётных данных и поиска групп, но `IdentityStore` также может управлять данными учётной записи пользователя.

Существует две встроенных реализации `IdentityStore` : хранилище идентификаторов LDAP и хранилище идентификаторов базы данных. Эти хранилища идентификаторов используют внешние хранилища, которые уже должны существовать. Реализации IdentityStore не предоставляют внешних хранилищ и не управляют ими. Для них настроены параметры связи с внешним хранилищем, используя следующие аннотации:

- `LdapIdentityStoreDefinition` — настраивает хранилище идентификаторов с параметрами, необходимыми для связи с внешним сервером LDAP, проверки учётных данных пользователя и/или поиска групп пользователей.
- `DatabaseIdentityStoreDefinition` — настраивает хранилище идентификаторов с параметрами, необходимыми для подключения к внешней базе данных, проверки учётных данных пользователя и/или поиска групп пользователей. Вы должны предоставить реализацию PasswordHash при настройке хранилища идентификаторов базы данных. Смотрите Интерфейс PasswordHash.

Приложение может предоставить собственное хранилище идентификаторов или использовать одно из встроенных хранилищ. Примеры обоих типов:

- Выполнение примера встроенного хранилища идентификаторов базы данных
- Запуск примера кастомного хранилища идентификаторов

Реализация `IdentityStore` должна быть компонентом CDI, который находится контейнером и развёртывается во время выполнения, и предполагается, что он имеет нормальную область видимости. Объекты IdentityStore прежде всего предназначены для использования реализациями `HttpAuthenticationMechanisms` , но это не является обязательным требованием. Они могут также использоваться другими типами механизмов аутентификации или контейнерами.

Может присутствовать несколько реализаций `IdentityStore` . Если это так, они вызываются под управлением `IdentityStoreHandler` .

IdentityStoreHandler

Механизмы аутентификации не взаимодействуют напрямую с `IdentityStore` . Вместо этого они вызывают `IdentityStoreHandler` . Реализация интерфейса `IdentityStoreHandler` предоставляет единственный метод `validate(Credential)` , который при вызове выполняет итерации по доступным объектам IdentityStore и возвращает агрегированный результат. `IdentityStoreHandler` также должен быть компонентом CDI, и предполагается, что он имеет нормальную область видимости. Во время выполнения механизм аутентификации инъецирует `IdentityStoreHandler` и вызывает его. `IdentityStoreHandler` , в свою очередь, просматривает доступные объекты IdentityStore и вызывает их для определения совокупного результата.

Существует встроенный `IdentityStoreHandler` , который реализует стандартный алгоритм, определённый Jakarta Security. Спецификация Jakarta Security содержит полное описание алгоритма, но её можно резюмировать следующим образом:

- Перебирайте доступные объекты IdentityStore в порядке приоритета до тех пор, пока предоставленные учётные данные не будут проверены или пока не останется больше объектов IdentityStore.
- Если учётные данные были проверены, выполните итерацию доступных объектов IdentityStore, предоставляющих группы, в порядке приоритетов, агрегируя группы, возвращаемые каждым хранилищем.
- Вернуть подтверждённого пользователя и информацию о группе.

Приложение может также предоставить свой собственный `IdentityStoreHandler`, который может использовать любой желаемый алгоритм для выбора и вызова в объектах IdentityStore и возврата агрегированного (или неагрегированного) результата.

Методы интерфейса IdentityStore

Сам интерфейс IdentityStore имеет четыре метода:

- `validate(Credential)` — проверяет учётные данные и возвращает результат этой проверки.
- `getCallerGroups(CredentialValidationResult)` — возвращает группы, связанные с вызывающим субъектом, указанным в `CredentialValidationResult`, который представляет собой результат предыдущей успешной проверки.
- `validationTypes()` — возвращает набор типов проверки (один или несколько из `VALIDATE`, `PROVIDE_GROUPS`), которые указывают операции, поддерживаемые этим объектом `IdentityStore`.
- `priority()` — возвращает положительное целое число, представляющее самообъявленный приоритет данного IdentityStore. Более низкие значения представляют более высокий приоритет.

Поскольку `getCallerGroups()` является конфиденциальной операцией — она может возвращать информацию о произвольных пользователях и не требует, чтобы вызывающий субъект предоставлял учётные данные пользователя или иным образом подтверждал свою личность — вызывающий субъект должен иметь разрешение `IdentityStorePermission("getGroups")`. Осуществление этой проверки зависит от реализации метода `getCallerGroups()`. Встроенные IdentityStore проверяет это разрешение, если SecurityManager настроен, а встроенный IdentityStoreHandler вызывает метод `getCallerGroups()` в контексте блока `PrivilegedAction`.

Интерфейс PasswordHash

В отличие от некоторых типов хранилищ идентификаторов, например каталогов LDAP, базы данных могут хранить и извлекать пароли пользователей, но не могут проверять их в исходном виде. Поэтому встроенное хранилище идентификаторов базы данных должно самостоятельно проверять пароли пользователей. Чаще всего это включает в себя создание хэша пароля пользователя для сравнения со значением хэш-функции, хранящимся в базе данных.

Чтобы обеспечить максимальную гибкость и совместимость, хранилище идентификаторов базы данных не реализует какие-либо конкретные алгоритмы хэширования паролей. Вместо этого он определяет интерфейс `PasswordHash` и ожидает, что приложение предоставит реализацию `PasswordHash`, которая может проверять пароли из определённого хранилища, которое будет использовать приложение. Реализация `PasswordHash` должна быть доступна в виде компонента с областью видимости dependent и настраиваться путём предоставления полного имени нужного типа в виде значения `hashAlgorithm` для `DatabaseIdentityStoreDefinition`.

Алгоритм `PasswordHash` определяет три метода:

- `initialize(Map<String, String> parameters)` — инициализирует PasswordHash с предоставленным отображением (Map) параметров. Хранилище идентификаторов базы данных вызывает этот метод при инициализации, передавая значение `hashAlgorithmParameters` аннотации `@DatabaseIdentityStoreDefinition` (после преобразования в отображение Map).
- `verify(char[] password, String hashedPassword)` — проверяет предоставленный вызывающим субъектом пароль по сохранённому хэшу пароля, полученному из базы данных. Значение `hashedPassword` должно быть указано точно так, как оно было возвращено из базы данных.
- `generate(char[] password)` — сгенерировать хэш предоставленного пароля. Возвращаемое значение должно быть отформатировано и закодировано точно так, как оно будет сохранено в базе данных. Хотя полезно генерировать хэш пароля, предоставленного вызывающим субъектом, во время `verify()`, этот метод предназначен в первую очередь для использования приложениями или реализациями `IdentityStore`, которые хотят поддерживать возможность управления паролями и сброса пароля без дублирования кода, используемого для проверки паролей.

Обратите внимание, что, хотя интерфейс ориентирован на хэширование паролей, он также может поддерживать альтернативные подходы, такие как двустороннее шифрование хранимых паролей.

Существует встроенная реализация `Pbkdf2PasswordHash`, которая поддерживает, как следует из названия, хэширование пароля PBKDF2. Она поддерживает несколько параметров, которые управляют генерацией хэш-значений (размер ключа, итерации и т. д. — см. Javadoc), и эти параметры кодируются в результирующее хэш-значение, поэтому хэши можно проверить, даже если в данный момент настроены параметры, отличные от действовавших при создании сохранённого хэша.

Хотя для обеспечения совместимости с устаревшим хранилище идентификаторов, хранящим хэши паролей в формате, отличном от `Pbkdf2PasswordHash`, необходимо создать пользовательский `PasswordHash`, разработчикам следует тщательно рассмотреть вопрос о том, является ли `Pbkdf2PasswordHash` достаточным для новых хранилищ удостоверений. Избегайте написания новой реализации PasswordHash без глубокого понимания связанных с этим криптографических и других соображений безопасности. Некоторые из соображений, касающихся хэширования паролей:

- Требования к хэшированию паролей значительно отличаются от требований к хэшированию в других контекстах. В частности, скорость обычно является преимуществом при генерации хэшей, но при генерации хэшей паролей, чем медленнее, тем лучше — замедлять атаки методом перебора.
- Сравнение сгенерированного хэша с сохранённым хэшем должно занимать постоянное время, независимо от того, успешно оно или неудачно, чтобы не дать злоумышленнику подсказки о значении пароля на основе времени неудачных попыток.
- Новая случайная "соль" должна использоваться каждый раз, когда генерируется новое значение хэша пароля.

Интерфейс RememberMeIdentityStore

Интерфейс `RememberMeIdentityStore` представляет специальный тип хранилища идентификаторов. Он не имеет прямого отношения к интерфейсу `IdentityStore`. То есть он не реализует и не расширяет его. Однако он выполняет аналогичную, хотя и специализированную функцию.

В некоторых случаях приложение хочет «запомнить» аутентифицированную сессию пользователя на длительный период. Например, веб-сайт может помнить вас при посещении и запрашивать пароль только периодически, возможно, раз в две недели, если вы явно не выходите из системы.

RememberMe работает следующим образом:

- При получении запроса от неаутентифицированного пользователя аутентификация осуществляется с помощью `HttpAuthenticationMechanism`, который предоставляется приложением (это необходимо — `RememberMeIdentityStore` можно использовать только в сочетании с предоставленным приложением `HttpAuthenticationMechanism`).
- После аутентификации сконфигурированный `RememberMeIdentityStore` сохраняет информацию об аутентифицированной личности пользователя, чтобы впоследствии её восстановить, и генерирует долгоживущий токен «запомнить меня» для входа в систему, который отправляется обратно клиенту, возможно, как cookie.
- При последующем посещении приложения клиент предоставляет токен для входа. Затем `RememberMeIdentityStore` проверяет токен и возвращает сохранённую идентификационную информацию пользователя, которая затем устанавливается как аутентифицированная идентификационная информация пользователя. Если токен недействителен или срок его действия истёк, он удаляется, пользователь снова проходит обычную аутентификацию и генерируется новый токен входа.

Интерфейс `RememberMeIdentityStore` определяет следующие методы:

- `generateLoginToken(CallerPrincipal caller, Set<String> groups)` — сгенерировать маркер входа для недавно аутентифицированного пользователя и связать его с предоставленной информацией о вызывающем субъекте и группах.
- `removeLoginToken(String token)` — удалить (предположительно истёкший или недействительный) маркер входа в систему и любую связанную информацию о вызывающем субъекте и группах.
- `validate(RememberMeCredential credential)` — проверить предоставленные учётные данные и, если они действительны, вернуть информацию о вызывающем субъекте и группах. (`RememberMeCredential` по сути является просто держателем токена для входа в систему).

Реализация `RememberMeIdentityStore` должна быть компонентом CDI, и предполагается, что она имеет нормальную область видимости. Это настраивается путём добавления аннотации `@RememberMe` к `HttpAuthenticationMechanism` приложения, которая указывает, что используется `RememberMeIdentityStore`, и предоставляет соответствующие параметры конфигурации. Затем Interceptor, предоставленный контейнером, перехватывает вызовы к `HttpAuthenticationMechanism`, при необходимости вызывает `RememberMeIdentityStore` до и после вызовов к механизму аутентификации и гарантирует, что идентификатор пользователя правильно установлен для сессии. Спецификация Jakarta Security предоставляет подробное описание требуемого поведения Interceptor-а.

Реализации `RememberMeIdentityStore` должны позаботиться о безопасном управлении токенами и идентификационной информацией пользователя. Например, токены входа в систему не должны содержать конфиденциальную информацию пользователя, такую как учётные данные или конфиденциальные атрибуты, чтобы избежать раскрытия этой информации, если злоумышленник сможет получить доступ к токену — даже зашифрованный токен потенциально уязвим для злоумышленника с достаточным количеством времени/ресурсов. Точно так же токены должны быть зашифрованы/подписаны везде, где это возможно, и отправляться только по защищённым каналам (HTTPS). Идентификационная информация пользователя, управляемая `RememberMeIdentityStore`, должна храниться как можно безопаснее (сохранность при этом не обязательна — единственное влияние «забытой» сессии состоит в том, что пользователю снова будет предложено войти в систему).

Выполнение примера встроенного хранилища идентификаторов базы данных

Пример, описанный в этом разделе, демонстрирует, как использовать встроенное хранилище идентификаторов базы данных для проверки учётных данных.

Темы включают в себя:

- Обзор примера built-in-db-identity-store
- Запуск built-in-db-identity-store

Обзор примера built-in-db-identity-store

Jakarta Security требует, чтобы контейнер Jakarta EE поддерживал встроенный `IdentityStore` с базой данных. Чтобы выполнить это обязательное требование, `DatabaseIdentityStore` поставляется в комплекте с GlassFish.

В этом примере демонстрируется, как вы можете настроить `DatabaseIdentityStore`, чтобы он указывал на встроенную базу данных, а затем использовал её в качестве IdentityStore. Используемый механизм аутентификации — `BasicAuthenticationMechanism`.

Исходный код для этого примера находится в каталоге *`tut-install`*`/examples/security/security-api/built-in-db-identity-store`.

В следующих разделах высокоуровнево описан процесс настройки `DatabaseIdentityStore`. Обратите внимание, что конфигурация, описанная в этих разделах, уже была завершена в файлах приложения, но приведена здесь, чтобы проиллюстрировать, что нужно сделать, чтобы использовать встроенное хранилище идентификаторов базы данных.

- Определите пользователей и группы в хранилище идентификаторов
- Связывание DatabaseIdentityStore с источником данных по умолчанию
- Укажите механизм аутентификации
- Объявите роли в контейнере сервлетов

Когда запрос, содержащий учётные данные, отправляется приложению, он запускает настроенный механизм проверки подлинности, и проверка подлинности выполняется для `DatabaseIdentityStore`, как определено в приложении.

После проверки подлинности приложение также проверяет роли вызывающего субъекта и отправляет подробности как часть ответа.

Обратите внимание, что в GlassFish, если пользователь предоставляет неверные учётные данные при использовании `BasicAuthenticationMechanism`, то `realmName` представляется пользователю в качестве подсказки.

SHELL

```
curl -I -u Joe http://localhost:8080/built-in-db-identity-store/servlet
Enter host password for user 'Joe':
HTTP/1.1 401 Unauthorized
Server: Eclipse GlassFish  6.0.0
X-Powered-By: Servlet/5.0 JSP/3.0(Eclipse GlassFish  6.0.0  Java/AdoptOpenJDK/1.8)
WWW-Authenticate: Basic realm="file"
Content-Length: 1056
Content-Language:
Content-Type: text/html
```

Определите пользователей и группы в хранилище идентификаторов

В следующей таблице показаны пользователи, пароли и группы, используемые в этом примере.

Пользователь	Пароль	Группа
Joe	secret1	foo, bar
Sam	secret2	foo, bar
Tom	secret2	foo
Sue	secret2	foo

В следующем коде показано, как определить учётные данные и роли, назначенные пользователям в файле `DatabaseSetup.java`.

С аннотацией `@Startup` этот Enterprise-бин-синглтон инициализируется во время запуска приложения, а учётные данные устанавливаются в базу данных.

JAVA

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

import jakarta.annotation.PostConstruct;
import jakarta.annotation.PreDestroy;
import jakarta.annotation.Resource;
import jakarta.annotation.sql.DataSourceDefinition;
import jakarta.ejb.Singleton;
import jakarta.ejb.Startup;
import jakarta.sql.DataSource;

@Singleton
@Startup
public class DatabaseSetup {

    // Источник данных по умолчанию поставляется в составе GlassFish и используется для сохранения реквизитов.
    @Resource(lookup="java:comp/DefaultDataSource")
    private DataSource dataSource;

    @PostConstruct
    public void init() {

        // ...
        executeUpdate(dataSource, "INSERT INTO caller VALUES('Joe', '" +
passwordHash.generate("secret1".toCharArray()) + "')");
        // ...
        executeUpdate(dataSource, "INSERT INTO caller_groups VALUES('Joe', 'foo')");
        executeUpdate(dataSource, "INSERT INTO caller_groups VALUES('Joe', 'bar')");
        // ...
    }

    @PreDestroy
    public void destroy() {
            // ...
    }

    private void executeUpdate(DataSource dataSource, String query) {
        // ...
    }
}
```

Связывание DatabaseIdentityStore с источником данных по умолчанию

Используйте аннотацию `@DatabaseIdentityStoreDefinition` для связи встроенного `DatabaseIdentityStore` с `DefaultDataSource` в файле `ApplicationConfig.java`. Этот пример также демонстрирует использование интерфейса `Pbkdf2PasswordHash`.

JAVA

```
// Определение базы данных для встроенного DatabaseIdentityStore
@DatabaseIdentityStoreDefinition(
    callerQuery = "#{'select password from caller where name = ?'}",
    groupsQuery = "select group_name from caller_groups where caller_name = ?",
    hashAlgorithm = Pbkdf2PasswordHash.class,
    priorityExpression = "#{100}",
    hashAlgorithmParameters = {
        "Pbkdf2PasswordHash.Iterations=3072",
        "${applicationConfig.dyna}"
    }
)

@ApplicationScoped
@Named
public class ApplicationConfig {

  public String[] getDyna() {
       return new String[]{"Pbkdf2PasswordHash.Algorithm=PBKDF2WithHmacSHA512",
"Pbkdf2PasswordHash.SaltSizeBytes=64"};
   }

}
```

Укажите механизм аутентификации

В этом приложении учётные данные проверяются с использованием механизма базовой аутентификации. Укажите аннотацию `@BasicAuthenticationMechanismDefinition` в `ApplicationConfig.java`, чтобы убедиться, что `BasicAuthenticationMechanism` используется для выполнения проверки учётных данных.

Когда выполняется запрос к сервлету, о котором идёт речь, контейнер делегирует его в `org.glassfish.soteria.mechanisms.jaspic.HttpBridgeServerAuthModule`, который затем вызывает метод `BasicAuthenticationMechanism.validateRequest` и получает учётные данные из запроса.

JAVA

```
@BasicAuthenticationMechanismDefinition(
        realmName = "file"
)
```

Объявите роли в контейнере сервлетов

Когда к приложению делается запрос, назначенные пользователю роли возвращаются как часть ответа. Обратите внимание, что контейнер должен быть осведомлён о поддерживаемых ролях, которые определяются с помощью аннотации `@DeclareRoles({"foo", "bar", "kaz"})`, как показано ниже.

JAVA

```
@WebServlet("/servlet")
@DeclareRoles({ "foo", "bar", "kaz" })
@ServletSecurity(@HttpConstraint(rolesAllowed = "foo"))
public class Servlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {

        String webName = null;
        if (request.getUserPrincipal() != null) {
            webName = request.getUserPrincipal().getName();
        }

        response.getWriter().write("web username: " + webName + "\n");

        response.getWriter().write("web user has role \"foo\": " + request.isUserInRole("foo") + "\n");
        response.getWriter().write("web user has role \"bar\": " + request.isUserInRole("bar") + "\n");
        response.getWriter().write("web user has role \"kaz\": " + request.isUserInRole("kaz") + "\n");
    }

}
```

В GlassFish 6.0 сопоставление групп и ролей включено по умолчанию. Поэтому не требуется связывать web.xml с приложением, чтобы обеспечить назначение ролей группам.

Запуск built-in-db-identity-store

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска приложения `built-in-db-identity-store`, как описано в следующих разделах:

- Сборка, упаковка и развёртывание built-in-db-identity-store с IDE NetBeans
- Сборка, упаковка и развёртывание built-in-db-identity-store с использованием Maven
- Запуск built-in-db-identity-store

Сборка, упаковка и развёртывание built-in-db-identity-store с IDE NetBeans

1. Если вы ещё этого не сделали, запустите базу данных по умолчанию. Это необходимо, потому что мы используем DefaultDataSource в комплекте с GlassFish для `DatabaseIdentityStore`. Смотрите Запуск и остановка Apache Derby.
2. Если вы ещё этого не сделали, запустите GlassFish Server. Смотрите Запуск и остановка GlassFish Server.
3. В меню «Файл» выберите «Открыть проект».
4. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/security/security-api
   ```

5. Выберите каталог `built-in-db-identity-store`.
6. Нажмите Открыть проект.
7. На вкладке «Проекты» кликните правой кнопкой мыши проект `built-in-db-identity-store` и выберите «Сборка».

   Эта команда собирает и развёртывает пример приложения в GlassFish Server.

Сборка, упаковка и развёртывание built-in-db-identity-store с использованием Maven

1. Если вы ещё этого не сделали, запустите базу данных по умолчанию. Это необходимо, потому что мы используем DefaultDataSource в комплекте с GlassFish для `DatabaseIdentityStore`. Смотрите Запуск и остановка Apache Derby.
2. Если вы ещё этого не сделали, запустите GlassFish Server. Смотрите Запуск и остановка GlassFish Server.
3. В окне терминала перейдите в:

   ```
   tut-install/examples/security/security-api/built-in-db-identity-store
   ```

4. Введите следующую команду:

   ```
   mvn install
   ```
   SHELL

   Эта команда собирает и упаковывает приложение в WAR-файл `built-in-db-identity-store.war`, который находится в каталоге `target`, а затем развёртывает этот WAR-файл.

Запуск built-in-db-identity-store

В этом примере используйте учётные данные пользователя Joe, чтобы сделать запрос и проверить ответ в соответствии с учётными данными/ролями, определёнными в `DatabaseSetup.java`.

1. Сделайте запрос к развёрнутому приложению, введя следующий URL запроса в вашем веб-браузере:

   URL запроса:

   ```
   http://localhost:8080/built-in-db-identity-store/servlet
   ```

   Поскольку здесь используется BASIC-аутентификация, контейнер отвечает обратно, запрашивая имя пользователя и пароль.

2. Введите имя пользователя `Joe` и пароль `secret1` в командной строке.

   После предоставления учётных данных происходит следующий процесс:

   - Клиент представляет запрос контейнеру со строкой в кодировке base64 и с заголовком `Authorization`, используя значение в формате, ожидаемом для базовой аутентификации.
   - Когда имя пользователя и пароль доступны для контейнера, проверка выполняется в `DatabaseIdentityStore`.
   - Соответствующий объект `UsernamePasswordCredential` передаётся в качестве параметра методу `DatabaseIdentityStore.validate()`.
   - Пароль извлекается из базы данных для пользователя Joe.
   - Пароль, хранящийся в базе данных, хэшируется с использованием алгоритма `PBKDF2` и проверяется встроенной реализацией `Pbkdf2PasswordHash`.
   - При успешной проверке запрос передаётся соответствующему сервлету, а конечному пользователю возвращается следующий ответ.

     Ответ:

     ```
     web username: Joe
     web user has role "foo": true
     web user has role "bar": true
     web user has role "kaz": false
     ```

3. Проверьте аутентификацию, используя недействительные учётные данные. Сделайте запрос к развёрнутому приложению, введя следующий URL запроса в вашем веб-браузере:

   URL запроса:

   ```
   http://localhost:8080/built-in-db-identity-store/servlet
   ```

   Опять же, поскольку здесь используется BASIC-аутентификация, контейнер отвечает обратно, запрашивая имя пользователя и пароль.

4. Введите неверное имя пользователя и пароль. Вам предлагается снова ввести учётные данные, так как вы не прошли аутентификацию.

   При клике кнопки «Отмена» в окне «Требуется аутентификация» возвращается следующий ответ:

   ```
   HTTP Status 401 – Unauthorized

   type Status report

   message Unauthorized

   description This request requires HTTP authentication.

   Eclipse GlassFish 6.0.0
   ```

Запуск примера кастомного хранилища идентификаторов

Пример, описанный в этом разделе, демонстрирует, как связать и использовать кастомное хранилище идентификаторов в вашем приложении для проверки учётных данных.

Темы включают в себя:

- Обзор примера кастомного хранилища идентификаторов
- Запуск примера custom-identity-store

Обзор примера кастомного хранилища идентификаторов

В качестве альтернативы использованию встроенного хранилища идентификаторов приложение может предоставить свой собственный IdentityStore. При связывании с приложением это кастомное хранилище идентификаторов может использоваться для аутентификации и авторизации.

В этом примере показано, как определить кастомное хранилище идентификаторов `TestIdentityStore` и сделать его частью развёртываемого приложения. Используемый механизм аутентификации — `BasicAuthenticationMechanism`.

Исходный код для этого примера находится в каталоге *`tut-install`*`/examples/security/security-api/custom-identity-store`.

В следующих разделах высокоуровнево описан процесс настройки `TestIdentityStore`. Обратите внимание, что конфигурация, описанная в этих разделах, уже была завершена в файлах приложения, но представлена здесь, чтобы проиллюстрировать, что нужно сделать, чтобы использовать кастомное хранилище идентификаторов.

- Определите пользователей и группы в хранилище идентификаторов
- Укажите механизм аутентификации

- Объявите роли в контейнере сервлетов

Когда в приложение отправляется запрос, содержащий учётные данные, настраиваемый механизм проверки подлинности задействуется, и проверка подлинности выполняется по `TestIdentityStore`, как определено в приложении.

После проверки подлинности приложение также проверяет роли вызывающего субъекта и отправляет подробности как часть ответа.

Обратите внимание, что в GlassFish, если пользователь предоставляет неверные учётные данные при использовании `BasicAuthenticationMechanism`, то `realmName` представляется пользователю в качестве подсказки.

```
curl -I -u Joe http://localhost:8080/custom-identity-store/servlet
Enter host password for user 'Joe':
HTTP/1.1 401 Unauthorized
Server: Eclipse GlassFish  6.0.0
X-Powered-By: Servlet/5.0 JSP/3.0(Eclipse GlassFish  6.0.0  Java/AdoptOpenJDK/1.8)
WWW-Authenticate: Basic realm="file"
Content-Length: 1056
Content-Language:
Content-Type: text/html
```

Определите пользователей и группы в хранилище идентификаторов

В следующей таблице показаны пользователь, пароль и группа, используемые в этом примере.

Пользователь	Пароль	Группа
Joe	secret1	foo, bar

В следующем фрагменте кода показано, как определить учётные данные и роли, назначенные пользователям в файле `TestIdentityStore.java`.

JAVA

```java
if (usernamePasswordCredential.compareTo("Joe", "secret1")) {
    return new CredentialValidationResult("Joe", new HashSet<>(asList("foo", "bar")));
}
```

Укажите механизм аутентификации

В этом приложении учётные данные проверяются с использованием механизма базовой аутентификации. Укажите аннотацию `@BasicAuthenticationMechanismDefinition` в `ApplicationConfig.java`, чтобы убедиться, что `BasicAuthenticationMechanism` используется для выполнения проверки учётных данных.

JAVA

```java
@BasicAuthenticationMechanismDefinition(
        realmName = "file"
)

@ApplicationScoped
@Named
public class ApplicationConfig {

}
```

Объявите роли в контейнере сервлетов

Когда к приложению делается запрос, назначенные пользователю роли возвращаются как часть ответа. Обратите внимание, что контейнер должен быть осведомлён о поддерживаемых ролях, которые определяются с помощью аннотации `@Declareroles({"foo", "bar", "kaz"})`, как показано ниже.

JAVA

```
@DeclareRoles({ "foo", "bar", "kaz" })
@WebServlet("/servlet")
public class Servlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {

        String webName = null;
        if (request.getUserPrincipal() != null) {
            webName = request.getUserPrincipal().getName();
        }

        response.getWriter().write("web username: " + webName + "\n");

        response.getWriter().write("web user has role \"foo\": " + request.isUserInRole("foo") + "\n");
        response.getWriter().write("web user has role \"bar\": " + request.isUserInRole("bar") + "\n");
        response.getWriter().write("web user has role \"kaz\": " + request.isUserInRole("kaz") + "\n");
    }

}
```

В GlassFish 6.0 сопоставление групп и ролей включено по умолчанию. Поэтому не требуется связывать `web.xml` с приложением, чтобы обеспечить сопоставление ролей и групп.

Запуск примера custom-identity-store

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска приложения `custom-identity-store`, как описано в следующих разделах:

- Сборка, упаковка и развёртывание custom-identity-store с IDE NetBeans
- Сборка, упаковка и развёртывание custom-identity-store с использованием Maven
- Запуск custom-identity-store

Сборка, упаковка и развёртывание custom-identity-store с IDE NetBeans

1. Если вы ещё этого не сделали, запустите GlassFish Server. Смотрите Запуск и остановка GlassFish Server.
2. В меню «Файл» выберите «Открыть проект».
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/security/security-api
   ```

4. Выберите каталог `custom-identity-store`.
5. Нажмите Открыть проект.
6. На вкладке «Проекты» кликните правой кнопкой мыши проект `custom-identity-store` и выберите «Сборка».

   Эта команда собирает и развёртывает пример приложения в GlassFish Server.

Сборка, упаковка и развёртывание custom-identity-store с использованием Maven

1. Если вы ещё этого не сделали, запустите GlassFish Server. Смотрите Запуск и остановка GlassFish Server.
2. В окне терминала перейдите в:

   ```
   tut-install/examples/security/security-api/custom-identity-store
   ```

3. Введите следующую команду:

   ```
   mvn install
   ```
   SHELL

   Эта команда собирает и упаковывает приложение в WAR-файл, `custom-identity-store.war`, который находится в каталоге `target`, а затем развёртывает WAR-файл.

Запуск custom-identity-store

В этом примере используйте учётные данные пользователя `Joe`, чтобы сделать запрос и проверить ответ в соответствии с учётными данными, определёнными в `TestIdentityStore`.

1. Сделайте запрос к развёрнутому приложению, используя действительные учётные данные, введя следующий URL запроса в веб-браузере:

   URL запроса:

   ```
   http://localhost:8080/custom-identity-store/servlet?name=Joe&password=secret1
   ```

   Ответ:

   ```
   web username: Joe
   web user has role "foo": true
   web user has role "bar": true
   web user has role "kaz": false
   ```

2. Проверьте аутентификацию, используя недействительные учётные данные. Сделайте запрос к развёрнутому приложению, введя следующий URL запроса в вашем веб-браузере:

   URL запроса:

   ```
   http://localhost:8080/custom-identity-store/servlet?name=Joe&password=secret3
   ```

   Ответ:

   ```
   HTTP Status 401 – Unauthorized

   type Status report

   message Unauthorized

   description This request requires HTTP authentication.

   Eclipse GlassFish 6.0.0
   ```

Глава 54. Безопасность Jakarta EE: дополнительные темы

В этой главе представлена дополнительная информация о защите приложений Jakarta EE.

Работа с цифровыми сертификатами

Цифровые сертификаты для GlassFish Server уже созданы и находятся в каталоге *domain-dir*`/config/`. Эти цифровые сертификаты являются самоподписанными и предназначены для использования в среде разработки. Они не предназначены для использования в производственной среде. В производственной среде создайте сертификаты и подпишите их в Центре сертификации (ЦС).

Чтобы использовать Secure Sockets Layer (SSL), приложение или веб-сервер должны иметь связанный сертификат для каждого внешнего интерфейса или IP-адреса, который принимает безопасные соединения. Теория, лежащая в основе этого дизайна, заключается в том, что сервер должен обеспечивать разумную уверенность в том, что его владелец является тем, за кого он себя выдаёт, особенно до получения какой-либо конфиденциальной информации. Может быть полезно рассматривать сертификат как «цифровое водительское удостоверение» для интернет-адреса. В сертификате указано, с какой компанией связан сайт, а также основные контактные данные о владельце сайта или администраторе.

Цифровой сертификат криптографически подписан его владельцем, и его трудно подделать кому-либо ещё. Для сайтов, участвующих в электронной коммерции или в любой другой бизнес-транзакции, в которой важна аутентификация личности, сертификат можно приобрести в известном центре сертификации, таком как VeriSign или Thawte. Если ваш серверный сертификат самоподписан, вы должны установить его в файл хранилища ключей GlassFish Server (`keystore.jks`). Если ваш клиентский сертификат самоподписан, вы должны установить его в файл хранилища доверенных сертификатов GlassFish Server (`cacerts.jks`).

Иногда проверка подлинности не является проблемой. Например, администратор может просто захотеть убедиться, что данные, передаваемые и получаемые сервером, являются частными и не могут быть отслежены кем-либо, подслушивающим соединение. В таких случаях вы можете сэкономить время и деньги, связанные с получением сертификата CA, и использовать самоподписанный сертификат.

SSL использует асимметричную криптографию, которая основана на парах ключей. Пары ключей содержат один открытый и один закрытый ключ. Данные, зашифрованные одним ключом, могут быть расшифрованы только другим ключом пары. Это свойство имеет основополагающее значение для установления доверия и конфиденциальности в сделках. Например, используя SSL, сервер вычисляет значение и шифрует его, используя свой закрытый ключ. Зашифрованное значение называется цифровой подписью. Клиент расшифровывает зашифрованное значение с помощью открытого ключа сервера и сравнивает значение с его собственным вычисленным значением. Если два значения совпадают, клиент может доверять подлинности подписи, поскольку для создания такой подписи мог использоваться только закрытый ключ.

Цифровые сертификаты используются с HTTPS для аутентификации веб-клиентов. Служба HTTPS большинства веб-серверов не будет работать, если не установлен цифровой сертификат. Используйте процедуру, описанную в разделе Создание сертификата сервера, чтобы настроить цифровой сертификат, который может использоваться вашим приложением или веб-сервером для включения SSL.

Одним из инструментов, который можно использовать для настройки цифрового сертификата, является `keytool` — утилита управления ключами и сертификатами, которая поставляется вместе с JDK. Этот инструмент позволяет пользователям управлять своими парами открытого/закрытого ключей и связанными сертификатами для использования при самостоятельной аутентификации, посредством чего пользователь

аутентифицирует себя для других пользователей или служб или служб целостности данных и аутентификации, используя цифровые подписи. Этот инструмент также позволяет пользователям кэшировать открытые ключи своих партнёров в виде сертификатов.

Для лучшего понимания `keytool` и асимметричной криптографии см. Дополнительная информация о разделах повышенной безопасности для ссылки на документацию `keytool` .

Создание серверного сертификата

Серверный сертификат уже создан для GlassFish Server и его можно найти в каталоге *`domain-dir`*`/config/` . Серверный сертификат находится в `keystore.jks` . Файл `cacerts.jks` содержит все доверенные сертификаты, включая клиентские сертификаты.

При необходимости вы можете использовать `keytool` для генерации сертификатов. Утилита `keytool` хранит ключи и сертификаты в файле, называемом хранилищем ключей, хранилищем сертификатов, используемых для идентификации клиента или сервера. Как правило, хранилище ключей — это файл, который содержит идентификационные данные одного клиента или одного сервера. Хранилище ключей защищает закрытые ключи с помощью пароля.

Если вы не указали каталог при указании имени файла хранилища ключей, хранилища ключей создаются в каталоге, из которого выполняется команда `keytool` . Это может быть каталог, в котором находится приложение, или каталог, общий для нескольких приложений.

Основные шаги для создания серверного сертификата следующие.

1. Создайте хранилище ключей.
2. Экспортируйте сертификат из хранилища ключей.
3. Подпишите сертификат.
4. Импортируйте сертификат в хранилище доверенных сертификатов: хранилище сертификатов, используемое для проверки сертификатов. Хранилище доверенных сертификатов обычно содержит несколько сертификатов.

В следующем разделе представлена конкретная информация об использовании утилиты `keytool` для выполнения этих действий.

Использование keytool для создания серверного сертификата

Запустите `keytool` , чтобы сгенерировать новую пару ключей в файле хранилища ключей разработки по умолчанию, `keystore.jks` . В этом примере используется псевдоним `server-alias` , чтобы сгенерировать новую пару открытого/закрытого ключа и включить открытый ключ в самоподписанный сертификат внутри `keystore.jks` . Пара ключей генерируется с использованием алгоритма типа RSA с паролем по умолчанию `changeit` . Дополнительные сведения и другие примеры создания файлов хранилища ключей и управления ими см. в документации `keytool` .

RSA — это технология шифрования с открытым ключом, разработанная RSA Data Security, Inc.

Из каталога, в котором вы хотите создать пару ключей, запустите `keytool` , как показано в следующих шагах.

1. Создайте серверный сертификат.

   Введите команду `keytool` в одну строку:

SHELL

```
java-home/bin/keytool -genkey -alias server-alias -keyalg RSA
-keypass changeit -storepass changeit -keystore keystore.jks
```

При нажатии Enter, `keytool` предлагает ввести имя сервера, организационную единицу, организацию, местность, штат и код страны.

Вы должны ввести имя сервера в ответ на первое приглашение `keytool`, в котором оно запрашивает имя и фамилию. В целях тестирования это может быть `localhost`.

2. Экспортируйте созданный сертификат сервера в `keystore.jks` в файл `server.cer`.

   Введите команду `keytool` в одну строку:

SHELL

```
java-home/bin/keytool -export -alias server-alias -storepass changeit
-file server.cer -keystore keystore.jks
```

3. Если вы хотите, чтобы сертификат был подписан центром сертификации, прочитайте пример в документации `keytool`.
4. Чтобы добавить серверный сертификат в файл хранилища доверенных сертификатов `cacerts.jks`, запустите `keytool` из каталога, в котором вы создали хранилище ключей и сертификат сервера.

   Используйте следующие параметры:

SHELL

```
java-home/bin/keytool -import -v -trustcacerts -alias server-alias
-file server.cer -keystore cacerts.jks -keypass changeit
-storepass changeit
```

   Появится информация о сертификате, такая как показано ниже:

```
Owner: CN=localhost, OU=My Company, O=Software, L=Santa Clara, ST=CA, C=US
Issuer: CN=localhost, OU=My Company, O=Software, L=Santa Clara, ST=CA, C=US
Serial number: 3e932169
Valid from: Mon Nov 26 18:15:47 EST 2012 until: Sun Feb 24 18:15:47 EST 2013
Certificate fingerprints:
         MD5: 52:9F:49:68:ED:78:6F:39:87:F3:98:B3:6A:6B:0F:90
         SHA1: EE:2E:2A:A6:9E:03:9A:3A:1C:17:4A:28:5E:97:20:78:3F:
         SHA256: 80:05:EC:7E:50:50:5D:AA:A3:53:F1:11:9B:19:EB:0D:20:67:C1:12:
AF:42:EC:CD:66:8C:BD:99:AD:D9:76:95
         Signature algorithm name: SHA256withRSA         Version: 3
         ...
Trust this certificate? [no]:
```

5. Введите `yes`, затем нажмите клавишу Enter или Return.

   Появится следующая информация:

```
Certificate was added to keystore
[Storing cacerts.jks]
```

Добавление пользователей в область безопасности certificate

В области `certificate` идентификатор пользователя устанавливается в контексте безопасности GlassFish Server и заполняется данными пользователя, полученными из криптографически проверенных клиентских сертификатов. Пошаговые инструкции по созданию сертификата этого типа см. в разделе Работа с цифровыми сертификатами.

Использование другого серверного сертификата с GlassFish Server

Следуйте инструкциям в Создание серверного сертификата, чтобы создать собственный серверный сертификат, подписать его центром сертификации и импортировать сертификат в `keystore.jks`.

Убедитесь, что при создании сертификата вы соблюдаете следующие правила.

- При создании серверного сертификата `keytool` предлагает ввести ваше имя и фамилию. В ответ на это приглашение нужно ввести имя сервера. В целях тестирования это может быть `localhost`.
- При желании заменить существующий `keystore.jks`, нужно изменить пароль своего хранилища ключей на пароль по умолчанию (`changeit`), либо изменить пароль по умолчанию на пароль хранилища ключей.

Указание другого серверного сертификата

Чтобы указать, что GlassFish Server должен использовать новое хранилище ключей для решений по аутентификации и авторизации, необходимо установить параметры JVM для GlassFish Server, чтобы они распознавали новое хранилище ключей. Чтобы использовать хранилище ключей, отличное от предоставленного в целях разработки, выполните следующие действия.

1. Запустите GlassFish Server, если вы этого ещё не сделали. Информацию о запуске GlassFish Server можно найти в Запуск и остановка GlassFish Server.
2. Откройте Консоль администрирования GlassFish Server в браузере по ссылке http://localhost:4848.
3. Разверните «Конфигурации», затем разверните «server-config» и нажмите «Настройки JVM».
4. Перейдите на вкладку «Параметры JVM».
5. Измените следующие параметры JVM, чтобы они указывали на местоположение и имя нового хранилища ключей. Текущие настройки показаны ниже:

   ```
   -Djavax.net.ssl.keyStore=${com.sun.aas.instanceRoot}/config/keystore.jks
   -Djavax.net.ssl.trustStore=${com.sun.aas.instanceRoot}/config/cacerts.jks
   ```

6. Если пароль хранилища ключей был изменён и отличается от значения по умолчанию, необходимо также добавить параметр с новым паролем:

   ```
   -Djavax.net.ssl.keyStorePassword=your-new-password
   ```

7. Нажмите Сохранить и перезапустите GlassFish Server.

Механизмы аутентификации

В этом разделе рассматриваются механизмы аутентификации клиента по цифровому сертификату и взаимной аутентификации.

Аутентификация клиента по цифровому сертификату

При аутентификации клиента по цифровому сертификату веб-сервер аутентифицирует клиента с помощью сертификата открытого ключа клиента. Аутентификация клиента по цифровому сертификату является более безопасным методом аутентификации, чем базовая аутентификация или аутентификация на основе форм. Она использует HTTP over SSL (HTTPS), при котором сервер аутентифицирует клиента, используя сертификат открытого ключа клиента. Технология SSL обеспечивает шифрование данных, проверку подлинности сервера, целостность сообщений и, опционально, аутентификацию клиента по цифровому сертификату для соединения TCP/IP. Вы можете думать о сертификате открытого ключа как о цифровом эквиваленте паспорта. Сертификат выдаётся доверенной организацией — центром сертификации (CA) и обеспечивает идентификацию для канала-носителя.

Перед использованием аутентификации клиента по цифровому сертификату убедитесь, что у клиента есть действующий сертификат открытого ключа. Для получения дополнительной информации о создании и использовании сертификатов открытых ключей прочитайте Работа с цифровыми сертификатами.

В следующем примере показано, как объявить аутентификацию клиента по цифровому сертификату в дескрипторе развёртывания:

