



Java EE 7 и сервер приложений GlassFish 4

Дэвид Хеффельфингер



Дэвид Хеффельфингер

Java EE 7 и сервер приложений GlassFish 4

Java EE 7 with GlassFish 4 Application Server

A practical guide to install and configure the GlassFish 4 application server and develop Java EE 7 applications to be deployed to this server

David R. Heffelfinger



BIRMINGHAM - MUMBAI

Java EE 7 и сервер приложений GlassFish 4

Практическое руководство по установке
и настройке сервера приложений GlassFish 4,
а также по разработке приложений Java EE 7
и их развертыванию на этом сервере

Дэвид Хеффельфингер



Москва, 2016

УДК 004.438Java ЕЕ

ББК 32.973.26-018.2

X41

X41 Дэвид Хеффельфингер

Java EE 7 и сервер приложений GlassFish 4 / пер. с англ. Киселев А. Н. – М.: ДМК Пресс, 2016. – 332 с.: ил.

ISBN 978-5-97060-332-1

Книга представляет собой практическое руководство с очень удобным подходом, позволяющим читателю быстрее освоить технологии Java EE 7. Все рассмотренные основные интерфейсы Java EE 7 и подробная информация о сервере GlassFish 4 подкреплены практическими примерами их использования.

Платформа Java Enterprise Edition (Java EE) 7 является отраслевым стандартом для корпоративных вычислений Java, а сервер приложений GlassFish представляет собой эталонную реализацию спецификации Java EE. В книге рассматриваются различные соглашения и аннотации Java EE 7, которые помогут существенно упростить разработку корпоративных приложений Java. Описываются последние версии технологий Servlet, JSP, JSF, JPA, EJB и JAX-WS, а также новые дополнения к спецификации Java EE, в частности JAX-RS и CDI. Рассмотрены задачи администрирования, конфигурирования и использования сервера GlassFish 4 для развертывания корпоративных приложений.

Настоящее издание предназначено для разработчиков Java, желающих стать специалистами в разработке корпоративных приложений с использованием платформы Java EE 7. Для изучения материала необходимо иметь некоторый опыт работы с Java, однако знаний в области Java EE или J2EE не требуется.

Книга официально рекомендуется компанией Oracle – разработчиком перечисленных технологий – в качестве учебного пособия.

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-78217-688-6 (англ.)
ISBN 978-5-97060-321-1 (рус.)

Copyright © 2014 Packt Publishing
© Оформление, перевод на русский язык,
ДМК Пресс, 2016



ОГЛАВЛЕНИЕ

Об авторе **10**

О рецензентах..... **11**

Предисловие **13**

Темы, освещаемые в книге	13
Что нужно для чтения этой книги	15
Кому адресована эта книга	15
Соглашения	15
Отзывы и пожелания	16
Загрузка исходного кода примеров	17
Список опечаток	17
Нарушение авторских прав	17
Вопросы	17

Глава 1.

Знакомство с сервером GlassFish..... **18**

Общий обзор Java EE и GlassFish.....	18
Новые возможности Java EE 7	19
Преимущества GlassFish.....	21
Получение GlassFish	22
Установка GlassFish	23
Зависимости GlassFish.....	24
Установка	24
Запуск GlassFish	25
Развертывание первого приложения Java EE	26
Домены GlassFish	33
Создание доменов	34
Удаление доменов	35
Остановка домена	36
Настройка подключения к базе данных.....	36
Создание пулов соединений	37
Создание источников данных.....	40
Резюме	40

Глава 2.