```xml
<login-config>
    <auth-method>CLIENT-CERT</auth-method>
</login-config>
```

Jakarta Security предоставляет альтернативные средства для настройки аутентификации клиента с помощью интерфейса `HttpAuthenticationMechanism`. Этот интерфейс определяет SPI для записи механизмов аутентификации, которые могут быть предоставлены приложением и развёрнуты с использованием CDI. Смотрите Обзор интерфейса механизма аутентификации HTTP.

Взаимная аутентификация

При взаимной аутентификации сервер и клиент аутентифицируют друг друга. Взаимная аутентификация бывает двух типов:

- На основе сертификатов (см. рисунок 54-1)
- На основе имени пользователя/пароля (см. рисунок 54-2)

При использовании взаимной аутентификации на основе сертификатов выполняются следующие действия.

1. Клиент запрашивает доступ к защищённому ресурсу.
2. Веб-сервер представляет свой сертификат клиенту.
3. Клиент проверяет сертификат сервера.
4. В случае успеха клиент отправляет свой сертификат на сервер.
5. Сервер проверяет учётные данные клиента.
6. В случае успеха сервер предоставляет доступ к защищённому ресурсу, запрошенному клиентом.

Рисунок 54-1 показывает, что происходит во время взаимной проверки подлинности на основе сертификатов.

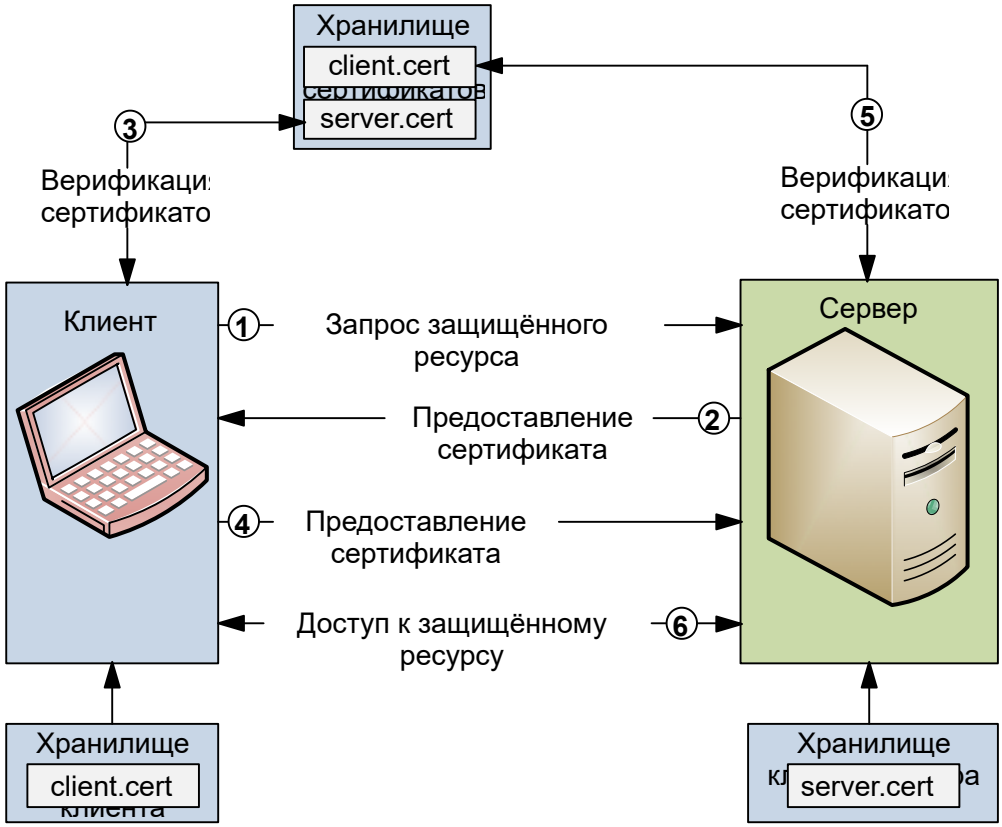

Рис. 54-1 Взаимная проверка подлинности на основе сертификатов

При взаимной аутентификации на основе имени пользователя и пароля выполняются следующие действия.

1. Клиент запрашивает доступ к защищённому ресурсу.
2. Веб-сервер представляет свой сертификат клиенту.
3. Клиент проверяет сертификат сервера.
4. В случае успеха клиент отправляет своё имя пользователя и пароль на сервер.
5. Сервер проверяет учётные данные клиента
6. Если проверка прошла успешно, сервер предоставляет доступ к защищённому ресурсу, запрошенному клиентом.

Рисунок 54-2 показывает, что происходит во время взаимной аутентификации на основе имени пользователя и пароля.

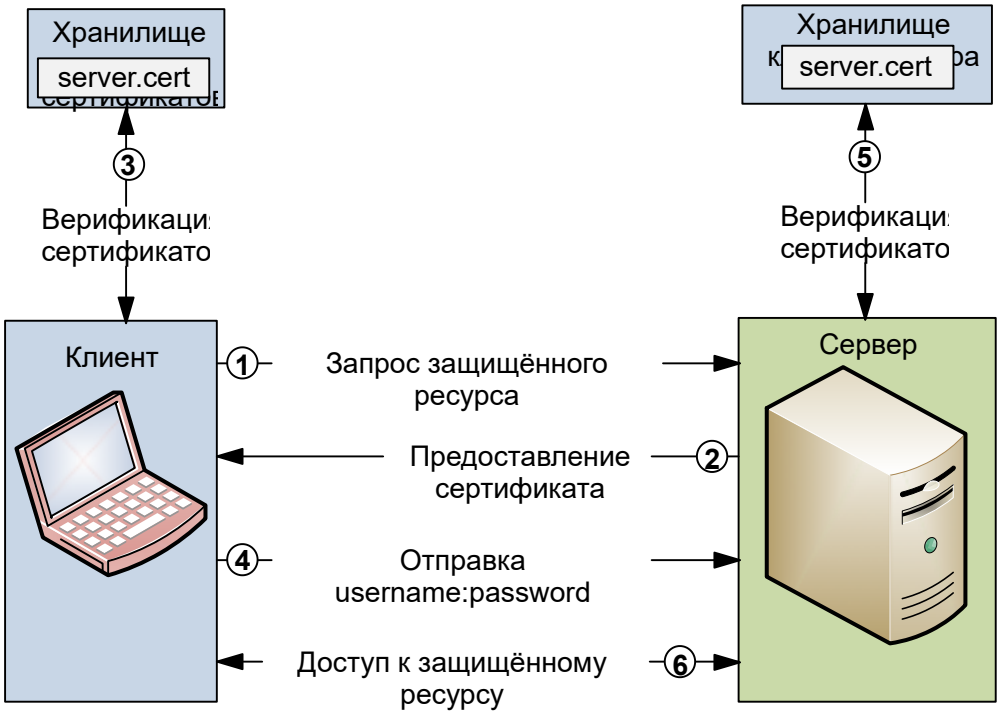

Рис. 54-2 Взаимная проверка подлинности на основе имени пользователя/пароля

Включение взаимной аутентификации по SSL

В этом разделе обсуждается настройка аутентификации на стороне клиента. Включение аутентификации на стороне сервера и на стороне клиента называется взаимной, или двусторонней аутентификацией. При аутентификации клиента по цифровому сертификату клиенты должны предоставлять сертификаты, выданные центром сертификации, который вы решили принять.

Существует как минимум два способа включить взаимную аутентификацию по SSL.

- Предпочтительным методом является установка метода аутентификации в `web.xml` (дескрипторе развёртывания приложения) на `CLIENT-CERT`. Это обеспечивает взаимную аутентификацию путём изменения дескриптора развёртывания данного приложения. Таким образом, аутентификация клиента по цифровому сертификату включена только для определённого ресурса, контролируемого ограничением безопасности, и проверка выполняется только тогда, когда приложение требует такой аутентификации.
- Менее часто используемый метод — установить свойство `clientAuth` в области `certificate` на `true`, если вы хотите, чтобы стек SSL требовал допустимой цепочки сертификатов от клиента, прежде чем принимать соединение. Значение `false` (по умолчанию) не потребует цепочки сертификатов, если только клиент не запросит ресурс, защищённый ограничением безопасности, использующим аутентификацию `CLIENT-CERT`. Когда вы включаете аутентификацию клиента по цифровому сертификату, устанавливая для свойства `clientAuth` значение `true`, такая аутентификация будет требоваться для всех запросов, проходящих через указанный порт SSL. Если вы включите `clientAuth`, он будет включён постоянно, что может серьезно снизить производительность.

Если аутентификация клиента по цифровому сертификату включена обоими этими способами, она будет выполняться дважды.

Создание сертификата клиента для взаимной аутентификации

Если у вас есть сертификат, подписанный доверенным центром сертификации (CA), таким как Verisign, и файл GlassFish Server `cacerts.jks` уже содержит сертификат, проверенный этим центром сертификации, выполнять этот шаг не нужно. Нужно установить свой сертификат в файл сертификатов GlassFish Server только в том случае, когда сертификат самоподписан.

Из каталога, в котором вы хотите создать сертификат клиента, запустите `keytool`, как описано ниже. При нажатии Enter `keytool` предлагает ввести имя сервера, организационную единицу, организацию, местность, регион и код страны.

Вы должны ввести имя сервера в ответ на первое приглашение `keytool`, в котором оно запрашивает имя и фамилию. В целях тестирования это может быть `localhost`. Если в этом примере проверяется взаимная аутентификация, и вы получаете сообщение об ошибке выполнения, в котором говорится, что имя хоста HTTPS неверно, заново создайте клиентский сертификат, убедившись, что используется то же имя хоста, которое будет использоваться при выполнении примера. Например, если имя вашего компьютера — `duke`, введите `duke` в качестве certificate CN или при запросе имени и фамилии. При доступе к приложению введите URL, указывающий на то же местоположение (например, `https://duke:8181/mutualauth/hello`). Это необходимо, потому что во время установления соединения SSL сервер проверяет сертификат клиента, сравнивая имя сертификата с именем хоста, с которого он приходит.

Чтобы создать хранилище ключей с именем `client_keystore.jks`, содержащее сертификат клиента с именем `client.cer`, выполните следующие действия.

1. Создайте резервную копию файла хранилища доверенных сертификатов сервера. Чтобы сделать это,
   a. Перейдите в каталог, содержащий файлы хранилища ключей и хранилища доверенных сертификатов сервера, *`domain-dir`*`\config`.
   b. Скопируйте `cacerts.jks` в `cacerts.backup.jks`.
   c. Скопируйте `keystore.jks` в `keystore.backup.jks`.

   Не помещайте клиентские сертификаты в файл `cacerts.jks`. Любой сертификат, который вы добавляете в файл `cacerts`, может быть доверенным корнем для любой цепочки сертификатов. После завершения экспериментов удалите экспериментальную версию файла `cacerts` и замените её исходной.
2. Сгенерируйте клиентский сертификат. Введите следующую команду из каталога, в котором вы хотите создать сертификат клиента:

   ```
   java-home/bin/keytool -genkey -alias client-alias -keyalg RSA -keypass changeit -storepass changeit -
   keystore client_keystore.jks
   ```
   SHELL
3. Экспортируйте сгенерированный сертификат клиента в файл `client.cer`:

   ```
   java-home/bin/keytool -export -alias client-alias -storepass changeit -file client.cer -keystore
   client_keystore.jks
   ```
   SHELL
4. Добавьте сертификат в файл хранилища доверенных сертификатов *`domain-dir`*`/config/cacerts.jks`. Запустите `keytool` из каталога, в котором вы создали хранилище ключей и клиентский сертификат. Используйте следующие параметры:

   ```
   java-home/bin/keytool -import -v -trustcacerts -alias client-alias
   -file client.cer -keystore domain-dir/config/cacerts.jks
   -keypass changeit -storepass changeit
   ```
   SHELL

   Утилита `keytool` возвращает сообщение, подобное следующему:

```
Owner: CN=localhost, OU=My Company, O=Software, L=Santa Clara, ST=CA, C=US
Issuer: CN=localhost, OU=My Company, O=Software, L=Santa Clara, ST=CA, C=US
Serial number: 3e39e66a
Valid from: Tue Nov 27 12:22:47 EST 2012 until: Mon Feb 25 12:22:47 EST 2013
Certificate fingerprints:
    MD5: 5A:B0:4C:88:4E:F8:EF:E9:E5:8B:53:BD:D0:AA:8E:5A
    SHA1:90:00:36:5B:E0:A7:A2:BD:67:DB:EA:37:B9:61:3E:26:B3:89:46:32
    Signature algorithm name: SHA1withRSA
    Version: 3
Trust this certificate? [no]: yes
Certificate was added to keystore
[Storing cacerts.jks]
```

5. Перезагрузите GlassFish Server.

Использование области безопасности JDBC для аутентификации пользователя

Область безопасности, иногда называемая доменом политики безопасности или доменом безопасности, является областью, в которой сервер приложений определяет и применяет общую политику безопасности. Область содержит список пользователей, которым может быть назначена (или не может быть назначена) группа. Сервер GlassFish поставляется с предустановленными областями безопасности file, certificate, and administration. Администратор также может настроить LDAP, JDBC, дайджест или кастомные области безопасности.

Приложение может указать в своём дескрипторе развёртывания, какую область безопасности использовать. Если приложение не указывает область, GlassFish Server использует свою область по умолчанию — область `file` . Если приложение указывает, что область JDBC должна использоваться для аутентификации пользователя, GlassFish Server будет получать учётные данные пользователя из базы данных. Сервер приложений использует информацию базы данных и включённую опцию области JDBC в файле конфигурации.

База данных предоставляет простой способ добавлять, редактировать или удалять пользователей во время выполнения и позволяет пользователям создавать свои собственные учётные записи без какой-либо административной помощи. Использование базы данных имеет дополнительное преимущество: предоставление места для безопасного хранения любой дополнительной пользовательской информации. Область может рассматриваться как база данных имён пользователей и паролей, которые идентифицируют действительных пользователей веб-приложения или набора веб-приложений с перечислением списка ролей, связанных с каждым пользователем. Доступ к определённым ресурсам веб-приложения предоставляется всем пользователям с определённой ролью вместо перечисления списка связанных пользователей. Имя пользователя может иметь любое количество ролей, связанных с ним.

Два из учебных примеров, глава 62 *Пример Duke's Tutoring* и глава 63 *Пример Duke's Forest* используют область безопасности JDBC для аутентификации пользователей.

Настройка аутентификации области безопасности JDBC

Сервер GlassFish позволяет администраторам указывать учётные данные пользователя (имя пользователя и пароль) в области JDBC, а не в пуле соединений. Это не позволяет другим приложениям просматривать таблицы базы данных для учётных данных пользователя. По умолчанию хранение паролей в виде открытого текста не поддерживается в области JDBC. В обычных условиях пароли не должны храниться в виде открытого текста.

1. Создайте таблицы базы данных, в которых будут храниться учётные данные пользователя для области.
2. Добавьте учётные данные пользователя в созданные таблицы базы данных.

3. Создайте пул соединений JDBC для базы данных.

   Вы можете использовать Консоль администрирования или командную строку для создания пула соединений.

4. Создайте ресурс JDBC для базы данных.

   Вы можете использовать Консоль администрирования или командную строку для создания ресурса JDBC.

5. Создайте область безопасности

   Этот шаг должен связать ресурс с областью, определить таблицы и столбцы для пользователей и групп, используемых для аутентификации, и определить алгоритм дайджеста, который будет использоваться для хранения паролей в базе данных.

   Вы можете использовать Консоль администрирования или командную строку для создания области.

6. Измените дескриптор развёртывания для своего приложения, чтобы указать область JDBC.

   - Для корпоративного приложения в файле EAR измените файл `glassfish-application.xml` .
   - Для веб-приложения в файле WAR измените файл `web.xml` .
   - Для Enterprise-бина в файле JAR EJB измените файл `glassfish-ejb-jar.xml` .

     Например, для гипотетического приложения файл `web.xml` может указывать область `jdbcRealm` следующим образом:

     XML

     ```xml
     <login-config>
         <auth-method>FORM</auth-method>
         <realm-name>jdbcRealm</realm-name>
         <form-login-config>
             <form-login-page>/login.xhtml</form-login-page>
             <form-error-page>/login.xhtml</form-error-page>
         </form-login-config>
     </login-config>
     <security-constraint>
         <web-resource-collection>
             <web-resource-name>Secure Pages</web-resource-name>
             <description/>
             <url-pattern>/admin/*</url-pattern>
         </web-resource-collection>
         <auth-constraint>
             <role-name>ADMINS</role-name>
         </auth-constraint>
     </security-constraint>
     ```

     Имя входа на основе формы указывается для всех веб-страниц в `/admin` . Доступ к этим страницам будет разрешён только пользователям с ролью `ADMINS` .

7. Назначьте роли безопасности пользователям или группам пользователей в области.

   Чтобы назначить роль безопасности группе или пользователю, добавьте элемент `security-role-mapping` в дескриптор развёртывания для конкретного сервера приложений, в данном случае `glassfish-web.xml` :

   XML

   ```xml
   <security-role-mapping>
       <role-name>USERS</role-name>
       <group-name>USERS</group-name>
   </security-role-mapping>
   <security-role-mapping>
       <role-name>ADMINS</role-name>
       <group-name>ADMINS</group-name>
   </security-role-mapping>
   ```

Поскольку пользователи GlassFish Server назначаются в группы в процессе создания пользователя, это более эффективно, чем сопоставление ролей безопасности отдельным пользователям.

Защита ресурсов HTTP

Когда URI запроса соответствует нескольким защищённым шаблонам URL, к запросу применяются те ограничения, которые связаны с шаблоном URL с наилучшим соответствием. Правила сопоставления сервлетов, определённые в главе 12 «Назначение запросов сервлетам» спецификации Jakarta Servlet 5.0, используются для определения наилучшего соответствующего URL-шаблона URI запросу. Никакие требования защиты не применяются к URI запроса, который не соответствует какому-либо защищённому шаблону URL. Метод HTTP запроса не играет роли в выборе наиболее подходящего шаблона URL для запроса.

Когда методы HTTP перечислены в определении ограничения, средства защиты, определённые этим ограничением, применяются только к перечисленным методам.

Когда методы HTTP не перечислены в определении ограничения, средства защиты, определённые этим ограничением, применяются ко всему набору методов HTTP, включая методы расширения HTTP.

Когда ограничения с различными требованиями защиты применяются к одной и той же комбинации URL-шаблонов и методов HTTP, правила объединения требований защиты определены в разделе 13.8.1 «Комбинирование ограничений» спецификации Jakarta Servlet 5.0.

Следуйте этим рекомендациям, чтобы правильно защитить веб-приложение.

- Не перечисляйте методы HTTP в определениях ограничений. Это самый простой способ убедиться, что вы не оставляете методы HTTP незащищёнными. Например:

```xml
<!-- SECURITY CONSTRAINT #1 -->
<security-constraint>
    <display-name>Do not enumerate Http Methods</display-name>
    <web-resource-collection>
        <url-pattern>/company/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>sales</role-name>
    </auth-constraint>
</security-constraint>
```

  Если вы перечислите методы в ограничении, все не перечисленные методы из фактически бесконечного набора возможных методов HTTP, включая методы расширения, будут не защищены. Используйте такое ограничение только при полной уверенности, что это схема защиты, которую вы намереваетесь определить. В следующем примере показано ограничение, которое указывает метод GET и, таким образом, оставляет незащищёнными все остальные HTTP методы:

```xml
<!-- SECURITY CONSTRAINT #2 -->
<security-constraint>
    <display-name>
        Protect GET only, leave all other methods unprotected
    </display-name>
    <web-resource-collection>
        <url-pattern>/company/*</url-pattern>
        <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>sales</role-name>
    </auth-constraint>
</security-constraint>
```

- Если требуется применить определённые типы защиты к определённым HTTP методам, убедитесь, что вы задаёте ограничения, чтобы охватить каждый метод, который хотите разрешить, с или без ограничения, для соответствующих шаблонов URL. Если есть какие-либо методы, которые вы не хотите разрешать, нужно также создать ограничение, которое запрещает доступ к этим методам по тем же шаблонам. Как пример см. ограничение безопасности № 5 в следующем пункте.

  Например, чтобы разрешить GET и POST, когда POST требует аутентификации, а GET разрешён без ограничений, вы можете определить следующие ограничения:

  XML
  ```
  <!-- SECURITY CONSTRAINT #3 -->
  <security-constraint>
      <display-name>Allow unprotected GET</display-name>
      <web-resource-collection>
          <url-pattern>/company/*</url-pattern>
          <http-method>GET</http-method>
      </web-resource-collection>
  </security-constraint>

  <!-- SECURITY CONSTRAINT #4 -->
  <security-constraint>
      <display-name>Require authentication for POST</display-name>
      <web-resource-collection>
          <url-pattern>/company/*</url-pattern>
          <http-method>POST</http-method>
      </web-resource-collection>
      <auth-constraint>
          <role-name>sales</role-name>
      </auth-constraint>
  </security-constraint>
  ```

- Самый простой способ убедиться, что вы запрещаете все HTTP методы, кроме тех, которые хотите разрешить, — это использовать элементы `http-method-omission` , чтобы исключить эти HTTP методы из ограничения безопасности, а также определить `auth-constraint` , который не называет роли. Ограничение безопасности будет применяться ко всем методам, кроме тех, которые были названы в пропусках, и ограничение будет применяться только к ресурсам, соответствующим шаблонам в ограничении.

  Например, следующее ограничение исключает доступ ко всем методам, кроме GET и POST, для ресурсов, соответствующих шаблону `/company/*` :

  XML
  ```
  <!-- SECURITY CONSTRAINT #5 -->
  <security-constraint>
      <display-name>Deny all HTTP methods except GET and POST</display-name>
      <web-resource-collection>
          <url-pattern>/company/*</url-pattern>
          <http-method-omission>GET</http-method-omission>
          <http-method-omission>POST</http-method-omission>
      </web-resource-collection>
      <auth-constraint/>
  </security-constraint>
  ```

  Если вы хотите распространить эти исключения на свободные части вашего приложения, также включите шаблон URL `/` (косая черта):

```xml
<!-- SECURITY CONSTRAINT #6 -->
<security-constraint>
    <display-name>Deny all HTTP methods except GET and POST</display-name>
    <web-resource-collection>
        <url-pattern>/company/*</url-pattern>
        <url-pattern>/</url-pattern>
        <http-method-omission>GET</http-method-omission>
        <http-method-omission>POST</http-method-omission>
    </web-resource-collection>
    <auth-constraint/>
</security-constraint>
```

- Если вы не хотите, чтобы какой-либо ресурс вашего веб-приложения был доступен, пока вы явно не определите ограничение, разрешающее доступ к нему, можете определить элемент `auth-constraint`, который не указывает роли и связывает его с шаблоном URL `/`. Шаблон URL `/` — самый слабый подходящий шаблон. Не перечисляйте никакие методы HTTP в этом ограничении:

```xml
<!-- SECURITY CONSTRAINT #7 -->
<security-constraint>
    <display-name>
        Switch from Constraint to Permission model
        (where everything is denied by default)
    </display-name>
    <web-resource-collection>
        <url-pattern>/</url-pattern>
    </web-resource-collection>
    <auth-constraint/>
</security-constraint>
```

Защита клиентских приложений

Требования к аутентификации Jakarta EE для клиентских приложений такие же, как и для других компонентов Jakarta EE, и могут использоваться те же методы аутентификации, что и для других компонентов приложения Jakarta EE. При доступе к незащищенным веб-ресурсам аутентификация не требуется.

При доступе к защищённым веб-ресурсам могут использоваться обычные варианты аутентификации: базовая аутентификация HTTP, аутентификация с формой входа HTTP или аутентификации клиента по цифровому сертификату SSL. Указание механизма аутентификации в дескрипторе развёртывания описывает, как задать базовую аутентификацию HTTP и аутентификацию с формой входа HTTP. Аутентификация клиента по цифровому сертификату описывает, как задать аутентификацию клиента по сертификату SSL.

Аутентификация требуется при доступе к защищённым Enterprise-бинам. Механизмы аутентификации для Enterprise-бинов обсуждаются в Защита Enterprise-бинов.

Клиентское приложение использует сервис аутентификации, предоставляемый контейнером клиентского приложения, для аутентификации своих пользователей. Сервис контейнера может быть интегрирован с системой аутентификации собственной платформы, так что используется возможность единого входа. Контейнер может аутентифицировать пользователя либо при запуске приложения, либо при доступе к защищённому ресурсу.

Клиентское приложение может предоставить класс, называемый модулем входа в систему, для сбора данных аутентификации. В этом случае должен быть реализован интерфейс `jakarta.security.auth.callback.CallbackHandler`, а имя класса должно быть указано в его дескрипторе развёртывания. Обработчик Callback-вызова приложения должен полностью поддерживать объекты `Callback`, указанные в пакете `jakarta.security.auth.callback`.

Использование модулей входа

Клиентское приложение может использовать сервис аутентификации и авторизации Java (JAAS) для создания модулей входа для аутентификации. Приложение на основе JAAS реализует интерфейс `jakarta.security.auth.callback.CallbackHandler` , чтобы оно могло взаимодействовать с пользователями для ввода определённых данных аутентификации, таких как имена пользователей или пароли, или для отображения сообщений об ошибках и предупреждениях.

Приложения реализуют интерфейс `CallbackHandler` и передают его в контекст входа в систему, который направляет его непосредственно в базовые модули входа в систему. Модуль входа в систему использует обработчик Callback-вызова для сбора ввода, такого как пароль или ПИН-код смарт-карты, от пользователей и для предоставления информации, такой как информация о состоянии, пользователям. Поскольку приложение определяет обработчик Callback-вызова, базовый модуль входа в систему может оставаться независимым от различных способов взаимодействия приложений с пользователями.

Например, реализация обработчика Callback-вызова для приложения с графическим интерфейсом может отображать окно, запрашивающее ввод данных пользователем, или реализация обработчика Callback-вызова для инструмента командной строки может просто запрашивать ввод данных непосредственно из командной строки.

Модуль входа передаёт массив соответствующих Callback-вызовов в метод `handle` обработчика Callback-вызова, такой как `NameCallback` для имени пользователя и `PasswordCallback` для пароля. Обработчик Callback-вызова выполняет запрошенное взаимодействие с пользователем и устанавливает соответствующие значения в Callback-вызовах. Например, чтобы обработать `NameCallback` , `CallbackHandler` может запросить имя, получить значение от пользователя и вызвать метод `setName NameCallback` для хранения имени.

Для получения дополнительной информации об использовании JAAS для аутентификации в модулях входа в систему см. документацию, приведённую в Дополнительная информация о разделах повышенной безопасности.

Использование программного входа

Программный вход в систему позволяет клиентскому коду предоставлять учётные данные пользователя. Если вы используете клиент EJB-компонента, вы можете использовать класс `com.sun.appserv.Security.ProgrammaticLogin` с его удобными методами `login` и `logout` из системы. Программный вход в систему зависит от сервера.

Защита информационных систем (ИС) предприятия

В приложениях информационных систем (ИС) предприятия компоненты запрашивают подключение к ресурсу ИС.

Обзор защиты приложений корпоративных информационных систем

В рамках этого соединения ИС может требовать входа запрашивающего субъекта для доступа к ресурсу. Разработчик компонента приложения может выбрать вариант входа в ИС двумя способами.

- Управляемый контейнером вход: компонент приложения позволяет контейнеру отвечать за настройку и управление входом в ИС. Контейнер определяет имя пользователя и пароль для установления соединения с объектом ИС.
- Вход с управлением компонентами: код прикладного компонента управляет входом в ИС.

Вы также можете настроить безопасность для адаптеров ресурсов. Смотрите Настройка безопасности адаптера ресурса.

Управляемый контейнером вход

При входе, управляемом контейнером, компоненту приложения не нужно передавать какую-либо информацию о безопасности входа в метод `getConnection()`. Информация о безопасности предоставляется контейнером, как показано в следующем примере (вызов метода выделен жирным шрифтом):

JAVA

```
// Бизнес-метод в компоненте приложения
Context initctx = new InitialContext();
// Поиск JNDI для получение фабрики соединений
jakarta.resource.cci.ConnectionFactory cxf =
    (jakarta.resource.cci.ConnectionFactory)initctx.lookup(
    "java:comp/env/eis/MainframeCxFactory");
// Вызов фабрики для получение соединения. Информация
// о безопасности не передаётся в метод getConnection
jakarta.resource.cci.Connection cx = cxf.getConnection();
...
```

Управляемый компонентом вход

При управляемом компонентом входе компонент приложения отвечает за передачу необходимой информации о безопасности входа для ресурса в метод `getConnection`. Например, информация о безопасности может быть именем пользователя и паролем, как показано здесь (вызов метода выделен жирным шрифтом):

JAVA

```
// Метод в компоненте приложения
Context initctx = new InitialContext();

// Поиск JNDI для получения фабрики соединений
jakarta.resource.cci.ConnectionFactory cxf =
    (jakarta.resource.cci.ConnectionFactory)initctx.lookup(
    "java:comp/env/eis/MainframeCxFactory");

// Получение нового ConnectionSpec
com.myeis.ConnectionSpecImpl properties = //..

// Вызов фабрики для получение соединения
properties.setUserName("...");
properties.setPassword("...");
jakarta.resource.cci.Connection cx =
    cxf.getConnection(properties);
...
```

Настройка безопасности адаптера ресурса

Адаптер ресурсов — это программный компонент системного уровня, который обычно реализует сетевое подключение к внешнему менеджеру ресурсов. Адаптер ресурсов может расширять функциональные возможности платформы Jakarta EE либо путём реализации одного из стандартных сервисных API Jakarta EE, например драйвера JDBC, либо путём определения и реализации адаптера ресурса для коннектора с внешней прикладной системой. Адаптеры ресурсов также могут предоставлять сервисы, которые являются полностью локальными, возможно, взаимодействуя с собственными ресурсами. Адаптеры ресурсов взаимодействуют с платформой Jakarta EE через интерфейсы провайдера услуг Jakarta EE (Jakarta EE SPI). Адаптер ресурсов, использующий SPI-интерфейсы Jakarta EE для подключения к платформе Jakarta EE, сможет работать со всеми продуктами Jakarta EE.

Чтобы настроить параметры безопасности для адаптера ресурса, необходимо отредактировать файл дескриптора адаптера ресурса `ra.xml`. Вот пример части файла `ra.xml`, который настраивает свойства безопасности для адаптера ресурса:

```xml
<authentication-mechanism>
    <authentication-mechanism-type>
        BasicPassword
    </authentication-mechanism-type>
    <credential-interface>
        jakarta.resource.spi.security.PasswordCredential
    </credential-interface>
</authentication-mechanism>
<reauthentication-support>false</reauthentication-support>
```

Дополнительные сведения о параметрах настройки безопасности адаптера ресурсов можно узнать, просмотрев *as-install*`/lib/schemas/connector_2_0.xsd`. Вы можете настроить следующие элементы в файле дескриптора развёртывания адаптера ресурса.

- Механизмы аутентификации: используйте элемент `authentication-mechanism`, чтобы указать механизм аутентификации, поддерживаемый адаптером ресурсов. Эта поддержка предназначена для адаптера ресурсов, а не для объекта ИС.

  Существует два поддерживаемых типа механизма:

  - `BasicPassword`, который поддерживает следующий интерфейс:

    ```
    jakarta.resource.spi.security.PasswordCredential
    ```

  - `Kerbv5`, который поддерживает следующий интерфейс:

    ```
    jakarta.resource.spi.security.GenericCredential
    ```

    GlassFish Server в настоящее время не поддерживает этот тип механизма.

- Поддержка повторной аутентификации: используйте элемент `reauthentication-support`, чтобы указать, поддерживает ли реализация адаптера ресурса повторную аутентификацию существующих объектов `Managed-Connection`. Возможные значения: `true` или `false`.
- Разрешения безопасности: используйте элемент `security-permission`, чтобы указать разрешение безопасности, которое требуется для кода адаптера ресурса. Поддержка разрешений безопасности является необязательной и не поддерживается в текущей версии GlassFish Server. Однако вы можете вручную обновить файл `server.policy`, чтобы добавить соответствующие разрешения для адаптера ресурсов.

  Разрешения безопасности, указанные в дескрипторе развёртывания, отличаются от тех, которые требуются набором разрешений по умолчанию, указанным в спецификации коннекторов.

  Для получения дополнительной информации о реализации спецификации разрешений безопасности см. документацию по файлу политики безопасности, приведённую в разделе Дополнительная информация о разделах повышенной безопасности.

Помимо указания защиты адаптера ресурса в файле `ra.xml`, вы можете создать отображение безопасности для пула соединений коннектора, чтобы сопоставить принципала приложения или группу пользователей с принципалами ИС. Отображение безопасности обычно используется, если один или несколько принципалов ИС используются для выполнения операций (в ИС), инициированных различными принципалами или группами пользователей в приложении.

Отображение принципала приложения на принципалов ИС

При использовании GlassFish Server вы можете использовать отображения безопасности для сопоставления идентификатора вызывающего субъекта приложения (принципала или группы пользователей) с соответствующим принципалом ИС в сценариях на основе транзакций, управляемых контейнером. Когда

принципал приложения инициирует запрос к ИС, GlassFish Server сначала проверяет принципала, используя отображение безопасности, определённое для пула соединений коннектора, для определения соответствующего принципала ИС. Если нет точного соответствия, GlassFish Server использует спецификацию подстановочных знаков, если таковая имеется, для определения соответствующего принципала ИС. Отображения безопасности используются, когда пользователь приложения должен выполнить операцию ИС, которая требует выполнения под определённым принципалом в ИС.

Для работы с отображениями безопасности используйте Консоль администрирования. В Консоли администрирования выполните следующие действия, чтобы перейти на страницу отображений безопасности.

1. В дереве навигации разверните узел Ресурсы.
2. Разверните узел Коннекторы.
3. Выберите узел Коннекторы пулов соединений.
4. На странице «Коннекторы пулов соединений» кликните на имени пула соединений, для которого вы хотите создать отображение безопасности.
5. Нажмите вкладку Отображения безопасности.
6. Нажмите «Создать», чтобы создать новое отображение безопасности для пула соединений.
7. Введите имя, под которым вы будете ссылаться на отображение безопасности, а также другую необходимую информацию.

   Нажмите «Справка» для получения дополнительной информации об отдельных параметрах.

Настройка безопасности с использованием дескрипторов развёртывания

Рекомендуемый способ настройки безопасности в платформе Jakarta EE 8 — аннотации. При желании переопределить параметры безопасности во время развёртывания вы можете использовать для этого элементы безопасности в дескрипторе развёртывания `web.xml`. В этом разделе описывается, как использовать дескриптор развёртывания, чтобы задать базовую аутентификацию и изменить назначение по умолчанию ролей для принципалов.

Настройки безопасности для базовой аутентификации в дескрипторе развёртывания

Элементы дескриптора развёртывания, которые добавляют базовую аутентификацию в пример, сообщают серверу или браузеру выполнить следующие задачи.

- Отправьте стандартное диалоговое окно входа в систему для получения данных об имени пользователя и пароле.
- Убедитесь, что пользователь авторизован для доступа к приложению.
- Если разрешено, отобразите сервлет пользователю.

Следующий пример кода показывает элементы безопасности для дескриптора развёртывания, которые можно использовать в примере базовой проверки подлинности, найденном в каталоге *`tut-install`*`/examples/security/hello2_basicauth/`:

```xml
    <security-constraint>
        <display-name>SecurityConstraint</display-name>
        <web-resource-collection>
             <web-resource-name>WRCollection</web-resource-name>
            <url-pattern>/greeting</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>TutorialUser</role-name>
        </auth-constraint>
        <user-data-constraint>
             <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>file</realm-name>
    </login-config>
    <security-role>
        <role-name>TutorialUser</role-name>
    </security-role>
```

Этот дескриптор развёртывания указывает, что URI запроса `/greeting` могут быть доступны только пользователям, которые ввели свои имена пользователей и пароли и были авторизованы для доступа к этому URL, поскольку они были проверены на наличие у них роли `TutorialUser`. Имя пользователя и пароль будут отправлены по защищённому транспортному протоколу, чтобы предотвратить их считывание при передаче.

Настройки назначения ролей принципалам в дескрипторе развёртывания

Jakarta Security требует, чтобы по умолчанию имена участников группы сопоставлялись с одноимёнными ролями. По умолчанию GlassFish придерживается этого стандарта и предоставляет принципала группы для сопоставления ролей. Реализации стандарта, однако, могут предоставить механизмы для настройки другого значения по умолчанию.

Чтобы сопоставить название роли, разрешённое приложением или модулем, с принципалами (пользователями) и группами, определёнными на сервере, используйте элемент `security-role-mapping` в файле дескриптора развёртывания среды выполнения (`glassfish-application.xml`, `glassfish-web.xml` или `glassfish-ejb-jar.xml`). Запись должна объявлять сопоставление между ролью безопасности, используемой в приложении, и одной или несколькими группами или принципалами, определёнными для соответствующей области GlassFish Server. Пример файла `glassfish-web.xml` показан ниже:

```xml
<glassfish-web-app>
    <security-role-mapping>
        <role-name>DIRECTOR</role-name>
        <principal-name>schwartz</principal-name>
    </security-role-mapping>
    <security-role-mapping>
        <role-name>DEPT-ADMIN</role-name>
        <group-name>dept-admins</group-name>
    </security-role-mapping>
</glassfish-web-app>
```

Название роли может быть сопоставлено с определённым принципалом (пользователем), группой или обоими. Указанные имена принципалов или групп должны быть действительными принципалами или группами в текущей области безопасности по умолчанию GlassFish Server. `role-name` в этом примере должно точно соответствовать `role-name` в элементе `security-role` соответствующего файла `web.xml` или название роли, определённые аннотациями `@DeclareRoles` и/или `@RolesAllowed`.

Дополнительная информация о разделах повышенной безопасности

Для получения дополнительной информации о темах безопасности, рассматриваемых в этой главе, см.

- Документация по команде `keytool` :
  https://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html
- Справочное руководство по Java Authentication and Authorization Service (JAAS):
  https://docs.oracle.com/javase/8/docs/technotes/guides/security/jaas/JAASRefGuide.html
- Руководство разработчика Java Authentication and Authorization Service (JAAS): LoginModule:
  https://docs.oracle.com/javase/8/docs/technotes/guides/security/jaas/JAASLMDevGuide.html
- Документация по синтаксису файла политик безопасности:
  https://docs.oracle.com/javase/8/docs/technotes/guides/security/PolicyFiles.html#FileSyntax

Часть XI: Технологии, поддерживаемые Jakarta EE

Часть XI исследует несколько технологий, которые поддерживаются платформой Jakarta EE.

Глава 55. Транзакции

В этой главе описываются типы транзакций и способы управления ими в разных приложениях.

Обзор транзакций

Типичное корпоративное приложение осуществляет доступ и сохраняет информацию в одной или нескольких базах данных. Поскольку эта информация важна для бизнес-операций, она должна быть точной, актуальной и надёжной. Целостность данных была бы потеряна, если бы нескольким программам было разрешено обновлять одну и ту же информацию одновременно или если система, которая не работала во время обработки бизнес-транзакции, оставляла данные только частично обновлёнными. Предотвращая оба эти случая, программные транзакции обеспечивают целостность данных. Транзакции контролируют одновременный доступ к данным несколькими программами. В случае сбоя системы транзакции следят за тем, чтобы после восстановления данные находились в согласованном состоянии.

Транзакции в приложениях Jakarta EE

В приложении Jakarta EE транзакция представляет собой последовательность действий, которые либо должны все успешно завершиться, либо все изменения каждого действия должны быть отменены. Транзакции заканчиваются фиксацией (коммитом) или откатом.

Jakarta Transactions позволяет приложениям получать доступ к транзакциям независимо от конкретных реализаций. Jakarta Transactions определяет стандартные интерфейсы Java между менеджером транзакций и сторонами, участвующими в системе распределённых транзакций: транзакционным приложением, сервером Jakarta EE и менеджером, который контролирует доступ к совместно используемым ресурсам, затронутым транзакциями.

Jakarta Transactions определяет интерфейс `UserTransaction`, который приложения используют для запуска, фиксации или отката транзакций. Компоненты приложения получают объект `UserTransaction` через поиск JNDI, используя имя `java:comp/UserTransaction` или инъецируя объект `UserTransaction`. Сервер приложений использует ряд интерфейсов, определённых Jakarta Transactions, для связи с менеджером транзакций; менеджер транзакций использует интерфейсы, определённые Jakarta Transactions, для взаимодействия с менеджером ресурсов.

Спецификация Jakarta Transactions 2.0 доступна по ссылке https://jakarta.ee/specifications/transactions/2.0/.

Что такое транзакция?

Для эмуляции бизнес-транзакции программе может потребоваться выполнить несколько шагов. Например, финансовая программа может переводить средства с текущего счёта на сберегательный счёт, используя шаги, перечисленные в следующем псевдокоде:

```
begin transaction
    debit checking account
    credit savings account
    update history log
commit transaction
```

Либо все шаги должны быть выполнены, либо ни один из них. В противном случае целостность данных теряется. Поскольку шаги внутри транзакции представляют собой единое целое, транзакция часто определяется как атомарная операция.

Транзакция может закончиться двумя способами: фиксацией или с откатом. Когда транзакция фиксируется, изменения данных, сделанные входящими в неё операторами, сохраняются. Если один из операторов транзакции завершается неудачей, транзакция откатывается, отменяя действие всех операторов в транзакции. Например, в псевдокоде, если во время шага `credit` произошёл сбой в работе с жёстким диском, транзакция откатится и отменит изменения данных, сделанные на шаге `debit`. Несмотря на то, что транзакция не удалась, целостность данных не пострадает, поскольку счета всё ещё сбалансированы.

В предыдущем псевдокоде операторы `begin` и `commit` отмечают границы транзакции. При разработке Enterprise-бина вы определяете, как устанавливаются границы, указывая транзакции, управляемые контейнером или управляемые компонентом.

Управляемые контейнером транзакции

В EJB-компоненте с разграничением транзакций, управляемым контейнером, EJB-контейнер устанавливает границы транзакций. Управляемые контейнером транзакции можно использовать с любым типом Enterprise-бина: сессионным или управляемым сообщениями. Управляемые контейнером транзакции упрощают разработку, поскольку код Enterprise-бина явно не отмечает границы транзакции. В коде отсутствуют операторы, которые начинают и заканчивают транзакцию. По умолчанию, если разграничение транзакции не указано, Enterprise-бины используют транзакцию, управляемую контейнером.

Как правило, контейнер начинает транзакцию непосредственно перед запуском метода Enterprise-бина и фиксирует транзакцию непосредственно перед завершением метода. Каждый метод может быть связан с одной транзакцией. Вложенные или множественные транзакции в методе не разрешены.

Управляемые контейнером транзакции не требуют, чтобы все методы были связаны с транзакциями. При разработке бина вы можете установить атрибуты транзакции, чтобы указать, какие методы бина связаны с транзакциями.

Enterprise-бины, использующие разграничение транзакций, управляемое контейнером, не должны использовать методы управления транзакциями, которые мешают разграничению транзакций контейнером. Примерами таких методов являются `commit`, `setAutoCommit` и `rollback` у `java.sql.Connection` или методы `commit` и `rollback` у `jakarta.jms.Session`. Если требуется контроль над разграничением транзакций, следует использовать разграничение транзакций, управляемое приложением.

Enterprise-бины, использующие разграничение транзакций, управляемых контейнером, также не должны использовать интерфейс `jakarta.transaction.UserTransaction`.

Атрибуты транзакции

Атрибут транзакции контролирует область действия транзакции. На рис. 55-1 показано, почему важнó управление областью видимости. На диаграмме `method-A` начинает транзакцию, а затем вызывает `method-B` у `Bean-2`. Когда `method-B` выполняется, он запускается в рамках транзакции, запущенной `method-A`, или выполняется в новой транзакции? Ответ зависит от атрибута транзакции `method-B`.

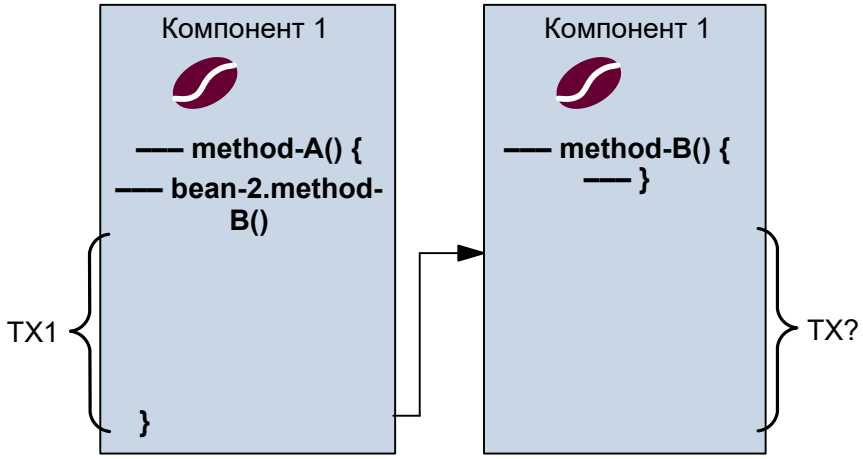

Рис. 55-1 Область действия транзакций

Атрибут транзакции может иметь одно из следующих значений:

- `Required`
- `RequiresNew`
- `Mandatory`
- `NotSupported`
- `Supports`
- `Never`

Атрибут Required

Если клиент работает в транзакции и вызывает метод Enterprise-бина, метод выполняется в транзакции клиента. Если клиент не связан с транзакцией, контейнер запускает новую транзакцию перед запуском метода.

Атрибут `Required` является неявным атрибутом транзакции для всех методов Enterprise-бина, работающих с разграничением транзакций, управляемым контейнером. Обычно атрибут `Required` не устанавливается, если не нужно переопределять другой атрибут транзакции. Поскольку атрибуты транзакции являются декларативными, их легко можно изменить позже.

Атрибут RequiresNew

Если клиент работает в транзакции и вызывает метод Enterprise-бина, контейнер выполняет следующие шаги:

1. Приостанавливает транзакцию клиента
2. Запускает новую транзакцию
3. Делегирует вызов метода
4. Возобновляет транзакцию клиента после завершения метода

Если клиент не связан с транзакцией, контейнер запускает новую транзакцию перед запуском метода.

Атрибут `requireNew` используется, если нужна уверенность, что метод всегда выполняется в новой транзакции.

Атрибут Mandatory

Если клиент работает в транзакции и вызывает метод Enterprise-бина, метод выполняется в транзакции клиента. Если клиент не связан с транзакцией, контейнер выбрасывает исключение `TransactionRequiredException`.

Атрибут `Mandatory` используется, если метод Enterprise-бина должен использовать транзакцию клиента.

Атрибут NotSupported

Если клиент работает в транзакции и вызывает метод Enterprise-бина, контейнер приостанавливает транзакцию клиента перед вызовом метода. После завершения метода контейнер возобновляет транзакцию клиента.

Если клиент не связан с транзакцией, контейнер не запускает новую транзакцию перед запуском метода.

Атрибут `NotSupported` используется для методов, которые не требуют транзакций. Поскольку с транзакциями связаны накладные расходы, этот атрибут может повысить производительность.

Атрибут Supports

Если клиент работает в транзакции и вызывает метод Enterprise-бина, метод выполняется в транзакции клиента. Если клиент не связан с транзакцией, контейнер не запускает новую транзакцию перед запуском метода.

Поскольку транзакционное поведение метода может отличаться, следует использовать атрибут `Supports` с осторожностью.

Атрибут Never

Если клиент работает в транзакции и вызывает метод Enterprise-бина, контейнер генерирует `RemoteException`. Если клиент не связан с транзакцией, контейнер не запускает новую транзакцию перед запуском метода.

Обзор атрибутов транзакции

В таблице 55-1 приводится сводная информация о влиянии атрибутов транзакции. Обе транзакции `T1` и `T2` управляются контейнером. Транзакция `T1` связана с клиентом, который вызывает метод в Enterprise-бине. В большинстве случаев клиент — это другой Enterprise-бин. Транзакция `T2` запускается контейнером непосредственно перед выполнением метода.

В последнем столбце таблицы 55-1 слово "None" означает, что бизнес-метод не выполняется в транзакции, управляемой контейнером. Однако вызовы базы данных в таком бизнес-методе могут управляться менеджером транзакций системы управления базами данных.

Таблица 55-1 Атрибуты и область действия транзакции

Атрибут транзакции	**Транзакция клиента**	**Транзакция бизнес-метода**
`Required`	None	T2
`Required`	T1	T1
`RequiresNew`	None	T2
`RequiresNew`	T1	T2
`Mandatory`	None	Error

Атрибут транзакции	Транзакция клиента	Транзакция бизнес-метода
`Mandatory`	T1	T1
`NotSupported`	None	None
`NotSupported`	T1	None
`Supports`	None	None
`Supports`	T1	T1
`Never`	None	None
`Never`	T1	Error

Установка атрибутов транзакции

Атрибуты транзакции задаются путём аннотирования класса или метода Enterprise-бина аннотацией `jakarta.ejb.TransactionAttribute` и установки для него одной из констант `jakarta.ejb.TransactionAttributeType`.

Если аннотировать класс Enterprise-бина с помощью `@TransactionAttribute`, указанный `TransactionAttributeType` будет применён ко всем бизнес-методам в классе. При аннотирования бизнес-метода с помощью `@TransactionAttribute TransactionAttributeType` применяется только к этому методу. Если аннотацией `@TransactionAttribute` аннотированы и класс, и метод, атрибут `TransactionAttributeType` метода переопределяет атрибут `TransactionAttributeType` класса.

Константы `TransactionAttributeType`, показанные в таблице 55-2, инкапсулируют атрибуты транзакции, описанные ранее в этом разделе.

Таблица 55-2 Константы TransactionAttributeType

Атрибут транзакции	Константа TransactionAttributeType
`Required`	`TransactionAttributeType.REQUIRED`
`RequiresNew`	`TransactionAttributeType.REQUIRES_NEW`
`Mandatory`	`TransactionAttributeType.MANDATORY`
`NotSupported`	`TransactionAttributeType.NOT_SUPPORTED`
`Supports`	`TransactionAttributeType.SUPPORTS`
`Never`	`TransactionAttributeType.NEVER`

В следующем фрагменте кода показано, как использовать аннотацию `@TransactionAttribute`:

JAVA