JavaServer Faces **42**

Введение в JSF.....	42
---------------------	----

Файлы	42
Необязательный файл faces-config.xml	43
Разработка первого JSF-приложения.....	44
Файлы	45
Этапы проекта	50
Проверка допустимости.....	53
Группировка компонентов.....	55
Отправка формы.....	55
Именованные компоненты.....	56
Навигация.....	58
Пользовательская проверка допустимости данных	60
Создание нестандартных валидаторов	60
Методы валидатора	63
Настройка сообщений JSF по умолчанию.....	66
Настройка стилей сообщения	67
Изменение текста сообщения.....	69
Поддержка Ajax в JSF-приложениях.....	71
Поддержка HTML5 в JSF 2.2	76
HTML5-совместимая разметка	76
Сквозные элементы	78
JSF 2.2 Faces Flows.....	80
Библиотеки дополнительных компонентов JSF	84
Резюме	84
Глава 3.	
Объектно-реляционное отображение в JPA	85
База данных CUSTOMERDB	85
Введение в Java Persistence API.....	87
Отношения между сущностями	92
Составные первичные ключи	110
Введение в язык запросов JPA.....	115
Введение в Criteria API	119
Поддержка проверки допустимости на стороне компонентов	126
Заключительные замечания	128
Резюме	129
Глава 4.	
Enterprise JavaBeans	130
Сеансовые компоненты.....	131
Простой сеансовый компонент	131
Более реалистический пример	135
Вызов сеансовых компонентов в веб-приложении.....	137
Сеансовые компоненты-одиночки (Singleton)	139
Асинхронные вызовы методов	140

Компоненты, Управляемые сообщениями	143
Транзакции в Enterprise JavaBeans	144
Транзакции, управляемые контейнером.....	145
Транзакции, управляемые компонентом	148
Жизненные циклы компонентов Enterprise JavaBeans.....	150
Жизненный цикл сеансового компонента с сохранением состояния ...	151
Жизненный цикл сеансового компонента без сохранения состояния ...	154
Жизненный цикл компонентов, управляемых сообщениями.....	156
Служба таймеров EJB.....	156
Таймеры EJB на основе календаря.....	160
Безопасность EJB	162
Аутентификация клиента.....	165
Резюме	167

Глава 5.

Контексты и внедрение зависимостей	169
Именованные компоненты	169
Внедрение зависимостей.....	171
Квалификаторы CDI.....	173
Контексты именованных компонентов	176
Резюме	184

Глава 6.

Обработка JSON с помощью JSON-P API	185
JSON-P Model API.....	186
Создание данных в формате JSON с использованием Model API	186
Парсинг данных в формате JSON с использованием Model API	189
JSON-P Streaming API	191
Создание данных в формате JSON с использованием Streaming API ...	191
Парсинг данных в формате JSON с использованием Streaming API	193
Резюме	196

Глава 7.

Веб-сокеты	197
Создание серверных конечных точек веб-сокетов	197
Создание серверной конечной точки веб-сокета с применением аннотаций.....	198
Создание клиентов веб-сокетов.....	200
Создание клиентов веб-сокетов на JavaScript	201
Создание клиентов веб-сокетов на Java.....	204
Дополнительная информация о Java API для веб-сокетов	208
Резюме	208

Глава 8.**Служба обмена сообщениями Java 210**

Настройка GlassFish для использования JMS	210
Настройка фабрики JMS-соединений	211
Создание очереди JMS-сообщений.....	213
Создание темы JMS-сообщений	214
Очереди сообщений.....	216
Отправка сообщений в очередь	216
Извлечение сообщений из очереди	219
Асинхронный прием сообщений из очереди	221
Просмотр очередей сообщений.....	224
Темы сообщений	225
Отправка сообщений в тему	225
Получение сообщений из темы	226
Создание долговременной подписки.....	228
Резюме	231

Глава 9.**Безопасность приложений Java EE 232**

Области безопасности	232
Предопределенные области безопасности.....	233
Стандартная аутентификация через область файла	238
Аутентификация на основе формы	248
Создание самоподписанных сертификатов.....	253
Настройка приложений для использования области сертификата	258
Определение дополнительных областей.....	262
Резюме	278

Глава 10.**Веб-службы JAX-WS 279**

Разработка веб-служб с использованием JAX-WS API.....	279
Создание клиента веб-службы.....	284
Отправка вложений веб-службам.....	291
Экспортирование компонентов EJB в виде веб-служб.....	294
Клиенты веб-служб EJB	295
Безопасность веб-служб	296
Безопасность веб-служб EJB	298
Резюме	300

Глава 11.**Веб-службы RESTful JAX-RS 301**

Введение в веб-службы RESTful и JAX-RS.....	301
Создание простой веб-службы RESTful.....	302
Настройка пути к ресурсам REST в приложении	305
Тестирование веб-службы	306
Преобразование данных в/из XML с помощью JAXB.....	308
Создание клиента веб-службы RESTful.....	312
Параметры запроса и пути	313
Параметры запроса	314
Отправка параметров запроса через клиентский JAX-RS API	316
Параметры пути	316
Отправка параметров пути через клиентский JAX-RS API.....	318
Резюме	320
Предметный указатель	321