```java
@TransactionAttribute(NOT_SUPPORTED)
@Stateful
public class TransactionBean implements Transaction {
...
    @TransactionAttribute(REQUIRES_NEW)
    public void firstMethod() {...}

    @TransactionAttribute(REQUIRED)
    public void secondMethod() {...}

    public void thirdMethod() {...}

    public void fourthMethod() {...}
}
```

В этом примере атрибут транзакции класса `TransactionBean` был установлен в `NotSupported`, `firstMethod` был установлен в `requireNew`, и `secondMethod` был установлен в `Required`. Поскольку `@TransactionAttribute` для метода переопределяет `@TransactionAttribute` для класса, вызовы `firstMethod` создадут новую транзакцию и вызов `secondMethod` будет либо запущен в текущей транзакции, либо начнёт новую транзакцию. Вызовы `thirdMethod` и `fourthMethod` происходят не в транзакции.

Откат транзакции, управляемой контейнером

Существует два способа откатить транзакцию, управляемую контейнером. Во-первых, если выдаётся системное исключение, контейнер автоматически откатит транзакцию. Во-вторых, вызывая метод `setRollbackOnly` интерфейса `EJBContext`, метод бина сигнализирует контейнеру откатить транзакцию. Если бин генерирует исключение приложения, откат не выполняется автоматическим, но может быть инициирован вызовом `setRollbackOnly`.

Синхронизация переменных объекта сессионного бина

Необязательный интерфейс `SessionSynchronization` позволяет объектам сессионного компонента с сохранением состояния получать уведомления о синхронизации транзакций. Например, вы можете синхронизировать переменные объекта Enterprise-бина с соответствующими значениями в базе данных. Контейнер вызывает методы `SessionSynchronization` (`afterBegin`, `beforeCompletion` и `afterCompletion`) на каждом из основных этапов транзакции.

Метод `afterBegin` информирует объект о начале новой транзакции. Контейнер вызывает `afterBegin` непосредственно перед вызовом бизнес-метода.

Контейнер вызывает метод `beforeCompletion` после завершения бизнес-метода, но непосредственно перед фиксацией транзакции. Метод `beforeCompletion` является последней возможностью для сессионного компонента откатить транзакцию (путём вызова `setRollbackOnly`).

Метод `afterCompletion` указывает, что транзакция завершена. Этот метод имеет единственный параметр `boolean`, значение которого равно `true`, если транзакция была зафиксирована, и `false`, если был выполнен откат.

Методы, не разрешённые в транзакциях, управляемых контейнером

Недопустимо вызывать какой-либо метод, который может помешать разграничению транзакции, установленному контейнером. Следующие методы запрещены:

- Методы `commit`, `setAutoCommit` и `rollback` для `java.sql.Connection`
- Метод `getUserTransaction` у `jakarta.ejb.EJBContext`
- Любой метод `jakarta.transaction.UserTransaction`

Однако допускается использовать эти методы для установки границ в управляемых приложением транзакциях.

Управляемые бином транзакции

При разграничении транзакций, управляемых компонентом, код в сессионном компоненте или в компоненте, управляемом сообщениями, явно отмечает границы транзакции. Хотя бины с управляемыми контейнером транзакциями требуют меньше кода, они имеют ограничение: когда метод выполняется, он может быть либо связан с одной транзакцией, либо вообще выполняться без транзакции. Если это ограничение затрудняет кодирование компонента, следует рассмотреть возможность использования транзакций, управляемых компонентом.

Следующий псевдокод иллюстрирует вид детального контроля, который вы можете получить с помощью управляемых приложением транзакций. Проверяя различные условия, псевдокод решает, запускать или останавливать определённые транзакции в рамках бизнес-метода:

```
begin transaction
...
    update table-a
...
if (condition-x)
   commit transaction
else if (condition-y)
   update table-b
   commit transaction
else
   rollback transaction
   begin transaction
   update table-c
   commit transaction
```

При кодировании управляемой приложением транзакции для сессионных или компонентов, управляемых сообщениями, необходимо решить, использовать ли транзакции Java Database Connectivity или Jakarta. В следующих разделах обсуждаются оба типа транзакций.

Транзакции Jakarta

Jakarta Transactions позволяет разграничить транзакции таким образом, который не зависит от реализации менеджера транзакций. GlassFish Server реализует менеджер транзакций с сервисом транзакций Java (Java Transaction Service — JTS). Однако ваш код не вызывает методы JTS напрямую, а вместо этого вызывает методы Jakarta Transactions, которые затем вызывают подпрограммы JTS нижнего уровня.

Транзакция Jakarta управляются менеджером транзакций Jakarta EE. Возможно, вы захотите использовать транзакцию Jakarta, поскольку она может охватывать обновления нескольких баз данных от разных поставщиков. Менеджер транзакций конкретной СУБД может не работать с гетерогенными базами данных. Однако менеджер транзакций Jakarta EE имеет одно ограничение: он не поддерживает вложенные транзакции. Другими словами, он не может начать транзакцию для объекта, пока не завершится предыдущая транзакция.

Чтобы разграничить транзакцию Jakarta, вызываются методы `begin`, `commit` и `rollback` интерфейса `jakarta.transaction.UserTransaction`.

Возврат без завершения

В сессионном компоненте без сохранения состояния с управляемыми компонентом транзакциями бизнес-метод должен зафиксировать или откатить транзакцию перед возвратом. Однако сессионный компонент с состоянием не имеет этого ограничения.

В сессионном компоненте с сохранением состояния с транзакцией Jakarta связь между объектом компонента и транзакцией сохраняется в течением нескольких клиентских вызовов. Даже если каждый бизнес-метод, вызванный клиентом, открывает и закрывает соединение с базой данных, связь сохраняется до тех пор, пока объект не завершит транзакцию.

В сессионном компоненте с состоянием с JDBC транзакцией соединение JDBC сохраняет связь между объектом компонента и транзакцией в нескольких вызовах. Если соединение закрыто, связь не сохраняется.

Методы, не разрешённые в управляемых бином транзакциях

Не вызывайте методы `getRollbackOnly` и `setRollbackOnly` интерфейса `EJBContext` в управляемых компонентом транзакциях. Эти методы должны использоваться только в управляемых контейнером транзакциях. Для транзакций, управляемых компонентом, вызывайте методы `getStatus` и `rollback` интерфейса `UserTransaction`.

Тайм-ауты транзакций

Для транзакций, управляемых контейнером, может использоваться Консоль администрирования для настройки тайм-аута транзакции. Смотрите Запуск Консоли администрирования.

Для EJB-компонентов с управляемыми компонентом транзакциями Jakarta вызывается метод `setTransactionTimeout` интерфейса `UserTransaction`.

Установка тайм-аута транзакции

1. В консоли администрирования разверните узел «Конфигурации», затем разверните узел «server-config» и выберите «Сервис транзакций».
2. На странице «Сервис транзакций» установите для поля «Тайм-аут транзакции» значение по вашему выбору (например, 5).

   С помощью этого параметра, если транзакция не была завершена в течение 5 секунд, EJB-контейнер откатывает её.

   Значение по умолчанию равно 0, что означает, что тайм-аут транзакции не задан.
3. Нажмите Сохранить.

Обновление нескольких баз данных

Менеджер транзакций Jakarta EE управляет всеми транзакциями Enterprise-бина, кроме управляемых компонентом транзакций JDBC. Менеджер транзакций Jakarta EE позволяет Enterprise-бину обновлять несколько баз данных в транзакции. Рисунок 55-2 и рисунок 55-3 показывают два сценария для обновления нескольких баз данных за одну транзакцию.

На рисунке 55-2 клиент вызывает бизнес-метод в `Bean-A`. Бизнес-метод начинает транзакцию, обновляет базу данных X, затем базу данных Y и вызывает бизнес-метод в `Bean-B`. Второй бизнес-метод обновляет базу данных Z и возвращает управление бизнес-методу в `Bean-A`, который фиксирует транзакцию. Все три обновления базы данных происходят в одной транзакции.

На рисунке 55-3 клиент вызывает бизнес-метод в `Bean-A`, который начинает транзакцию и обновляет базу данных X. Затем `Bean-A` вызывает метод в `Bean-B`, который находится на удалённом сервере Jakarta EE. Метод в `Bean-B` обновляет базу данных Y. Менеджеры транзакций серверов Jakarta EE обеспечивают

обновление обеих баз данных в одной транзакции.

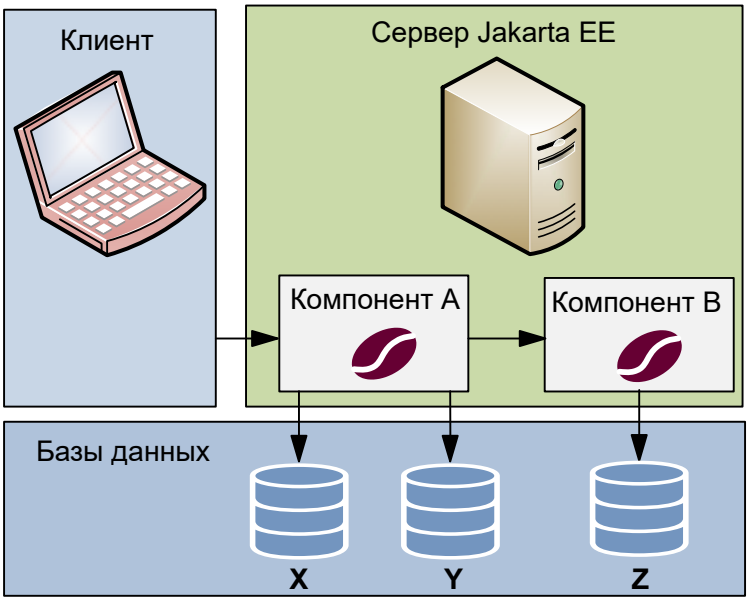

Рисунок 55-2 Обновление нескольких баз данных

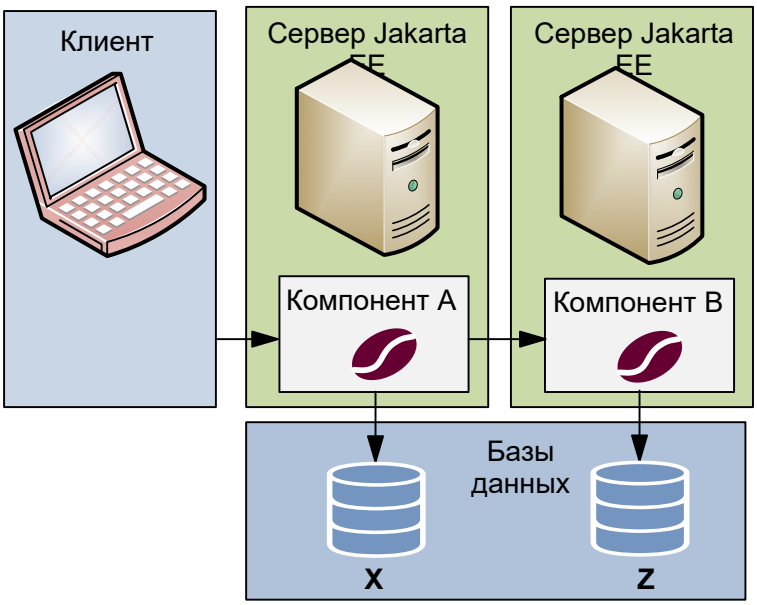

Рис. 55-3 Обновление нескольких баз данных на серверах Jakarta EE

Транзакции в веб-компонентах

Можно разграничить транзакцию в веб-компоненте, используя интерфейс `java.sql.Connection` или `jakarta.transaction.UserTransaction` . Это те же интерфейсы, которые может использовать сессионный компонент с транзакциями, управляемыми компонентом. Транзакции, разграниченные с помощью интерфейса `UserTransaction` , обсуждаются в Jakarta Transactions.

Дополнительная информация о транзакциях

Дополнительные сведения о транзакциях см. в спецификации Jakarta Transactions 2.0 по ссылке https://jakarta.ee/specifications/transactions/2.0/.

Глава 56. Адаптеры и контракты ресурсов

В этой главе рассматриваются адаптеры ресурсов и объясняется, как они обеспечивают связь между серверами Jakarta EE и системами ИС.

Что такое адаптер ресурсов?

Адаптер ресурсов — это компонент Jakarta EE, реализующий API коннекторов Jakarta для конкретной ИС. Примерами ИС могут быть информационные системы планирования ресурсов предприятия, обработки транзакций мэйнфреймов и системы управления базами данных. В Jakarta EE сервер Jakarta Messaging и Jakarta Mail также действуют как ИС, доступ к которым можно получить с помощью адаптеров ресурсов. Как показано на рис. 56-1, адаптер ресурсов упрощает взаимодействие между приложением Jakarta EE и ИС.

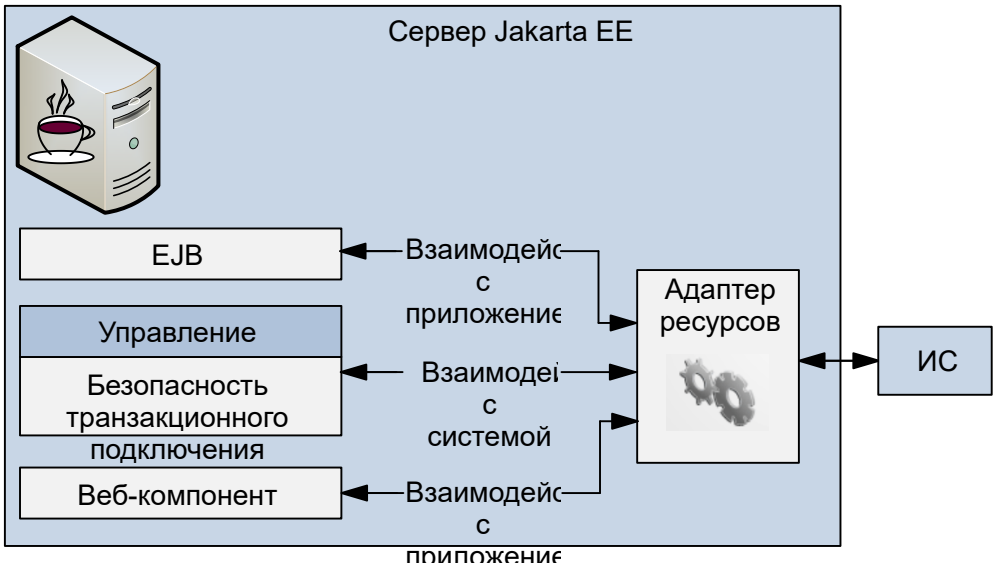

Рис. 56-1 Адаптеры ресурсов

Адаптер ресурса упаковывается в RAR-файл и может быть развёрнут на любом сервере Jakarta EE как обычное приложение Jakarta EE. RAR-файл может быть включён в состав EAR-файла или существовать отдельно от него.

Адаптер ресурса аналогичен драйверу JDBC. Оба предоставляют стандартный API, через который приложение может получить доступ к ресурсу, который находится за пределами сервера Jakarta EE. Для адаптера ресурса целевой системой является ИС. Для драйвера JDBC это СУБД. Адаптеры ресурсов и драйверы JDBC редко создаются разработчиками приложений. В большинстве случаев оба типа программного обеспечения разрабатываются провайдерами, которые предоставляют инструменты, серверы или интеграционное ПО.

Адаптер ресурсов обеспечивает связь между сервером Jakarta EE и ИС посредством контрактов. Контракт приложения определяет API, через который компонент Jakarta EE, такой как Enterprise-бин, получает доступ к ИС. Этот API является единственным представлением ИС. Системные контракты связывают адаптер ресурсов с важными сервисами, которые управляются сервером Jakarta EE. Сам адаптер ресурсов и его системные контракты прозрачны для компонента Jakarta EE.

Контракты управления

Jakarta Connectors определяет системные контракты, которые обеспечивают жизненный цикл адаптера ресурсов и управление потоками

Управление жизненным циклом

Jakarta Connectors определяет контракт на управление жизненным циклом, который позволяет серверу приложений управлять жизненным циклом адаптера ресурсов. Этот контракт предоставляет серверу приложений механизм начальной загрузки объекта адаптера ресурсов во время развёртывания или запуска сервера приложений. Этот контракт также предоставляет серверу приложений средство уведомления объекта адаптера ресурсов, когда оно удаляется или когда происходит корректное завершение работы сервера приложений.

Контракт управления работами

Контракт на управление работой Jakarta Connectors гарантирует, что адаптеры ресурсов используют потоки надлежащим, рекомендуемым образом. Этот контракт также позволяет серверу приложений управлять потоками для адаптеров ресурсов.

Адаптеры ресурсов, некорректно использующие потоки, могут поставить под угрозу всю среду сервера приложений. Например, адаптер ресурсов может создать слишком много потоков или может некорректно освободить созданные им потоки. Плохая обработка потоков препятствует завершению работы сервера приложений и влияет на его производительность, поскольку создание и удаление потоков — дорогостоящие операции.

Контракт управления работами устанавливает для сервера приложений средства для объединения и повторного использования потоков, аналогично пулу и повторному использованию соединений. Придерживаясь этого контракта, адаптер ресурсов не должен сам управлять потоками. Вместо этого адаптер ресурсов обеспечивается необходимыми потоками сервером приложений. Когда текущий поток завершается, адаптер ресурса возвращает его серверу приложений. Сервер приложений управляет потоком, либо возвращая его в пул для последующего повторного использования, либо уничтожая его. Такая обработка потоков приводит к повышению производительности сервера приложений и более эффективному использованию ресурсов.

Помимо перемещения управления потоками на сервер приложений, архитектура коннекторов предоставляет гибкую модель для адаптера ресурсов, использующего потоки.

- Запрашивающий поток может выбрать блокировку (остановить собственное выполнение) до завершения рабочего потока.
- Запрашивающий поток может блокироваться, пока он ожидает получения рабочего потока. Когда сервер приложений предоставляет рабочий поток, запрашивающий поток и рабочий поток выполняются параллельно.
- Адаптер ресурса может поставить выполнение некоторой задачи для потока в очередь. Поток выполнит задачу из очереди позже. Адаптер ресурса продолжает выполняться с того момента, когда он поставил задачу в очередь, независимо от её выполнения потоком.

При последних двух подходах отправляющий поток и рабочий поток могут выполняться одновременно или независимо. Для этих подходов в контракте указывается механизм слушателя, который уведомляет адаптер ресурсов о том, что поток завершил свою работу. Адаптер ресурсов также может указывать контекст выполнения для потока, а контракт управления работами управляет контекстом, в котором выполняется поток.

Контракт общего вида работ

Контракт управления работами между сервером приложений и адаптером ресурсов позволяет адаптеру ресурсов выполнять задачу, такую как обмен данными с ИС или доставка сообщений, путём предоставления объектов `Work` для выполнения.

Контракт общего вида работ позволяет адаптеру ресурсов управлять контекстами, в которых отправляемые им объекты `Work` выполняются `WorkManager` -ом сервера приложений. Механизм общего вида работ также позволяет серверу приложений поддерживать схемы поступления и доставки сообщений. Он также предоставляет более богатую контекстную среду выполнения `Work` адаптеру ресурсов, сохраняя при этом контроль над параллелизмом в управляемой среде.

Контракт общего вида работ стандартизирует контекст транзакции и контекст безопасности.

Исходящие и входящие контракты

Архитектура коннекторов определяет следующие исходящие контракты, контракты системного уровня между сервером приложений и ИС, которые обеспечивают исходящее подключение к ИС.

- Контракт управления соединениями поддерживает пул соединений, метод, который повышает производительность и масштабируемость приложений. Пул соединений прозрачен для приложения, которое просто получает соединение с ИС.
- Контракт управления транзакциями расширяет контракт управления соединениями и обеспечивает поддержку для управления как локальными, так и XA-транзакциями.

  Локальная транзакция ограничена по объёму одной системой ИС, и менеджер ресурсов ИС сам управляет такой транзакцией. Транзакция XA или глобальная транзакция может охватывать несколько менеджеров ресурсов. Эта форма транзакции требует координации транзакции внешним менеджером транзакций, как правило, в комплекте с сервером приложений. Менеджер транзакций использует протокол двухфазной фиксации для управления транзакцией, охватывающей несколько менеджеров ресурсов или ИС, и использует оптимизацию однофазной фиксации, если в транзакции XA участвует только один менеджер ресурсов.
- Контракт управления безопасностью предоставляет механизмы для аутентификации, авторизации и безопасного обмена данными между сервером Jakarta EE и ИС для защиты информации в ИС.

  Рабочее отображение безопасности сопоставляет идентификаторы ИС с идентификаторами домена сервера приложений.

Входящие контракты — это системные контракты между сервером Jakarta EE и ИС, которые обеспечивают входящее соединение из ИС: контракты на подключение для поставщиков сообщений и контракты на импорт транзакций.

Аннотации

Jakarta Connectors предоставляет набор аннотаций, чтобы снизить необходимость в файлах описания.

- Разработчик адаптера ресурсов может использовать аннотацию `@Connector` , чтобы указать, что компонент JavaBeans является адаптером ресурсов. Эта аннотация используется для предоставления метаданных о возможностях адаптера ресурсов. При желании вы можете предоставить компонент JavaBeans, реализующий интерфейс `ResourceAdapter` , как в следующем примере:

```java
@Connector(
    displayName = "TrafficRA",
    vendorName = "Jakarta EE Tutorial",
    version = "9.0"
)
public class TrafficResourceAdapter implements ResourceAdapter,
                                                Serializable {
    ...
}
```

- Аннотация `@ConnectionDefinition` определяет набор интерфейсов и классов соединений, относящихся к конкретному типу соединения, как в следующем примере:

```java
@ConnectionDefinition(
    connectionFactory = TradeConnectionFactory.class,
    connectionFactoryImpl = TradeConnectionFactoryImpl.class,
    connection = TradeConnection.class,
    connectionImpl = TradeConnectionImpl.class
)
public class TradeManagedConnectionFactory ... {
    ...
}
```

- Аннотация `@AdministeredObject` обозначает компонент JavaBeans как управляемый объект.
- Аннотация `@Activation` содержит информацию о конфигурации, относящуюся к входящим подключениям из ИС, как в следующем примере:

```java
@Activation(
        messageListeners = { TrafficListener.class }
)
public class TrafficActivationSpec implements ActivationSpec,
                                               Serializable {
    ...
    @ConfigProperty()
    /* порт для прослушивания запросов от ИС */
    private String port;
    ...
}
```

- Аннотация `@ConfigProperty` может использоваться в компонентах JavaBeans для предоставления дополнительной информации о конфигурации, которая может использоваться поставщиком развёртывания и адаптером ресурсов. Код предыдущего примера показывает несколько аннотаций `@ConfigProperty`.
- Аннотация `@ConnectionFactoryDefinition` является аннотацией определения ресурса, которая используется для определения фабрики соединений коннектора и регистрации её в JNDI под именем, указанным в обязательном элементе `name` аннотации. Обязательный элемент `interfaceName` аннотации указывает полное имя класса интерфейса фабрики соединений. Элемент аннотации `@TransactionsSupport` указывает уровень поддержки транзакций, который должна поддерживать фабрика соединений. Элементы `minPoolSize` и `maxPoolSize` аннотации определяют минимальное или максимальное количество соединений, которые должны быть выделены для пула соединений, поддерживающего этот ресурс фабрики соединений. Дополнительные свойства, связанные с определяемой фабрикой соединений, можно указать с помощью элемента `properties`.

  Поскольку повторы аннотаций недопустимы, аннотация `@ConnectionFactoryDefinitions` действует как контейнер для определений фабрики соединений с несколькими коннекторами. Элемент аннотации `value` содержит определения фабрики соединений с несколькими коннекторами.
- Аннотация `@AdministeredObjectDefinition` является аннотацией определения ресурса, которая используется для определения администрируемого объекта и регистрации его в JNDI под именем, указанным в обязательном элементе `name` аннотации. Обязательное полное имя класса администрируемого объекта должно указываться элементом `className`. Дополнительные свойства, которые необходимо настроить в администрируемом объекте, можно указать с помощью элемента `properties`.

Поскольку повторы аннотаций недопустимы, аннотация `@AdministeredObjectDefinitions` действует как контейнер для нескольких определений администрируемых объектов. Элемент аннотации `value` содержит несколько определений администрируемых объектов.

Спецификация позволяет разрабатывать адаптер ресурсов в смешанном режиме, то есть разработчик имеет возможность использовать для адаптера ресурса как аннотации, так и дескрипторы развёртывания в приложениях. Установщик приложения может использовать дескриптор развёртывания для переопределения аннотаций, указанных разработчиком адаптера ресурса.

Дескриптор развёртывания для адаптера ресурса, если он есть, называется `ra.xml`. Атрибут `metadata-complete` определяет, завершён ли дескриптор развёртывания для модуля адаптера ресурсов и нужно ли проверять файлы классов, доступные для модуля и упакованные с адаптером ресурсов, на наличие аннотаций, в которых указана информация о развёртывании.

Полный список аннотаций и компонентов JavaBeans, предоставляемых платформой Jakarta EE, см. в спецификации Jakarta Connectors 2.0.

Общий клиентский интерфейс

В этом разделе объясняется, как компоненты используют API общего клиентского интерфейса Jakarta Connectors Common Client Interface (CCI) и адаптер ресурсов для доступа к данным из ИС. API CCI определяет набор интерфейсов и классов, методы которых позволяют клиенту выполнять типичные операции доступа к данным. Интерфейсы и классы CCI следующие.

- `ConnectionFactory`: предоставляет компоненту приложения объект `Connection` к ИС.
- `Connection`: представляет соединение с ИС.
- `ConnectionSpec`: предоставляет компоненту приложения возможность передавать специфичные для запроса соединения свойства в `ConnectionFactory` при создании запроса на соединение.
- `Interaction`: предоставляет компоненту приложения средства для выполнения функций ИС, таких как хранимые процедуры базы данных.
- `InteractionSpec`: содержит свойства, относящиеся к взаимодействию компонента приложения с ИС.
- `Record`: родительский интерфейс для различных типов записей. Записи могут быть объектами `MappedRecord`, `IndexedRecord` или `ResultSet`, каждый из которых наследуется от интерфейса `Record`.
- `RecordFactory`: предоставляет компоненту приложения объект `Record`.
- `IndexedRecord`: представляет упорядоченную коллекцию объектов `Record` на основе интерфейса `java.util.List`.

Клиентский или прикладной компонент, который использует CCI для взаимодействия с ИС, должен соблюдать определённый порядок. Компонент должен установить соединение с менеджером ресурсов ИС, и он делает это с помощью `ConnectionFactory`. Объект `Connection` представляет соединение с ИС и используется для последующих взаимодействий с ИС.

Компонент выполняет взаимодействие с ИС, такое как доступ к данным из определённой таблицы, используя объект `Interaction`. Компонент приложения определяет объект Interaction с помощью объекта `InteractionSpec`. Когда он читает данные из ИС, например из таблиц базы данных, или записывает в эти таблицы, компонент приложения делает это, используя особый тип объекта `Record` — `MappedRecord`, `IndexedRecord` или `ResultSet`.

Также обратите внимание, что клиентское приложение, использующее адаптер ресурсов CCI, очень похоже на любой другой клиент Jakarta EE, использующий методы EJB.

Использование адаптеров ресурсов с инъецированим контекстов и зависимостей Jakarta (CDI)

Подробнее о CDI см. главу 25 *Введение в Jakarta Contexts and Dependency Injection* и главу 27 *Jakarta Contexts and Dependency Injection: дополнительные темы*.

Не указывайте следующие классы в адаптере ресурсов как Managed-бины CDI (т.е. не инъецируйте их), поскольку поведение этих классов как Managed-бинов CDI не было определено переносимым образом.

- Компоненты адаптера ресурсов: эти компоненты являются классами, которые аннотированы `jakarta.resource.spi.Connector` или объявлены как соответствующие элементы в дескрипторе развёртывания адаптера ресурсов `ra.xml`.
- Компоненты фабрики управляемых соединений: эти компоненты являются классами, которые аннотированы `jakarta.resource.spi.ConnectorDefinition`, или `jakarta.resource.spi.ConnectorDefinitions` или объявляются как соответствующие элементы в `ra.xml`.
- Компоненты спецификации активации: эти компоненты представляют собой классы, аннотированные `jakarta.resource.spi.Activation` или объявленные как соответствующие элементы в `ra.xml`.
- Администрируемые объекты: эти компоненты являются классами, которые аннотированы `jakarta.resource.spi.AdministeredObject` или объявлены как соответствующие элементы в `ra.xml`.

Другие типы классов в адаптере ресурсов могут быть управляемыми компонентами CDI и будут вести себя переносимым образом.

Дополнительная информация об адаптерах ресурсов

Для получения дополнительной информации об адаптерах ресурсов и аннотациях см.

- Спецификация платформы Jakarta EE 9:
  https://jakarta.ee/specifications/platform/
- Спецификация Jakarta Connectors 2.0:
  https://jakarta.ee/specifications/connectors/2.0/
- Спецификация Jakarta Enterprise Beans:
  https://jakarta.ee/specifications/enterprise-beans/4.0/
- Jakarta Annotations:
  https://jakarta.ee/specifications/annotations/2.0/

Глава 57. Примеры адаптеров ресурсов

В этой главе описываются два примера, которые демонстрируют, как использовать адаптеры ресурсов в приложениях Jakarta EE и как реализовать простые адаптеры ресурсов.

Обзор примеров адаптеров ресурсов

В примере `trading` показано, как использовать простой пользовательский интерфейс клиента для подключения к ИС из веб-приложения. В этом примере адаптер ресурсов реализует исходящий контракт и пользовательский интерфейс клиента. Пример `traffic` показывает, как использовать управляемый сообщениями компонент (message-driven bean — MDB) для обработки обновлений информации о трафике из ИС. Адаптер ресурсов в этом примере реализует входящие контракты и контракты управления работами.

Пример trading

Пример `trading` демонстрирует, как реализовать и использовать простой адаптер исходящих ресурсов, который отправляет запросы в ИС с использованием сокета TCP. Пример демонстрирует сценарий на рисунке 57-1 и состоит из следующих модулей:

- `trading-eis` : программа на Java SE, которая имитирует ИС
- `trading-rar` : реализация адаптера исходящих ресурсов
- `trading-war` : веб-приложение, использующее адаптер ресурсов
- `trading-ear` : корпоративный архив, содержащий адаптер ресурсов и веб-приложение

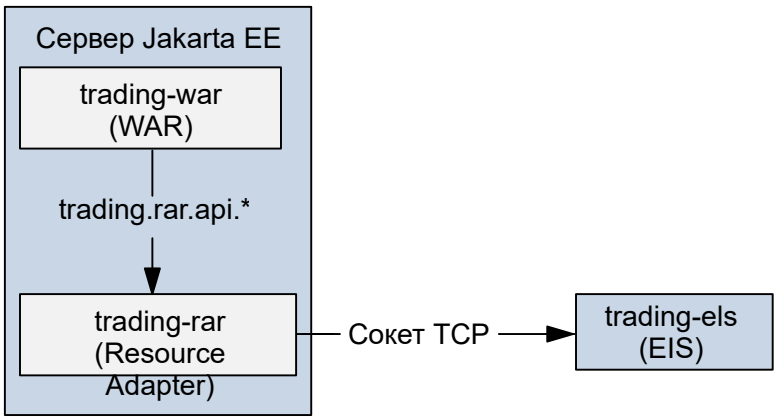

Рисунок 57-1 Пример trading

Модуль `trading-eis` — это вспомогательный проект, имитирующий платформу для проведения торговых операций с акциями. Он содержит программу Java SE, которая слушает торговые запросы в виде простого текста по сокету TCP. Программа отвечает на торговые запросы со значением статуса, номером подтверждения и суммой в долларах за запрошенные акции и сборы. Например, пара запрос-ответ будет выглядеть так:

```
>> BUY 1000 ZZZZ MARKET
<< EXECUTED #1234567 TOTAL 50400.00 FEE 252.00
```

Модуль `trading-rar` реализует исходящий контракт Jakarta Connectors для отправки запросов и получения ответов от ИС биржевой торговли. Модуль `trading-rar` предоставляет и реализует пользовательский интерфейс клиента для использования приложениями Jakarta EE. Этот интерфейс проще, чем Common Client Interface (CCI).

Модуль `trading-war` — это веб-приложение с интерфейсом Jakarta Faces и Managed-бином. Это приложение позволяет клиентам отправлять заявки в ИС, используя адаптер ресурсов, предоставленный модулем `trading-rar`. Модуль `trading-war` использует пользовательский интерфейс клиента, предоставленный адаптером ресурсов, для получения соединений с ИС.

Использование адаптера исходящих ресурсов

В большинстве случаев разработчики приложений Jakarta EE используют адаптеры исходящих ресурсов, разработанные третьей стороной. Адаптеры исходящих ресурсов либо реализуют общий клиентский интерфейс (CCI), либо предоставляют пользовательский интерфейс для взаимодействия приложений с ИС. Адаптеры исходящих ресурсов предоставляют приложениям Jakarta EE следующие элементы:

- Фабрики соединений
- Дескрипторы соединений
- Другие интерфейсы и объекты, специфичные для ИС

Приложения Jakarta EE получают объект фабрики соединений посредством инъецирования ресурсов, а затем используют его для получения дескрипторов соединения с ИС. Дескрипторы соединений позволяют приложению отправлять запросы и получать информацию из ИС.

Модуль `trader-rar` предоставляет пользовательский интерфейс клиента, состоящий из классов, перечисленных в таблице 57-1.

Таблица 57-1 Классы и интерфейсы в модуле trading-rar

API-компонент	**Описание**
`TradeOrder`	Представляет торговую заявку для ИС
`TradeResponse`	Представляет ответ от ИС на торговую заявку
`TradeConnection`	Представляет дескриптор подключения к ИС Предоставляет способ подачи заявок в ИС
`TradeConnectionFactory`	Позволяет приложениям получать дескрипторы подключения к ИС
`TradeProcessingException`	Указывает, что возникла проблема при обработке торгового запроса

Managed-бин `ResourceAccessBean` в модуле `trading-war` настраивает фабрику соединений для адаптера ресурсов `trading-rar` с помощью аннотации `@ConnectionFactoryDefinition` следующим образом:

JAVA

```
@Named
@SessionScoped
@ConnectionFactoryDefinition(
    name = "java:comp/env/eis/TradeConnectionFactory",
    interfaceName = "ee.jakarta.tutorial.trading.rar.api.TradeConnectionFactory",
    resourceAdapter = "#trading-rar",
    minPoolSize = 5,
    transactionSupport =
            TransactionSupport.TransactionSupportLevel.NoTransaction
)
public class ResourceAccessBean implements Serializable { ... }
```

Параметр `name` указывает имя JNDI для фабрики соединений. В этом примере регистрируется фабрика соединений в области `java:comp`. Вы можете использовать аннотацию `@ConnectionFactoryDefinition`, чтобы указать другую область, такую как `java:global`, `java:app` или `java:module`. Аннотация `@AdministeredObjectDefinition` также позволяет регистрировать объекты администрируемых коннекторов в пространстве имён JNDI.

Параметр `interfaceName` указывает интерфейс, реализованный фабрикой соединений, включённой в адаптер ресурсов. В этом примере это пользовательский интерфейс.

Параметр `resourceAdapter` указывает имя адаптера ресурса, который содержит реализацию фабрики соединений. Префикс `#` в `#trading-rar` указывает, что `trading-rar` является встроенным адаптером ресурсов, который входит в тот же EAR, что и это веб-приложение.

> Вы также можете настроить фабрику соединений для ранее развёрнутого адаптера исходящих ресурсов, используя команды администрирования с вашего сервера приложений. Однако эта процедура зависит от производителя ПО.

Managed-бин получает объект фабрики соединений, используя инъецирование ресурсов следующим образом:

JAVA

```
...
public class ResourceAccessBean implements Serializable {
    @Resource(lookup = "java:comp/env/eis/TradeConnectionFactory")
    private TradeConnectionFactory connectionFactory;
    ...
}
```

Managed-бин использует фабрику соединений для получения дескрипторов соединений следующим образом:

JAVA

```
TradeConnection connection = connectionFactory.getConnection();
```

Адаптер ресурса возвращает дескриптор соединения, связанный с физическим соединением с ИС. Как только дескриптор соединения становится доступным, Managed-бин отправляет заявку и получает ответ следующим образом:

JAVA

```
TradeOrder order = new TradeOrder();
order.setNShares(1000);
order.setTicker(TradeOrder.Ticker.YYYY);
order.setOrderType(TradeOrder.OrderType.BUY);
order.setOrderClass(TradeOrder.OrderClass.MARKET);
...
try {
    TradeResponse response = connection.submitOrder(order);
    ...
} catch (TradeProcessingException ex) { ... }
```

Реализация адаптера исходящих ресурсов

Модуль `trading-rar` реализует исходящий контракт и пользовательский интерфейс клиента для ИС, используемой в этом примере. Архитектура адаптера исходящего ресурса показана на рисунке 57-2.

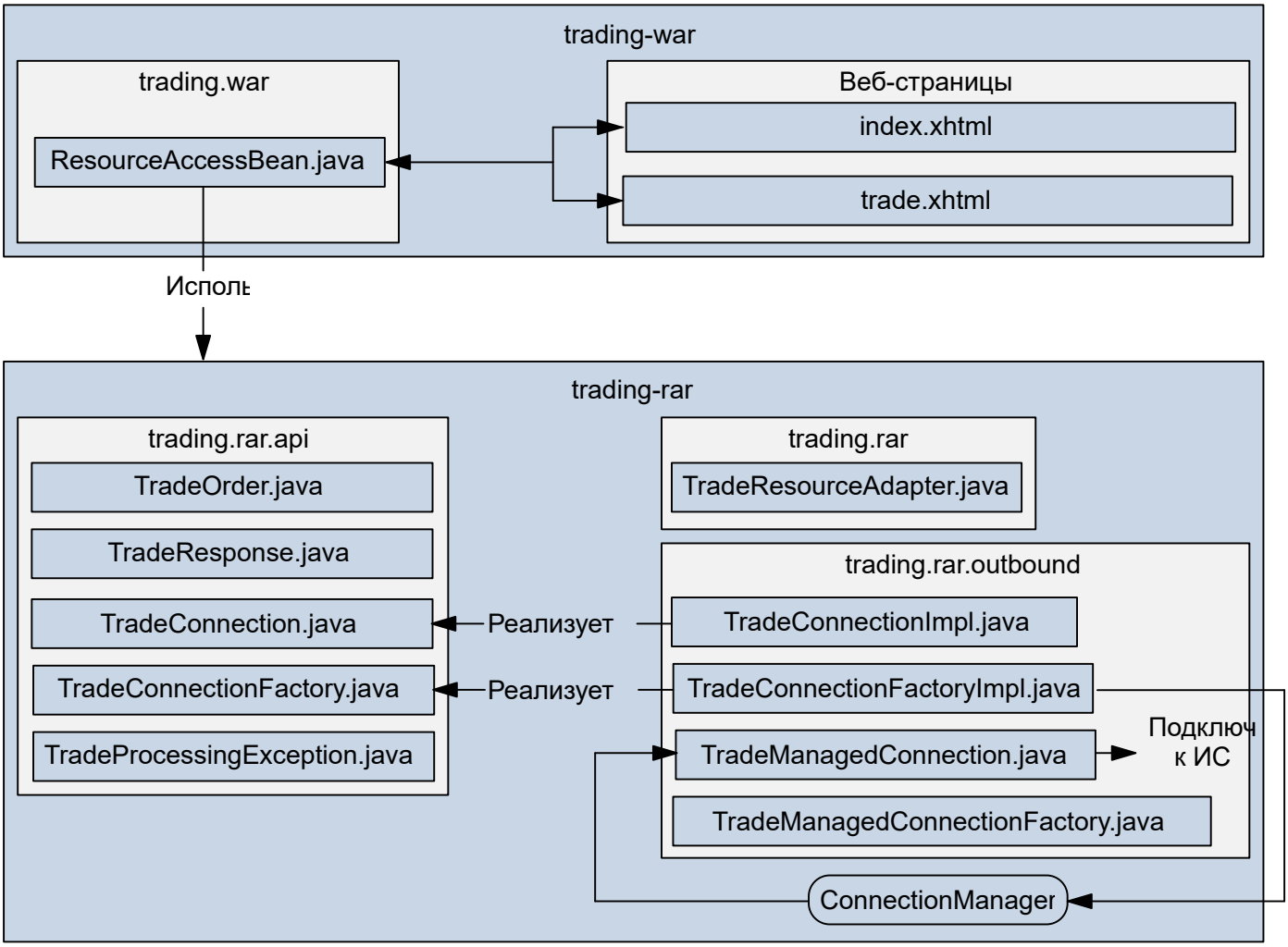

Рисунок 57-2 Архитектура примера trading

Модуль `trading-rar` реализует интерфейсы, перечисленные в таблице 57-2.

Таблица 57-2 Интерфейсы, реализованные в модуле trading-rar

Пакет	Интерфейс	Описание
`jakarta.resource.spi`	`ResourceAdapter`	Определяет методы жизненного цикла адаптера ресурса
`jakarta.resource.spi`	`ManagedConnectionFactory`	Определяет фабрику соединений, которую менеджер соединений с сервера приложений использует для получения физических соединений с ИС
`jakarta.resource.spi`	`ManagedConnection`	Определяет физическое соединение с ИС, которым может управлять менеджер соединений
`trading.rar.api`	`TradeConnectionFactory`	Определяет фабрику соединений, которую приложения используют для получения дескрипторов соединений
`trading.rar.api`	`TradeConnection`	Определяет дескриптор соединения, который приложения используют для взаимодействия с ИС

Когда архив `trading-ear` развёрнут и ресурс пула соединений настроен, как описано в Использование адаптера исходящих ресурсов, сервер приложений создаёт объекты `TradeConnectionFactory`, которые приложения могут получить, используя инъецирование ресурсов. Реализация `TradeConnectionFactory` делегирует создание соединений с менеджером соединений, предоставляемым сервером приложений.

Менеджер соединений использует реализацию `ManagedConnectionFactory` для получения физических соединений с ИС и поддерживает пул активных физических соединений. Когда приложение запрашивает дескриптор соединения, менеджер соединений связывает соединение из пула с новым дескриптором соединения, который может использовать приложение. Пул соединений повышает производительность приложений и упрощает разработку адаптера ресурсов.

Для получения дополнительной информации см. код и комментарии в модуле `trading-rar`.

Запуск trading

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска примера `trading`.

Запуск trading с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню «Файл» выберите «Открыть проект».
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/connectors
   ```

4. Выберите каталог `trading`.
5. Нажмите Открыть проект.
6. На вкладке «Проекты» разверните узел `trading`.
7. Кликните правой кнопкой мыши модуль `trading-eis` и выберите «Открыть проект».
8. Кликните правой кнопкой мыши проект `trading-eis` и выберите «Выполнить».