ОБ АВТОРЕ

Дэвид Хеффельфингер (David R. Heffelfinger) является техническим директором Ensoode Technology LLC – консалтинговой компании, специализирующейся на разработке программного обеспечения, расположенной в районе большого Вашингтона, округ Колумбия. Дэвид – профессиональный архитектор, проектировщик и разработчик программного обеспечения с 1995 года. Использует Java в качестве основного языка программирования с 1996 года. Ему довелось работать во многих крупных проектах для ряда клиентов, в числе которых департамент США по Национальной безопасности, Freddie Mac, Fannie Mae и Министерство обороны США. Дэвид имеет степень магистра в области разработки программного обеспечения Южного методического университета. Также является главным редактором Ensoode.net (<http://www.ensoode.net>), веб-сайта, посвященного Java, Linux и другим технологиям. Часто выступает на конференциях Java-разработчиков, таких как JavaOne. Вы можете следовать за Дэвидом в Твиттере, его учетная запись: [@ensoode](#).



О РЕЦЕНЗЕНТАХ

Штефан Хорошовец (Stefan Horochovec) из Бразилии. Имеет учёную степень в области проектирования программного обеспечения и в настоящее время работает как программный архитектор.

Последние 10 лет занимался разработкой корпоративных приложений на Java, с использованием серверов приложений, таких как GlassFish, JBoss, Weblogic и WildFly.

Имеет 4-летний опыт работы с такими технологиями создания пользовательских интерфейсов, как Apache Flex (три года подряд выступал с докладами на конференции FlexMania – небывалое событие в Apache Flex в Латинской Америке), Struts и JSF. В настоящее время занимается проектами на основе фреймворка JSF 2 и JavaScript, сильным уклоном в сторону AngularJS.

Имеет опыт работы с мобильными технологиями в течение 6 лет, хорошо знает платформу Android. Был одним из первых преподавателей по платформе Android в Бразилии выступал с докладами на конференции, посвященной Android, в Бразилии. Два года занимался разработкой корпоративных веб-приложений на основе HTML для мобильных устройств, используя такие фреймворки, как PhoneGap.

В 2014 Штефана пригласили присоединиться к программе «BlackBerry Elite Member», объединившей около 100 специалистов по всему миру, осознающих важность развития мобильных технологий и использующих операционную систему BlackBerry на одноименных устройствах в качестве мобильной платформы.

Штефан преподает в университете дисциплины, связанные с разработкой мобильных и веб-приложений, а также ведет курсы на предприятиях, где преподает Java, HTML/JS/CSS3, PhoneGap, Git и серверы приложений для Java.

Тим Пинет (Tim Pinet) – практикующий программист и веб-разработчик. В настоящее время живет в Оттаве, Канада. С раннего возраста начал интересоваться всем, что связано с электроникой и в

свое время поступил на факультет проектирования и разработки программного обеспечения. Оттава – крупный столичный город с развитым промышленным производством, предоставляющим богатые возможности, поэтому Тим получил отличный шанс попрактиковаться в разработке программного обеспечения в частных (Computer Associates, Emergis, Telus, Nortel) и общественных компаниях (City of Ottawa) в самых разных областях, таких как транспорт, здравоохранение, связь и муниципальные услуги.

Как приверженец идеи открытого программного обеспечения, Тим готов работать за невысокую плату, но с большой отдачей и в любых окружениях. Часто вносит свой вклад в развитие открытых проектов (таких как Apache и SourceForge) и делится своими знаниями в сообществах (таких как Stackoverflow и его персональный блог). В своей работе широко использует инструменты с открытым исходным кодом, чем экономит своим работодателям тысячи долларов, привносит передовые приемы ускоренной разработки и тестирования, не требующие финансовых затрат и не влекущих потерю качества.