   Сообщения от ИС появляются на вкладке Вывод:

   ```
   Trade execution server listening on port 4004.
   ```

9. Кликните правой кнопкой мыши проект `trading-ear` и выберите Сборка.

   Эта команда упаковывает адаптер ресурса и веб-приложение в EAR-файл и развёртывает его в GlassFish Server.

10. Откройте следующий URL в веб-браузере:

    ```
    http://localhost:8080/trading/
    ```

    Веб-интерфейс позволяет подключаться к ИС и отправлять сделки. В журнале сервера отображаются запросы от веб-приложения и последовательность вызовов, которые предоставляют дескрипторы подключения от адаптера ресурсов.

11. Перед удалением приложения `trading-ear`, закройте приложение `trading-eis` из строки состояния.

Запуск trading с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).

2. В окне терминала перейдите в:

```
tut-install/examples/connectors/trading/
```

3. Введите следующую команду:

```
mvn install
```
SHELL

Эта команда собирает и упаковывает адаптер ресурсов и веб-приложение в архив EAR и развёртывает его в GlassFish Server.

4. В том же окне терминала перейдите в каталог `trading-eis` :

```
cd trading-eis
```
SHELL

5. Введите следующую команду для запуска платформы исполнения сделок:

```
mvn exec:java
```
SHELL

Сообщения от ИС появляются в окне терминала:

```
Trade execution server listening on port 4004.
```

6. Откройте следующий URL в веб-браузере:

```
http://localhost:8080/trading/
```

Веб-интерфейс позволяет подключаться к ИС и отправлять сделки. В журнале сервера отображаются запросы от веб-приложения и последовательность вызовов, которые предоставляют дескрипторы подключения от адаптера ресурсов.

7. Перед удалением приложения `trading-ear` нажмите Ctrl+C в окне терминала, чтобы закрыть приложение `trading-eis` .

Пример traffic

Пример `traffic` демонстрирует, как реализовать и использовать простой адаптер входящих ресурсов, который получает данные из ИС с использованием сокета TCP.

Пример находится в каталоге *`tut-install`*`/examples/connectors/traffic` . См. главу 2 *Использование учебных примеров* для получения базовой информации о сборке и запуске примеров.

Пример демонстрирует сценарий на рисунке 57-3 и состоит из следующих модулей:

- `traffic-eis` : программа Java SE, которая имитирует ИС
- `traffic-rar` : реализация адаптера входящих ресурсов
- `traffic-ejb` : управляемый сообщениями компонент, являющийся конечной точкой для входящих сообщений
- `traffic-war` : веб-приложение, отображающее информацию из бина, управляемого сообщениями
- `traffic-ear` : корпоративный архив, содержащий адаптер ресурса, компонент, управляемый сообщениями, и веб-приложение.

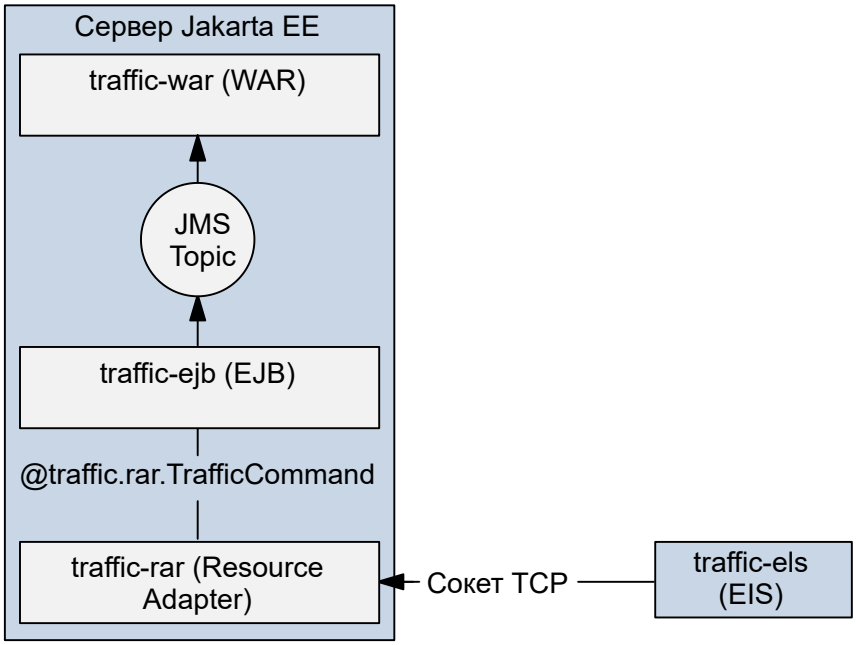

Рис. 57-3 Пример traffic

Модуль `traffic-eis` — это вспомогательный проект, имитирующий информационную систему управления трафиком. Он содержит программу Java SE, которая отправляет обновления статуса трафика для нескольких городов любому подписанному клиенту. Программа отправляет обновления в формате JSON через TCP-сокет. Например, обновление трафика выглядит так:

JSON

```
{"report":[
    {"city":"City1", "access":"AccessA", "status":"GOOD"},
    {"city":"City1", "access":"AccessB", "status":"CONGESTED"},
    ...
    {"city":"City5", "access":"AccessE", "status":"SLOW"}
 ]}
```

Модуль `traffic-rar` реализует входящий контракт спецификации Jakarta Connectors. Этот модуль подписывается на систему информации о трафике, используя порт TCP, указанный в конфигурации, предоставленной MDB, и вызывает методы MDB для обработки обновлений информации о трафике.

Модуль `traffic-ejb` содержит управляемый сообщениями компонент, который активирует адаптер ресурсов с параметром конфигурации (порт TCP для подписки на систему информации о трафике). MDB содержит метод для обработки обновлений информации о трафике. Этот метод фильтрует обновления для конкретного города и публикует результаты в теме Jakarta Messaging.

Модуль `traffic-war` содержит управляемый сообщениями компонент, который асинхронно получает отфильтрованные обновления информации о трафике из раздела Jakarta Messaging и отправляет их клиентам с помощью конечной точки веб-сокета.

Использование входящего адаптера ресурсов

В большинстве случаев разработчики приложений Jakarta EE используют адаптеры входящих ресурсов, разработанные третьей стороной. Чтобы использовать адаптер входящих ресурсов, приложение Jakarta EE включает компонент, управляемый сообщениями, со следующими характеристиками.

- MDB реализует бизнес-интерфейс, определённый адаптером ресурсов.
- MDB указывает параметры конфигурации для активации адаптера ресурса.

Бизнес-интерфейс, определённый адаптером ресурсов, не указан в спецификации Jakarta Connectors: он специфичен для ИС.

MDB в этом примере определяется следующим образом:

JAVA

```
@MessageDriven(
    activationConfig = {
      @ActivationConfigProperty(propertyName = "port",
                                propertyValue = "4008")
    }
)
public class TrafficMdb implements TrafficListener { ... }
```

Интерфейс `TrafficListener` определён в пакете API адаптера ресурсов. Адаптеру ресурсов требуется, чтобы MDB предоставил свойство `port`.

Когда MDB развёрнут, он активирует адаптер ресурсов `traffic-rar`, который вызывает методы MDB для обработки обновлений информации о трафике. Затем MDB фильтрует обновления для определённого города и публикует результаты в теме JMS.

В этом конкретном примере интерфейс `TrafficListener` пуст. В дополнение к этому интерфейсу адаптер ресурсов предоставляет аннотацию `@TrafficCommand` и использует reflection, чтобы выяснить, какие методы в MDB ею аннотированы:

JAVA

```
@MessageDriven(...)
public class TrafficMdb implements TrafficListener {

    @TrafficCommand(name="report", info="Process report")
    public void processReport(String jsonReport) { ... }
    ...
}
```

Этот подход позволяет адаптировать MDB для поддержки новых функций в ИС без необходимости изменять интерфейс `TrafficListener` или модуль адаптера ресурсов.

Реализация адаптера входящих ресурсов

Модуль `traffic-rar` реализует контракт адаптера входящих ресурсов Jakarta Connectors для простой системы информирования о дорожном движении, используемой в этом примере. Архитектура адаптера входящих ресурсов показана на рисунке 57-4.

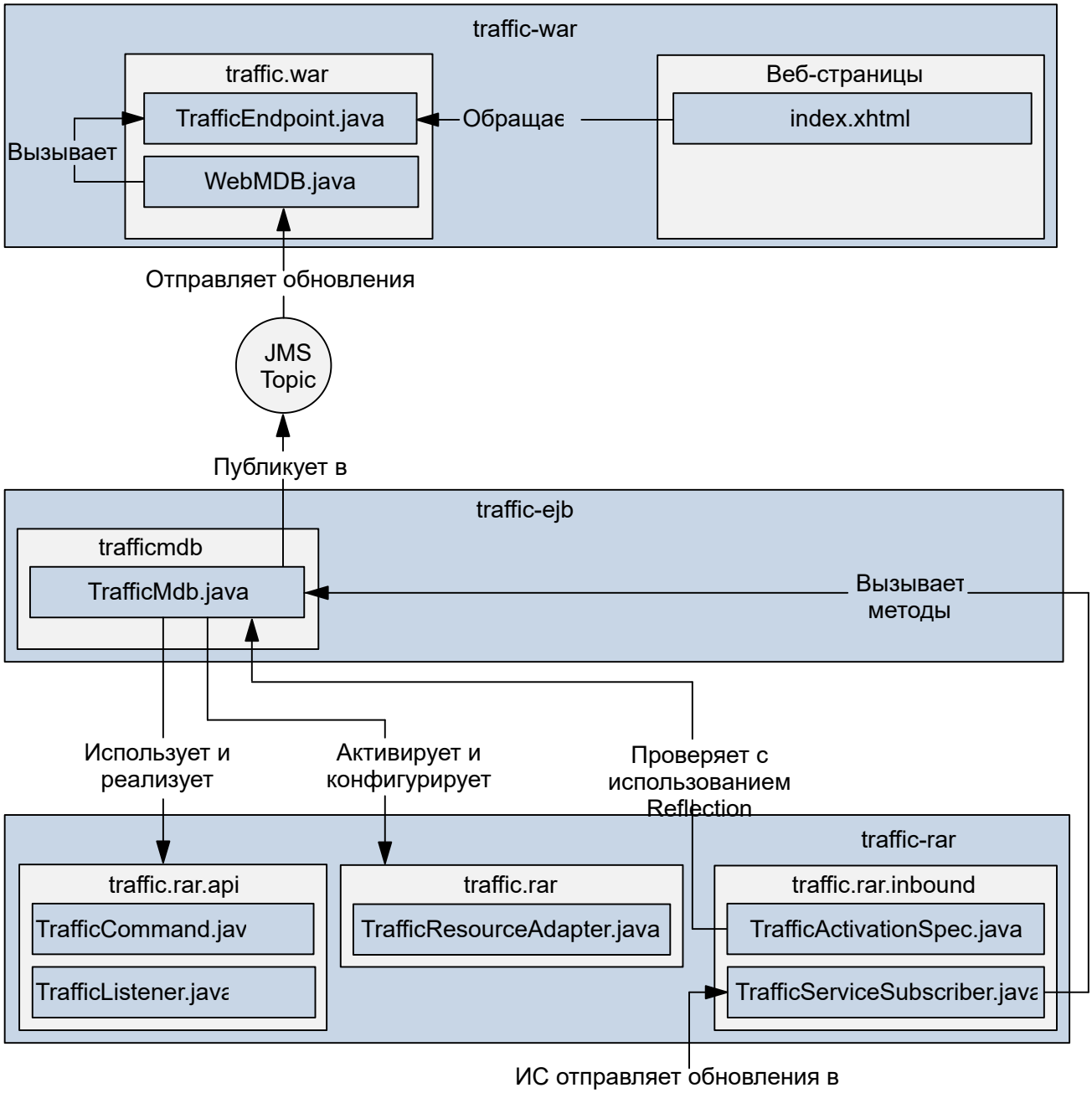

Рисунок 57-4 Архитектура примера traffic

Модуль `traffic-rar` реализует интерфейсы, перечисленные в таблице 57-3.

Таблица 57-3 Интерфейсы, реализованные в модуле traffic-rar

Пакет	Интерфейс	Описание
`jakarta.resource.spi`	`ResourceAdapter`	Определяет методы жизненного цикла адаптера ресурса.
`jakarta.resource.spi`	`ActivationSpec`	Определяет параметры конфигурации, которые MDB предоставляет для активации адаптера входящих ресурсов.
`jakarta.resource.spi`	`Work`	Подписчик сервиса трафика реализует этот интерфейс из контракта управления работами, чтобы ожидать обновления трафика в отдельном потоке.

Когда MDB активирует адаптер входящих ресурсов, контейнер вызывает метод `endpointActivation` класса `TrafficResourceAdapter`:

JAVA

```
@Connector(...)
public class TrafficResourceAdapter implements ResourceAdapter, Serializable {
    ...
    @Override
    public void endpointActivation(MessageEndpointFactory endpointFactory,
                                   ActivationSpec spec)
                                   throws ResourceException {
        Class endpointClass = endpointFactory.getEndpointClass();
        /* этот метод вызывается из нового потока, например:
        MessageEndpoint endpoint = endpointFactory.createEndpoint(null); */
    }
}
```

Метод `getEndpointClass` возвращает `Class` типа MDB, выполняющего активацию, что позволяет адаптеру ресурсов использовать reflection для поиска методов, аннотированных `@TrafficCommand` в MDB.

Метод `createEndpoint` возвращает объект MDB. Адаптер ресурсов использует этот объект для вызова методов MDB, когда он получает запросы от ИС.

После получения объекта конечной точки сообщения (MDB) адаптер ресурсов использует контракт управления работами для создания потока подписчика сервиса трафика, который получает обновления трафика от ИС. Адаптер ресурса получает объект `WorkManager` из контекста начальной загрузки следующим образом:

JAVA

```
WorkManager workManager;
...
@Override
public void start(BootstrapContext ctx) ... {
    workManager = ctx.getWorkManager();
}
```

Адаптер ресурса планирует поток подписчика сервиса трафика с помощью менеджера работ:

JAVA

```
tSubscriber = new TrafficServiceSubscriber(tSpec, endpoint);
workManager.scheduleWork(tSubscriber);
```

Класс `TrafficServiceSubscriber` реализует интерфейс `jakarta.resource.spi.Work` из контракта на управление работами.

Поток подписчика сервиса трафика использует reflection для вызова методов в MDB:

JAVA

```
private String callMdb(MessageEndpoint mdb, Method command,
                       String... params) ... {
    String resp;
    /* этот код содержит обработку исключений */
    mdb.beforeDelivery(command);
    Object ret = command.invoke(mdb, (Object[]) params);
    resp = (String) ret;
    mdb.afterDelivery();
    return resp;
}
```

Подробнее см. в коде и комментариях модуля traffic-rar.

Выполнение примера трафика

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска примера `traffic`.

Запуск traffic с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню «Файл» выберите «Открыть проект».
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/connectors
   ```

4. Выберите каталог `traffic`.
5. Нажмите Открыть проект.
6. На вкладке «Проекты» разверните узел `traffic`.
7. Кликните правой кнопкой мыши модуль `traffic-eis` и выберите Открыть проект.
8. Кликните правой кнопкой мыши проект `traffic-eis` и выберите «Выполнить».

   Сообщения от ИС появляются на вкладке Вывод:

   ```
   Traffic EIS accepting connections on port 4008
   ```

9. На вкладке «Проекты» кликните правой кнопкой мыши проект `traffic` и выберите команду "Очистить и собрать".

   Эта команда собирает и упаковывает адаптер ресурсов, MDB и веб-приложение в архив EAR и развёртывает его. Журнал сервера показывает последовательность вызовов, которая активирует адаптер ресурсов, и обновления отфильтрованного трафика для City1.
10. Откройте следующий URL в веб-браузере:

    ```
    http://localhost:8080/traffic/
    ```

    Веб-интерфейс показывает отфильтрованные обновления трафика для City1 каждые несколько секунд.
11. После удаления приложения `traffic-ear` закройте приложение `traffic-eis` из строки состояния.

Запуск traffic с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/connectors/traffic/traffic-eis/
   ```

3. Введите следующую команду в окне терминала:

   ```
   mvn install
   ```
   SHELL

   Эта команда собирает и упаковывает ИС traffic.
4. Введите следующую команду в окне терминала:

   ```
   mvn exec:java
   ```
   SHELL

Сообщения от ИС появляются в окне терминала:

```
Traffic EIS accepting connections on port 4008
```

Оставьте это окно терминала открытым.

5. Откройте новое окно терминала и перейдите по ссылке:

```
tut-install/examples/connectors/traffic/
```

6. Введите следующую команду:

```
mvn install
```
SHELL

Эта команда собирает и упаковывает адаптер ресурсов, MDB и веб-приложение в архив EAR и развёртывает его. Журнал сервера показывает последовательность вызовов, которая активирует адаптер ресурсов, и обновления отфильтрованного трафика для City1.

7. Откройте следующий URL в веб-браузере:

```
http://localhost:8080/traffic/
```

Веб-интерфейс показывает отфильтрованные обновления трафика для City1 каждые несколько секунд.

8. После удаления приложения `traffic-ear` нажмите Ctrl+C в первом окне терминала, чтобы закрыть приложение `traffic-eis`.

Глава 58. Использование Interceptor-ов Jakarta EE

В этой главе обсуждается, как создавать классы Interceptor-ов и методы, которые встраиваются в вызовы методов или события жизненного цикла целевого класса.

Обзор Interceptor-ов

Interceptor-ы используются в сочетании с управляемыми классами Jakarta EE, чтобы позволить разработчикам вызывать методы Interceptor-а в связанном целевом классе в сочетании с вызовами методов или событиями жизненного цикла. Типичное использование Interceptor-ов — это регистрация, аудит и профилирование.

Вы можете использовать Interceptor-ы с сессионными компонентами, управляемыми сообщениями компонентами и Managed-бинами CDI. Во всех этих случаях целевым классом Interceptor-а является класс бина.

Interceptor может быть определён в целевом классе как метод Interceptor-а или в связанном классе, называемом классом Interceptor-а. Классы Interceptor-ов содержат методы, которые вызываются вместе с методами или событиями жизненного цикла целевого класса.

Классы и методы Interceptor-ов определяются аннотациями или в дескрипторе развёртывания приложения, который содержит Interceptor-ы и целевые классы.

Приложения, использующие дескриптор развёртывания для определения Interceptor-ов, не переносимы между серверами Jakarta EE.

Методы Interceptor-а в целевом классе или в классе Interceptor-а аннотируются одной из аннотаций метаданных, определённых в таблице 58-1.

Таблица 58-1 Аннотации метаданных Interceptor-а

Аннотации Interceptor-ов	**Описание**
`jakarta.interceptor.AroundConstruct`	Обозначает метод как метод Interceptor-а, который получает Callback-вызов после создания целевого класса
`jakarta.interceptor.AroundInvoke`	Обозначает метод как метод Interceptor-а
`jakarta.interceptor.AroundTimeout`	Обозначает метод как Interceptor тайм-аута для вставки в методы тайм-аута для EJB-таймеров
`jakarta.annotation.PostConstruct`	Обозначает метод как метод Interceptor-а для события "конструирование завершено" жизненного цикла
`jakarta.annotation.PreDestroy`	Обозначает метод как метод Interceptor-а для события "непосредственно перед уничтожением" жизненного цикла

Классы Interceptor-ов

Классы Interceptor-ов могут быть обозначены аннотацией `jakarta.interceptor.Interceptor`, но это не обязательно. Класс Interceptor-а должен иметь публичный конструктор без аргументов.

Целевой класс может иметь любое количество классов Interceptor-ов, связанных с ним. Порядок вызова классов Interceptor-ов, определяется порядком, в котором классы Interceptor-ов определены в аннотации `jakarta.interceptor.Interceptors`. Однако этот порядок может быть переопределён в дескрипторе развёртывания.

Классы Interceptor-ов могут быть инъецируемыми объектами. Инъецирование зависимостей происходит при создании объекта класса Interceptor-а с использованием контекста именования связанного целевого класса и до вызова любых Callback-вызовов `@PostConstruct`.

Жизненный цикл Interceptor-а

Классы Interceptor-ов имеют тот же жизненный цикл, что и связанный с ними целевой класс. Когда создаётся объект целевого класса, объект класса Interceptor-а также создаётся для каждого объявленного класса Interceptor-а в целевом классе. То есть, если целевой класс объявляет несколько классов Interceptor-ов, объект каждого класса создаётся при создании объекта целевого класса. Объект целевого класса и все объекты классов Interceptor-ов полностью создаются перед вызовом любых Callback-вызовов `@PostConstruct`, а любые Callback-вызовы `@PreDestroy` вызываются до того, как объекты целевого класса и класса Interceptor-а уничтожены.

Interceptor-ы и CDI

Спецификация Jakarta Contexts and Dependency Injection (CDI) основываются на базовой функциональности Jakarta EE Interceptors. Для получения информации об Interceptor-ах CDI, включая обсуждение типов связывания Interceptor-ов, см. Использование Interceptor-ов в приложениях CDI.

Использование Interceptor-ов

Чтобы определить Interceptor, используйте одну из аннотаций метаданных Interceptor-а, перечисленных в таблице 58-1 в целевом классе или в отдельном классе Interceptor-а. Следующий код объявляет метод Interceptor-а `@AroundTimeout` в целевом классе:

JAVA

```java
@Stateless
public class TimerBean {
    ...
    @Schedule(minute="*/1", hour="*")
    public void automaticTimerMethod() { ... }

    @AroundTimeout
    public void timeoutInterceptorMethod(InvocationContext ctx) { ... }
    ...
}
```

Если вы используете классы Interceptor-ов, используйте аннотацию `jakarta.interceptor.Interceptors` для объявления одного или нескольких Interceptor-ов на уровне класса или метода целевого класса. Следующий код объявляет Interceptor-ы на уровне класса:

JAVA

```java
@Stateless
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class})
public class OrderBean { ... }
```

Следующий код объявляет класс Interceptor-а на уровне метода:

JAVA

```java
@Stateless
public class OrderBean {
    ...
    @Interceptors(OrderInterceptor.class)
    public void placeOrder(Order order) { ... }
    ...
}
```

Перехват вызовов методов

Используйте аннотацию `@AroundInvoke` для обозначения методов Interceptor-ов для методов управляемого объекта. Для класса разрешается только один метод Interceptor-а типа Around-invoke. Методы Interceptor-а типа Around-invoke имеют следующую форму:

JAVA

```java
@AroundInvoke
visibility Object method-name(InvocationContext) throws Exception { ... }
```

Например:

JAVA

```java
@AroundInvoke
public void interceptOrder(InvocationContext ctx) { ... }
```

Методы Interceptor-а типа Around-invoke могут иметь публичный, приватный, защищённый или пакетный доступ и не должны объявляться как статические или final.

Interceptor типа Around-invoke может вызывать любой компонент или ресурс, вызывающийся целевым методом, может иметь тот же контекст безопасности и транзакции, что и целевой метод, и может работать в том же стеке вызовов виртуальной машины Java, что и целевой метод.

Interceptor-ы типа Around-invoke могут генерировать исключения во время выполнения и любое исключение, разрешённое предложением `throws` целевого метода. Они могут перехватывать и подавлять исключения, а затем восстанавливаться, вызывая метод `InvocationContext.proceed`.

Использование нескольких Interceptor-ов с методом

Используйте аннотацию `@Interceptors`, чтобы объявить несколько Interceptor-ов для целевого метода или класса:

JAVA

```java
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class,
        LastInterceptor.class})
public void updateInfo(String info) { ... }
```

Порядок Interceptor-ов в аннотации `@Interceptors` определяет порядок вызова Interceptor-ов.

Вы также можете определить несколько Interceptor-ов в дескрипторе развёртывания. Порядок Interceptor-ов в дескрипторе развёртывания определяет порядок, в котором будут вызываться Interceptor-ы:

XML

```
...
<interceptor-binding>
    <target-name>myapp.OrderBean</target-name>
    <interceptor-class>myapp.PrimaryInterceptor.class</interceptor-class>
    <interceptor-class>myapp.SecondaryInterceptor.class</interceptor-class>
    <interceptor-class>myapp.LastInterceptor.class</interceptor-class>
    <method-name>updateInfo</method-name>
</interceptor-binding>
...
```

Чтобы явно передать управление следующему Interceptor-у в цепочке, вызовите метод `InvocationContext.proceed`.

Данные могут быть разделены между Interceptor-ами.

- Один и тот же объект `InvocationContext` передаётся в качестве входного параметра каждому методу Interceptor-а в цепочке Interceptor-ов для конкретного целевого метода. Свойство `contextData` объекта `InvocationContext` используется для передачи данных через методы Interceptor-а. Свойство `contextData` является объектом `java.util.Map<String, Object>`. Данные, хранящиеся в `contextData`, доступны для методов Interceptor-а далее по цепочке Interceptor-ов.
- Данные, хранящиеся в `contextData`, не могут использоваться совместно для отдельных вызовов методов целевого класса. То есть для каждого вызова метода в целевом классе создаётся отдельный объект `InvocationContext`.

Доступ к параметрам целевого метода из класса Interceptor-а

Вы можете использовать объект `InvocationContext`, передаваемый каждому методу типа Around-invoke, для доступа и изменения параметров целевого метода. Свойство `parameters` в `InvocationContext` представляет собой массив объектов `Object`, который соответствует порядку параметров целевого метода. Например, для следующего целевого метода свойство `parameters` в объекте `InvocationContext`, передаваемом методу Interceptor-а типа Around-invoke в `PrimaryInterceptor`, имеет вид массива `Object`, содержащий два объекта `String` (`firstName` и `lastName`) и объект `Date` (`date`):

JAVA

```
@Interceptors(PrimaryInterceptor.class)
public void updateInfo(String firstName, String lastName, Date date) { ... }
```

Вы можете получить доступ к параметрам и изменить их с помощью методов `InvocationContext.getParameters` и `InvocationContext.setParameters` соответственно.

Перехват событий Callback-вызова жизненного цикла

Interceptor-ы для событий Callback-вызова жизненного цикла (around-construct, post-construct и pre-destroy) могут быть определены в целевом классе или в классах Interceptor-ов. Аннотация `jakarta.interceptor.AroundConstruct` обозначает метод как метод Interceptor-а, который выполняется при вызове конструктора целевого класса. Аннотация `jakarta.annotation.PostConstruct` используется для обозначения метода как Interceptor-а событий post-construct жизненного цикла. Аннотация `jakarta.annotation.PreDestroy` используется для обозначения метода как Interceptor-а событий pre-destroy жизненного.

Interceptor-ы событий жизненного цикла, определённые в целевом классе, имеют следующую форму:

JAVA

```
void method-name() { ... }
```

Например:

```java
@PostConstruct
void initialize() { ... }
```

Interceptor-ы событий жизненного цикла, определённые в классе Interceptor-ов, имеют следующую форму:

```java
void method-name(InvocationContext) { ... }
```

Например:

```java
@PreDestroy
void cleanup(InvocationContext ctx) { ... }
```

Методы Interceptor-а жизненного цикла могут иметь публичный, приватный, защищённый или пакетный доступ и не должны объявляться как статические или final. Interceptor-ы жизненного цикла могут генерировать исключения во время выполнения, но не могут генерировать проверяемые исключения.

Методы Interceptor-а жизненного цикла вызываются в неопределённом контексте безопасности и транзакции. То есть переносимые приложения Jakarta EE не должны предполагать, что метод Interceptor-а событий жизненного цикла имеет доступ к контексту безопасности или транзакции. Только один метод Interceptor-а для каждого события жизненного цикла (post-create и pre-destroy) разрешён для каждого класса.

Использование методов перехвата AroundConstruct

Методы `@AroundConstruct` вставляются в вызов конструктора целевого класса. Методы, аннотированные `@AroundConstruct`, могут быть определены только внутри классов Interceptor-ов или родительских классов Interceptor-ов. Нельзя использовать методы `@AroundConstruct` в целевом классе.

Метод `@AroundConstruct` вызывается после завершения инъецирования всех Interceptor-ов, связанных с целевым классом. Целевой класс создаётся, и инъецирование конструктора целевого класса выполняется после того, как все связанные методы `@AroundConstruct` вызвали метод `Invocation.proceed`. В этот момент инъецирование зависимостей для целевого класса завершается, и затем выполняются вызовы Callback-метода `@PostConstruct`.

Методы `@AroundConstruct` могут обращаться к созданному целевому объекту после вызова `Invocation.proceed` путём вызова метода `InvocationContext.getTarget`.

> Вызов методов на целевом объекте из метода `@AroundConstruct` опасен, поскольку инъецирование на целевом объекте может быть не завершено.

Методы `@AroundConstruct` должны вызывать `Invocation.proceed` для создания целевого объекта. Если метод `@AroundConstruct` не вызывает `Invocation.proceed`, целевой объект не будет создан.

Использование нескольких Interceptor-ов Callback-методов жизненного цикла

Вы можете определить несколько Interceptor-ов жизненного цикла для целевого класса, указав классы Interceptor-ов в аннотации `@Interceptors`:

JAVA

```
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class,
        LastInterceptor.class})
@Stateless
public class OrderBean { ... }
```

Данные, хранящиеся в свойстве `contextData` у `InvocationContext`, не доступны для разных событий жизненного цикла.

Перехват событий таймера

Вы можете определить Interceptor-ы для методов-обработчиков событий сервиса EJB-таймера, используя аннотацию `@AroundTimeout` на методе в целевом классе или в классе Interceptor-а. Для класса разрешается только один метод `@AroundTimeout`.

Interceptor-ы событий таймера имеют следующий формат:

JAVA

```
Object method-name(InvocationContext) throws Exception { ... }
```

Например:

JAVA

```
@AroundTimeout
protected void timeoutInterceptorMethod(InvocationContext ctx) { ... }
```

Методы Interceptor-ов событий таймера могут иметь публичный, приватный, защищённый или пакетный доступ и не должны быть статическими или final.

Interceptor-ы могут вызывать любой компонент или ресурс, вызываемый целевым методом таймера, и вызываются в той же транзакции и в том же контексте безопасности, что и целевой метод.

Interceptor-ы могут обращаться к объекту таймера, связанному с целевым методом таймера, с помощью метода `getTimer` объекта `InvocationContext`.

Использование нескольких Interceptor-ов событий таймера

Вы можете определить несколько Interceptor-ов для целевого класса, указав классы Interceptor-ов, содержащие методы `@AroundTimeout`, в аннотации `@Interceptors` на уровне класса.

Если целевой класс задаёт Interceptor-ы событий таймера и сам также имеет метод `@AroundTimeout`, то методы, задаваемые Interceptor-ами событий таймера, вызываются первыми, а только затем метод Interceptor-а в целевом классе. Например, в следующем примере предположим, что классы `PrimaryInterceptor` и `SecondaryInterceptor` содержат методы Interceptor-а событий таймера:

JAVA

```
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class})
@Stateful
public class OrderBean {
    ...
    @AroundTimeout
    private void last(InvocationContext ctx) { ... }
    ...
}
```

Сначала будет вызван метод Interceptor-а в `PrimaryInterceptor`, затем в `SecondaryInterceptor` и, наконец, метод `last`, определённый в целевом классе.

Связывание Interceptor-ов с компонентами

Типы привязки Interceptor-а — это аннотации, которые можно применять к компонентам, чтобы связать их с конкретным Interceptor-ом. Типы привязки Interceptor-а, как правило, представляют собой пользовательские типы аннотаций времени выполнения, которые определяют цель Interceptor-а. Используйте аннотацию `jakarta.interceptor.InterceptorBinding` для задания кастомной аннотации и укажите целевой объект с помощью `@Target`, задав один или несколько `TYPE` (Interceptor-ы уровня класса), `METHOD` (Interceptor-ы уровня метода), `CONSTRUCTOR` (Interceptor-ы конструктора) или любые другие допустимые целевые значения:

JAVA

```
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
@Inherited
pubic @interface Logged { ... }
```

Типы связывания с Interceptor-ами могут также применяться к другим типам связывания с Interceptor-ами:

JAVA

```
@Logged
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
@Inherited
public @interface Secured { ... }
```

Объявление привязок Interceptor-а на классе Interceptor-а

Аннотируйте класс Interceptor-а с типом привязки Interceptor-а и аннотацией `@Interceptor`, чтобы связать привязку Interceptor-а с классом Interceptor-а:

JAVA

```
@Logged
@Interceptor
public class LoggingInterceptor {
    @AroundInvoke
    public Object logInvocation(InvocationContext ctx) throws Exception { ... }
    ...
}
```

Класс Interceptor-а может объявлять несколько типов привязки Interceptor-а, и более одного класса Interceptor-а может объявлять тип привязки Interceptor-а.

Если класс Interceptor-а перехватывает вызовы Callback-методов жизненного цикла, он может объявлять только типы привязки Interceptor-а с помощью `Target(TYPE)` или, в случае `@AroundConstruct` вызовов Callback-методов жизненного цикла, `Target(CONSTRUCTOR)`.

Привязка компонента к Interceptor-у

Добавьте аннотацию типа привязки Interceptor-а к классу, методу или конструктору целевого компонента. Типы привязки Interceptor-ов применяются с использованием тех же правил, что и аннотации `@Interceptor`:

JAVA

```
@Logged
public class Message {
    ...
    @Secured
    public void getConfidentialMessage() { ... }
    ...
}
```

Если компонент имеет привязку Interceptor-а на уровне класса, он не должен быть `final` или иметь какие-либо не-`static`, не-`private final` методы. Если к не-`static`, не-`private` методу применяется привязка Interceptor-а, он не должен быть `final`, а класс компонента не может быть `final`.

Упорядочение Interceptor-ов

Если Interceptor-ов несколько, порядок их вызова определяется следующими правилами.

- Interceptor-ы по умолчанию определены в дескрипторе развёртывания и вызываются первыми. Они могут указывать порядок вызова или переопределять порядок, указанный аннотациями. Interceptor-ы по умолчанию вызываются в том порядке, в котором они определены в дескрипторе развёртывания.
- Порядок перечисления классов Interceptor-ов в аннотации `@Interceptors` определяет порядок вызова этих Interceptor-ов. Любые настройки `@Priority` для Interceptor-ов, перечисленные в аннотации `@Interceptors`, игнорируются.
- Если класс Interceptor-а имеет родительские классы, Interceptor-ы, определённые в родительских классах, вызываются первыми, начиная с самого общего родительского класса.
- Классы Interceptor-ов могут устанавливать приоритет методов Interceptor-а, установив значение в аннотации `jakarta.annotation.Priority`.
- После вызова Interceptor-ов, определённых в классах Interceptor-ов, конструктор целевого класса, Interceptor-ы типа Around-invoke или тайм-аута запускаются в том же порядке, что и Interceptor-ы в аннотации `@Interceptors`.
- Если целевой класс имеет родительские классы, любые Interceptor-ы, определённые в родительских классах, вызываются первыми, начиная с самого общего родительского класса.

Аннотация `@Priority` требует указание значения типа `int`. Чем меньше число, тем выше приоритет соответствующего Interceptor-а.

Порядок вызова Interceptor-ов с одинаковым значением приоритета зависит от реализации.

Класс `jakarta.interceptor.Interceptor.Priority` определяет константы приоритета, перечисленные в таблице 58-2.

Таблица 58-2 Константы приоритета Interceptor-а

Приоритет Константа	**Значение**	**Описание**
`PLATFORM_BEFORE`	0	Interceptor-ы, определённые платформой Jakarta EE и предназначенные для раннего вызова в цепочке вызовов, должны использовать диапазон между `PLATFORM_BEFORE` и `LIBRARY_BEFORE`. Эти Interceptor-ы имеют самый высокий приоритет.
`LIBRARY_BEFORE`	1000	Interceptor-ы, определённые библиотеками расширений, которые должны вызываться на ранних этапах цепочки Interceptor-ов, должны использовать диапазон между `LIBRARY_BEFORE` и `APPLICATION`.
`APPLICATION`	2000	Interceptor-ы, определённые приложениями, должны использовать диапазон между `APPLICATION` и `LIBRARY_AFTER`.

Приоритет Константа	Значение	Описание
`LIBRARY_AFTER`	3000	Interceptor-ы с низким приоритетом, определённые библиотеками расширений, должны использовать диапазон между `LIBRARY_AFTER` и `PLATFORM_AFTER`.
`PLATFORM_AFTER`	4000	Interceptor-ы с низким приоритетом, определённые платформой Jakarta EE, должны иметь значения выше, чем `PLATFORM_AFTER`.

Отрицательные значения приоритета зарезервированы спецификацией Interceptors для будущего использования и не должны использоваться.

В следующем фрагменте кода показано, как использовать константы приоритета в определяемом приложением Interceptor-е:

JAVA

```java
@Interceptor
@Priority(Interceptor.Priority.APPLICATION+200)
public class MyInterceptor { ... }
```

Пример interceptor

Пример `interceptor` демонстрирует, как использовать класс Interceptor-а, содержащий метод `@AroundInvoke`, с сессионным компонентом без сохранения состояния.

Сессионный компонент `HelloBean` без сохранения состояния — это простой Enterprise-бин с двумя бизнес-методами, `getName` и `setName`, для извлечения и изменения строки. Бизнес-метод `setName` имеет аннотацию `@Interceptors`, которая задаёт класс Interceptor-а `HelloInterceptor` для этого метода:

JAVA

```java
@Interceptors(HelloInterceptor.class)
public void setName(String name) {
    this.name = name;
}
```

Класс `HelloInterceptor` определяет метод Interceptor-а `@AroundInvoke`, `modifyGreeting`, который преобразует строку, переданную в `HelloBean.setName`, к нижнему регистру:

JAVA

```java
@AroundInvoke
public Object modifyGreeting(InvocationContext ctx) throws Exception {
    Object[] parameters = ctx.getParameters();
    String param = (String) parameters[0];
    param = param.toLowerCase();
    parameters[0] = param;
    ctx.setParameters(parameters);
    try {
        return ctx.proceed();
    } catch (Exception e) {
        logger.warning("Error calling ctx.proceed in modifyGreeting()");
        return null;
    }
}
```

Параметры для `HelloBean.setName` извлекаются и сохраняются в массиве `Object` путём вызова метода `InvocationContext.getParameters`. Поскольку `setName` имеет только один параметр, он является первым и единственным элементом в массиве. Строка преобразуется в нижний регистр и сохраняется в массиве `parameters`, а затем передаётся в `InvocationContext.setParameters`. Чтобы вернуть управление сессионному компоненту, вызывается `InvocationContext.proceed`.

Пользовательский интерфейс `interceptor` — это веб-приложение JavaServer Faces, состоящее из двух представлений Facelets: `index.xhtml`, содержащем форму для ввода имени и `response.xhtml`, в котором отображается окончательное имя.

Запуск interceptor

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска примера `interceptor`.