Обожая все, что связано с программированием и Интернетом, Тим постоянно удовлетворяет свои желания использовать новейшие технологии для повышения качества обслуживания клиентов. Он обладает обширным опытом практического применения корпоративных технологий Java и веб-служб, разработки пользовательских интерфейсов и серверных компонентов приложений, управления базами данных и интеграции служб SOAP. Он очень ценный командный игрок и лучше всего проявляет себя в руководстве коллективами и принятии архитектурных решений.

Чираг Сангани (Chirag Sangani) – программист, живет в пригороде Сиэтла. Получил степень магистра в Стэнфордском университете (штат Калифорния), и бакалавра – в Индийском технологическом институте, в городе Канпур (Индия). В настоящее время работает инженером-программистом в Microsoft.



ПРЕДИСЛОВИЕ

Java Enterprise Edition 7, последняя версия Java EE, добавила несколько новых особенностей в спецификацию. Некоторые существующие Java EE API претерпели значительные усовершенствования в этой версии спецификации; другие, совершенно новые API, были добавлены в спецификацию Java EE. Эта книга охватывает самые последние версии наиболее популярных спецификаций Java EE, включая JavaServer Faces (JSF), Java Persistence API (JPA), Enterprise JavaBeans (EJB), Contexts and Dependency Injection (CDI), новый Java API for JSON Processing (JSON-P), WebSocket, полностью обновленный Java Messaging Service (JMS) API 2.0, Java API for XML Web Services (JAX-WS) и Java API for RESTful Web Services (JAX-RS), а также приемы обеспечения безопасности в приложениях Java EE.

Сервер приложений GlassFish является эталонной реализацией для Java EE; это первый сервер приложений для Java EE на рынке, обеспечивший поддержку Java EE 7. Эта книга охватывает GlassFish 4.0, последнюю версию мощного сервера приложений с открытым исходным кодом.

Темы, освещаемые в книге

В главе 1, «Знакомство с сервером *GlassFish*», объясняется, как установить и настроить сервер GlassFish и как развертывать приложения Java EE через веб-консоль GlassFish. Наконец, здесь рассматриваются основные задачи администрирования GlassFish, такие как настройка доменов и соединений с базой данных, добавление пулов соединений и источников данных.

В главе 2, «*JavaServer Faces*», рассказывается о разработке приложений с использованием фреймворка JSF и его новых особенностей, включая поддержку HTML5-подобной разметки и потоков Faces Flows. Кроме того, здесь рассказывается о приемах проверки ввода пользователя с применением стандартных валидаторов JSF, а также о том, как создавать собственные валидаторы и определять методы, осуществляющие проверку.

Глава 3, «Объектно-реляционное отображение в JPA», обсуждает вопросы взаимодействий с системами управления реляционными базами данных (СУРБД), такими как Oracle или MySQL с применением Java Persistence API.

Глава 4, «Enterprise JavaBeans», объясняет, как разрабатывать приложения с использованием сеансовых компонентов и компонентов, управляемых сообщениями. Описывает основные особенности EJB, такие как поддержка управления транзакциями, служба таймеров EJB и безопасность. Рассматривает жизненные циклы разных типов компонентов Enterprise JavaBeans и рассказывает, как обеспечить автоматический вызов методов компонентов контейнером EJB в определенные моменты жизненного цикла.

Глава 5, «Контексты и внедрение зависимостей», представляет собой введение в механизм поддержки контекстов и внедрения зависимостей (Contexts and Dependency Injection, CDI). Эта глава познакомит вас с именованными компонентами CDI, квалифициаторами CDI и расскажет, как внедрять зависимости с помощью CDI.

Глава 6, «Обработка JSON с помощью JSON-P API», рассказывает, как создавать и обрабатывать данные в формате JSON с применением нового прикладного интерфейса JSON-P. Она охватывает оба API для обработки JSON: Model API и Streaming API.

Глава 7, «Веб-сокеты», рассказывает, как создавать веб-приложения, поддерживающие полноценные двусторонние взаимодействия между клиентом и сервером, в противоположность традиционным, опирающимся на традиционный цикл запрос/ответ.

Глава 8, «Служба обмена сообщениями Java», повествует о том, как настроить в GlassFish фабрики соединений JMS, очереди и темы сообщений JMS, используя веб-консоль GlassFish. Здесь также рассказывается, как организовать обмен сообщениями между приложениями с использованием полностью обновленного JMS 2.0 API.