Запуск interceptor с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню «Файл» выберите «Открыть проект».
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/ejb
   ```

4. Выберите каталог `interceptor` и нажмите «Открыть проект».
5. На вкладке «Проекты» кликните правой кнопкой мыши проект `interceptor` и выберите «Выполнить».

   Это скомпилирует, развернёт и запустит пример `interceptor`, открыв веб-браузер по следующему URL:

   ```
   http://localhost:8080/interceptor/
   ```

6. Введите имя в форму и нажмите Отправить.

   Имя будет преобразовано в строчные буквы с помощью метода Interceptor-а, определённого в классе `HelloInterceptor`.

Запуск interceptor с помощью Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. Перейдите в следующий каталог:

   ```
   tut-install/examples/ejb/interceptor/
   ```

3. Чтобы скомпилировать исходные файлы и упаковать приложение, используйте следующую команду:

   ```
   mvn install
   ```
   SHELL

   Эта команда собирает и упаковывает приложение в WAR-файл, `interceptor.war`, расположенный в каталоге `target`. WAR-файл затем развёртывается в GlassFish Server.
4. Откройте следующий URL в веб-браузере:

   ```
   http://localhost:8080/interceptor/
   ```

5. Введите имя в форму и нажмите Отправить.

Имя будет преобразовано в строчные буквы с помощью метода Interceptor-а, определённого в классе `HelloInterceptor`.

Глава 59. Пакетная обработка

В этой главе описывается Jakarta Batch, который обеспечивает поддержку для определения, реализации и выполнения пакетных заданий. Пакетные задачи — это задачи, которые могут быть выполнены без взаимодействия с пользователем. Фреймворк пакетной обработки состоит из языка спецификации задач, основанного на XML, Java API и пакетной среды выполнения.

Введение в пакетную обработку

Некоторые корпоративные приложения содержат задачи, которые могут быть выполнены без взаимодействия с пользователем. Эти задачи выполняются периодически или при низком использовании ресурсов, и они часто обрабатывают больши́е объёмы информации, такие как файлы журналов, записи базы данных или изображения. Например, выставление счетов, генерация отчётов, преобразование формата данных и обработка изображений. Эти задачи называются пакетными заданиями.

Пакетная обработка относится к запуску пакетных заданий в компьютерной системе. Jakarta EE содержит фреймворк пакетной обработки, который обеспечивает инфраструктурную поддержку выполнения пакетных приложений и позволяет разработчикам сосредоточиться на бизнес-логике. Фреймворк пакетной обработки состоит из языка спецификации задания на основе XML, набора аннотаций пакета и интерфейсов для классов приложений, реализующих бизнес-логику, контейнера пакета, управляющего выполнением пакетных заданий, и поддержки классов и интерфейсов для взаимодействия с контейнером.

Пакетное задание может быть выполнено без вмешательства пользователя. Например, рассмотрим приложение для выставления счетов по телефону, которое считывает записи телефонных звонков из информационных систем предприятия и генерирует ежемесячный счёт для каждой учётной записи. Поскольку это приложение не требует взаимодействия с пользователем, оно может запускаться как пакетное задание.

Приложение для выставления счетов по телефону состоит из двух фаз: на первой фазе каждый звонок из реестра связывается с ежемесячным счётом, на второй фазе рассчитываются налог и общая сумма, причитающиеся за каждый счёт. Каждая из этих фаз является шагом пакетного задания.

Пакетные приложения определяют набор шагов и порядок их выполнения. Различные фреймворки пакетной обработки могут указывать дополнительные элементы, такие как элементы решения или группы шагов, которые выполняются параллельно. В следующих разделах шаги описываются более подробно и предоставляется информация о других общих характеристиках фреймворка пакетной обработки.

Шаги в пакетных заданиях

Шаг — это независимая и последовательная фаза пакетного задания. Пакетные задания содержат шаги, ориентированные на обработку фрагментов данных, и шаги, ориентированные на задачи.

- Ориентированные на обработку фрагментов данных шаги (шаги фрагментов) обрабатывают данные путём чтения элементов из источника данных, применения некоторой бизнес-логики к каждому элементу и сохранения результатов. Шаги фрагментов читают и обрабатывают один элемент за раз и группируют фрагмент с результатами. Результаты сохраняются, когда фрагмент достигает заранее установленного размера. Ориентированная на фрагменты обработка делает хранение результатов более эффективным и облегчает разграничение транзакций.

  Шаги фрагментов состоят из трёх частей.

  - Часть извлечения ввода считывает один элемент за раз из источника данных, например записи в базе данных, файлы в каталоге или записи в файле журнала.

- Часть бизнес-обработки управляет одним элементом за раз, используя бизнес-логику, определённую приложением. Например, фильтрация, форматирование и доступ к данным из элемента для вычисления результата.
- Часть записи вывода хранит фрагмент одновременно обработанных элементов.

Шаги фрагментов часто выполняются длительное время, поскольку обрабатывают большие объёмы данных. Фреймворки пакетной обработки позволяют шагам фрагментов отмечать прогресс по контрольным точкам. Прерванный шаг фрагмента может быть перезапущен с последней контрольной точки. Части извлечения ввода и записи вывода шага фрагмента сохраняют свою текущую позицию после обработки каждого фрагмента и могут восстанавливать его после перезапуска шага.

Рисунок 59-1 показывает три части из двух шагов в пакетном задании.

Рис. 59-1 Шаги фрагмента в пакетном задании

Например, приложение для выставления счетов за телефон состоит из двух фрагментов.

- На первом этапе часть извлечения ввода считывает записи вызовов из реестра. Часть бизнес-обработки связывает каждый вызов со счётом и создаёт счёт, если он не существует для учётной записи. Часть записи вывода сохраняет каждый счёт в базе данных.
- На втором шаге часть извлечения ввода считывает счета из базы данных. Часть бизнес-обработки вычисляет налог и общую сумму, причитающуюся за каждый счёт. Часть записи вывода обновляет записи базы данных и генерирует печатные версии каждого счёта.

Это приложение также может содержать шаг задачи, который очищает файлы от счетов, сгенерированных за предыдущий месяц.

Параллельная обработка

Пакетные задания часто обрабатывают большие объёмы данных или выполняют дорогие в вычислительном отношении операции. Пакетные приложения могут выиграть от параллельной обработки в двух случаях.

- Шаги, которые не зависят друг от друга, могут выполняться в разных потоках (thread).
- Ориентированные на фрагменты шаги, на которых обработка каждого элемента не зависит от результатов обработки предыдущих элементов, могут выполняться более чем в одном потоке (thread).

Фреймворки пакетной обработки предоставляют разработчикам механизмы для определения групп независимых шагов и разделения ориентированных на фрагменты шагов на части, которые могут выполняться параллельно.

Элементы статуса и принятия решения

Фреймворки пакетной обработки отслеживают статус для каждого шага в задании. Статус указывает, выполняется шаг или завершён. Если шаг завершён, статус имеет одно из следующих значений:

- Выполнение шага прошло успешно.
- Шаг был прерван.
- Произошла ошибка при выполнении шага.

Дополнительно к шагам пакетные задания могут также содержать элементы принятия решения. Элементы принятия решения используют статус выхода предыдущего шага, чтобы определить следующий шаг или завершить выполнение пакетного задания. Элементы принятия решения устанавливают статус пакетного задания при его завершении. Как и шаг, пакетное задание может завершиться успешно, с ошибкой или быть прервано.

Рисунок 59-2 показывает пример задания, которое содержит шаги фрагмента, шаги задачи и элемент принятия решения.

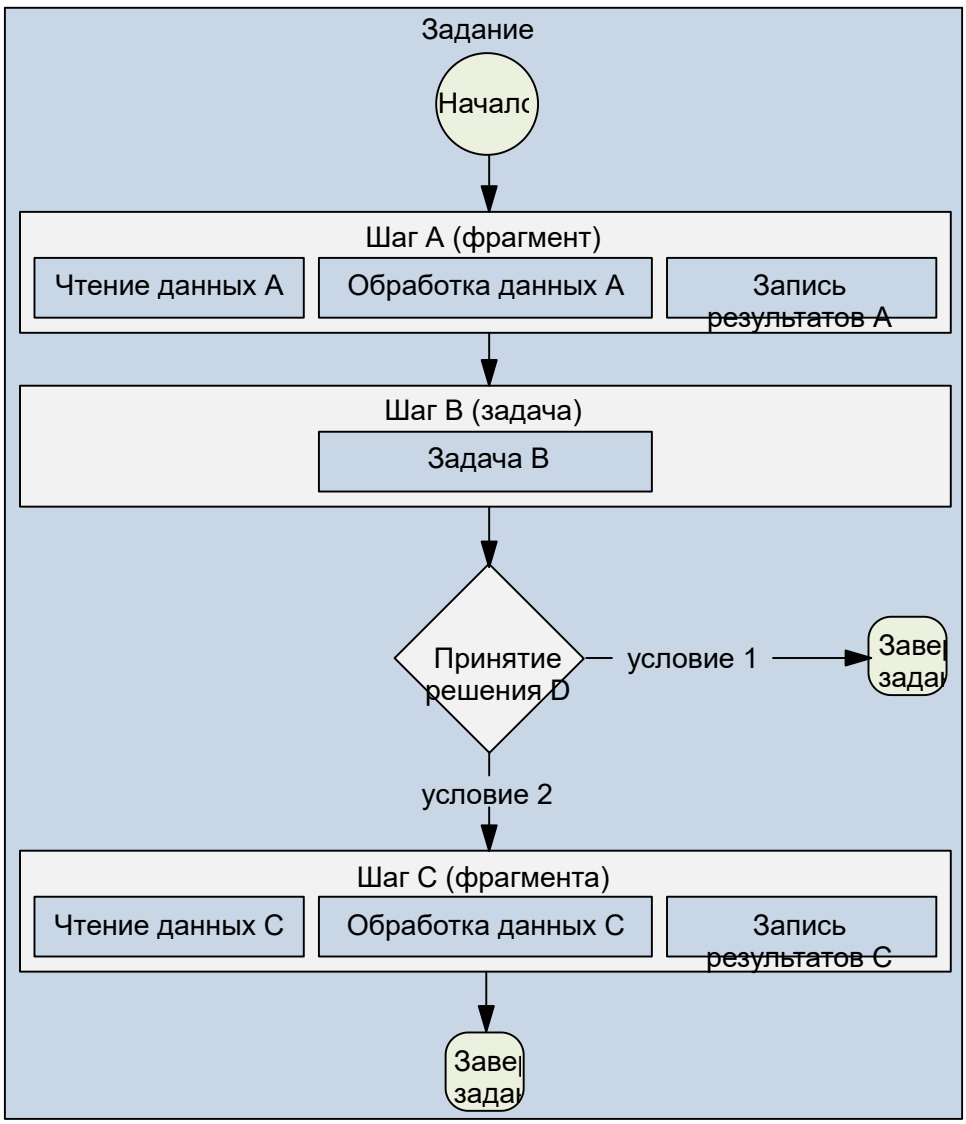

Рисунок 59-2 Шаги и элементы принятия решения в пакетной задаче

Функциональность фреймворка пакетной обработки

К пакетным приложениям предъявляются следующие требования:

- Определение задания, шагов, элементов принятия решения и отношений между ними.

- Выполнение несколько групп шагов или частей шага параллельно.
- Сохранение информации о состоянии заданий и шагов.
- Запуск задания и возобновление прерванных заданий.
- Обработка ошибок.

Фреймворки пакетной обработки предоставляют инфраструктуру, которая отвечает общим требованиям выполнения всех пакетных приложений, позволяя разработчикам сосредоточиться на бизнес-логике приложений. Фреймворки пакетной обработки состоят из формата описания заданий и шагов, API и сервиса, доступного во время выполнения, который управляет выполнением пакетных заданий.

Пакетная обработка в Jakarta EE

В этом разделе перечислены компоненты фреймворка пакетной обработки Jakarta EE и представлен обзор шагов, которые необходимо выполнить для создания пакетного приложения.

Фреймворк пакетной обработки

Jakarta EE включает фреймворк пакетной обработки, который состоит из следующих элементов:

- Пакетная среда выполнения, которая управляет выполнением заданий
- Язык спецификации задания на основе XML
- Java API для взаимодействия с пакетной средой выполнения
- Java API для реализации шагов, элементов принятия решения и других пакетных артефактов

Пакетные приложения в Jakarta EE содержат файлы XML и классы Java. XML-файлы определяют структуру задания в терминах пакетных артефактов и отношений между ними. (Артефакт пакетной обработки является частью ориентированного на фрагменты шага, ориентированного на задачу шага, элемента принятия решения или другого компонента пакетного приложения). Классы Java реализуют прикладную логику пакетных артефактов, определённых в файлах XML. Среда выполнения пакета анализирует файлы XML и загружает артефакты пакета как классы Java для запуска заданий в пакетном приложении.

Создание пакетных приложений

Процесс создания пакетного приложения в Jakarta EE заключается в следующем:

1. Разработайте пакетное приложение.
    a. Определите источники ввода, формат входных данных, желаемый конечный результат и необходимые этапы обработки.
    b. Организуйте приложение как задание с шагами, ориентированными на фрагменты или задачи, и элементами принятия решения. Определите зависимости между ними.
    c. Определите порядок выполнения в терминах переходов между шагами.
    d. Определите шаги, которые могут выполняться параллельно, и шаги, которые могут выполняться в нескольких потоках (thread).
2. Создайте пакетные артефакты как классы Java, реализуя интерфейсы, определённые фреймворком для шагов, элементов принятия решения и т. д. Эти классы Java содержат код для чтения данных из входных источников, элементов форматирования, элементов обработки и сохранения результатов. Пакетные артефакты могут обращаться к объектам контекста из пакетной среды выполнения с помощью инъецирования зависимостей.

3. Определите задания, шаги и поток (flow) их выполнения в файлах XML с помощью языка спецификации заданий. Элементы в файлах XML ссылаются на пакетные артефакты, реализованные в классах Java. Пакетные артефакты могут обращаться к свойствам, объявленным в файлах XML, таким как имена файлов и баз данных.
4. Используйте Java API, предоставляемый пакетной средой выполнения, для запуска пакетного приложения.

В следующих разделах подробно описывается, как использовать компоненты фреймворка пакетной обработки в Jakarta EE для создания пакетных приложений.

Элементы пакетного задания

Пакетное задание может содержать один или несколько из следующих элементов:

- Шаги
- Потоки (Flow)
- Разделители
- Элементы принятия решения

Шаги описаны во Введении в пакетную обработку и могут быть ориентированы на фрагменты или задачи. Ориентированные на фрагменты шаги могут быть разделёнными шагами. В разделённом шаге обработка одного фрагмента не зависит от других фрагментов, поэтому эти шаги могут выполняться в нескольких потоках (thread).

Поток (flow) — это последовательность шагов, которые выполняются как единое целое. Последовательность связанных шагов может быть сгруппирована в поток (flow). Шаги в потоке (flow) не могут переходить к шагам вне потока (flow). Поток (flow) переходит к следующему элементу, когда завершается его последний шаг.

Разделитель — это набор потоков (flow), которые выполняются параллельно. Каждый поток (flow) работает в отдельном потоке (thread). Разделитель переходит к следующему элементу, когда все его потоки (flow) завершены.

Элементы принятия решения используют статус выхода предыдущего шага, чтобы определить следующий шаг или завершить выполнение пакетного задания.

Свойства и параметры

Задания и шаги могут иметь ряд связанных свойств. Свойства определяются в файле определения задания, и артефакты пакета получают доступ к этим свойствам с помощью объектов контекста из пакетной среды выполнения. Таким образом, использование свойств позволяет отделить статические параметры задания от бизнес-логики и повторно использовать пакетные артефакты в разных файлах определения задания.

Определение свойств описано в Использование языка спецификации задания, а доступ к свойствам в пакетных артефактах описан в Создание пакетных артефактов.

Приложения Jakarta EE также могут передавать параметры в задание, когда они отправляют его в пакетную среду выполнения. Это позволяет указать динамические параметры, которые известны только во время выполнения. Параметры также необходимы для разделённых шагов, поскольку каждое разделение должно знать, например, какой диапазон элементов обрабатывать.

Задание параметров при отправке заданий описано в разделе Отправка заданий в пакетную среду выполнения. Задание параметров для разделённых шагов и доступ к ним в пакетных артефактах продемонстрированы в Пример phonebilling.

Задания и их выполнение

Задание может быть инстанцировано несколько раз, каждый раз с разными параметрами. Выполнение задания — это попытка запустить на выполнение его инстанцированный объект. Пакетная среда выполнения содержит информацию об объектах задания и его выполнении, как описано в Проверка статуса задания.

Статус пакета и статус завершения

Состояние заданий, шагов, разделителей и потоков (flow) представляется во время выполнения пакета в виде значений статуса пакета. Значения статуса пакетного выполнения перечислены таблице 59-1. Они представлены в виде строк.

Таблица 59-1 Значения состояния пакетного выполнения

Значение	Описание
`STARTING`	Задание было отправлено на выполнение.
`STARTED`	Задание выполняется.
`STOPPING`	Была запрошена остановка задания.
`STOPPED`	Задание остановлено.
`FAILED`	Выполнение задания завершено из-за ошибки.
`COMPLETED`	Задание успешно завершено.
`ABANDONED`	Задание было помечено как брошенное.

Приложения Jakarta EE могут отправлять задания и получать доступ к состоянию пакета задания с помощью интерфейса `JobOperator`, как описано в Отправке заданий в пакетную среду выполнения. Файлы определения задания могут ссылаться на значения состояния пакета с использованием языка спецификации задания (JSL), как описано в Использование языка спецификации задания. Пакетные артефакты могут обращаться к значениям состояния пакета, используя объекты контекста, как описано в Использование объектов контекста из пакетной среды выполнения.

Для потоков (flow) статусом выполнения является статус последнего шага. Для разделителей статус выполнения может принимать следующие значения:

- `COMPLETED`: если все его потоки (flow) имеют статус выполнения `COMPLETED`
- `FAILED`: если какой-либо поток (flow) имеет статус выполнения `FAILED`
- `STOPPED`: если какой-либо поток (flow) имеет статус выполнения `STOPPED`, и ни один из потоков (flow) не имеет статуса выполнения `FAILED`

Статус выполнения для заданий, шагов, разделителей и потоков (flow) задаётся во время выполнения пакета. Задания, шаги, разделители и потоки (flow) также имеют статус завершения, который определяется пользователем в зависимости от статуса обработки. Установить статус завершения можно внутри артефактов пакета или в файле определения задания. Получить доступ к статусу завершения можно так же, как и к статусу выполнения, как описано выше. Значение по умолчанию для статуса завершения совпадает со статусом выполнения.

Простой вариант использования

В этом разделе показано, как определить простое задание с использованием языка спецификации заданий (Job Specification Language — JSL) и как реализовать соответствующие пакетные артефакты. Обратитесь к остальным разделам этой главы для подробного описания элементов во фреймворке пакетной обработки.

Следующее определение задания описывает шаг фрагмента и шаг задачи следующим образом:

XML

```xml
<job id="simplejob" xmlns="https://jakarta.ee/xml/ns/jakartaee"
                    version="2.0">
  <properties>
    <property name="input_file" value="input.txt"/>
    <property name="output_file" value="output.txt"/>
  </properties>
  <step id="mychunk" next="mytask">
    <chunk>
      <reader ref="MyReader"></reader>
      <processor ref="MyProcessor"></processor>
      <writer ref="MyWriter"></writer>
    </chunk>
  </step>
  <step id="mytask">
    <batchlet ref="MyBatchlet"></batchlet>
    <end on="COMPLETED"/>
  </step>
</job>
```

Шаг фрагмента

В большинстве случаев нужно реализовать класс контрольных точек для шагов, ориентированных на обработку фрагментов. Следующий класс просто отслеживает номер строки в текстовом файле:

JAVA

```java
public class MyCheckpoint implements Serializable {
    private long lineNum = 0;
    public void increase() { lineNum++; }
    public long getLineNum() { return lineNum; }
}
```

Следующая реализация средства чтения элементов продолжает чтение входного файла с предоставленной контрольной точки, если задание было перезапущено. Элементами являются отдельные строки в текстовом файле (в более сложных сценариях элементы являются пользовательскими типами Java, а источником ввода может быть база данных):

JAVA

```java
@Dependent
@Named("MyReader")
public class MyReader implements jakarta.batch.api.chunk.ItemReader {
    private MyCheckpoint checkpoint;
    private BufferedReader breader;
    @Inject
    JobContext jobCtx;

    public MyReader() {}

    @Override
    public void open(Serializable ckpt) throws Exception {
        if (ckpt == null)
            checkpoint = new MyCheckpoint();
        else
            checkpoint = (MyCheckpoint) ckpt;
        String fileName = jobCtx.getProperties()
                                .getProperty("input_file");
        breader = new BufferedReader(new FileReader(fileName));
        for (long i = 0; i < checkpoint.getLineNum(); i++)
            breader.readLine();
    }

    @Override
    public void close() throws Exception {
        breader.close();
    }

    @Override
    public Object readItem() throws Exception {
        String line = breader.readLine();
        return line;
    }
}
```

В следующем случае обработчик элементов приводит строку к верхнему регистру. Более сложные примеры могут обрабатывать элементы различными способами или преобразовывать их в пользовательские выходные типы Java:

JAVA

```java
@Dependent
@Named("MyProcessor")
public class MyProcessor implements jakarta.batch.api.chunk.ItemProcessor {
    public MyProcessor() {}

    @Override
    public Object processItem(Object obj) throws Exception {
        String line = (String) obj;
        return line.toUpperCase();
    }
}
```

> API пакетной обработки не поддерживает обобщённые типы (generic). В большинстве случаев нужно привести объекты к конкретному типу перед обработкой.

Средство записи элементов записывает обработанные элементы в выходной файл. Оно перезаписывает выходной файл, если контрольная точка не указана. В противном случае возобновляет запись в конец файла. Элементы записываются фрагментами:

JAVA

```java
@Dependent
@Named("MyWriter")
public class MyWriter implements jakarta.batch.api.chunk.ItemWriter {
    private BufferedWriter bwriter;
    @Inject
    private JobContext jobCtx;

    @Override
    public void open(Serializable ckpt) throws Exception {
        String fileName = jobCtx.getProperties()
                                .getProperty("output_file");
        bwriter = new BufferedWriter(new FileWriter(fileName,
                                                    (ckpt != null)));
    }

    @Override
    public void writeItems(List<Object> items) throws Exception {
        for (int i = 0; i < items.size(); i++) {
            String line = (String) items.get(i);
            bwriter.write(line);
            bwriter.newLine();
        }
    }

    @Override
    public Serializable checkpointInfo() throws Exception {
        return new MyCheckpoint();
    }
}
```

Шаг задачи

Шаг задачи отображает длину выходного файла. В более сложных сценариях шаги задачи выполняют функции, которые не соответствуют программной модели обработки фрагментами:

JAVA

```java
@Dependent
@Named("MyBatchlet")
public class MyBatchlet implements jakarta.batch.api.chunk.Batchlet {
    @Inject
    private JobContext jobCtx;

    @Override
    public String process() throws Exception {
        String fileName = jobCtx.getProperties()
                                .getProperty("output_file");
        System.out.println(""+(new File(fileName)).length());
        return "COMPLETED";
    }
}
```

Использование языка спецификации задания

Язык спецификации задания (JSL) позволяет определить шаги в задании и порядок их выполнения, используя файл XML. В следующем примере показано, как определить простое задание, содержащее один шаг фрагмента и один шаг задачи:

```xml
<job id="loganalysis" xmlns="https://jakarta.ee/xml/ns/jakartaee"
                      version="2.0">
  <properties>
    <property name="input_file" value="input1.txt"/>
    <property name="output_file" value="output2.txt"/>
  </properties>

  <step id="logprocessor" next="cleanup">
    <chunk checkpoint-policy="item" item-count="10">
      <reader ref="com.example.pkg.LogItemReader"></reader>
      <processor ref="com.example.pkg.LogItemProcessor"></processor>
      <writer ref="com.example.pkg.LogItemWriter"></writer>
    </chunk>
  </step>

  <step id="cleanup">
    <batchlet ref="com.example.pkg.CleanUp"></batchlet>
    <end on="COMPLETED"/>
  </step>
</job>
```

В этом примере определяется задание `loganalysis`, которое состоит из шага фрагмента `logprocessor` и шага задачи `cleanup`. Шаг `logprocessor` переходит к шагу `cleanup`, при завершении которого завершается и всё задание.

Элемент `job` определяет два свойства: `input_file` и `output_file`. Указание свойств таким способом позволяет запускать пакетное задание с различными параметрами конфигурации без необходимости перекомпиляции его пакетных артефактов Java. Пакетные артефакты могут обращаться к этим свойствам с помощью объектов контекста из пакетной среды выполнения.

Шаг `logprocessor` является шагом фрагмента, который указывает пакетные артефакты для чтения (`LogItemReader`), обработки (`LogItemProcessor`) и записи (`LogItemWriter`). Этот шаг создаёт контрольную точку для каждых десяти обработанных элементов.

Шаг `cleanup` — это шаг задачи, который определяет класс `CleanUp` в качестве своего пакетного артефакта. Задание в целом завершается вместе с завершением этого шага.

В следующих разделах более подробно описываются элементы языка спецификации заданий (JSL) и демонстрируются наиболее распространённые атрибуты и дочерние элементы.

Элемент job

Элемент `job` всегда является элементом верхнего уровня в файле определения задания. Его основные атрибуты `id` и `restartable`. Элемент `job` может содержать один элемент `properties` и любое количество каждого из следующих элементов: `listener`, `step`, `flow` и `split`. Например:

```xml
<job id="jobname" restartable="true">
  <listeners>
    <listener ref="com.example.pkg.ListenerBatchArtifact"/>
  </listeners>
  <properties>
    <property name="propertyName1" value="propertyValue1"/>
    <property name="propertyName2" value="propertyValue2"/>
  </properties>
  <step ...> ... </step>
  <step ...> ... </step>
  <decision ...> ... </decision>
  <flow ...> ... </flow>
  <split ...> ... </split>
</job>
```

Элемент `listener` указывает пакетный артефакт, методы которого вызываются до и после выполнения задания. Пакетный артефакт представляет собой реализацию интерфейса `jakarta.batch.api.listener.JobListener`. Смотрите Пакетные артефакты слушателя для примера реализации слушателя задания.

Первым выполняется первый элемент `step`, `flow` или `split` внутри элемента `job`.

Элемент step

Элемент `step` может быть дочерним элементом элементов `job` и `flow`. Его основные атрибуты `id` и `next`. Элемент `step` может содержать следующие элементы:

- Один элемент `chunk` для шагов, ориентированных на фрагменты, или один элемент `batchlet` для шагов, ориентированных на задачи.
- Один элемент `properties` (необязательно).

  Этот элемент определяет набор свойств, к которым пакетные артефакты могут получить доступ с помощью объектов контекста пакета.
- Один элемент `listener` (необязательно). Один элемент `listeners`, если указано более одного слушателя.

  Этот элемент определяет артефакты слушателя, которые перехватывают различные фазы выполнения шага.

  Для шагов фрагмента пакетные артефакты для этих слушателей могут быть реализациями следующих интерфейсов: `StepListener`, `ItemReadListener`, `ItemProcessListener`, `ItemWriteListener`, `ChunkListener`, `RetryReadListener`, `RetryProcessListener`, `RetryWriteListener`, `SkipReadListener`, `SkipProcessListener` и `SkipWriteListener`.

  Для шагов задачи артефакт пакета для этих слушателей должен реализовывать интерфейс `StepListener`.

  Посмотрите Пакетные артефакты слушателя для примера реализации слушателя.
- Один элемент `partition` (необязательно).

  Этот элемент используется в разделённых шагах, которые выполняются в более чем одном потоке (thread).
- Один элемент `end`, если это последний шаг в задании.

  Этот элемент устанавливает статус пакета в `COMPLETED`.
- Один элемент `stop` (необязательно), чтобы остановить работу на этом шаге.

  Этот элемент устанавливает статус пакета в `STOPPED`.
- Один элемент `fail` (необязательно), который завершает работу на этом этапе.

Этот элемент устанавливает статус пакета на `FAILED`.

- Один или несколько элементов `next`, если не указан атрибут `next`.

  Этот элемент связан со статусом завершения и относится к другому шагу, потоку (flow), разделителю или элементу принятия решения.

Ниже приведён пример шага фрагмента:

```xml
<step id="stepA" next="stepB">
  <properties> ... </properties>
  <listeners>
    <listener ref="MyItemReadListenerImpl"/>
    ...
  </listeners>
  <chunk ...> ... </chunk>
  <partition> ... </partition>
  <end on="COMPLETED" exit-status="MY_COMPLETED_EXIT_STATUS"/>
  <stop on="MY_TEMP_ISSUE_EXIST_STATUS" restart="step0"/>
  <fail on="MY_ERROR_EXIT_STATUS" exit-status="MY_ERROR_EXIT_STATUS"/>
</step>
```

Ниже приведён пример шага задачи:

```xml
<step id="stepB" next="stepC">
  <batchlet ...> ... </batchlet>
  <properties> ... </properties>
  <listener ref="MyStepListenerImpl"/>
</step>
```

Элемент chunk

Элемент `chunk` является дочерним элементом элемента `step` для шагов, ориентированных на фрагменты. Атрибуты этого элемента перечислены в таблице 59-2.

Таблица 59-2 Атрибуты элемента chunk

Название атрибута	Описание	Значение по умолчанию
`checkpoint-policy`	Определяет, как зафиксировать результаты обработки каждого фрагмента: • `"item"`: фрагмент фиксируется после обработки `item-count` элементов • `"custom"`: фрагмент фиксируется в соответствии с алгоритмом контрольной точки, указанном в элементе `checkpoint-algorithm` Контрольная точка обновляется при фиксации результатов обработки фрагмента. Каждый фрагмент обрабатывается в глобальной транзакции Jakarta EE. Если обработка одного элемента во фрагменте не удалась, транзакция откатывается и никакие обработанные элементы из этого фрагменты не сохраняются.	`"item"`

Название атрибута	Описание	Значение по умолчанию
`item-count`	Определяет количество элементов для обработки перед фиксацией фрагмента и принятием контрольной точки.	10
`time-limit`	Указывает количество секунд до фиксации фрагмента и получения контрольной точки, когда `checkpoint-policy = "item"`. Если элементы `item-count` не были обработаны в течение `time-limit` секунд, фрагмент фиксируется и берётся контрольная точка.	0 (без ограничений)
`buffer-items`	Указывает, буферизуются ли обработанные элементы до тех пор, пока не настанет время проходить контрольную точку. Если значение true, выполняется один вызов к элементу записи со списком буферизованных элементов перед фиксацией фрагмента и принятием контрольной точки.	true
`skip-limit`	Задаёт максимальное количество пропускаемых исключений, которые можно пропустить на этом шаге при обработке фрагмента. Классы исключений, которые можно пропустить, указываются с помощью элемента `skippable-exception-classes`.	Без ограничений
`retry-limit`	Задаёт количество попыток выполнить этот шаг, если возникают повторяющиеся исключения. Классы повторяющихся исключений указываются с помощью элемента `retryable-exception-classes`.	Без ограничений

Элемент `chunk` может содержать следующие элементы:

- Один элемент `reader`.

  Этот элемент указывает пакетный артефакт, который реализует интерфейс `ItemReader`.

- Один элемент `processor`.

  Этот элемент указывает пакетный артефакт, который реализует интерфейс `ItemProcessor`.

- Один элемент `writer`.

  Этот элемент указывает пакетный артефакт, который реализует интерфейс `ItemWriter`.

- Один элемент `checkpoint-algorithm` (необязательно).

  Этот элемент указывает пакетный артефакт, который реализует интерфейс `CheckpointAlgorithm` и предоставляет настраиваемую политику контрольных точек.

- Один элемент `skippable-exception-classes` (необязательно).

  Этот элемент определяет набор исключений, выдаваемых из артефактов чтения, записи и обработки, которые при обработке фрагмента следует пропустить. Атрибут `skip-limit` из элемента `chunk` указывает максимальное количество пропущенных исключений.

- Один элемент `retryable-exception-classes` (необязательно).

  Этот элемент задаёт набор исключений, выдаваемых артефактами пакета чтения, записи и обработки, которые повторяется при обработке фрагмента. Атрибут `retry-limit` из элемента `chunk` указывает максимальное количество попыток.

- Один элемент `no-rollback-exception-classes` (необязательно).

  Этот элемент определяет набор исключений, выдаваемых артефактами пакетного чтения, записи и процессора, которые не должны приводить к тому, что среда выполнения пакета выполняет откат текущего фрагмента, а вместо этого повторяет текущую операцию без отката.

  Для типов исключений, не указанных в этом элементе, текущий фрагмент по умолчанию откатывается при возникновении исключения.

Ниже приведён пример шага, ориентированного на фрагменты:

XML

```
<step id="stepC" next="stepD">
  <chunk checkpoint-policy="item" item-count="5" time-limit="180"
         buffer-items="true" skip-limit="10" retry-limit="3">
    <reader ref="pkg.MyItemReaderImpl"></reader>
    <processor ref="pkg.MyItemProcessorImpl"></processor>
    <writer ref="pkg.MyItemWriterImpl"></writer>
    <skippable-exception-classes>
      <include class="pkg.MyItemException"/>
      <exclude class="pkg.MyItemSeriousSubException"/>
    </skippable-exception-classes>
    <retryable-exception-classes>
      <include class="pkg.MyResourceTempUnavailable"/>
    </retryable-exception-classes>
  </chunk>
</step>
```

В этом примере определяется шаг фрагмента и указываются артефакты чтения, обработки и записи. Шаг обновляет контрольную точку и фиксирует каждый фрагмент после обработки пяти элементов. Он пропускает все исключения `MyItemException` и все его подтипы, кроме `MyItemSeriousSubException`, но не более десяти пропущенных исключений. Шаг повторяет фрагмент, когда возникает исключение `MyResourceTempUnavailable`, максимум до трёх попыток.

Элемент batchlet

Элемент `batchlet` является дочерним элементом элемента `step` для шагов, ориентированных на задачи. Этот элемент имеет только атрибут `ref`, указывающий артефакт пакета, реализующего интерфейс `Batchlet`. Элемент `batch` может содержать элемент `properties`.

Ниже приведён пример шага, ориентированного на задачу:

XML

```
<step id="stepD" next="stepE">
  <batchlet ref="pkg.MyBatchletImpl">
    <properties>
      <property name="pname" value="pvalue"/>
    </properties>
  </batchlet>
</step>
```

В этом примере определяется шаг пакета и указывается его пакетный артефакт.

Элемент partition

Элемент `partition` является дочерним элементом элемента `step`. Он указывает на то, что шаг разделён. Большинство разделённых шагов являются шагами, когда обработка каждого элемента не зависит от результатов обработки предыдущих элементов. Вы указываете количество разделений в шаге и предоставляете каждому разделению конкретную информацию о том, какие элементы обрабатывать. Как пример:

- Диапазон элементов. Например, разделение 1 обрабатывает элементы с 1 по 500, а разделение 2 обрабатывает элементы с 501 по 1000.
- Источник ввода. Например, разделение 1 обрабатывает элементы в `input1.txt` , а разделение 2 обрабатывает элементы в `input2.txt` .

Когда число разделений, количество элементов и источники ввода для шага разделения известны во время разработки или развёртывания, вы можете использовать свойства разделения в файле определения задания, чтобы указать информацию о разделении и получить доступ к этим свойствам из пакетных артефактов. Во время выполнения инстанцируется столько объектов пакетных артефактов (читателей, обработчиков и писателей), сколько разделений, и каждый объект артефакта получает свойства, характерные для его разделения.

В большинстве случаев количество разделений, количество элементов или входные источники для шага разделения могут быть определены только во время выполнения. Вместо статического указания специфичных для разделения свойств в файле определения задания вы предоставляете пакетный артефакт, который может обращаться к вашим источникам данных во время выполнения и определять, сколько разделений необходимо и какой диапазон элементов должно обрабатывать каждое разделение. Этот пакетный артефакт является реализацией интерфейса `PartitionMapper` . Пакетная среда выполнения вызывает этот артефакт, а затем использует информацию, которую он предоставляет, для инстанцирования объектов пошаговых пакетных артефактов (читателя, обработчика и писателя) для каждого разделения и для передачи им данных, специфичных для разделения, в качестве параметров.

В оставшейся части этого раздела подробно описывается элемент `partition` и показаны два примера файлов определения задания: один, использующий свойства разделения для указания диапазона элементов для каждого разделения, и другой, который опирается на `PartitionMapper` реализация для определения специфичной для разделения информации.

См. Шаг фрагмента телефонного биллинга в Примере phonebilling для полного примера разделённого шага фрагмента.

Элемент `partition` может содержать следующие элементы:

- Один элемент `plan` , если элемент `mapper` не указан.

  Этот элемент определяет количество разделений, количество потоков (thread) и свойства для каждого разделения в файле определения задания. Элемент `plan` полезен, когда эта информация известна во время разработки или развёртывания.

- Один элемент `mapper` , если элемент `plan` не указан.

  Этот элемент указывает пакетный артефакт, который предоставляет количество разделений, количество потоков (thread) и свойства для каждого разделения. Пакетный артефакт является реализацией интерфейса `PartitionMapper` . Этот параметр используется, когда информация, необходимая для каждого разделения, известна только во время выполнения.

- Один элемент `reducer` (необязательно).

  Этот элемент указывает пакетный артефакт, который получает управление, когда разделённый шаг начинается, заканчивается или откатывается. Пакетный артефакт позволяет объединять результаты из разных разделений и выполнять другие сопутствующие операции. Пакетный артефакт является реализацией интерфейса `PartitionReducer` .

- Один элемент `collector` (необязательно).

Этот элемент указывает пакетный артефакт, который отправляет промежуточные результаты из каждого разделения в анализатор разделений. Пакетный артефакт отправляет промежуточные результаты после каждой контрольной точки для шагов фрагментов и в конце шага для шагов задач. Пакетный артефакт является реализацией интерфейса `PartitionCollector`.

- Один элемент `analyzer` (необязательно).

  Этот элемент указывает пакетный артефакт, который анализирует промежуточные результаты от объектов сборщика разделений. Пакетный артефакт является реализацией интерфейса `PartitionAnalyzer`.

Ниже приведён пример разделённого шага с использованием элемента `plan`:

```xml
<step id="stepE" next="stepF">
  <chunk>
    <reader ...></reader>
    <processor ...></processor>
    <writer ...></writer>
  </chunk>
  <partition>
    <plan partitions="2" threads="2">
      <properties partition="0">
        <property name="firstItem" value="0"/>
        <property name="lastItem" value="500"/>
      </properties>
      <properties partition="1">
        <property name="firstItem" value="501"/>
        <property name="lastItem" value="999"/>
      </properties>
    </plan>
  </partition>
  <reducer ref="MyPartitionReducerImpl"/>
  <collector ref="MyPartitionCollectorImpl"/>
  <analyzer ref="MyPartitionAnalyzerImpl"/>
</step>
```

В этом примере элемент `plan` определяет свойства для каждого разделения в файле определения задания.

В следующем примере вместо элемента `plan` используется элемент `mapper`. Реализация `PartitionMapper` динамически предоставляет ту же информацию, что и элемент `plan` в файле определения задания:

```xml
<step id="stepE" next="stepF">
  <chunk>
    <reader ...></reader>
    <processor ...></processor>
    <writer ...></writer>
  </chunk>
  <partition>
    <mapper ref="MyPartitionMapperImpl"/>
    <reducer ref="MyPartitionReducerImpl"/>
    <collector ref="MyPartitionCollectorImpl"/>
    <analyzer ref="MyPartitionAnalyzerImpl"/>
  </partition>
</step>
```

Обратитесь к Примеру phonebilling для примера реализации интерфейса `PartitionMapper`.

Элемент flow

Элемент `flow` может быть дочерним элементом элементов `job`, `flow` и `split`. Его атрибуты `id` и `next`. Потоки (flow) могут переходить к потокам (flow), шагам, разделителям и элементам принятия решения. Элемент `flow` может содержать следующие элементы:

- Один или несколько элементов `step`
- Один или несколько элементов `flow` (необязательно)
- Один или несколько элементов `split` (необязательно)
- Один или несколько элементов `decision` (необязательно)

Последний `step` в потоке (flow) — это тот, у которого нет атрибута `next` или элемента `next`. Шаги и другие элементы в потоке (flow) не могут переходить к элементам вне потока (flow).

Ниже приведён пример элемента `flow`:

```xml
<flow id="flowA" next="stepE">
  <step id="flowAstepA" next="flowAstepB">...</step>
  <step id="flowAstepB" next="flowAflowC">...</step>
  <flow id="flowAflowC" next="flowAsplitD">...</flow>
  <split id="flowAsplitD" next="flowAstepE">...</split>
  <step id="flowAstepE">...</step>
</flow>
```

Этот пример потока (flow) содержит три шага, один поток (flow) и один разделитель. Последний шаг не имеет атрибута `next`. Поток (flow) переходит на `stepE`, когда его последний шаг завершается.

Элемент split

Элемент `split` может быть дочерним по отношению к элементам `job` и `flow`. Его атрибуты `id` и `next`. Разделители могут переходить в разделители, шаги, потоки (flow) и элементы принятия решения. Элемент `split` может содержать только один или несколько элементов `flow`, которые могут переходить только к другим элементам `flow` в группе split.

Ниже приведён пример разделителя с тремя потоками (flow), которые выполняются одновременно:

```xml
<split id="splitA" next="stepB">
  <flow id="splitAflowA">...</flow>
  <flow id="splitAflowB">...</flow>
  <flow id="splitAflowC">...</flow>
</split>
```

Элемент decision

Элемент `decision` может быть дочерним для элементов `job` и `flow`. Его атрибуты `id` и `next`. Шаги, потоки (flow) и разделители могут переходить к элементу `decision`. Этот элемент указывает пакетный артефакт, который решает, какой следующий шаг, поток (flow) или разделитель выполнить, основываясь на информации из выполнения предыдущего шага, потока (flow) или разделителя. Пакетный артефакт реализует интерфейс `Decider`. Элемент `decision` может содержать следующие элементы:

- Один или несколько элементов `end` (необязательно).

  Этот элемент устанавливает статус пакета в `COMPLETED`.
- Один или несколько элементов `stop` (необязательно).

  Этот элемент устанавливает статус пакета в `STOPPED`.

- Один или несколько элементов `fail` (необязательно).

  Этот элемент устанавливает статус пакета на `FAILED`.
- Один или несколько элементов `next` (необязательно).
- Один элемент `properties` (необязательно).

Ниже приведён пример элемента `decider`:

```xml
<decision id="decisionA" ref="MyDeciderImpl">
  <fail on="FAILED" exit-status="FAILED_AT_DECIDER"/>
  <end on="COMPLETED" exit-status="COMPLETED_AT_DECIDER"/>
  <stop on="MY_TEMP_ISSUE_EXIST_STATUS" restart="step2"/>
</decision>
```

Создание пакетных артефактов

После определения задания с точки зрения его пакетных артефактов с помощью языка спецификации задания (JSL) эти артефакты создаются как классы Java, реализующие интерфейсы из `jakarta.batch.api` и его подпакеты.

В этом разделе перечислены основные интерфейсы артефактов пакета, показано, как получить доступ к объектам контекста из пакетной среды выполнения, и приведены некоторые примеры.

Интерфейсы пакетных артефактов

В следующих таблицах перечислены интерфейсы, которые реализуются при создании пакетных артефактов. Реализации интерфейса ссылаются на элементы, описанные в Использование языка спецификации задания.

Таблица 59-3 перечисляет интерфейсы для реализации пакетных артефактов для шагов фрагмента, шагов задачи и элементов принятия решений.

Таблица 59-4 перечисляет интерфейсы для реализации пакетных артефактов для шагов разделения.

Таблица 59-5 перечисляет интерфейсы для реализации пакетных артефактов для слушателей заданий и шагов.

Таблица 59-3 Основные интерфейсы пакетных артефактов

Пакет	Интерфейс	Описание
`jakarta.batch.api`	`Batchlet`	Реализует бизнес-логику шага задачи. На него ссылается элемент `batchlet`.
`jakarta.batch.api`	`Decider`	Принимает решение о следующем шаге, потоке (flow) или разделителе для выполнения на основе информации от предыдущего шага, потока (flow) или разделителя. На него ссылается элемент `decision`.
`jakarta.batch.api.chunk`	`CheckPointAlgorithm`	Реализует настраиваемую политику контрольных точек для шагов фрагментов. На него ссылается элемент `checkpoint-algorithm` внутри элемента `chunk`.

Пакет	Интерфейс	Описание
`jakarta.batch.api.chunk`	`ItemReader`	Читает элементы из входного источника в шаге фрагмента. На него ссылается элемент `reader` внутри элемента `chunk`.
`jakarta.batch.api.chunk`	`ItemProcessor`	Обрабатывает входные элементы для получения выходных элементов в шагах фрагмента. На него ссылается элемент `processor` внутри элемента `chunk`.
`jakarta.batch.api.chunk`	`ItemWriter`	Записывает выходные элементы в шагах фрагментов. На него ссылается элемент `writer` внутри элемента `chunk`.

Таблица 59-4 Интерфейсы пакетного артефакта разделения

Пакет	Интерфейс	Описание
`jakarta.batch.api.partition`	`PartitionPlan`	Сведения о выполнении разделённого шага, такие как количество разделений, количество потоков (flow) и параметры для каждого разделения. На этот артефакт нет прямой ссылки из файла определения задания.
`jakarta.batch.api.partition`	`PartitionMapper`	Предоставляет объект `PartitionPlan`. На него ссылается элемент `mapper` внутри элемента `partition`.
`jakarta.batch.api.partition`	`PartitionReducer`	Получает контроль, когда разделённый шаг начинается, заканчивается или откатывается. На него ссылается элемент `reducer` внутри элемента `partition`.
`jakarta.batch.api.partition`	`PartitionCollector`	Посылает промежуточные результаты из каждого разделения в анализатор разделений. На него ссылается элемент `collector` внутри элемента `partition`.
`jakarta.batch.api.partition`	`PartitionAnalyzer`	Обрабатывает данные и окончательные результаты из каждого разделения. На него ссылается элемент `analyzer` внутри элемента `partition`.

Таблица 59-5 Интерфейсы пакетных артефактов слушателей

Пакет	Интерфейс	Описание
`jakarta.batch.api.listener`	`JobListener`	Перехватывает выполнение задания до и после запуска задания. На него ссылается элемент `listener` внутри элемента `job`.

Пакет	Интерфейс	Описание
`jakarta.batch.api.listener`	`StepListener`	Перехватывает выполнение шага до и после выполнения шага. На него ссылается элемент `listener` внутри элемента `step`
`jakarta.batch.api.chunk.listener`	`ChunkListener`	Перехватывает обработку фрагмента в шагах фрагмента до и после обработки каждого фрагмента, а также при ошибках. На него ссылается элемент `listener` внутри элемента `step`.
`jakarta.batch.api.chunk.listener`	`ItemReadListener`	Перехватывает чтение элементов в шагах фрагментов до и после чтения каждого элемента и при ошибках. На него ссылается элемент `listener` внутри элемента `step`.
`jakarta.batch.api.chunk.listener`	`ItemProcessListener`	Перехватывает обработку элементов в шагах фрагментов до и после обработки каждого элемента и при ошибках. На него ссылается элемент `listener` внутри элемента `step`.
`jakarta.batch.api.chunk.listener`	`ItemWriteListener`	Перехватывает запись элементов в шагах фрагментов до и после записи каждого элемента и при ошибках. На него ссылается элемент `listener` внутри элемента `step`.
`jakarta.batch.api.chunk.listener`	`RetryReadListener`	Перехватывает повторное считывание элемента при возникновении исключения. На него ссылается элемент `listener` внутри элемента `step`.
`jakarta.batch.api.chunk.listener`	`RetryProcessListener`	Перехватывает повторную обработку элементов в шагах фрагментов при возникновении исключений. На него ссылается элемент `listener` внутри элемента `step`.
`jakarta.batch.api.chunk.listener`	`RetryWriteListener`	Перехватывает повторную запись элементов в шагах фрагментов при возникновении исключения. На него ссылается элемент `listener` внутри элемента `step`.

Пакет	Интерфейс	Описание
`jakarta.batch.api.chunk.listener`	`SkipReadListener`	Перехватывает пропущенную обработку исключений при чтении элементов в шагах фрагментов. На него ссылается элемент `listener` внутри элемента `step`.
`jakarta.batch.api.chunk.listener`	`SkipProcessListener`	Перехватывает пропущенную обработку исключений при обработке элементов в шагах фрагментов. На него ссылается элемент `listener` внутри элемента `step`.
`jakarta.batch.api.chunk.listener`	`SkipWriteListener`	Перехватывает пропущенную обработку исключений при записи элементов в шагах фрагментов. На него ссылается элемент `listener` внутри элемента `step`.

Инъецирование зависимостей в пакетных артефактах

Чтобы убедиться, что инъецирование контекстов и зависимостей Jakarta (CDI) работает в пакетных артефактах, выполните следующие действия.

1. Определите свои реализации пакетного артефакта как именованные компоненты CDI, используя аннотацию `@Named`.

   Например, определите реализацию читателя элементов в шаге фрагмента следующим образом:

   JAVA
   ```java
   @Named("MyItemReaderImpl")
   public class MyItemReaderImpl implements ItemReader {
       /* ... Переопределение методов интерфейса ItemReader... */
   }
   ```

2. Предоставьте публичный пустой конструктор без аргументов для ваших пакетных артефактов.

   Например, предоставьте следующий конструктор для артефакта выше:

   JAVA
   ```java
   public MyItemReaderImpl() {}
   ```

3. Укажите имя CDI для пакетных артефактов в файле определения задания вместо использования полного имени класса.

   Например, определите шаг для артефакта выше следующим образом:

   XML
   ```xml
   <step id="stepA" next="stepB">
     <chunk>
       <reader ref="MyItemReaderImpl"></reader>
       ...
     </chunk>
   </step>
   ```

   В этом примере используется имя CDI (`MyItemReaderImpl`) вместо полностью определённого имени класса (`com.example.pkg.MyItemReaderImpl`) для указания пакетного артефакта.

4. Убедитесь, что модуль является архивом компонентов CDI, аннотируя пакетные артефакты аннотацией `jakarta.enterprise.context.Dependent` или включив пустой файл `beans.xml` в приложение. Например, следующий пакетный артефакт помечен `@Dependent`:

JAVA

```
@Dependent
@Named("MyItemReaderImpl")
public class MyItemReaderImpl implements ItemReader { ... }
```

Для получения дополнительной информации об архивах компонентов см. Упаковка приложений CDI в главе 27 *Jakarta Contexts and Dependency Injection: дополнительные темы*.

Jakarta Contexts and Dependency Injection (CDI) требуется для доступа к объектам контекста из пакетной среды выполнения в пакетных артефактах.

Если не выполнить эту процедуру, можно столкнуться со следующими ошибками.

- Пакетная среда выполнения не сможет найти некоторые пакетные артефакты.
- Пакетные артефакты сгенерируют исключения NullPointerException при доступе к инъецированным объектам.

Использование объектов контекста из пакетной среды выполнения

Пакетная среда выполнения предоставляет контекстные объекты, реализующие интерфейсы `JobContext` и `StepContext` интерфейсов в пакете `jakarta.batch.runtime.context`. Эти объекты связаны с текущим заданием и шагом соответственно, и позволяют выполнять следующие действия:

- Получение информации от текущего задания или шага, например его имя, идентификатор объекта, идентификатор выполнения, статус пакета и статус завершения
- Установка пользовательского статуса завершения
- Сохранение данных пользователя
- Получение значений свойств из определения задания или шага

Вы можете инъецировать объекты контекста из пакетной среды выполнения при реализации пакетного артефакта, такие как читатели, обработчики, писатели элементов, слушатели и т. д. В следующем примере показано, как получить доступ к значениям свойств из файла определения задания в реализации читателя элементов:

JAVA

```
@Dependent
@Named("MyItemReaderImpl")
public class MyItemReaderImpl implements ItemReader {
    @Inject
    JobContext jobCtx;

    public MyItemReaderImpl() {}

    @Override
    public void open(Serializable checkpoint) throws Exception {
        String fileName = jobCtx.getProperties()
                                .getProperty("log_file_name");
        ...
    }
    ...
}
```

Смотрите Инъецирование зависимостей в пакетных артефактах для получения инструкций о том, как определить пакетные артефакты для использования инъецирования зависимостей.

Не обращайтесь к объектам пакетного контекста внутри конструкторов артефактов.

Поскольку задание не запускается до тех пор, пока оно не отправлено на выполнение, объекты контекста пакета не будут доступны, когда CDI инстанцирует объекты артефактов при загрузке приложения. Инстанцирование этих бинов завершится неудачно, и пакетная среда выполнения не сможет найти пакетные артефакты, когда приложение отправит задание.

Отправка заданий в пакетную среду выполнения

Интерфейс `JobOperator` в пакете `jakarta.batch.operations` позволяет передавать задания в пакетную среду выполнения и получать информацию о существующих заданиях. Этот интерфейс обеспечивает следующую функциональность.

- Получение имён всех известных задач.
- Запуск, остановка, перезапуск и отмена заданий.
- Получение заданий и выполнение заданий.

Класс `BatchRuntime` в пакете `jakarta.batch.runtime` предоставляет фабричный метод `getJobOperator` для получения объектов `JobOperator` .

Запуск задания

В следующем примере кода показано, как получить объект `JobOperator` и отправить пакетное задание:

JAVA

```
JobOperator jobOperator = BatchRuntime.getJobOperator();
Properties props = new Properties();
props.setProperty("parameter1", "value1");
...
long execID = jobOperator.start("simplejob", props);
```

Первым аргументом метода `JobOperator.start` является имя задания, указанное в файле определения задания. Второй параметр — это объект `Properties` , который представляет параметры для выполнения этого задания. Вы можете использовать параметры задания для передачи информации о задании, известной только во время выполнения.

Проверка статуса задания

Интерфейс `JobExecution` в пакете `jakarta.batch.runtime` предоставляет методы для получения информации об отправленных заданиях. Этот интерфейс обеспечивает следующую функциональность.

- Получение пакета и статуса завершения выполнения задания.
- Получение времени запуска, изменения или завершения выполнения.
- Получение названия задания.
- Получение ID выполнения.

В следующем примере кода показано, как получить статус пакета задания, используя его идентификатор выполнения:

JAVA

```
JobExecution jobExec = jobOperator.getJobExecution(execID);
String status = jobExec.getBatchStatus().toString();
```

Вызов пакетной среды выполнения из приложения

Компонент, из которого вызывается пакетная среда выполнения, зависит от архитектуры конкретного приложения. Например, вы можете вызвать пакетную среду выполнения из Enterprise-бина, сервлета, Managed-бина и т. д.

См. Пример webserverlog и Пример phonebilling для получения подробной информации о том, как вызвать пакетную среду выполнения из Managed-бина, управляемого пользовательским интерфейсом Jakarta Faces.

Пакетные приложения

Файлы определения задания и пакетные артефакты не требуют отдельной упаковки и могут быть включены в любое приложение Jakarta EE.

Упакуйте классы пакетных артефактов с остальными классами приложения и включите файлы определения задания в один из следующих каталогов:

- `META-INF/batch-jobs/` для пакетов `jar`
- `WEB-INF/classes/META-INF/batch-jobs/` для пакетов `war`

Имя каждого файла определения задания должно соответствовать его идентификатору задания. Например, если определяется задание следующим образом и приложение упаковывается как WAR-файл, включите файл определения задания в `WEB-INF/classes/META-INF/batch-jobs/simplejob.xml`:

```xml
<job id="simplejob" xmlns="https://jakarta.ee/xml/ns/jakartaee"
                    version="2.0">
  ...
</job>
```

Пример webserverlog

Пример `webserverlog`, расположенный в каталоге *`tut-install`*`/examples/batch/webserverlog/`, демонстрирует, как использовать пакетный фреймворк Jakarta EE для анализа файла журнала с веб-сервера. В этом примере приложение читает файл журнала и находит, какой процент просмотров страниц с планшетных устройств приходится на продажи продукта.

Архитектура webserverlog

Приложение `webserverlog` состоит из следующих элементов:

- Файл определения задания (`webserverlog.xml`), который использует язык спецификации задания (JSL) для определения пакетного задания с шагом фрагмента и шагом задачи. Шаг фрагмента действует как фильтр, а шаг задачи вычисляет статистику по оставшимся записям.
- Файл журнала (`log1.txt`), который содержит входные данные для пакетного задания.
- Два Java-класса (`LogLine` и `LogFilteredLine`), которые представляют элементы ввода и вывода для шага фрагмента.
- Три пакетных артефакта (`LogLineReader`, `LogLineProcessor` и `LogFilteredLineWriter`), которые реализуют шаг фрагмента приложения. Этот шаг считывает элементы из файла журнала веб-сервера, фильтрует их с помощью браузера, используемого клиентом, и записывает результаты в текстовый файл.
- Два пакетных артефакта (`InfoJobListener` и `InfoItemProcessListener`), которые реализуют два простых слушателя.

- Пакетный артефакт (`MobileBatchlet.java`), который вычисляет статистику по отфильтрованным элементам.
- Две страницы Facelets (`index.xhtml` и `jobstarted.xhtml`), которые предоставляют внешний интерфейс пакетного приложения. Первая страница показывает файл журнала, который будет обработан пакетным заданием, а вторая страница позволяет пользователю проверить состояние задания и показать результаты.
- Managed-бин (`JsfBean`), доступ к которому осуществляется со страниц Facelets. Компонент отправляет задание в пакетную среду выполнения, проверяет состояние задания и считывает результаты из текстового файла.

Файл определения задания

Файл определения задания `webserverlog.xml` находится в каталоге `WEB-INF/classes/META-INF/batch-jobs/` . Файл определяет семь свойств уровня задания и два шага:

XML

```
<job id="webserverlog" xmlns="https://jakarta.ee/xml/ns/jakartaee"
     version="2.0">
    <properties>
        <property name="log_file_name" value="log1.txt"/>
        <property name="filtered_file_name" value="filtered1.txt"/>
        <property name="num_browsers" value="2"/>
        <property name="browser_1" value="Tablet Browser D"/>
        <property name="browser_2" value="Tablet Browser E"/>
        <property name="buy_page" value="/auth/buy.html"/>
        <property name="out_file_name" value="result1.txt"/>
    </properties>
    <listeners>
        <listener ref="InfoJobListener"/>
    </listeners>
    <step id="mobilefilter" next="mobileanalyzer"> ... </step>
    <step id="mobileanalyzer"> ... </step>
</job>
```

Первый шаг определяется следующим образом:

XML

```
<step id="mobilefilter" next="mobileanalyzer">
    <listeners>
        <listener ref="InfoItemProcessListeners"/>
    </listeners>
    <chunk checkpoint-policy="item" item-count="10">
        <reader ref="LogLineReader"></reader>
        <processor ref="LogLineProcessor"></processor>
        <writer ref="LogFilteredLineWriter"></writer>
    </chunk>
</step>
```

Этот шаг является обычным шагом фрагмента, указывающего пакетные артефакты, реализующие каждую фазу шага. Имена пакетных артефактов не являются полностью квалифицированными именами классов, потому что пакетные артефакты являются бинами CDI, аннотированными `@Named` .

Второй шаг определяется следующим образом:

XML

```
<step id="mobileanalyzer">
    <batchlet ref="MobileBatchlet"></batchlet>
    <end on="COMPLETED"/>
</step>
```

Этот шаг является шагом задачи, указывающей пакетный артефакт, который реализует этот шаг. Это последний шаг задания.

Элементы LogLine и LogFilteredLine

Класс `LogLine` представляет записи в файле журнала веб-сервера и определяется следующим образом:

JAVA

```
public class LogLine {
    private final String datetime;
    private final String ipaddr;
    private final String browser;
    private final String url;

    /* ... Конструкторы, get- и set-методы... */
}
```

Класс `LogFileteredLine` аналогичен этому классу, но имеет только два поля: IP-адрес клиента и URL.

Пакетные артефакты шага фрагмента

Первый шаг состоит из артефактов пакета `LogLineReader`, `LogLineProcessor` и `LogFilteredLineWriter`.

Артефакт `LogLineReader` считывает записи из файла журнала веб-сервера:

JAVA

```
@Dependent
@Named("LogLineReader")
public class LogLineReader implements ItemReader {
    private ItemNumberCheckpoint checkpoint;
    private String fileName;
    private BufferedReader breader;
    @Inject
    private JobContext jobCtx;

    public LogLineReader() { }

    /* ... Переопределение методов open, close, readItem и
     *     checkpointInfo... */
}
```

Метод `open` считывает свойство `log_file_name` и открывает файл журнала с буферизованным читателем. В этом примере файл журнала был включён в приложение в `webserverlog/WEB-INF/classes/log1.txt`:

JAVA

```
fileName = jobCtx.getProperties().getProperty("log_file_name");
ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
InputStream iStream = classLoader.getResourceAsStream(fileName);
breader = new BufferedReader(new InputStreamReader(iStream));
```

Если предоставляется объект контрольной точки, метод `open` продвигает читателя до последней контрольной точки. В противном случае этот метод создаёт новый объект контрольной точки. Объект контрольной точки отслеживает номер строки из последнего подтверждённого фрагмента.

Метод `readItem` возвращает новый объект `LogLine` или null в конце файла журнала:

JAVA

```
@Override
public Object readItem() throws Exception {
    String entry = breader.readLine();
    if (entry != null) {
        checkpoint.nextLine();
        return new LogLine(entry);
    } else {
        return null;
    }
}
```

Артефакт `LogLineProcessor` получает список браузеров из свойств задания и фильтрует записи журнала в соответствии со списком:

JAVA

```
@Override
public Object processItem(Object item) {
    /* Получение списка интересующих браузеров */
    if (nbrowsers == 0) {
        Properties props = jobCtx.getProperties();
        nbrowsers = Integer.parseInt(props.getProperty("num_browsers"));
        browsers = new String[nbrowsers];
        for (int i = 1; i < nbrowsers + 1; i++)
            browsers[i - 1] = props.getProperty("browser_" + i);
    }

    LogLine logline = (LogLine) item;
    /* Фильтрация только мобильных/планшетных браузеров */
    for (int i = 0; i < nbrowsers; i++) {
        if (logline.getBrowser().equals(browsers[i])) {
            return new LogFilteredLine(logline);
        }
    }
    return null;
}
```

Артефакт `LogFilteredLineWriter` считывает имя выходного файла из свойств задания. Метод `open` открывает файл для записи. Если предоставлен объект контрольной точки, артефакт продолжает запись в конец файла. В противном случае файл перезаписывается, если он существует. Метод `writeItems` записывает отфильтрованные элементы в выходной файл:

JAVA

```
@Override
public void writeItems(List<Object> items) throws Exception {
    /* Запись фильтрованных строк в выходной файл */
    for (int i = 0; i < items.size(); i++) {
        LogFilteredLine filtLine = (LogFilteredLine) items.get(i);
        bwriter.write(filtLine.toString());
        bwriter.newLine();
    }
}
```

Пакетные артефакты слушателя

Пакетный артефакт `InfoJobListener` реализует простой слушатель, который записывает сообщения журнала, когда задание начинается и когда оно заканчивается:

JAVA

```
@Dependent
@Named("InfoJobListener")
public class InfoJobListener implements JobListener {
    ...
    @Override
    public void beforeJob() throws Exception {
        logger.log(Level.INFO, "The job is starting");
    }

    @Override
    public void afterJob() throws Exception { ... }
}
```

Пакетный артефакт `InfoItemProcessListener` реализует интерфейс `ItemProcessListener` для шагов фрагмента:

JAVA

```
@Dependent
@Named("InfoItemProcessListener")
public class InfoItemProcessListener implements ItemProcessListener {
    ...
    @Override
    public void beforeProcess(Object o) throws Exception {
        LogLine logline = (LogLine) o;
        llogger.log(Level.INFO, "Processing entry {0}", logline);
    }
    ...
}
```

Пакетный артефакт шага задачи

Шаг задачи реализуется артефактом `MobileBatchlet` , который вычисляет, какой процент отфильтрованных записей журнала являются покупками:

JAVA

```
@Override
public String process() throws Exception {
    /* Получение свойств из файла определения задачи */
    ...
    /* Количество из предыдущего шага фрагмента */
    breader = new BufferedReader(new FileReader(fileName));
    String line = breader.readLine();
    while (line != null) {
        String[] lineSplit = line.split(", ");
        if (buyPage.compareTo(lineSplit[1]) == 0)
            pageVisits++;
        totalVisits++;
        line = breader.readLine();
    }
    breader.close();
    /* Запись результата */
    ...
}
```

Страницы Jakarta Faces

Страница `index.xhtml` содержит текстовую область (textarea), в которой отображается журнал веб-сервера. На странице есть кнопка, позволяющая пользователю отправить пакетное задание и перейти на следующую страницу:

XML

```
<body>
    ...
    <textarea cols="90" rows="25"
              readonly="true">#{jsfBean.getInputLog()}</textarea>
    <p> </p>
    <h:form>
        <h:commandButton value="Start Batch Job"
                         action="#{jsfBean.startBatchJob()}" />
    </h:form>
</body>
```

Эта страница вызывает методы Managed-бина, чтобы показать файл журнала и отправить пакетное задание.

Страница `jobstarted.xhtml` предоставляет кнопку для проверки текущего состояния пакетного задания и отображает результаты после его завершения:

XML

```
<p>Current Status of the Job: <b>#{jsfBean.jobStatus}</b></p>
<p>#{jsfBean.showResults()}</p>
<h:form>
    <h:commandButton value="Check Status"
                     action="jobstarted"
                     rendered="#{jsfBean.completed==false}" />
</h:form>
```

Managed-бин

Managed-бин `JsfBean` передаёт задание в пакетную среду выполнения, проверяет статус задания и считывает результаты из текстового файла.

Метод `startBatchJob` передаёт пакетное задание на выполнение:

JAVA

```
/* Отправка пакетной задачи в пакетную среду выполнения.
 * метод навигации JSF (возвращает имя следующей страницы) */
public String startBatchJob() {
    jobOperator = BatchRuntime.getJobOperator();
    execID = jobOperator.start("webserverlog", null);
    return "jobstarted";
}
```

Метод `getJobStatus` проверяет статус задания:

JAVA

```
/* Получение статуса задачи от пакетной среды выполнения */
public String getJobStatus() {
    return jobOperator.getJobExecution(execID).getBatchStatus().toString();
}
```

Метод `showResults` считывает результаты из текстового файла.

Запуск webserverlog

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска примера приложения `webserverlog`.

Запуск webserverlog с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню «Файл» выберите «Открыть проект».

3. В диалоговом окне «Открыть проект» перейдите к:

```
tut-install/examples/batch
```

4. Выберите каталог `webserverlog` .
5. Нажмите Открыть проект.
6. На вкладке «Проекты» кликните правой кнопкой мыши проект `webserverlog` и выберите «Выполнить».

   Эта команда собирает и упаковывает приложение в WAR-файл `webserverlog.war` , расположенный в каталоге `target/` , развёртывает его на сервере и запускает окно веб-браузера по следующему URL:

```
http://localhost:8080/webserverlog/
```

Запуск webserverlog с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

```
tut-install/examples/batch/webserverlog/
```

3. Введите следующую команду для развёртывания приложения:

```
mvn install
```
SHELL

4. Откройте окно веб-браузера по следующему URL:

```
http://localhost:8080/webserverlog/
```

Пример phonebilling

Пример `phonebilling` , расположенное в каталоге *`tut-install`*`/examples/batch/phonebilling/` , демонстрирует, как использовать пакетный фреймворк Jakarta EE для реализации системы выставления счетов по телефону. В этом примере приложение обрабатывает файл журнала телефонных звонков и создаёт счёт для каждого клиента.

Архитектура phonebilling

Приложение `phonebilling` состоит из следующих элементов.

- Файл определения задания (`phonebilling.xml`), который использует язык спецификации задания (JSL) для определения пакетного задания с двумя шагами фрагмента. Первый шаг читает записи вызовов из файла журнала и связывает их со счётом. Второй шаг вычисляет сумму к оплате и записывает каждый счёт в текстовый файл.
- Класс Java (`CallRecordLogCreator`), который создаёт файл журнала для пакетного задания. Это вспомогательный бин, который не содержит никаких значимых для этого примера функций.
- Две сущности Jakarta Persistence (`CallRecord` и `PhoneBill`), которые представляют записи вызовов и счета клиентов. Приложение использует диспетчер сущностей Jakarta Persistence для хранения объектов этих сущностей в базе данных.
- Три пакетных артефакта (`CallRecordReader` , `CallRecordProcessor` и `CallRecordWriter`), которые реализуют первый шаг приложения. Этот шаг считывает записи вызовов из файла журнала, связывает их со счётом и сохраняет их в базе данных.

- Четыре пакетных артефакта (`BillReader` , `BillProcessor` , `BillWriter` и `BillPartitionMapper`), которые реализуют второй шаг приложения. Этот шаг является разделённым и получает каждый счёт из базы данных, рассчитывает сумму к оплате и записывает её в текстовый файл.
- Две страницы Facelets (`index.xhtml` и `jobstarted.xhtml`), которые предоставляют внешний интерфейс пакетного приложения. Первая страница показывает файл журнала, который будет обработан пакетным заданием, а вторая страница позволяет пользователю проверить статус задания и показывает итоговый счёт для каждого клиента.
- Managed-бин (`JsfBean`), доступ к которому осуществляется со страниц Facelets. Компонент отправляет пакетное задание на выполнение, проверяет статус задания и считывает текстовые файлы для каждого счёта.

Файл определения задания

Файл определения задания `phonebilling.xml` находится в каталоге `WEB-INF/classes/META-INF/batch-jobs/` . Файл определяет три свойства уровня задания и два шага:

XML

```xml
<job id="phonebilling" xmlns="https://jakarta.ee/xml/ns/jakartaee"
     version="2.0">
    <properties>
        <property name="log_file_name" value="log1.txt"/>
        <property name="airtime_price" value="0.08"/>
        <property name="tax_rate" value="0.07"/>
    </properties>
    <step id="callrecords" next="bills"> ... </step>
    <step id="bills"> ... </step>
</job>
```

Первый шаг определяется следующим образом:

XML

```xml
<step id="callrecords" next="bills">
    <chunk checkpoint-policy="item" item-count="10">
        <reader ref="CallRecordReader"></reader>
        <processor ref="CallRecordProcessor"></processor>
        <writer ref="CallRecordWriter"></writer>
    </chunk>
</step>
```

Этот шаг является обычным шагом фрагмента, указывающего пакетные артефакты, реализующие каждую фазу шага. Имена пакетных артефактов не являются полностью квалифицированными именами классов, потому что пакетные артефакты являются бинами CDI, аннотированными `@Named` .

Второй шаг определяется следующим образом:

XML

```
<step id="bills">
    <chunk checkpoint-policy="item" item-count="2">
        <reader ref="BillReader">
            <properties>
                <property name="firstItem" value="#{partitionPlan['firstItem']}"/>
                <property name="numItems" value="#{partitionPlan['numItems']}"/>
            </properties>
        </reader>
        <processor ref="BillProcessor"></processor>
        <writer ref="BillWriter"></writer>
    </chunk>
    <partition>
        <mapper ref="BillPartitionMapper"/>
    </partition>
    <end on="COMPLETED"/>
</step>
```

Этот шаг является разделённым шагом. План разделения указывается с помощью артефакта `BillPartitionMapper` вместо использования элемента `plan`.

Сущности CallRecord и PhoneBill

Сущность `CallRecord` определяется следующим образом:

JAVA

```
@Entity
public class CallRecord implements Serializable {
    @Id @GeneratedValue
    private Long id;
    @Temporal(TemporalType.DATE)
    private Date datetime;
    private String fromNumber;
    private String toNumber;
    private int minutes;
    private int seconds;
    private BigDecimal price;

    public CallRecord() { }

    public CallRecord(String datetime, String from,
            String to, int min, int sec)            throws ParseException { ... }

    public CallRecord(String jsonData) throws ParseException { ... }

    /* ... get- и set-методы... */
}
```

Поле `id` автоматически создаётся реализацией Jakarta Persistence для сохранения объектов `CallRecord` в базу данных и извлечения из неё.

Второй конструктор создаёт объект `CallRecord` из JSON в файле журнала с помощью Jakarta JSON Processing. Записи в журнале выглядят следующим образом:

JAVA

```
{"datetime":"03/01/2013 04:03","from":"555-0101",
"to":"555-0114","length":"03:39"}
```

Сущность `PhoneBill` определяется следующим образом:

JAVA

```
@Entity
public class PhoneBill implements Serializable {
    @Id
    private String phoneNumber;
    @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.PERSIST)
    @OrderBy("datetime ASC")
    private List<CallRecord> calls;
    private BigDecimal amountBase;
    private BigDecimal taxRate;
    private BigDecimal tax;
    private BigDecimal amountTotal;

    public PhoneBill() { }

    public PhoneBill(String number) {
        this.phoneNumber = number;
        calls = new ArrayList<>();
    }

    public void addCall(CallRecord call) {
        calls.add(call);
    }

    public void calculate(BigDecimal taxRate) { ... }

    /* ... get- и set-методы... */
}
```

Аннотация `@OneToMany` определяет связь между счётом и его записями вызовов. Атрибут `FetchType.EAGER` требует раннего (eager) извлечения коллекции. Атрибут `CascadeType.PERSIST` указывает, что элементы в списке вызовов должны автоматически сохраняться при сохранении счёта за телефон. Аннотация `@OrderBy` определяет порядок извлечения элементов списка вызовов из базы данных.

Пакетные артефакты используют объекты этих двух сущностей в качестве элементов для чтения, обработки и записи.

Дополнительные сведения о Jakarta Persistence см. в разделе Введение в Jakarta Persistence. Дополнительные сведения о Jakarta JSON Processing см. в разделе Обработка JSON.

Шаг фрагментов записи вызовов

Первый шаг состоит из артефактов пакета `CallRecordReader`, `CallRecordProcessor` и `CallRecordWriter`.

Артефакт `CallRecordReader` считывает записи вызовов из файла журнала:

JAVA

```
@Dependent
@Named("CallRecordReader")
public class CallRecordReader implements ItemReader {
    private ItemNumberCheckpoint checkpoint;
    private String fileName;
    private BufferedReader breader;
    @Inject
    JobContext jobCtx;

    /* ... Переопределение методов open, close, readItem,
     *     и checkpointInfo... */
}
```

Метод `open` считывает свойство `log_filename` и открывает файл журнала с буферизованным читателем:

JAVA

```
fileName = jobCtx.getProperties().getProperty("log_file_name");
breader = new BufferedReader(new FileReader(fileName));
```

Если предоставляется объект контрольной точки, метод `open` продвигает читателя до последней контрольной точки. В противном случае этот метод создаёт новый объект контрольной точки. Объект контрольной точки отслеживает номер строки из последнего подтверждённого фрагмента.

Метод `readItem` возвращает новый объект `CallRecord` или null в конце файла журнала:

JAVA

```
@Override
public Object readItem() throws Exception {
    /* Чтение строки из файла лога и
     * создание CallRecord из JSON */
    String callEntryJson = breader.readLine();
    if (callEntryJson != null) {
        checkpoint.nextItem();
        return new CallRecord(callEntryJson);
    } else
        return null;
}
```

Артефакт `CallRecordProcessor` получает цену эфирного времени из свойств задания, вычисляет цену каждого вызова и возвращает объект вызова. Этот артефакт переопределяет только метод `processItem`.

Артефакт `CallRecordWriter` связывает каждую запись о вызове со счётом и сохраняет счёт в базе данных. Этот артефакт переопределяет методы `open`, `close`, `writeItems` и `checkpointInfo`. Метод `writeItems` выглядит следующим образом:

JAVA

```
@Override
public void writeItems(List<Object> callList) throws Exception {

    for (Object callObject : callList) {
        CallRecord call = (CallRecord) callObject;
        PhoneBill bill = em.find(PhoneBill.class, call.getFromNumber());
        if (bill == null) {
            /* Для этого пользователя нет счёта, создаётся */
            bill = new PhoneBill(call.getFromNumber());
            bill.addCall(call);
            em.persist(bill);
        } else {
            /* Добавление вызова к существующему счёту */
            bill.addCall(call);
        }
    }
}
```

Шаг фрагмента телефонного биллинга

Второй шаг состоит из артефактов пакетной обработки `BillReader`, `BillProcessor`, `BillWriter` и `BillPartitionMapper`. Этот шаг получает счета за телефон из базы данных, вычисляет сумму налога и общую сумму задолженности и записывает каждый счёт в текстовый файл. Поскольку обработка каждого счёта не зависит от других, этот этап может быть разделён на несколько потоков (thread).

Артефакт `BillPartitionMapper` указывает количество разделений и параметры для каждого разделения. В этом примере параметры представляют диапазон элементов, которые должно обрабатывать каждое разделение. Артефакт получает количество счетов в базе данных для расчёта этих диапазонов. Он

предоставляет объект плана разделения, который переопределяет методы `getPartitions` и `getPartitionProperties` интерфейса `PartitionPlan`. Метод `getPartitions` выглядит следующим образом:

JAVA

```
@Override
public Properties[] getPartitionProperties() {
    /* Добавление (примерного) числа элементов
     * к каждому разделению. */
    long totalItems = getBillCount();
    long partItems = (long) totalItems / getPartitions();
    long remItems = totalItems % getPartitions();

    /* Обработка массива свойств. Каждый элемент Properties
* в массиве соответствует разделению. */
    Properties[] props = new Properties[getPartitions()];

    for (int i = 0; i < getPartitions(); i++) {
        props[i] = new Properties();
        props[i].setProperty("firstItem",
                String.valueOf(i * partItems));
        /* Последнее разделение берёт оставшиеся элементы */
        if (i == getPartitions() - 1) {
            props[i].setProperty("numItems",
                    String.valueOf(partItems + remItems));
        } else {
            props[i].setProperty("numItems",
                    String.valueOf(partItems));
        }
    }
    return props;
}
```

Артефакт `BillReader` получает параметры разделения следующим образом:

JAVA

```
@Dependent
@Named("BillReader")
public class BillReader implements ItemReader {

    @Inject @BatchProperty(name = "firstItem")
    private String firstItemValue;
    @Inject @BatchProperty(name = "numItems")
    private String numItemsValue;
    private ItemNumberCheckpoint checkpoint;
    @PersistenceContext
    private EntityManager em;
    private Iterator iterator;

    @Override
    public void open(Serializable ckpt) throws Exception {
        /* Получение диапазона записей для обработки в этом разделении */
        long firstItem0 = Long.parseLong(firstItemValue);
        long numItems0 = Long.parseLong(numItemsValue);

        if (ckpt == null) {
            /* Создание объекта checkpoint для этого разделения */
            checkpoint = new ItemNumberCheckpoint();
            checkpoint.setItemNumber(firstItem0);
            checkpoint.setNumItems(numItems0);
        } else {
            checkpoint = (ItemNumberCheckpoint) ckpt;
        }

        /* Adjust range for this partition from the checkpoint */
        long firstItem = checkpoint.getItemNumber();
        long numItems = numItems0 - (firstItem - firstItem0);
        ...
    }
    ...
}
```

Этот артефакт также получает итератор для чтения элементов из entity manager-a Jakarta Persistence:

JAVA

```
/* Получение итератора для счетов этого разделения */
String query = "SELECT b FROM PhoneBill b ORDER BY b.phoneNumber";
Query q = em.createQuery(query).setFirstResult((int) firstItem)
        .setMaxResults((int) numItems);
iterator = q.getResultList().iterator();
```

Артефакт `BillProcessor` перебирает список записей вызовов в счёте и рассчитывает налог и общую сумму, причитающуюся за каждый счёт.

Артефакт `BillWriter` записывает каждый счёт в простой текстовый файл.

Страницы Jakarta Faces

Страница `index.xhtml` содержит текстовую область, в которой отображается файл журнала записей вызовов. На странице есть кнопка, позволяющая пользователю отправить пакетное задание и перейти на следующую страницу:

XML

```
<body>
    <h1>The Phone Billing Example Application</h1>
    <h2>Log file</h2>
    <p>The batch job analyzes the following log file:</p>
    <textarea cols="90" rows="25"
              readonly="true">#{jsfBean.createAndShowLog()}</textarea>
    <p> </p>
    <h:form>
        <h:commandButton value="Start Batch Job"
                         action="#{jsfBean.startBatchJob()}" />
    </h:form>
</body>
```

Эта страница вызывает методы Managed-бина, чтобы показать файл журнала и отправить пакетное задание.

На странице `jobstarted.xhtml` есть кнопка для проверки текущего статуса пакетного задания и отображения счетов по окончании задания:

XML

```
<p>Current Status of the Job: <b>#{jsfBean.jobStatus}</b></p>
<h:dataTable var="_row" value="#{jsfBean.rowList}"
             border="1" rendered="#{jsfBean.completed}">
    <!-- ... show results from jsfBean.rowList ... -->
</h:dataTable>
<!-- Render the check status button if the job has not finished -->
<h:form>
    <h:commandButton value="Check Status"
                     rendered="#{jsfBean.completed==false}"
                     action="jobstarted" />
</h:form>
```

Managed-бин

Managed-бин `JsfBean` отправляет задание в пакетную среду выполнения, проверяет статус задания и считывает текстовые файлы для каждого счёта.

Метод `startBatchJob` компонента отправляет задание в пакетную среду выполнения:

JAVA

```
/* Отправка пакетной задачи в пакетную среду выполнения.
 * Метод навигации JSF (возвращает имя следующей страницы) */
public String startBatchJob() {
    jobOperator = BatchRuntime.getJobOperator();
    execID = jobOperator.start("phonebilling", null);
    return "jobstarted";
}
```

Метод `getJobStatus` компонента проверяет статус задания:

JAVA

```
/* Получение статуса задачи от пакетной среды выполнения */
public String getJobStatus() {
    return jobOperator.getJobExecution(execID).getBatchStatus().toString();
}
```

Метод `getRowList` EJB-компонента создает список счетов для отображения на странице Faces `jobstarted.xhtml` с использованием таблицы.

Запуск phonebilling

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска приложения `phonebilling`.

Запуск phonebilling в IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню «Файл» выберите «Открыть проект».
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/batch
   ```

4. Выберите каталог `phonebilling`.
5. Нажмите Открыть проект.
6. На вкладке «Проекты» кликните правой кнопкой мыши проект `phonebilling` и выберите «Выполнить».

   Эта команда собирает и упаковывает приложение в WAR-файл `phonebilling.war`, расположенный в каталоге `target/`, развёртывает его на сервере и запускает окно веб-браузера по следующему URL:

   ```
   http://localhost:8080/phonebilling/
   ```

Запуск phonebilling с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

   ```
   tut-install/examples/batch/phonebilling/
   ```

3. Введите следующую команду для развёртывания приложения:

   ```
   mvn install
   ```
   SHELL

4. Откройте окно веб-браузера по следующему URL:

   ```
   http://localhost:8080/phonebilling/
   ```

Дополнительная информация о пакетной обработке

Для получения дополнительной информации о пакетной обработке в Jakarta EE см. Jakarta Batch: https://jakarta.ee/specifications/batch/2.0/

Глава 60. Параллелизм Jakarta

В этой главе описывается спецификация Jakarta Concurrency.

Основы параллелизма

Параллелизм — это концепция выполнения двух или более задач одновременно (параллельно). Задачи могут включать методы (функции), части программы или даже другие программы. В современных компьютерных архитектурах поддержка нескольких ядер и нескольких процессоров в одном процессорном модуле очень распространена.

Платформа Java всегда предлагала поддержку параллелизма, которое было основой для реализации многих сервисов, предлагаемых контейнерами Jakarta EE. Начиная с Java SE 5, дополнительная поддержка высокоуровневого API для параллелизма была предоставлена в пакете `java.util.concurrent`.

Потоки и процессы

Двумя основными понятиями параллелизма являются процессы и потоки (thread).

Процессы в основном связаны с приложениями, работающими в операционной системе (ОС). У любого процесса (в том числе у процесса JVM) есть определённые ресурсы времени выполнения для взаимодействия с ОС и распределения других ресурсов, таких как выделенная ему оперативная память. JVM — это типичный процесс.

Язык программирования и платформа Java используют в основном потоки.

Потоки разделяют некоторые функции с процессами, поскольку оба потребляют ресурсы ОС или среды выполнения. Но потоки легче создавать и они потребляют гораздо меньше ресурсов, чем процессы.

Поскольку потоки — очень легковесные объекты, любой современный процессор, имеющий пару ядер и несколько гигабайт оперативной памяти, может обрабатывать тысячи потоков в одном процессе JVM. Точное количество потоков будет зависеть от совокупных выходных данных процессора, ОС и оперативной памяти, а также от правильной настройки JVM.

Хотя параллельное программирование решает многие проблемы и может повысить производительность для большинства приложений, существует ряд ситуаций, когда несколько выполняемых объектов (потоки или процессы) могут вызвать серьёзные проблемы. Эти ситуации включают в себя следующее:

- Тупики
- "Голодание" потока
- Параллельный доступ к общим ресурсам
- Ситуации, когда программа генерирует неверные данные

Основные компоненты утилит параллелизма

Ресурсы параллелизма — это управляемые объекты, которые предоставляют возможности параллелизма приложениям Jakarta EE. В GlassFish Server сначала нужно настроить ресурсы параллелизма, а затем сделать их доступными для использования компонентами приложения, такими как сервлеты и Enterprise-бины. Доступ к ресурсам параллелизма осуществляется через поиск JNDI или инъецирование ресурсов.

Основными компонентами утилит параллелизма являются следующие.

- `ManagedExecutorService` : управляемый ExecutorService используется приложениями для асинхронного выполнения задач. Задачи выполняются в потоках, которые запускаются и управляются контейнером. Контекст контейнера распространяется на поток, выполняющий задачу.

  Например, с помощью вызова `ManagedExecutorService.submit()` задача, такая как GenerateReportTask, может быть передана для отложенного выполнения, а затем с помощью Callback-метода объекта `Future` может получить результат, когда тот станет доступным.

- `ManagedScheduledExecutorService` : управляемый ScheduledExecutorService используется приложениями для асинхронного выполнения задач в определённое время. Задачи выполняются в потоках, которые запускаются и управляются контейнером. Контекст контейнера распространяется на поток, выполняющий задачу. API предоставляет функциональность планирования, которая позволяет пользователям программно устанавливать конкретную дату/время для выполнения прикладной задачи.

- `ContextService` : контекстный сервис используется для создания динамических прокси-объектов, которые захватывают контекст контейнера и позволяют приложениям запускаться в этом контексте позднее или передаются управляемому ExecutorService-у. Контекст контейнера распространяется на поток, выполняющий задачу.

- `ManagedThreadFactory` : фабрика управляемых потоков используется приложениями для создания управляемых потоков. Потоки запускаются и управляются контейнером. Контекст контейнера распространяется на поток, выполняющий задачу. Этот объект также можно использовать для предоставления кастомных фабрик для конкретных случаев использования (с пользовательскими потоками) и, например, для установки специфичных свойств этих объектов.

Параллелизм и транзакции

Основными операциями для транзакций являются фиксация и откат, но в распределённой среде с параллельной обработкой может оказаться сложным гарантировать, что операции фиксации или отката будут успешно обработаны, а транзакция может быть распределена между различными потоками, процессорными ядрами, физическими машинами и сетями.

Обеспечение успешного выполнения операции отката в таком сценарии имеет решающее значение. Concurrency Utilities полагается на Jakarta Transactions для реализации и поддержки транзакций в своих компонентах через `jakarta.transaction.UserTransaction` , что позволяет разработчикам приложений явно управлять границами транзакций. Более подробная информация содержится в спецификации Jakarta Transactions.

Опционально, объекты контекста могут начинать, фиксировать или откатывать транзакции, но эти объекты не могут присоединяться к транзакциям родительских компонентов.

Следующий фрагмент кода иллюстрирует задачу `Runnable` , которая получает `UserTransaction` , а затем запускает и фиксирует транзакцию при взаимодействии с другими транзакционными компонентами, такими как Enterprise-бин и база данных:

JAVA

```
public class MyTransactionalTask implements Runnable {

   UserTransaction ut = ... // получение с JNDI или инъецированием

   public void run() {

       // Запуск транзакции
       ut.begin();

       // Вызов сервиса или EJB
       myEJB.businessMethod();

       // Обновление в базе данные используя XA JDBC driver
       myEJB.updateCustomer(customer);

       // Фиксация транзакции
       ut.commit();

   }
}
```

Параллелизм и безопасность

Jakarta Concurrency защищает большинство решений по безопасности от реализации сервера приложений. Однако, если контейнер поддерживает контекст безопасности, этот контекст может быть передан выполняющемуся потоку. `ContextService` может поддерживать различное поведение во время выполнения, а атрибут `security`, если он установлен, будет использовать принципала контейнера.

Пример jobs

В этом разделе описан очень простой пример, который показывает, как использовать некоторые функции параллелизма в корпоративном приложении. В частности, в этом примере используется один из основных компонентов Jakarta Concurrency — Managed Executor Service.

В этом примере демонстрируется сценарий, в котором RESTful веб-сервис, предоставляющий общедоступный API, используется для отправки заданий на выполнение. Эти задания обрабатываются в фоновом режиме. Каждое задание выводит сообщение «Starting» в начале выполнения и «Finished» — в конце. Кроме того, для имитации фоновой обработки каждое задание занимает 10 секунд.

RESTful сервис предоставляет два метода:

- `/token`: предоставляется как метод GET, который регистрирует и возвращает валидные токены API
- `/process`: предоставляется как метод POST, получающий параметр запроса `jobID`, являющийся идентификатором задания для выполнения, и настраиваемый заголовок HTTP с именем `X-REST-API-Key`, который будет использоваться для внутренней проверки запросов с токенами

Маркер используется для предоставления обслуживания разного качества (QoS), предлагаемого API. Пользователи, предоставляющие токен в запросе на обслуживание, могут обрабатывать несколько одновременных заданий. Однако пользователи, которые не предоставляют токен, могут одновременно обрабатывать только одно задание. Поскольку на выполнение каждого задания уходит 10 секунд, пользователи, не предоставляющие токен, смогут выполнять только один вызов службы каждые 10 секунд. Для пользователей, предоставляющих токен, обработка будет идти намного быстрее.

Такое разграничение стало возможным благодаря использованию двух разных управляемых ExecutorService-ов, по одному для каждого типа запроса.

Пример выполнения работ

После настройки GlassFish Server путём добавления двух управляемых ExecutorService-ов вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска примера `jobs`.

Настройка GlassFish Server для базового примера параллелизма

Чтобы настроить GlassFish Server, выполните следующие действия.

1. Откройте Консоль администрирования http://localhost:4848.
2. Разверните узел Ресурсы.
3. Разверните узел «Ресурсы параллелизма».
4. Нажмите управляемые ExecutorService-ы.
5. На странице управляемых ExecutorService-ов нажмите Создать, чтобы открыть страницу Создание управляемого ExecutorService-а.
6. В поле Имя JNDI введите `MES_High`, чтобы создать высокоприоритетный управляемый ExecutorService. Используйте следующие настройки (оставьте значения по умолчанию для других настроек):
   - Приоритет потока: 10
   - Количество ядер: 2
   - Максимальный размер пула: 5
   - Ёмкость очереди задач: 2
7. Нажмите ОК.
8. На странице «Управляемые ExecutorService-ы» снова нажмите «Создать».
9. В поле Имя JNDI введите `MES_Low`, чтобы создать управляемый ExecutorService-а приоритетом. Используйте следующие настройки (оставьте значения по умолчанию для других настроек):
   - Приоритет потока: 1
   - Количество ядер: 1
   - Максимальный размер пула: 1
   - Ёмкость очереди задач: 0
10. Нажмите ОК.