Глава 9, «Безопасность приложений Java EE», рассказывает, как обезопасить приложения Java EE с применением стандартных областей безопасности (security realms), а также о том, как добавлять собственные области безопасности.

Глава 10, «Веб-службы JAX-WS», охватывает приемы разработки веб-служб и их клиентов с помощью JAX-WS API. Здесь также рассматривается возможность автоматического создания программного кода с использованием инструментов сборки ANT и Maven.

Глава 11, «Веб-службы RESTful JAX-RS», описывает приемы создания веб-служб RESTful с использованием нового прикладного ин-

терфейса Java API for RESTful Web Services, а также приемы создания клиентов веб-служб RESTful с применением совершенно нового стандартного клиентского JAX-RS API. Наконец, здесь рассказывается, как автоматически преобразовывать данные между Java и XML, используя возможности Java API для связывания с XML (Java API for XML Binding, JAXB).

Что нужно для чтения этой книги

Для чтения этой книги и опробования примеров из нее потребуется установить следующее программное обеспечение:

- комплект разработчика Java (Java Development Kit, JDK) версии 1.7 или выше;
- GlassFish 4.0;
- Maven версии 3 или выше (необходим для сборки примеров);
- среда разработки на Java, такая как NetBeans, Eclipse или IntelliJ IDEA (необязательно, но рекомендуется).

Кому адресована эта книга

Эта книга предполагает знакомство читателя с языком программирования Java, поэтому она адресована разработчикам приложений на Java, желающим освоить Java EE, а также разработчикам приложений Java EE, желающим освежить свои знания и познакомиться с последней версией спецификации Java EE.

Соглашения

В этой книге вы обнаружите несколько стилей оформления текста, которые разделяют различные виды информации. Ниже приводятся примеры этих стилей и поясняется их значение.

Элементы программного кода в тексте, имена таблиц в базах данных, имена папок и файлов, расширения файлов, пути к каталогам в файловой системе, фиктивные адреса URL, ввод пользователя и учетные записи в Twitter оформляются так: «Аннотация @Named на уровне класса указывает, что этот компонент является именованным компонентом CDI».

```
if (!emailValidator.isValid(email)) {  
    FacesMessage facesMessage =  
        new FacesMessage(htmlInputText.getLabel()  
            + ": email format is not valid");
```

```
        throw new ValidatorException(facesMessage);  
    }  
}
```

Чтобы привлечь ваше внимание к определенной части в блоке кода, соответствующие строки или элементы будут выделены жирным шрифтом:

```
<ejb>  
  <ejb-name>CustomerDaoBean</ejb-name>  
  <ior-security-config>  
    <as-context>  
      <auth-method>username_password</auth-method>  
      <realm>file</realm>  
      <required>true</required>  
    </as-context>  
  </ior-security-config>  
</ejb>
```

Любой ввод или вывод в командной строке оформляется так:

```
$ ~/GlassFish/glassfish4/bin $ ./asadmin start-domain  
Waiting for domain1 to start .....
```

Новые термины и важные (ключевые) слова в тексте выделяются жирным. Элементы интерфейса программ, например пункты меню или поля в диалогах выделяются так: «Щелкните на кнопке **Next** (Далее), чтобы перейти к следующему экрану».



Предупреждения или важные примечания отмечены в тексте таким образом.



Советы и рекомендации обозначены так.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или может быть не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору

по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Загрузка исходного кода примеров

Загрузить файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.dmk.ru в разделе «Читателям – Файлы к книгам».

Список опечаток

Хотя мы приняли все возможные меры, чтобы удостовериться в качестве наших текстов, ошибки всё равно случаются. Если вы найдёте ошибку в одной из наших книг – возможно, ошибку в тексте или в коде – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдёте какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в Интернете по-прежнему остается насущной проблемой. Издательства ДМК Пресс и Packt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в Интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли принять меры.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, и помогающую нам предоставлять вам качественные материалы.

Вопросы

Вы можете присыпать любые вопросы, касающиеся данной книги, по адресу dm@dmk-press.ru или questions@packtpub.com. Мы постараемся разрешить возникшие проблемы.



ГЛАВА 1.