Пример создания, упаковки и развёртывания jobs с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню «Файл» выберите «Открыть проект».
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/concurrency
   ```

4. Выберите каталог `jobs`.
5. Нажмите Открыть проект.
6. На вкладке «Проекты» кликните правой кнопкой мыши проект `jobs` и выберите «Сборка».

   Эта команда собирает и развёртывает приложение.

Пример создания, упаковки и развёртывания jobs с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

```
tut-install/examples/concurrency/jobs
```

3. Введите следующую команду, чтобы собрать и развернуть приложение:

```
mvn install
```
SHELL

Запуск примера jobs и отправка заданий с низким приоритетом

Чтобы запустить пример как пользователь, который отправляет задания с низким приоритетом, выполните следующие действия:

1. В веб-браузере введите следующий URL:

```
http://localhost:8080/jobs
```

2. На странице «Клиент задания» введите значение 1 в поле «Ввести идентификатор задания», ничего не вводите в поле «Ввести токен» и нажмите «Отправить задание».

   Следующее сообщение должно отображаться внизу страницы:

```
Job 1 successfully submitted
```

   Журнал сервера включает в себя следующие сообщения:

```
INFO:   Invalid or missing token!
INFO:   Task started LOW-1
INFO:   Job 1 successfully submitted
INFO:   Task finished LOW-1
```

   Вы отправили работу с низким приоритетом. Это означает, что вы не можете отправить другую работу в течение 10 секунд. Если вы попытаетесь это сделать, RESTful API вернёт ответ «Служба недоступна» (HTTP 503), а в нижней части страницы будет показано следующее сообщение:

```
Job 2 was NOT submitted
```

   Журнал сервера будет содержать следующие сообщения:

```
INFO:   Invalid or missing token!
INFO:   Job 1 successfully submitted
INFO:   Task started LOW-1
INFO:   Invalid or missing token!
INFO:   Job 2 was NOT submitted
INFO:   Task finished LOW-1
```

Запуск примера jobs и отправка заданий с высоким приоритетом

Чтобы запустить пример как пользователь, который отправляет задания с высоким приоритетом, выполните следующие действия:

1. В веб-браузере введите следующий URL:

```
http://localhost:8080/jobs
```

2. На странице «Клиент задания» введите значение от одной до десяти цифр в поле «Введите идентификатор задания».
3. Нажмите на ссылку здесь в строке «Получить токен здесь», чтобы получить токен. Страница, на которой отображается токен, откроется в новой вкладке.
4. Скопируйте токен и вернитесь на страницу клиента заданий.
5. Вставьте токен в поле «Введите токен», затем нажмите «Отправить задание».

   Сообщение, подобное следующему, должно отображаться внизу страницы:

   ```
   Job 11 successfully submitted
   ```

   Журнал сервера включает в себя следующие сообщения:

   ```
   INFO:   Token accepted. Execution with high priority.
   INFO:   Task started HIGH-11
   INFO:   Job 11 successfully submitted
   INFO:   Task finished HIGH-11
   ```

   Вы представили задание с высоким приоритетом. Это означает, что вы можете отправлять несколько заданий, каждое с токеном, и не сталкиваться с ограничением в 10 секунд на задание, с которым сталкиваются отправители с низким приоритетом. Если вы отправите 3 задания с токенами в быстрой последовательности, в нижней части страницы будут отображаться сообщения, подобные следующим:

   ```
   Job 1 was submitted
   Job 2 was submitted
   Job 3 was submitted
   ```

   Журнал сервера будет содержать следующие сообщения:

   ```
   INFO:   Token accepted. Execution with high priority.
   INFO:   Task started HIGH-1
   INFO:   Job 1 successfully submitted
   INFO:   Token accepted. Execution with high priority.
   INFO:   Task started HIGH-2
   INFO:   Job 2 successfully submitted
   INFO:   Task finished HIGH-1
   INFO:   Token accepted. Execution with high priority.
   INFO:   Task started HIGH-3
   INFO:   Job 3 successfully submitted
   INFO:   Task finished HIGH-2
   INFO:   Task finished HIGH-3
   ```

Пример taskcreator

Пример `taskcreator` демонстрирует, как использовать Jakarta Concurrency для немедленного, периодического выполнения задачи или выполнения её после фиксированной задержки. В этом примере представлен интерфейс Jakarta Faces, который позволяет пользователям отправлять задания для выполнения и отображает информационные сообщения для каждого задания. В этом примере управляемый ExecutorService используется для немедленного запуска заданий, а управляемый ScheduledExecutorService — для запуска периодических заданий или заданий с фиксированной задержкой. (См. Основные компоненты утилит параллелизма для получения информации об этих сервисах.)

Пример `taskcreator` состоит из следующих компонентов.

- Страница Jakarta Faces (`index.xhtml`), которая содержит три элемента: форму для отправки заданий, журнал их выполнения и форму для отмены периодических заданий. Эта страница отправляет Ajax-запросы для создания и отмены заданий. Она также получает сообщения веб-сокетов, используя код JavaScript для обновления журнала выполнения заданий.
- Managed-бин CDI (`TaskCreatorBean`), который обрабатывает запросы со страницы Jakarta Faces. Этот компонент вызывает методы из `TaskEJB` для отправки новых заданий и отмены периодических заданий.
- Enterprise-бин (`TaskEJB`), который получает объекты ExecutorService-а, используя инъецирование ресурсов, и отправляет задания на выполнение. Этот компонент также является конечной точкой RESTful веб-сервиса Jakarta. Задачи отправляют информационные сообщения этой конечной точке.
- Конечная точка веб-сокета (`InfoEndpoint`), используемая Enterprise-бином для отправки информационных сообщений клиентам.
- Класс задач (`Task`), реализующий интерфейс `Runnable` . Метод `run` в этом классе отправляет информационные сообщения конечной точке веб-сервиса в `TaskEJB` и засыпает на 1,5 секунды.

Рисунок 60-1 демонстрирует архитектуру `taskcreator` .

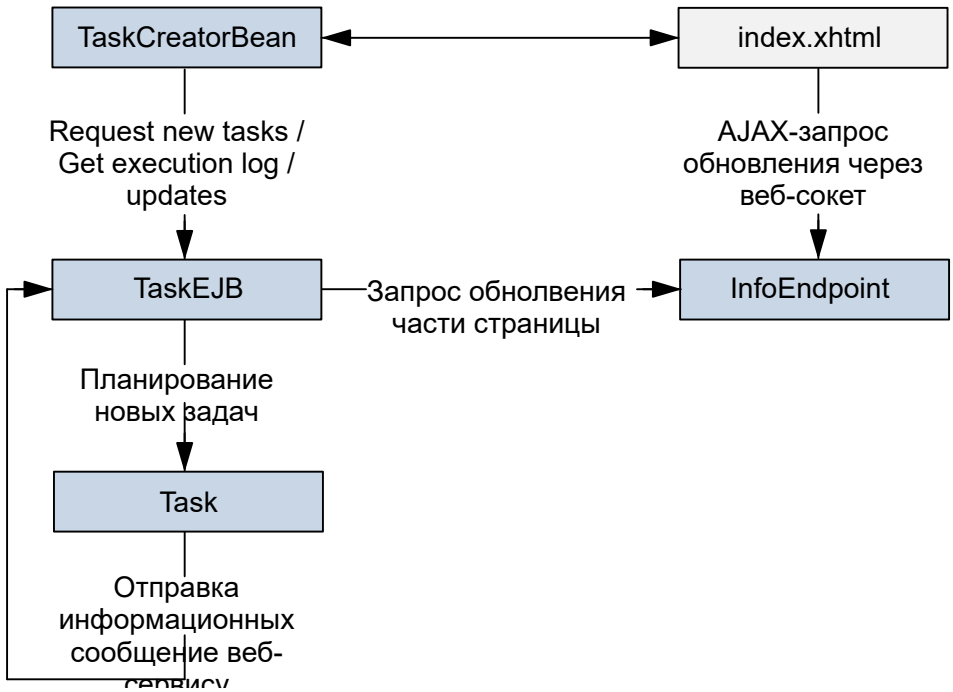

Рисунок 60-1 Архитектура taskcreator

Класс `TaskEJB` получает объекты ExecutorService по умолчанию с сервера приложений следующим образом:

JAVA

```
@Resource(name="java:comp/DefaultManagedExecutorService")
ManagedExecutorService mExecService;

@Resource(name="java:comp/DefaultManagedScheduledExecutorService")
ManagedScheduledExecutorService sExecService;
```

Метод `submitTask` в `TaskEJB` использует эти объекты для отправки задач на выполнение следующим образом:

JAVA

```
public void submitTask(Task task, String type) {
    /* Использование объекта managed executor objects с сервера приложений */
    switch (type) {
        case "IMMEDIATE":
            mExecService.submit(task);
            break;
        case "DELAYED":
            sExecService.schedule(task, 3, TimeUnit.SECONDS);
            break;
        case "PERIODIC":
            ScheduledFuture fut;
            fut = sExecService.scheduleAtFixedRate(task, 0, 8,
                    TimeUnit.SECONDS);
            periodicTasks.put(task.getName(), fut);
            break;
    }
}
```

Для периодических задач `TaskEJB` хранит ссылку на объект `ScheduledFuture`, чтобы пользователь мог отменить задание в любое время.

Запуск taskcreator

В этом разделе описывается, как собрать, упаковать, развернуть и запустить `taskcreator` с IDE NetBeans или Maven.

Сборка, упаковка и развёртывание taskcreator с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню «Файл» выберите «Открыть проект».
3. В диалоговом окне «Открыть проект» перейдите к:

    ```
    tut-install/examples/concurrency
    ```

4. Выберите каталог `taskcreator`.
5. Нажмите Открыть проект.
6. На вкладке «Проекты» кликните правой кнопкой мыши проект `taskcreator` и выберите «Сборка».

    Эта команда собирает и развёртывает приложение.

Сборка, упаковка и развёртывание taskcreator с использованием Maven

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В окне терминала перейдите в:

    ```
    tut-install/examples/concurrency/taskcreator
    ```

3. Введите следующую команду, чтобы собрать и развернуть приложение:

    SHELL

    ```
    mvn install
    ```

Запуск taskcreator

1. Откройте следующий URL в веб-браузере:

    ```
    http://localhost:8080/taskcreator/
    ```

Страница содержит форму для отправки заданий, журнал их выполнения и форму для отмены периодических заданий.

2. Выберите тип задания «Немедленно», введите имя задания и нажмите кнопку «Отправить». Сообщения, подобные следующим, появляются в журнале выполнения задания:

```
IMMEDIATE Task TaskA started
IMMEDIATE Task TaskA finished
```

3. Выберите тип задания с задержкой (3 секунды), введите имя задания и нажмите кнопку «Отправить». Сообщения, подобные следующим, появляются в журнале выполнения задания:

```
DELAYED Task TaskB submitted
DELAYED Task TaskB started
DELAYED Task TaskB finished
```

4. Выберите тип периодического (8 секунд) задания, введите имя задания и нажмите кнопку «Отправить». Сообщения, подобные следующим, появляются в журнале выполнения задания:

```
PERIODIC Task TaskC started run #1
PERIODIC Task TaskC finished run #1
PERIODIC Task TaskC started run #2
PERIODIC Task TaskC finished run #2
```

Вы можете добавить более одного периодического задания. Чтобы отменить периодическое задание, выберите его в форме и нажмите «Отменить задание».

Дополнительная информация о Jakarta Concurrency

Для получения дополнительной информации о параллелизме см.

- Спецификация Jakarta Concurrency 2.0: https://jakarta.ee/specifications/concurrency/2.0/
- Урок параллелизма в учебниках Java: https://docs.oracle.com/javase/tutorial/essential/concurrency/

Часть XII: Учебные примеры

В части XII представлены примеры, в которых используются различные технологии Jakarta EE.

Глава 61. Пример Duke's Bookstore

Пример Duke’s Bookstore — это простое приложение для электронной коммерции, которое иллюстрирует некоторые из дополнительных функций Jakarta Faces в сочетании с Jakarta Contexts and Dependency Injection (CDI), EJB-компонентами и Jakarta Persistence. Пользователи могут выбирать книги на карте изображения, просматривать каталог книжного магазина и покупать книги. Безопасность не используется в этом приложении.

Дизайн и архитектура Duke's Bookstore

Duke's Bookstore — это веб-приложение, которое использует многие функции Jakarta Faces в дополнение к функциям Jakarta EE:

- Jakarta Faces и Jakarta Contexts and Dependency Injection (CDI)
  - Пользовательский интерфейс приложения представлен набором страниц Facelets с шаблонами.
  - С каждой страницей Facelets связаны Managed-бины CDI.
  - Компонент карты изображения на главной странице позволяет выбрать книгу для входа в магазин. Каждая область карты представлена Managed-бином Jakarta Faces. Для большей доступности присутствуют текстовые гиперссылки.
  - Слушатели действий зарегистрированы на карте изображения и в текстовых ссылках. Эти слушатели получают значение идентификатора для выбранной книги и сохраняют его в сессии, чтобы Managed-бин мог получить его на следующей странице.
  - Тег `h:dataTable` используется для динамической отрисовки каталога книг и корзины покупок.
  - Кастомный конвертер зарегистрирован в поле кредитной карты на странице оформления заказа `bookcashier.xhtml`, где также используется тег `f:validateRegEx` для обеспечения корректного форматирования ввода.
  - Слушатель изменения значения зарегистрирован в поле имени на странице `bookcashier.xhtml`. Этот слушатель сохраняет имя в параметре, чтобы следующая страница — `bookreceipt.xhtml` — могла получить к нему доступ.
- Enterprise-бины: локальный сессионный компонент без сохранения состояния и компонент-синглтон
- Сущность Jakarta Persistence

Пакеты приложения taskcreator, расположенные в каталоге *`tut-install`*`/examples/case-studies/dukes-bookstore/src/main/java/ee/jakarta/tutorial/dukesbookstore/`, выглядят следующим образом:

- `components`: включает в себя кастомные классы компонентов пользовательского интерфейса `MapComponent` и `AreaComponent`
- `converters`: включает класс кастомного конвертера `CreditCardConverter`
- `ejb`: включает два Enterprise-бина:
  - Компонент-синглтон `ConfigBean`, который инициализирует данные в базе данных
  - Сессионный компонент без сохранения состояния `BookRequestBean`, который содержит бизнес-логику управления объектами сущности
- `entity`: включает класс сущности `Book`
- `exceptions`: включает три класса исключений

- `listeners` : включает обработчик событий и классы слушателей событий
- `model` : включает класс модели JavaBeans
- `renderers` : включает кастомные отрисовщики для кастомных классов компонентов пользовательского интерфейса
- `web.managedbeans` : включает Managed-бины для страниц Facelets
- `web.messages` : включает файлы bundle-ресурсов для локализованных сообщений

Интерфейс Duke's Bookstore

В этом разделе приведены дополнительные сведения о компонентах примера Duke's Bookstore и о том, как они взаимодействуют.

Сущность Book

Сущность `Book` , расположенная в пакете `dukesbookstore.entity` , инкапсулирует данные книги, хранящиеся в Duke's Bookstore.

Сущность `Book` определяет атрибуты, используемые в примере:

- Идентификатор книги
- Имя автора
- Фамилия автора
- Название
- Цена
- Есть ли книга в продаже
- Год публикации
- Описание книги
- Количество копий в инвентаре

Сущность `Book` также определяет простой именованный запрос `findBooks` .

Enterprise-бины, используемые в Duke's Bookstore

Два Enterprise-бина, расположенных в пакете `dukesbookstore.ejb` , обеспечивают бизнес-логику для Duke's Bookstore.

- `BookRequestBean` — это сессионный компонент с состоянием, который содержит бизнес-методы для приложения. Методы создания, поиска и покупки книг, а также обновления инвентаря для книги. Чтобы получить книги, метод `getBooks` вызывает именованный запрос `findBooks` , определённый в сущности `Book` .
- `ConfigBean` — это сессионный компонент-синглтон, используемый для создания книг в каталоге при первоначальном развёртывании приложения. Он вызывает метод `createBook` , определённый в `BookRequestBean` .

Страницы Facelets и Managed-бины, используемые в Duke's Bookstore

Приложение Duke's Bookstore использует Facelets и его шаблоны для отрисовки пользовательского интерфейса. Страницы Facelets взаимодействуют с набором Managed-бинов CDI, которые действуют как бины поддержки для свойств и методов пользовательского интерфейса. Главная страница также взаимодействует с

кастомными компонентами приложения.

Приложение использует следующие страницы Facelets, которые расположены в каталоге *tut-install*/examples/case-studies/dukes-bookstore/src/main/webapp/ .

- bookstoreTemplate.xhtml : файл шаблона, который определяет заголовок, используемый на каждой странице, а также таблицу стилей, используемую всеми страницами. Шаблон также извлекает язык из настроек веб-браузера.

  Использует Managed-бин LocaleBean .

- index.xhtml : страница входа, на которой размещаются настраиваемые компоненты карты и области с использованием Managed-бинов, настроенных в файле faces-config.xml , позволяет пользователю выбирать книгу и перейти на страницу bookstore.xhtml .

- bookstore.xhtml : страница, позволяющая пользователю получить подробную информацию о выбранной книге, добавить книгу в корзину покупок и перейти на страницу bookcatalog.xhtml .

  Использует Managed-бин BookstoreBean .

- bookdetails.xhtml : страница, которая отображает сведения о книге, выбранной на bookstore.xhtml или других страницах, и позволяет пользователю добавить книгу в корзину и/или перейти на страницу bookcatalog.xhtml .

  Использует Managed-бин BookDetailsBean .

- bookcatalog.xhtml : страница, которая отображает книги в каталоге и позволяет пользователю добавлять книги в корзину, просматривать сведения о любой книге, просматривать и очищать корзину или совершать покупки книг, находящихся в корзине.

  Используются Managed-бины BookstoreBean , CatalogBean и ShoppingCart .

- bookshowcart.xhtml : страница, которая отображает содержимое корзины и позволяет пользователю удалять и просматривать содержимое, очищать корзину, приобретать книги в корзине или вернуться в каталог.

  Использует Managed-бины ShowCartBean и ShoppingCart .

- bookcashier.xhtml : страница, позволяющая пользователю приобретать книги, указывать способ доставки, подписываться на рассылку новостей или вступать в фан-клуб Duke при покупке выше определённой суммы.

  Использует Managed-бины CashierBean и ShoppingCart .

- bookreceipt.xhtml : страница, которая подтверждает покупку пользователя и позволяет пользователю вернуться на страницу каталога, чтобы продолжить покупки.

  Использует Managed-бин CashierBean .

- bookordererror.xhtml : страница, отображаемая CashierBean , если в книжном магазине не осталось ни одного экземпляра заказанной книги.

Приложение использует следующие Managed-бины, которые находятся в каталоге *tut-install*/examples/case-studies/dukes-bookstore/src/main/java/ee/jakarta/tutorial/dukesbookstore/web/managedbeans/ .

- AbstractBean : содержит служебные методы, вызываемые другими Managed-бинами.
- BookDetailsBean : вспомогательный бин для страницы bookdetails.xhtml . Определяет имя details .
- BookstoreBean : вспомогательный бин для страницы bookstore.xhtml . Определяет имя store .

- `CashierBean` : вспомогательный бин для страниц `bookcashier.xhtml` и `bookreceipt.xhtml` .
- `CatalogBean` : вспомогательный бин для страницы `bookcatalog.xhtml` . Определяет имя `catalog` .
- `LocaleBean` : Managed-бин, который получает текущую локаль. Используется на каждой странице.
- `ShoppingCart` : вспомогательный бин, используемый страницами `bookcashier.xhtml` , `bookcatalog.xhtml` и `bookshowcart.xhtml` . Определяет имя `cart` .
- `ShoppingCartItem` : содержит методы, вызываемые `ShoppingCart` , `CatalogBean` и `ShowCartBean` .
- `ShowCartBean` : вспомогательный бин для страницы `bookshowcart.xhtml` . Определяет имя `showcart` .

Пользовательские компоненты и другие кастомные объекты, используемые в Duke's Bookstore

Пользовательские компоненты карты и области для Duke's Bookstore, а также связанные с ними классы визуализации, слушателя и модели определены в следующих пакетах в каталоге *`tut-install`*`/examples/case-studies/dukes-bookstore/src/main/java/ee/jakarta/tutorial/dukesbookstore/` .

- `components` : содержит классы `MapComponent` и `AreaComponent` . Смотрите Создание классов кастомного компонента.
- `listeners` : содержит класс `AreaSelectedEvent` , а также другие классы слушателей. Смотрите Обработка событий для кастомных компонентов.
- `model` : содержит класс `ImageArea` . Смотрите Конфигурирование данных модели для получения дополнительной информации.
- `renderers` : содержит классы `MapRenderer` и `AreaRenderer` . Смотрите Делегирование отрисовки отрисовщику.

Каталог *`tut-install`*`/examples/case-studies/dukes-bookstore/src/java/dukesbookstore/` также содержит кастомный конвертер и другие кастомные слушатели, не связанные с нестандартными компонентами.

- `converters` : содержит класс `CreditCardConverter` . Смотрите Создание и использование кастомного конвертера.
- `listeners` : содержит классы `LinkBookChangeListener` , `MapBookChangeListener` и `NameChanged` . Смотрите Реализация слушателя событий.

Файлы свойств, используемые в Duke's Bookstore

Строки, используемые в приложении Duke's Bookstore, инкапсулированы в bundle-ресурсы, что позволяет отображать локализованные строки в нескольких локалях. Файлы свойств, расположенные в *`tut-install`*`/examples/case-studies/dukes-bookstore/src/main/java/ee/jakarta/tutorial/dukesbookstore/web/messages/` , состоят из файла по умолчанию, содержащего английские строки, и трёх дополнительных файлов для других локалей. Файлы следующие:

- `Messages.properties` : файл по умолчанию, содержащий английские строки
- `Messages_de.properties` : файл, содержащий немецкие строки
- `Messages_es.properties` : файл, содержащий испанские строки
- `Messages_fr.properties` : файл, содержащий французские строки

Настройка языка в веб-браузере пользователя определяет, какой язык используется. Тег `html` в `bookstoreTemplate.xhtml` извлекает языковые настройки из свойства `language` в `LocaleBean` :

```xml
<html lang="#{localeBean.language}">
...
```

Для получения дополнительной информации о пакетах ресурсов см. главу 22 *Интернационализация и локализация веб-приложений*.

Пакет ресурсов настраивается в файле `faces-config.xml` следующим образом:

```xml
<application>
    <resource-bundle>
        <base-name>
            ee.jakarta.tutorial.dukesbookstore.web.messages.Messages
        </base-name>
        <var>bundle</var>
    </resource-bundle>
    <locale-config>
        <default-locale>en</default-locale>
        <supported-locale>de</supported-locale>
        <supported-locale>es</supported-locale>
        <supported-locale>fr</supported-locale>
    </locale-config>
</application>
```

Эта конфигурация означает, что на страницах Facelets сообщения извлекаются с помощью префиксов `bundle` с ключом, находящимся в файле `Messages_'locale'.properties` , как показано в следующем примере со страницы `index.xhtml` :

```xml
<h:outputText style="font-weight:bold"
              value="#{bundle.ChooseBook}" />
```

В `Messages.properties` ключевая строка определяется следующим образом:

```java
ChooseBook=Choose a Book from our Catalog
```

Дескрипторы развёртывания, используемые в Duke's Bookstore

В Duke's Bookstore используются следующие дескрипторы развёртывания:

- `src/main/resources/META-INF/persistence.xml` : Файл конфигурации Jakarta Persistence
- `src/main/webapp/WEB-INF/bookstore.taglib.xml` : файл дескриптора библиотеки тегов для кастомных компонентов
- `src/main/webapp/WEB-INF/faces-config.xml` : файл конфигурации Jakarta Faces, в котором настраиваются Managed-бины для компонента карты, а также пакеты ресурсов для приложения
- `src/main/webapp/WEB-INF/web.xml` : файл конфигурации веб-приложения

Запуск Duke's Bookstore

Вы можете использовать IDE NetBeans или Maven для сборки, упаковки, развёртывания и запуска приложения Duke's Bookstore.

Сборка и развёртывание Duke's Bookstore с IDE NetBeans

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. В меню «Файл» выберите «Открыть проект».

3. В диалоговом окне «Открыть проект» перейдите к:

```
tut-install/examples/case-studies
```

4. Выберите каталог `dukes-bookstore`.
5. Нажмите Открыть проект.
6. На вкладке «Проекты» кликните правой кнопкой мыши проект `dukes-bookstore` и выберите «Сборка».

   Это позволит создать, упаковать и развернуть Duke's Bookstore в GlassFish Server.

Сборка и развёртывание Duke's Bookstore с помощью Maven

1. Убедитесь, что GlassFish Server (см. Запуск и остановка GlassFish Server), а также сервер базы данных (см. Запуск и остановка Apache Derby) запущены.
2. В окне терминала перейдите в:

```
tut-install/examples/case-studies/dukes-bookstore/
```

3. Введите следующую команду:

```
mvn install
```
SHELL

   Эта команда собирает приложение и упаковывает его в WAR-файл в каталоге *`tut-install`*`/examples/case-studies/dukes-bookstore/target/`. Затем она развёртывает приложение в GlassFish Server.

Запуск Duke's Bookstore

1. В веб-браузере введите следующий URL:

```
http://localhost:8080/dukes-bookstore/
```

2. На главной странице Duke's Bookstore кликните книгу на рисунке или нажмите одну из ссылок внизу страницы.
3. Используйте страницы в приложении для просмотра и покупки книг.

Глава 62. Пример Duke’s Tutoring

Приложение Duke’s Tutoring — это система контроля посещаемости для учебного центра. Может быть проверена посещаемость занятий студентами. Учебный центр может отслеживать посещаемость, обновлять статус и хранить контактную информацию опекунов и студентов. Система учебного центра поддерживается администраторами.

Дизайн и архитектура Duke’s Tutoring

Duke's Tutoring — это веб-приложение, которое включает в себя несколько технологий Jakarta EE. Он предоставляет как основной интерфейс (для студентов, опекунов и сотрудников учебного центра), так и административный интерфейс (для обслуживающего систему персонала). Бизнес-логика для обоих интерфейсов обеспечивается Enterprise-бинами. EJB-компоненты используют Jakarta Persistence для создания и хранения данных приложения в базе данных. Рисунок 62-1 иллюстрирует архитектуру приложения.

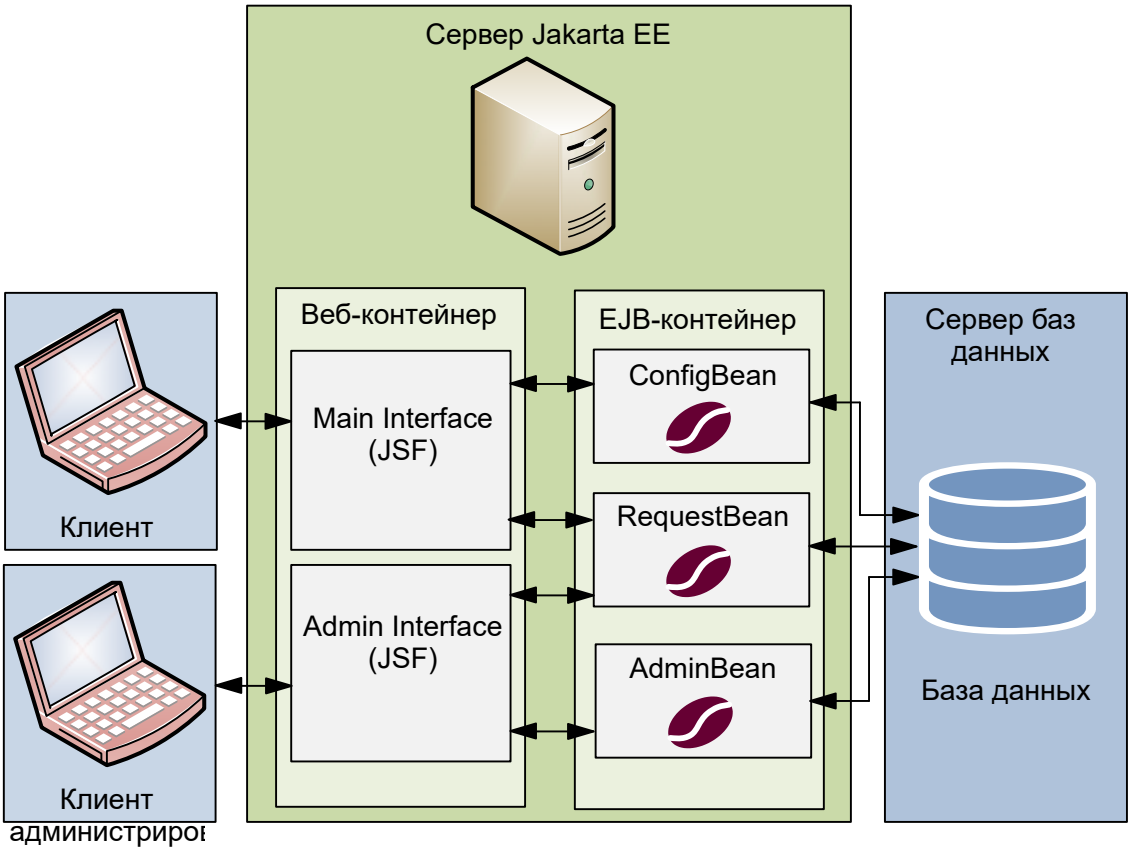

Рисунок 62-1 Архитектура Duke’s Tutoring

Приложение Duke’s Tutoring состоит из двух основных проектов: библиотеки `dukes-tutoring-common` и веб-приложения `dukes-tutoring-war` . Проект библиотеки `dukes-tutoring-common` содержит классы сущностей и вспомогательные классы, используемые веб-приложением `dukes-tutoring-war` , и `dukes-tutoring-common` упакован и развёртывается вместе с `dukes-tutoring-war` . JAR-файл библиотеки удобен для повторного использования классов сущностей и вспомогательных классов в других приложениях, таких как клиентское приложение JavaFX.

Duke’s Tutoring использует следующие функции платформы Jakarta EE:

- Сущности Jakarta Persistence
  - Кастомная аннотация `@Email` Jakarta Bean Validation для валидации адресов электронной почты
  - Стандартное определение `jta-data-source` , которое создаёт ресурс JDBC при развёртывании

    - Стандартное свойство в дескрипторе развёртывания `persistence.xml` для автоматического и переносимого создания и удаления таблиц в `jta-data-source`
- Enterprise-бины
    - Локальная сессия представления без интерфейса и компоненты-синглтоны
    - Ресурсы RESTful веб-сервисов Jakarta в сессионных компонентах
    - Ограничения безопасности Jakarta EE для бизнес-методов административного интерфейса
    - Все Enterprise-бины упакованы в WAR
- Веб-сокеты
    - Серверная конечная точка веб-сокета, которая автоматически публикует статус учащихся на клиентских конечных точках
- Инъецирование контекстов и зависимостей Jakarta
    - Событие CDI, которое запускается при изменении статуса студента
    - Методы-обработчики для обновления приложения после запуска события состояния
    - Managed-бины CDI для страниц Facelets
    - Аннотации Bean Validation в Managed-бинах CDI
- Jakarta Faces, использующая Facelets для веб-интерфейса
    - Шаблонизация
    - Составные компоненты
    - Кастомный форматтер, `PhoneNumberFormatter`
    - Ограничения безопасности на административном интерфейсе
    - Компоненты Facelets с поддержкой Ajax

Приложение Duke's Tutoring имеет два основных пользовательских интерфейса, оба упакованы в один WAR-файл:

- Основной интерфейс для студентов, опекунов и сотрудников
- Административный интерфейс, используемый персоналом для управления студентами и опекунами, а также для создания отчётов о посещаемости

Основной интерфейс

Основной интерфейс позволяет студентам и сотрудникам контролировать посещаемость учащихся, а также фиксировать присутствие студентов на игровой площадке.

Сущности Jakarta Persistence, используемые в основном интерфейсе

Следующие сущности, используемые в основном интерфейсе, инкапсулируют данные, хранящиеся и управляемые Duke's Tutoring, и находятся в пакете `dukestutoring.entity` проекта `dukes-tutoring-common`.

- `Person`: сущность `Person` определяет атрибуты, общие для студентов и опекунов, отслеживаемых приложением. Этими атрибутами являются имя человека и контактная информация, включающая номера телефонов и адрес электронной почты. У этой сущности есть два дочерних класса: `Student` и `Guardian`.

- `PersonDetails`: сущность `PersonDetails` используется для хранения дополнительных данных, общих для всех людей, таких как фотография и день рождения человека, которые не включены в сущность `Person` по причинам производительности.
- `Student` и `Guardian`: сущность `Student` хранит атрибуты, специфичные для студентов, которые проходят обучение. Они включают в себя такую информацию, как класс ученика и школа. Атрибуты сущности `Guardian` специфичны для родителей или опекунов `Student`. Студенты и опекуны связаны отношением "многие ко многим". То есть у студента может быть один или несколько опекунов, а у опекуна может быть один или несколько студентов.
- `Address`: сущность `Address` представляет почтовый адрес и связана с сущностями `Person`. Адреса и люди связаны отношением "многие к одному". То есть у одного человека может быть много адресов.
- `TutoringSession`: сущность `TutoringSession` представляет определённый день в обучающем центре. Отдельная учебная сессия отслеживает, какие студенты посетили этот день, а какие студенты пошли в парк.
- `StatusEntry`: сущность `StatusEntry`, которая регистрирует изменения статуса студента, связана с сущностью `TutoringSession`. Статусы студентов меняются в момент их регистрации в учебный день или когда они идут в парк, и когда они их покидают. Запись статуса позволяет сотрудникам учебного центра точно отслеживать, какие студенты посещали занятия, когда они регистрировались, какие студенты ходили в парк, когда они были в учебном центре, и когда они приходили и возвращались из парка.

Дополнительные сведения о создании сущностей Jakarta Persistence см. в главе 40 *Введение в Jakarta Persistence*. Дополнительные сведения о валидации данных сущностей см. в Валидация персистентных полей и свойств и главе 24 *Bean Validation: дополнительные темы*.

Enterprise-бины, используемые в основном интерфейсе

Следующие Enterprise-бины, используемые в основном интерфейсе, предоставляют бизнес-логику для Duke's Tutoring и находятся в пакете `dukestutoring.ejb` проекта `dukes-tutoring-war`:

- `ConfigBean` — это сессионный компонент-синглтон, используемый для создания записей студентов при первоначальном развёртывании приложения и для задания автоматического EJB-таймера, который создаёт сущности каждый день недели.
- `RequestBean` — это сессионный компонент без сохранения состояния, содержащий бизнес-методы для основного интерфейса. Бин также имеет бизнес-методы для получения списка студентов. Эти бизнес-методы используют строго типизированные запросы Criteria API для извлечения данных из базы данных. `RequestBean` также инъецирует объект события CDI, `StatusEvent`. Это событие вызывается бизнес-методами при изменении статуса студента.

Сведения о создании и использовании Enterprise-бинов см. в части VII «Enterprise-бины». Для получения информации о создании строго типизированных запросов Criteria API см. главу 43, *Использование Criteria API для создания запросов*. Дополнительные сведения о событиях CDI см. в Использование событий в приложениях CDI.

Конечная точка веб-сокета, используемая в основном интерфейсе

Класс `ee.jakarta.tutorial.dukestutoring.web.websocket.StatusEndpoint` — это конечная точка сервера веб-сокета, которая возвращает студентов и их состояние клиентским конечным точкам. Метод `StatusEndpoint.updateStatus` является методом наблюдателя CDI для события `StatusEvent`. При изменении статуса студента в основном интерфейсе запускается `StatusEvent`. Контейнер вызывает метод-наблюдатель `updateStatus` и передаёт изменение состояния всем клиентским конечным точкам, зарегистрированным в `StatusEndpoint`.

Страница `index.xhtml` Jakarta Faces содержит код JavaScript для подключения к конечной точке веб-сокета. Метод `onMessage` на этой странице кликает кнопку Jakarta Faces, которая отправляет Ajax-запрос на обновление таблицы, отображающей текущий статус студентов.

Дополнительные сведения о конечных точках веб-сокета см. в главе 19 *Jakarta WebSocket*. Дополнительные сведения о событиях CDI см. в Использование событий в приложениях CDI.

Файлы Facelets, используемые в основном интерфейсе

Приложение Duke's Tutoring использует Facelets для отображения пользовательского интерфейса и широко использует шаблоны Facelets. Facelets, технология отображения по умолчанию для Jakarta Faces, состоит из файлов XHTML, расположенных в каталоге *`tut-install`*`/examples/case-studies/dukes-tutoring-war/src/main/webapp/`.

В основном интерфейсе используются следующие файлы Facelets:

- `template.xhtml` : файл шаблона для основного интерфейса
- `error.xhtml` : файл ошибки, если что-то идёт не так
- `index.xhtml` : страница входа для основного интерфейса
- `park.xhtml` : страница, показывающая, кто в данный момент находится в парке
- `current.xhtml` : страница, показывающая, кто в данный момент участвует в сегодняшней учебной сессии
- `statusEntries.xhtml` : страница, показывающая подробный журнал записей статуса для сегодняшней учебной сессии
- `resources/components/allStudentsTable.xhtml` : составной компонент для таблицы, отображающей всех активных студентов
- `resources/components/allInactiveStudentsTable.xhtml` : составной компонент для таблицы, отображающей всех неактивных студентов
- `resources/components/currentSessionTable.xhtml` : составной компонент для таблицы, отображающей всех студентов на сегодняшней учебной сессии
- `resources/components/parkTable.xhtml` : составной компонент для таблицы, отображающей всех студентов, которые в данный момент находятся в парке
- `WEB-INF/includes/mainNav.xhtml` : фрагмент XHTML для панели навигации основного интерфейса

Дополнительные сведения об использовании Facelets см. в главе 8 *Введение в Facelets*.

Вспомогательные классы, используемые в основном интерфейсе

Следующие вспомогательные классы, находящиеся в пакете `dukestutoring.util` проекта `dukes-tutoring-common` , используются в основном интерфейсе.

- `CalendarUtil` : класс, предоставляющий метод для удаления ненужных временны́х данных из объектов `java.util.Calendar` .
- `Email` : класс кастомной аннотации Bean Validation для валидации адресов электронной почты в сущности `Person` .
- `StatusType` : перечислимый тип, определяющий различные статусы, которые может иметь студент. Возможные значения: `IN` , `OUT` и `PARK` . `StatusType` используется во всём приложении, в том числе в сущности `StatusEntry` и во всём основном интерфейсе. `StatusType` также определяет метод `toString` , который возвращает локализованное название статуса исходя из локали.

Файлы свойств

Строки, используемые в основном интерфейсе, инкапсулированы в bundle-ресурсы, что позволяет отображать локализованные сообщения в нескольких локалях. Каждый из файлов свойств имеет специфичные для локали файлы, к которым добавлены коды локали, содержащие переведённые строки для каждой локали. Например, `Messages_es.properties` содержит локализованные строки для испанских локалей.

Проект `dukes-tutoring-common` имеет следующий bundle-ресурс в `src/main/resources/`.

- `ee.jakarta.tutorial/dukestutoring/util/StatusMessages.properties` : строки для каждого из типов статуса, определённых в перечисляемом типе `StatusType` для локали по умолчанию. Каждая поддерживаемая локаль имеет файл свойств вида `StatusMessages_`*`locale prefix`*`.properties` содержащий локализованные строки. Например, строки для испаноязычных локалей находятся в `StatusMessages_es.properties`.

Проект `dukes-tutoring-war` имеет следующие bundle-ресурсы в `src/main/resources/`.

- `ValidationMessages.properties` : строки для локали по умолчанию, используемой средой выполнения Bean Validation для отображения сообщений валидации. Этот файл должен иметь имя `ValidationMessages.properties` и находиться в пакете по умолчанию, как того требует спецификация Bean Validation. Каждая поддерживаемая локаль имеет файл свойств вида `ValidationMessages_`*`locale prefix`*`.properties`, содержащий локализованные строки. Например, строки для немецкоязычных локалей находятся в `ValidationMessages_de.properties`.
- `ee.jakarta.tutorial/dukestutoring/web/messages/Messages.properties` : строки для локали по умолчанию для основного и административного интерфейса Facelets. Каждая поддерживаемая локаль имеет файл свойств вида `Messages_`*`locale prefix`*`.properties` содержащий локализованные строки. Например, строки для упрощённых китайскоязычных локалей находятся в `Messages_zh.properties`.

Для получения информации о локализации веб-приложений см. Регистрация сообщений приложения.

Дескрипторы развёртывания, используемые в Duke's Tutoring

Duke's Tutoring хранит дескрипторы развёртывания в каталоге `src/main/webapp/WEB-INF` проекта `dukes-tutoring-war`:

- `faces-config.xml` : файл конфигурации Jakarta Faces
- `glassfish-web.xml` : файл конфигурации, специфичный для GlassFish Server, который определяет сопоставление ролей безопасности
- `web.xml` : файл конфигурации веб-приложения

Duke's Tutoring также использует следующий дескриптор развёртывания в каталоге `src/main/resources/META-INF` проекта `dukes-tutoring-common`:

- `persistence.xml` : файл конфигурации Jakarta Persistence

Дескриптор развёртывания Enterprise-бина не используется в Duke's Tutoring. Аннотации в файлах классов Enterprise-бинов используются для настройки Enterprise-бинов в этом приложении.

Интерфейс администрирования

Интерфейс администрирования Duke's Tutoring используется персоналом учебного центра для управления данными, используемыми основным интерфейсом: студентами, их опекунами и адресами. Интерфейс администрирования использует многие из тех же компонентов, что и основной интерфейс. Дополнительные компоненты, которые используются только в интерфейсе администрирования, описаны здесь.

Enterprise-бины, используемые в интерфейсе администрирования

Следующий Enterprise-бин в пакете `dukestutoring.ejb` проекта `dukes-tutoring-war` используется в интерфейсе администрирования.

- `AdminBean` : сессионный компонент без сохранения состояния для всей бизнес-логики, используемой в интерфейсе администрирования. Вызывает методы безопасности, чтобы разрешить вызов бизнес-методов только авторизованным пользователям.

Файлы Facelets, используемые в интерфейсе администрирования

Следующие файлы Facelets в `src/main/webapp/` используются в интерфейсе администрирования:

- `admin/adminTemplate.xhtml` : шаблон для интерфейса администрирования
- `admin/index.xhtml` : страница входа для интерфейса администрирования
- `login.xhtml` : страница входа для безопасного интерфейса администрирования
- `loginError.xhtml` : страница отображается при наличии ошибок при аутентификации пользователя-администратора
- Каталог `admin/address` : страницы, позволяющие создавать, редактировать и удалять объекты сущности `Address`
- Каталог `admin/guardian` : страницы, позволяющие создавать, редактировать и удалять объекты сущности `Guardian`
- Каталог `admin/student` : страницы, позволяющие создавать, редактировать и удалять объекты сущности `Student`
- `resources/components/formLogin.xhtml` : составной компонент для формы входа в систему с использованием Jakarta Security
- `WEB-INF/includes/adminNav.xhtml` : фрагмент XHTML для панели навигации интерфейса администратора

Managed-бины CDI, используемые в интерфейсе администрирования

Managed-бины CDI, используемые в интерфейсе администрирования, находятся в пакете `dukestutoring.web` проекта `dukes-tutoring-war` .

- `StudentBean.java` : Managed-бин для страниц Facelets, используемый для создания и редактирования студентов. Имя и фамилия имеют аннотации Bean Validation, которые требуют заполненности полей. Телефонные номера имеют аннотации Bean Validation, чтобы гарантировать, что представленные данные корректны.
- `GuardianBean.java` : Managed-бин для страниц Facelets, используемый для создания опекунов и назначения опекунов студентам. Имя и фамилия имеют аннотации Bean Validation, которые требуют заполненности полей. Телефонные номера имеют аннотации Bean Validation, чтобы гарантировать, что представленные данные корректны.
- `AddressBean.java` : Managed-бин для страниц Facelets, используемый для создания адресов для студентов. Атрибуты улицы, города, регионы и почтового индекса имеют аннотации Bean Validation, которые требуют заполненности полей, а атрибут почтового индекса имеет дополнительную аннотацию для обеспечения

корректности данных.

Вспомогательные классы, используемые в интерфейсе администрирования

Следующие вспомогательные классы, найденные в пакете `dukestutoring.web.util` проекта `dukes-tutoring-war`, используются в интерфейсе администрирования.

- `EntityConverter`: родительский класс для `StudentConverter` и `GuardianConverter`, который определяет кэш для хранения классов сущностей при конвертации сущностей для использования в пользовательском интерфейсе Jakarta Faces. Кэш помогает повысить производительность. Кэш хранится в контексте Jakarta Faces.
- `StudentConverter`: конвертер Jakarta Faces для класса сущностей `Student`. Этот класс содержит методы для преобразования объектов `Student` в строки и обратно, чтобы их можно было использовать в компонентах пользовательского интерфейса приложения.
- `GuardianConverter`: Подобно `StudentConverter`, этот класс является конвертером для класса сущностей `Guardian`.

Запуск Duke's Tutoring

В этом разделе описывается, как собрать, упаковать, развернуть и запустить приложение Duke's Tutoring.

Запуск Duke's Tutoring

Вы можете использовать IDE NetBeans или Maven для создания, упаковки, развёртывания и запуска Duke's Tutoring.

Сборка и развёртывание Duke's Tutoring с IDE NetBeans

Прежде чем начать

Вы должны настроить GlassFish Server в качестве сервера Jakarta EE в IDE NetBeans, как описано в Добавление GlassFish Server в IDE NetBeans.

1. Удостоверьтесь, чтобы GlassFish Server был запущен (см. Запуск и остановка GlassFish Server).
2. Если сервер базы данных ещё не запущен, запустите его, как описано в Запуск и остановка Apache Derby.
3. В меню «Файл» выберите «Открыть проект».
4. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/case-studies
   ```

5. Выберите каталог `dukes-tutoring`.
6. Установите флажок «Открыть требуемые проекты» и нажмите «Открыть проект».

   

   Когда вы в первый раз откроете Duke's Tutoring в NetBeans, увидите глифы ошибок на вкладке Проекты. Это ожидаемо, поскольку файлы метамодели, используемые Enterprise-бинами для запросов Criteria API, ещё не созданы.

7. На вкладке «Проекты» кликните правой кнопкой мыши проект `dukes-tutoring` и выберите «Сборка».

   Эта команда создаёт область безопасности JDBC с именем tutoringRealm, собирает и упаковывает проекты `dukes-tutoring-common` и `dukes-tutoring-war` и развёртывает `dukes-tutoring-war` в GlassFish Server, запустив Derby и GlassFish Server, если они ещё не были запущены.

Сборка и развёртывание Duke's Tutoring с помощью Maven

1. Убедитесь, что GlassFish Server запущен (см. Запуск и остановка GlassFish Server).
2. Если сервер базы данных ещё не запущен, запустите его, как описано в Запуск и остановка Apache Derby.
3. В окне терминала перейдите в:

   ```
   tut-install/examples/case-studies/dukes-tutoring/
   ```

4. Введите следующую команду:

   ```
   mvn install
   ```
   SHELL

   Эта команда создаёт область безопасности JDBC с именем `tutoringRealm`, собирает и упаковывает проекты `dukes-tutoring-common` и `dukes-tutoring-war` и развёртывает `dukes-tutoring-war` в GlassFish Server.

Использование Duke's Tutoring

Когда программа Duke’s Tutoring будет запущена в GlassFish Server, используйте основной интерфейс, чтобы поэкспериментировать с входом и выходом студентов или отправкой их в парк.

Использовать основной интерфейс Duke's Tutoring

1. В веб-браузере откройте основной интерфейс по следующему URL:

   ```
   http://localhost:8080/dukes-tutoring-war/
   ```

2. Используйте основной интерфейс для входа и выхода студентов, а также для входа, когда студенты идут в парк.

Использовать интерфейс администрирования Duke's Tutoring

Следуйте этим инструкциям, чтобы войти в интерфейс администрирования Duke's Tutoring и добавить новых учеников, опекунов и адреса.

1. В главном интерфейсе откройте интерфейс администрирования, кликнув главную страницу Администрирование в левом меню.

   Это перенаправляет вас на страницу входа по следующему URL:

   ```
   http://localhost:8080/dukes-tutoring-war/admin/index.xhtml
   ```

2. На странице входа введите `admin@example.com` в поле "Имя пользователя" и введите `jakartaee` в поле "Пароль".
3. Используйте интерфейс администрирования, чтобы добавлять или изменять студентов, добавлять опекунов или добавлять адреса.
   - Чтобы добавить нового студента, нажмите «Создать нового студента» в левом меню, заполните поля (два обязательных) в открывшейся форме и нажмите «Отправить». В полях «Электронная почта», «Домашний телефон» и «Мобильный телефон» установлены требования к форматированию, применяемые в сквозном соединении HTML5 или в проверке бина.
   - Чтобы изменить студента, нажмите «Изменить» рядом с его именем, измените поля в открывшейся форме и нажмите «Отправить». Чтобы отредактировать другого студента, выберите его в раскрывающемся меню в верхней части страницы и нажмите «Изменить студента»

- Чтобы удалить студента, нажмите «Удалить» рядом с его именем, затем нажмите «Подтвердить» на появившейся странице. Это действие удаляет учащегося из сессии обучения, но не удаляет учащегося из базы данных. Чтобы снова добавить студента в сессию обучения, нажмите «Активировать студента» в левом меню, затем нажмите «Активировать» рядом с его именем на открывшейся странице.
- Чтобы добавить опекуна для студента, нажмите «Добавить опекуна» рядом с его именем. На появившейся странице отображаются имя студента, доступные опекуны и текущие опекуны для студента, если таковые имеются. Чтобы добавить существующего опекуна для этого ученика, выберите опекуна из списка и нажмите «Добавить опекуна». Чтобы создать нового опекуна для студента, заполните поля и нажмите «Отправить». Чтобы удалить опекуна из студента, выберите одного из текущих опекунов студента из списка и нажмите «Удалить опекуна».
- Чтобы добавить адрес учащегося, нажмите «Добавить адрес» рядом с его именем. На появившейся странице заполните соответствующие поля в появившейся форме и нажмите «Отправить». Четыре поля обязательны для заполнения.

Интерфейс администрирования реализован не полностью. Невозможно редактировать опекуна или просматривать или редактировать адрес, хотя страницы Facelets для этих функций существуют. Приложение также не использует свойства в сущности `PersonDetails`. Не стесняйтесь изменять приложение, чтобы добавить эти функции.

Глава 63. Пример Duke's Forest

В этой главе описывается Duke's Forest, простое приложение для электронной коммерции, которое содержит несколько веб-приложений и иллюстрирует использование нескольких API Jakarta EE.

Обзор примера Duke’s Forest

Duke's Forest — это простое приложение электронной коммерции, которое содержит несколько веб-приложений и иллюстрирует использование следующих API Jakarta EE:

- Jakarta Faces, включая Ajax
- Jakarta Contexts and Dependency Injection (CDI)
- Jakarta RESTful Web Service
- Персистентность Jakarta
- Jakarta Bean Validation
- Jakarta Enterprise Beans
- Jakarta Messaging

Приложение состоит из следующих проектов.

- Duke's Store: веб-приложение, в котором есть каталог товаров, самостоятельная регистрация клиентов и корзина покупок. Оно также имеет интерфейс администрирования для управления продуктами, категориями и пользователями. Название проекта: `dukes-store`.
- Duke's Shipment: веб-приложение, предоставляющее интерфейс для управления доставкой заказов. Название проекта — `dukes-shipment`.
- Duke's Payment: приложение веб-сервиса, которое содержит RESTful веб-сервис для оплаты заказа. Название проекта: `dukes-payment`.
- Duke’s Resources: проект Java-архива, содержащий все ресурсы, используемые веб-проектами. Он включает в себя сообщения, таблицы стилей CSS, изображения, файлы JavaScript и составные компоненты Jakarta Faces. Название проекта: `dukes-resources`.
- Entities: простой JAR-архив, содержащий все персистентные сущности Jakarta. Этот проект используется совместно с другими проектами, которые работают с сущностями. Имя проекта: `entity`.
- Events: проект, содержащий класс POJO, использующийся в качестве события CDI. Название проекта: `events`.

Дизайн и архитектура Duke's Forest

Duke's Forest — это сложное приложение, состоящее из трёх основных проектов и трёх подпроектов. Рисунок 63-1 показывает архитектуру трёх основных проектов, которые будут развёрнуты: Duke’s Store, Duke’s Shipment и Duke’s Payment. В нём также показано, как Duke's Store использует проекты «Events» и «Entities».

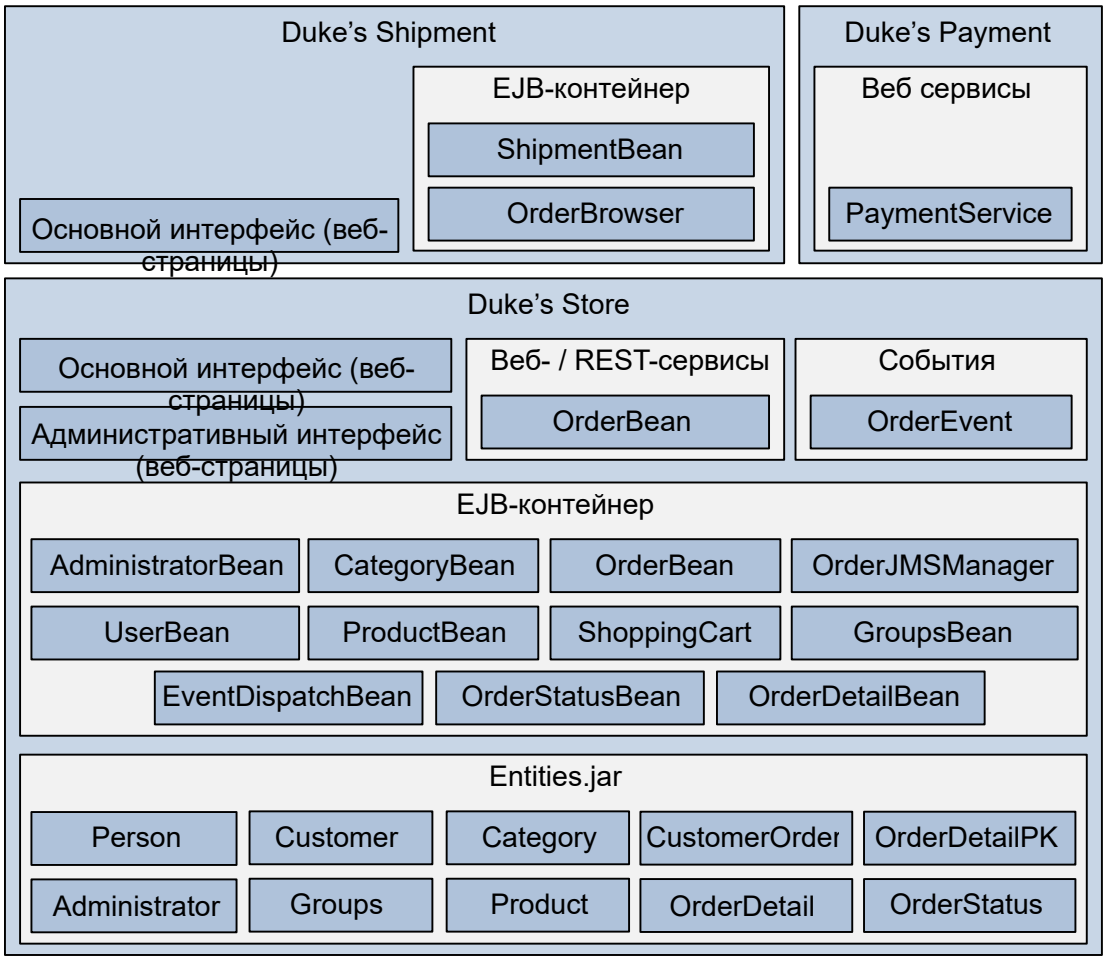

Рис. 63-1. Архитектура Duke's Forest.

В Duke's Forest используются следующие функции платформы Jakarta EE:

- Сущности Jakarta Persistence
  - Аннотации Bean Validation для валидации данных сущностей
  - Аннотации XML для сериализации XML Jakarta binding
- Веб-сервисы
  - Веб-сервис Jakarta REST для оплаты, с ограничениями безопасности
  - Веб-сервис Jakarta REST на основе Jakarta Enterprise Beans
- Enterprise-бины
  - Локальные сессионные бины
  - Все Enterprise-бины упакованы в WAR
- Jakarta Contexts and Dependency Injection (CDI)
  - CDI-аннотации для компонентов Jakarta Faces
  - Managed-бин CDI, используемый как корзина для покупок, с областью видимости диалога
  - Квалификаторы
  - События и обработчики событий
- Сервлеты
  - Сервлет для динамического представления изображений
- Jakarta Faces, использующая Facelets для веб-интерфейса

    - Шаблонизация
    - Составные компоненты
    - Загрузка файла
    - Ресурсы, упакованные в JAR-файл, чтобы их можно было найти в classpath
- Безопасность
    - Ограничения безопасности для бизнес-методов административного интерфейса (Enterprise-бины)
    - Ограничения безопасности для клиентов и администраторов (веб-компоненты)
    - Единый вход (SSO) для распространения аутентифицированного идентификатора пользователя из Duke's Store в Duke's Shipment

Приложение Duke's Forest имеет два основных пользовательских интерфейса, оба упакованы в WAR-файл Duke's Store:

- Основной интерфейс, для клиентов и гостей
- Административный интерфейс, используемый для выполнения служебных операций, таких как добавление новых элементов в каталог

Приложение Duke's Shipment также имеет пользовательский интерфейс, доступный для администраторов.

Рисунок 63-2 показывает, как веб-приложения взаимодействуют с веб-сервисом.

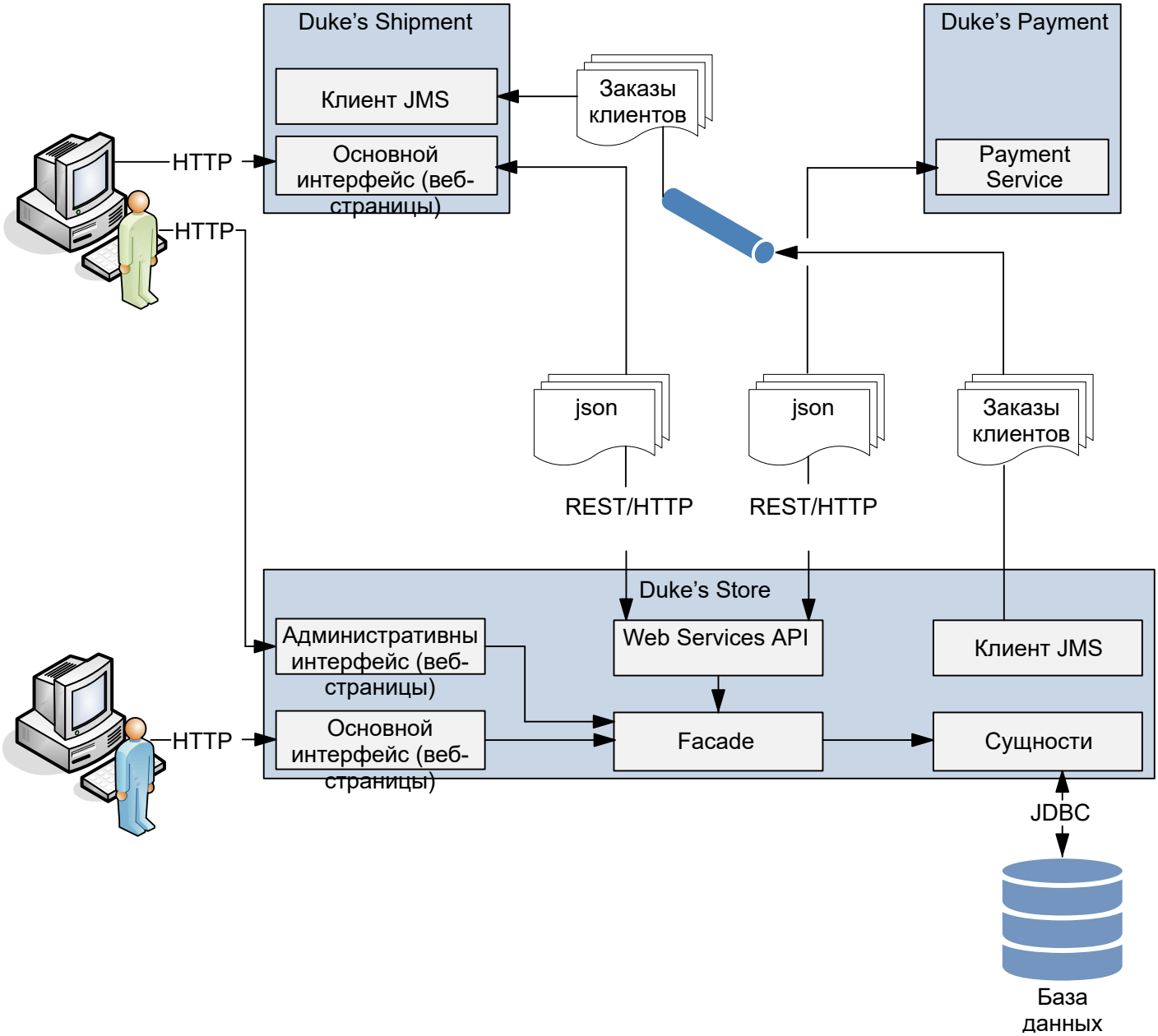

Рисунок 63-2 Взаимодействие между компонентами Duke's Forest

Как показано на рис. 63-2, клиент взаимодействует с основным интерфейсом Duke's Store, а администратор — с интерфейсом администрирования. Оба интерфейса получают доступ к фасаду, состоящему из Managed-бинов и сессионных бинов без сохранения состояния, которые, в свою очередь, взаимодействуют с объектами, представляющими таблицы базы данных. Фасад также взаимодействует с API веб-сервисов, которые получают доступ к веб-сервису Duke's Payment. После подтверждения оплаты заказа Duke's Store отправляет заказ в очередь JMS. Администратор также взаимодействует с интерфейсом Duke's Shipment, к которому можно получить доступ либо непосредственно через Duke's Shipment, либо из интерфейса администрирования Duke's Store с помощью веб-сервиса. Когда администратор утверждает отправку заказа, Duke's Shipment получает заказ из очереди JMS.

Фундаментальными строительными блоками приложения являются проекты «Events» и «Entities», которые объединяются в Duke's Store и Duke's Shipment вместе с проектом Duke's Resources.

Проект Events

События являются одними из ключевых компонентов Duke 's Forest. Проект `events` , включённый во все три основных проекта, является наиболее простым проектом приложения. У него есть только один класс, `OrderEvent` , но этот класс отвечает за большинство сообщений между объектами в приложении.

Приложение может отправлять сообщения на основе событий в различные компоненты и реагировать на них в зависимости от квалификации события. Приложение поддерживает следующие квалификаторы:

- `@LoggedIn` : для аутентифицированных пользователей
- `@New` : когда компонент корзины покупок создаёт новый заказ
- `@Paid` : когда заказ оплачен и готов к отправке

Следующий фрагмент кода из класса `PaymentHandler` в магазине Duke's Store показывает, как обрабатывается событие `@Paid` :

JAVA

```
@Inject @Paid Event<OrderEvent> eventManager;

...
public void onNewOrder(@Observes @New OrderEvent event) {

    if (processPayment(event)) {
        orderBean.setOrderStatus(event.getOrderID(),
                String.valueOf(OrderBean.Status.PENDING_PAYMENT.getStatus()));
        logger.info("Payment Approved");
        eventManager.fire(event);
    } else {
        orderBean.setOrderStatus(event.getOrderID(),
                String.valueOf(OrderBean.Status.CANCELLED_PAYMENT.getStatus()));
        logger.info("Payment Denied");
    }
}
```

Чтобы облегчить пользователям добавление событий в проект или добавление в классы событий большего количества полей для нового клиента, этот компонент является отдельным проектом в приложении.

Проект entities

Проект `entities` — это проект Jakarta Persistence, используемый как Duke’s Store, так и Duke’s Shipment. Он генерируется из схемы базы данных, показанной на Таблицы и связи между ними базы данных Duke’s Forest а также используется в качестве базы для организаций, потребляемых и производимых веб-сервисами с помощью Jakarta XML Binding. У каждой сущности есть правила проверки, основанные на бизнес-требованиях, заданных при помощи Jakarta Bean Validation

Таблицы базы данных Duke's Forest и отношения между ними

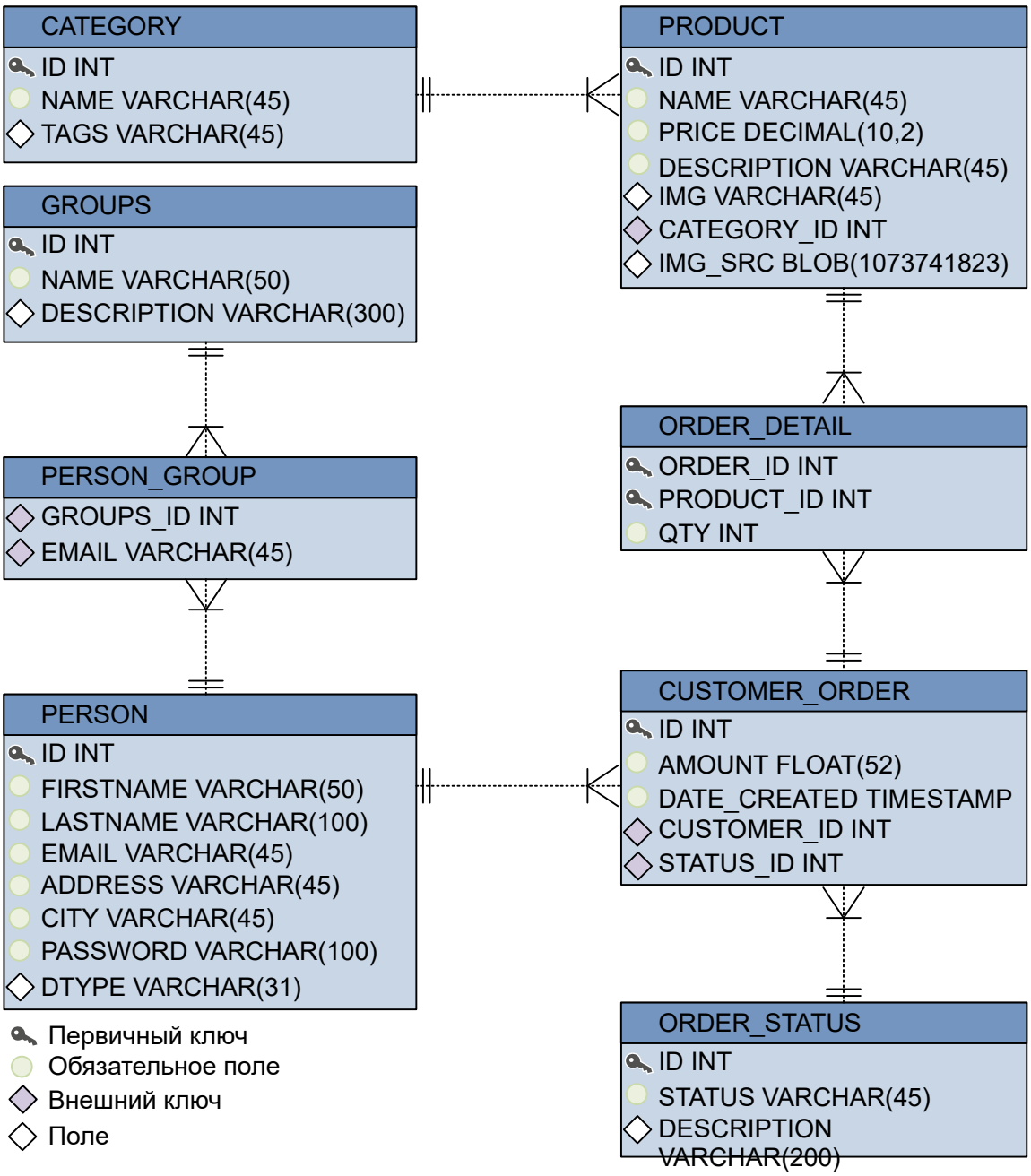

Схема базы данных содержит восемь таблиц:

- `PERSON` имеет отношение "один ко многим" с `PERSON_GROUPS` и `CUSTOMER_ORDER`
- `GROUPS` имеет отношение "один ко многим" с `PERSON_GROUPS`
- `PERSON_GROUPS` имеет отношение "многие к одному" с `PERSON` и `GROUPS` (промежуточная таблица между этими двумя таблицами)
- `PRODUCT` имеет отношение "многие к одному" с `CATEGORY` и отношение "один ко многим" с `ORDER_DETAIL`
- `CATEGORY` имеет отношение "один ко многим" с `PRODUCT`
- `CUSTOMER_ORDER` имеет отношение "один ко многим" с `ORDER_DETAIL` и отношение "многие к одному" с `PERSON` и `ORDER_STATUS`
- `ORDER_DETAIL` имеет отношение "многие к одному" с `PRODUCT` и `CUSTOMER_ORDER` (промежуточная таблица между этими двумя таблицами)
- `ORDER_STATUS` имеет отношение "один ко многим" с `CUSTOMER_ORDER`

Классы сущностей, которые соответствуют этим таблицам, следующие.

- `Person` определяет атрибуты, общие для клиентов и администраторов. Этими атрибутами являются имя человека и контактная информация, включая улицу и адреса электронной почты. Адрес электронной почты имеет аннотацию Bean Validation, чтобы гарантировать, что представленные данные правильно сформированы. Сгенерированная таблица для сущности `Person` также имеет поле `DTYPE`, которое представляет столбец дискриминатора. Его значение идентифицирует дочерний класс (`Customer` или `Administrator`), к которому принадлежит человек.
- `Customer`, дочерний для `Person` с определённым полем для объектов `CustomerOrder`.
- `Administrator`, дочерний для `Person` с полями для привилегий администратора.
- `Groups`, представляющая группу (`USERS` или `ADMINS`), к которой принадлежит пользователь.
- `Product`, который определяет атрибуты для продуктов. Эти атрибуты включают имя, цену, описание, ассоциированное изображение и категорию.
- `Category`, которая определяет атрибуты для категорий товаров. Эти атрибуты включают в себя имя и набор тегов.
- `CustomerOrder`, который определяет атрибуты для заказов, размещаемых клиентами. Эти атрибуты включают сумму и дату, а также значения идентификатора для клиента и детали заказа.
- `OrderDetail`, который определяет атрибуты для деталей заказа. Эти атрибуты включают значения количества и идентификатора для продукта и клиента.
- `OrderStatus`, который определяет атрибут статуса для каждого заказа.

Проект dukes-payment

Проект `dukes-payment` — это веб-проект, содержащий простой веб-сервис оплаты. Поскольку это пример приложения, оно не получает никакой реальной кредитной информации или даже статуса клиента для подтверждения платежа. На данный момент единственное правило, наложенное платёжной системой, — это запрещать все заказы на сумму свыше 1000 долларов США. Это приложение иллюстрирует общий сценарий, когда сторонняя платёжная система используется для проверки кредитных карт или банковских платежей.

Для идентификации клиента в веб-сервисе Jakarta REST в проекте используются HTTP Basic Authentication and JAAS (Java Authentication and Authorization Service). Сама реализация предоставляет простой метод `processPayment`, который получает `OrderEvent` для оценки и утверждения или отклонения оплаты заказа. Метод вызывается из процесса оформления покупки в Duke's Store.

Проект dukes-resources

Проект `dukes-resources` содержит несколько файлов, используемых как Duke's Store, так и Duke's Shipment, связанных в JAR-файл, размещённый в classpath. Ресурсы находятся в каталоге `src/main/resources`:

- `META-INF/resources/css`: две таблицы стилей, `default.css` и `jsfcrud.css`
- `META-INF/resources/img`: изображения, используемые в проектах
- `META-INF/resources/js`: файл JavaScript `util.js`
- `META-INF/resources/util`: составные компоненты, используемые в проектах
- `bundles/Bundle.properties`: сообщения приложений на английском языке
- `bundles/Bundle_es.properties`: сообщения приложений на испанском языке
- `ValidationMessages.properties`: сообщения Bean Validation на английском языке
- `ValidationMessages_es.properties`: сообщения Bean Validation на испанском языке

Проект Duke’s Store

Duke’s Store, веб-приложение, является основным приложением Duke's Forest. Оно отвечает за интерфейс основного магазина для клиентов, а также за интерфейс администрирования.

Основной интерфейс Duke's Store позволяет пользователю выполнять следующие задачи:

- Просмотр каталога продукции
- Регистрация в качестве нового клиента
- Добавление товаров в корзину
- Проверка
- Просмотр статуса заказа

Интерфейс администрирования Duke's Store позволяет администраторам выполнять следующие задачи:

- Обслуживание продукта (создание, редактирование, обновление, удаление)
- Обслуживание категории (создание, редактирование, обновление, удаление)
- Обслуживание клиентов (создание, редактирование, обновление, удаление)
- Обслуживание групп (создание, редактирование, обновление, удаление)

Проект также использует сессионные компоненты без сохранения состояния в качестве фасадов для взаимодействия с сущностями Jakarta Persistence, описанными в Проект entities, и Managed-бины CDI в качестве контроллеров для взаимодействия со страницами Facelets. Таким образом, проект следует шаблону MVC (Model-View-Controller) и применяет тот же шаблон ко всем сущностям и страницам, как в следующем примере.

- `AbstractFacade` — это абстрактный класс, который получает `Type<T>` и реализует общие операции (CRUD) для этого типа, где `<T>` — персистентная сущность.
- `ProductBean` — это сессионный компонент без сохранения состояния, который расширяет `AbstractFacade`, применяя `Product` как `Type<T>` и инъецирует `PersistenceContext` для `EntityManager`. Этот бин реализует все пользовательские методы, необходимые для взаимодействия с сущностью `Product` или для вызова пользовательского запроса.
- `ProductController` — это Managed-бин CDI, который взаимодействует с необходимыми Enterprise-бинами и страницами Facelets для управления отображением данных.

`ProductBean` начинается следующим образом:

JAVA

```
@Stateless
public class ProductBean extends AbstractFacade<Product> {
    private static final Logger logger =
        Logger.getLogger(ProductBean.class.getCanonicalName());

    @PersistenceContext(unitName="forestPU")
    private EntityManager em;

    @Override
    protected EntityManager getEntityManager() {
        return em;
    }
    ...
}
```

Enterprise-бины, используемые в Duke's Store

Enterprise-бины, используемые в Duke's Store, обеспечивают бизнес-логику для приложения и находятся в пакете `com.forest.ejb`. Все сессионные компоненты без состояния.

`AbstractFacade` — это не Enterprise-бин, а абстрактный класс, реализующий общие операции для `Type<T>`, где `<T>` — это персистентная сущность.

Большинство других бинов расширяют `AbstractFacade`, инъецируют `PersistenceContext` и реализуют все необходимые пользовательские методы:

- `AdministratorBean`
- `CategoryBean`
- `EventDispatcherBean`
- `GroupsBean`
- `OrderBean`
- `OrderDetailBean`
- `OrderJMSManager`
- `OrderStatusBean`
- `ProductBean`
- `ShoppingCart`
- `UserBean`

Класс `ShoppingCart`, хотя он находится в пакете `ejb`, является Managed-бином CDI с областью видимости диалога, что означает, что информация запроса будет сохраняться в нескольких запросах. Кроме того, `ShoppingCart` отвечает за запуск цепочки событий для заказов клиентов, которая вызывает RESTful веб-сервис в `dukes-payment` и публикует заказ в очереди Jakarta Messaging для подтверждения доставки, если оплата прошла успешно.

Файлы Facelets, используемые в основном интерфейсе Duke's Store

Как и в других примерах, в Duke's Store для отображения пользовательского интерфейса используются Facelets. Основной интерфейс использует большое количество страниц Facelets для отображения. Страницы сгруппированы в каталоги в зависимости от того, какой модуль они обрабатывают.

- `template.xhtml`: файл шаблона, используемый как для основного, так и для административного интерфейсов. Сначала выполняется проверка браузера, чтобы убедиться, что браузер пользователя поддерживает HTML 5, что необходимо для Duke's Forest. Он разделяет экран на несколько областей и определяет страницу клиента для каждой области.
- `topbar.xhtml`: страница для области входа в систему в верхней части экрана.
- `top.xhtml`: страница для области заголовка.
- `left.xhtml`: страница для левой боковой панели.
- `index.xhtml`: страница для содержимого основного экрана.
- `login.xhtml`: страница входа указана в `web.xml`. Основной интерфейс входа в систему представлен в `topbar.xhtml`, но эта страница появляется в случае ошибки входа.

- Каталог `admin` : страницы, связанные с интерфейсом администрирования, описанным в Файлы Facelets, используемые в интерфейсе администрирования Duke's Store.
- Каталог `customer` : страницы, связанные с клиентами (`Create.xhtml` , `Edit.xhtml` , `List.xhtml` , `Profile.xhtml` , `View.xhtml`).
- Каталог `order` : страницы, связанные с заказами (`Create.xhtml` , `List.xhtml` , `MyOrders.xhtml` , `View.xhtml`).
- Каталог `orderDetail` : всплывающая страница, позволяющая пользователям просматривать детали заказа (`View_popup.xhtml`).
- Каталог `product` : страницы, связанные с продукцией (`List.xhtml` , `ListCategory.xhtml` , `View.xhtml`).

Файлы Facelets, используемые в интерфейсе администрирования Duke's Store

Страницы Facelets для интерфейса администрирования Duke's Store находятся в каталоге `web/admin` :

- Каталог `administrator` : страницы, связанные с управлением администраторами (`Create.xhtml` , `Edit.xhtml` , `List.xhtml` , `View.xhtml`)
- Каталог `category` : страницы, связанные с управлением категориями товаров (`Create.xhtml` , `Edit.xhtml` , `List.xhtml` , `View.xhtml`)
- Каталог `customer` : страницы, связанные с управлением клиентами (`Create.xhtml` , `Edit.xhtml` , `List.xhtml` , `Profile.xhtml` , `View.xhtml`)
- Каталог `groups` : страницы, связанные с управлением группами (`Create.xhtml` , `Edit.xhtml` , `List.xhtml` , `View.xhtml`)
- Каталог `order` : страницы, связанные с управлением заказами (`Create.xhtml` , `Edit.xhtml` , `List.xhtml` , `View.xhtml`)
- Каталог `orderDetail` : всплывающая страница, позволяющая администратору просматривать детали заказа (`View_popup.xhtml`)
- Каталог `product` : страницы, связанные с управлением продукцией (`Confirm.xhtml` , `Create.xhtml` , `Edit.xhtml` , `List.xhtml` , `View.xhtml`)

Managed-бины, используемые в Duke's Store

В Duke's Store используются следующие Managed-бины CDI, соответствующие Enterprise-бинам. Бины находятся в пакете `com.forest.web` :

- `AdministratorController`
- `CategoryController`
- `CustomerController`
- `CustomerOrderController`
- `GroupsController`
- `OrderDetailController`
- `OrderStatusController`
- `ProductController`
- `UserController`

Вспомогательные классы, используемые в Duke's Store

Managed-бины CDI в главном интерфейсе Duke's Store используют следующие вспомогательные классы, которые находятся в пакете `com.forest.web.util`:

- `AbstractPaginationHelper`: абстрактный класс с методами, используемыми Managed-бинами
- `ImageServlet`: класс сервлета, который извлекает содержимое изображения из базы данных и отображает его
- `JsfUtil`: класс, используемый для операций Jakarta Faces, таких как сообщения очереди в объекте `FacesContext`
- `MD5Util`: класс, используемый Managed-бином `CustomerController` для генерации зашифрованного пароля пользователя

Квалификаторы, используемые в Duke's Store

Duke's Store определяет следующие квалификаторы в пакете `com.forest.qualifiers`:

- `@LoggedIn`: квалифицирует пользователя как вошедшего в систему
- `@New`: квалифицирует заказ как новый
- `@Paid`: квалифицирует заказ как оплаченный

Обработчики событий, используемые в Duke's Store

В Duke's Store определены обработчики событий, связанные с классом `OrderEvent`, упакованным в проекте `events` (см. Проект events). Обработчики событий находятся в пакете `com.forest.handlers`.

- `IOrderHandler`: интерфейс `IOrderHandler` определяет метод `onNewOrder`, реализованный двумя классами-обработчиками.
- `PaymentHandler`: компонент `ShoppingCart` запускает `OrderEvent`, квалифицированный как `@New`. Метод `onNewOrder` в `PaymentHandler` наблюдает за этими событиями и, когда он их перехватывает, обрабатывает платёж с помощью веб-сервиса Duke's Payment. После успешного ответа от веб-сервиса `PaymentHandler` снова запускает `OrderEvent`, на этот раз квалифицированный как `@Paid`.
- `DeliveryHandler`: метод `onNewOrder` для `DeliveryHandler` наблюдает за объектами `OrderEvent`, квалифицированными как `@Paid` (заказы оплачены и готовы к доставке) и изменяет статус заказа на `PENDING_SHIPMENT`. Когда администратор получает доступ к Duke's Shipment, он вызывает сервис заказов, RESTful веб-сервис, и запрашивает все заказы в базе данных, которые готовы к доставке.

Дескрипторы развёртывания, используемые в Duke's Store

В Duke's Store используются следующие дескрипторы развёртывания, расположенные в каталоге `web/WEB-INF`:

- `faces-config.xml`: файл конфигурации Jakarta Faces
- `glassfish-web.xml`: файл конфигурации, специфичный для GlassFish Server
- `web.xml`: файл конфигурации веб-приложения

Проект Duke's Shipment

Duke's Shipment — это веб-приложение со страницей входа, главной страницей Facelets и некоторыми другими объектами. Это приложение, которое доступно только администраторам, получает заказы из очереди Jakarta Messaging и вызывает RESTful веб-сервис, предоставляемый Duke's Store, для обновления статуса заказа. На главной странице Duke's Shipment отображается список заказов, ожидающих отправки, и

список отправленных заказов. Администратор может утверждать или отклонять заказы на доставку. В случае одобрения заказ доставляется и отображается под заголовком «Отправлено». В случае отклонения заказ исчезает со страницы, а в списке заказов клиента он отображается как отменённый.

В списке «Ожидание» также есть значок шестерёнки, который выполняет Ajax-вызов в службу заказов, чтобы обновить список без обновления страницы. Код выглядит так:

```xml
<h:commandLink>
    <h:graphicImage library="img" title="Check for new orders"
                    style="border:0px" name="refresh.png"/>
    <f:ajax execute="@form" render="@form" />
</h:commandLink>
```

Enterprise-бины, используемые в Duke's Shipment

`UserBean` сессионный компонент без сохранения состояния, используемый в Duke's Shipment, обеспечивает бизнес-логику для приложения и находится в пакете `com.forest.shipment.session`.

Как и Duke's Store, в Duke's Shipment используется класс `AbstractFacade`. Этот класс является не EJB-компонентом, а абстрактным классом, который реализует общие операции для `Type<T>`, где `<T>` является сущностью Jakarta Persistence.

`OrderBrowser` сессионный компонент без сохранения состояния, расположенный в пакете `com.forest.shipment.ejb`, имеет один метод, который просматривает очередь Jakarta Messaging заказов, и другой, который использует сообщение заказа после того как администратор утверждает или отклоняет заказ на отправку.

Файлы Facelets, используемые в Duke's Shipment

В Duke's Shipment есть только одна страница, поэтому в ней больше файлов Facelets, чем в Duke's Store.

- `template.xhtml`: файл шаблона, как и файл в Duke's Store, сначала выполняет проверку браузера, чтобы убедиться, что браузер пользователя поддерживает HTML 5, что требуется для Duke's Forest. Он делит экран на области и определяет страницу клиента для каждой области.
- `topbar.xhtml`: страница для области входа в систему в верхней части экрана.
- `top.xhtml`: страница для области заголовка.
- `index.xhtml`: страница для начального содержимого основного экрана.
- `login.xhtml`: страница входа указана в `web.xml`. Основной интерфейс входа в систему представлен в `topbar.xhtml`, но эта страница появляется в случае ошибки входа.
- `admin/index.xhtml`: страница для содержимого основного экрана после аутентификации.

Managed-бины, используемые в Duke's Shipment

Duke's Shipment использует следующие Managed-бины CDI в пакете `com.forest.shipment`:

- `web.ShippingBean`: Managed-бин, действующий в качестве клиента для сервиса заказов
- `web.UserController`: Managed-бин, соответствующий сессионному компоненту `UserBean`

Класс помощника, используемый в Duke's Shipment

Managed-бины Duke's Shipment используют только один вспомогательный класс, который находится в пакете `com.forest.shipment.web.util`:

- `JsfUtil` : класс, используемый для операций Jakarta Faces, таких как сообщения очереди в объекте `FacesContext`

Квалификатор, используемый в Duke's Shipment

Duke's Shipment включает в себя квалификатор `@LoggedIn` , описанный в Квалификаторы, используемые в Duke's Store.

Дескрипторы развёртывания, используемые в Duke's Shipment

Duke's Shipment использует следующие дескрипторы развёртывания:

- `faces-config.xml` : файл конфигурации Jakarta Faces
- `glassfish-web.xml` : файл конфигурации, специфичный для GlassFish Server
- `web.xml` : файл конфигурации веб-приложения

Сборка и развёртывание Duke's Forest

Вы можете использовать IDE NetBeans или Maven для сборки и развёртывания Duke's Forest.

Сборка и развёртывание Duke's Forest с IDE NetBeans

1. Убедитесь, что GlassFish Server (см. Запуск и остановка GlassFish Server), а также сервер базы данных (см. Запуск и остановка Apache Derby) запущены.
2. В меню «Файл» выберите «Открыть проект».
3. В диалоговом окне «Открыть проект» перейдите к:

   ```
   tut-install/examples/case-studies
   ```

4. Выберите каталог `dukes-forest` .
5. Установите флажок «Открыть требуемые проекты» и нажмите «Открыть проект».
6. Кликните правой кнопкой мыши каталог `dukes-forest` и выберите Сборка.

   Эта задача настраивает сервер, создаёт и заполняет базу данных, создаёт все подпроекты, упаковывает их в файлы JAR и WAR и развёртывает приложения `dukes-payment` , `dukes-store,` и `dukes-shipment` .

   Для настройки сервера эта задача создаёт область безопасности JDBC с именем `jdbcRealm` , включает отображение принципала на роли по умолчанию и включает единый вход (SSO) для службы HTTP.

Сборка и развёртывание Duke's Forest с помощью Maven

1. Убедитесь, что GlassFish Server (см. Запуск и остановка GlassFish Server), а также сервер базы данных (см. Запуск и остановка Apache Derby) запущены.
2. В окне терминала перейдите в:

   ```
   tut-install/examples/case-studies/dukes-forest/
   ```

3. Введите следующую команду, чтобы настроить сервер, создать и заполнить базу данных, собрать все подпроекты, упаковать их в файлы JAR и WAR и развернуть `dukes-payment` , `dukes-store,` и `dukes-shipment` :

   SHELL

   ```
   mvn install
   ```

Для настройки сервера эта задача создаёт область безопасности JDBC с именем `jdbcRealm`, включает отображение принципала на роли по умолчанию и включает единый вход (SSO) для службы HTTP.

Запуск Duke's Forest

Запуск приложения Duke's Forest включает в себя несколько задач:

- Регистрация клиента в Duke’s Store
- Покупка продукции клиентом
- Одобрение или отклонение отправки продукции администратором
- Создание новой продукции, клиента, группы или категории администратором

Регистрация клиента в Duke’s Store

1. В веб-браузере введите следующий URL:

   ```
   http://localhost:8080/dukes-store
   ```

   Откроется страница магазина Duke's Forest.
2. Нажмите Зарегистрироваться вверху страницы.
3. Заполните поля формы, затем нажмите Сохранить.

   Все поля обязательны для заполнения, а длина пароля должно быть не менее 7 символов.

Покупка продукции

1. Чтобы войти в систему как созданный вами пользователь или как один из двух пользователей, уже находящихся в базе данных, введите имя пользователя и пароль и нажмите «Войти».

   Существующие пользователи имеют имена пользователей `jack@example.com` и `robert@example.com`, и у них обоих одинаковый пароль `1234`.
2. Нажмите Продукция в левой боковой панели.
3. На открывшейся странице выберите одну из категорий («Растения», «Еда», «Услуги» или «Инструменты»).
4. Выберите продукт и нажмите «Добавить в корзину».

   Вы можете заказать один или несколько продуктов в нескольких категориях. Продукты и промежуточные суммы отображаются в корзине на левой боковой панели.
5. Когда вы закончите выбирать товары, нажмите «Оформить заказ».

   Появится сообщение: «Ваш заказ обрабатывается. Для просмотра статуса заказа перейдите на страницу Заказы.»
6. Нажмите Заказы в левой боковой панели, чтобы подтвердить свой заказ.

   Если общая сумма заказа превышает 1000 долларов США, статус заказа «Заказ отменён», поскольку веб-сервис оплаты отклоняет заказы сверх этого лимита. В противном случае статус «Готов к отправке».
7. Когда вы закончите размещать заказы, нажмите Выход из системы в верхней части страницы.

Утверждение отгрузки продукции

1. Войдите в Duke's Store как администратор.

   Имя пользователя `admin@example.com`, пароль `1234`.

Главная страница администрирования позволяет просматривать категории, клиентов, администраторов, группы, продукты и заказы, а также создавать новые объекты всех типов, кроме заказов.

2. В нижней части страницы нажмите «Подтвердить отправку».

   Это действие перенаправляет вас в Duke's Shipment, сохраняя ваш логин администратора.

3. В списке «Ожидание» нажмите «Утвердить», чтобы утвердить заказ и переместить его в область «Отправлено» на странице.

   Если вы нажмёте Отклонить, заказ исчезнет со страницы. Если вы снова войдёте в Duke's Store в качестве клиента, в списке «Заказы» появится сообщение «Заказ отменён».

Чтобы вернуться в Duke's Store из Duke's Shipment, нажмите «Вернуться в Duke's Store».

Создание нового продукта

Вы можете создавать другие виды объектов, а также продукты. Создание продуктов более сложное, чем другие процессы создания, поэтому оно описано здесь.

1. Войдите в Duke's Store как администратор.
2. На главной странице администрирования нажмите «Создать новый продукт».
3. Введите значения в поля «Имя», «Цена» и «Описание».
4. Выберите категорию, затем нажмите «Далее».
5. На странице «Загрузка изображения продукта» нажмите «Обзор», чтобы найти изображение в файловой системе с помощью средства выбора файлов.
6. Нажмите кнопку Далее.
7. На следующей странице просмотрите поля продукта, затем нажмите «Готово».
8. Нажмите «Продукты» на левой боковой панели, затем кликните категорию, чтобы убедиться, что продукт был добавлен.
9. Нажмите «Администрирование» в верхней части страницы, чтобы вернуться на главную страницу администрирования, или нажмите «Выйти», чтобы выйти из системы.

Release 9.1
Последнее обновление: 2021-12-27