Знакомство с сервером GlassFish

В этой главе мы обсудим, как приступить к работе с сервером GlassFish. Вот некоторые из обсуждаемых тем:

- ❖ общий обзор Java EE и GlassFish;
- ❖ получение сервера приложений GlassFish;
- ❖ установка и запуск сервера приложений GlassFish;
- ❖ описание понятия доменов GlassFish;
- ❖ развертывание приложений Java EE;
- ❖ установка соединения с базой данных.

Общий обзор Java EE и GlassFish

Спецификация Java Enterprise Edition (Java EE, ранее называвшаяся J2EE, или Java 2 Enterprise Edition) включает стандартный набор технологий для разработки серверных приложений на Java. В число таких технологий входят: фреймворк **JavaServer Faces (JSF)**, компоненты корпоративных приложений (**Enterprise Java Beans, EJB**), служба обмена сообщениями Java (**Java Messaging Service, JMS**), прикладной интерфейс хранения данных (**Java Persistence API, JPA**), Java API для веб-сокетов (**Java API for WebSocket**), механизм контекстов и внедрения зависимостей (**Contexts and Dependency Injection, CDI**), Java API для веб-служб XML (**Java API for XML Web Services, JAX-WS**), Java API для веб-служб RESTful (**Java API for RESTful Web Services, JAX-RS**) и Java API для обработки данных в формате JSON (**Java API for JSON Processing, JSON-P**).

Существует несколько коммерческих версий серверов приложений и несколько версий с открытым исходным кодом. Серверы приложений Java EE позволяют создавать и развертывать Java EE-совместимые приложения; одним из таких серверов приложений яв-

ляется сервер GlassFish. В числе других серверов приложений с открытым исходным кодом для Java EE можно назвать Red Hat WildFly (ранее известный как JBoss), Apache Software Foundation Geronimo и ObjectWeb JOnAS. Из коммерческих версий серверов можно назвать Oracle Weblogic, IBM Websphere и Oracle Application Server.

GlassFish – эталонная реализация сервера приложений для Java EE 7. В нем в самом первом были реализованы самые последние Java EE API. GlassFish является открытым и бесплатным продуктом, и распространяется на условиях общей лицензии разработки и распространения (Common Development and Distribution License, CDDL).



Полный текст лицензии CDDL можно найти по адресу: <http://opensource.org/licenses/CDDL-1.0>.

Как сервер приложений, полностью совместимый с Java EE, GlassFish предоставляет все необходимые библиотеки, позволяющие разрабатывать и развертывать Java-приложения, соответствующие спецификации Java EE.

Новые возможности Java EE 7

На сегодняшний день Java EE 7 является самой последней версией спецификации Java EE, включающей несколько усовершенствований и дополнений. В следующих разделах перечислены основные усовершенствования спецификации, которые представляют интерес для разработчиков корпоративных приложений.

JavaServer Faces (JSF) 2.2

Java EE 7 включает новую версию фреймворка **JavaServer Faces (JSF)**. В версии фреймворка JSF 2.2 появились следующие важные возможности:

- JSF 2.2 поддерживает HTML5-подобную разметку, то есть, веб-страницы теперь можно писать с использованием стандартной разметки HTML 5 и атрибутов JSF;
- JSF 2.2 включает также Faces Flows, средство для объединения взаимосвязанных страниц в последовательность с определенными точками входа и выхода;
- Третьей важной особенностью, появившейся в версии JSF 2.2, являются контракты библиотек ресурсов (resource library contracts). Контракты библиотек ресурсов упрощают разра-

ботку веб-приложений, которые могут выглядеть и действовать по-разному для разных пользователей.

Java Persistence API (JPA) 2.1

Прикладной интерфейс JPA был введен в спецификации Java EE 5. Главной его целью было заменить объектные компоненты (Entity Beans), использовавшиеся на тот момент в качестве стандартной основы для создания объектно-реляционных отображений в Java EE. JPA перенял идеи сторонних объектно-реляционных фреймворков, таких как Hibernate и JDO, и сделал их частью стандарта.

В JPA 2.1 появились следующие новые возможности:

- введено понятие **конвертеров** (Converters), обеспечивающих возможность писать свой код для выполнения преобразований между значениями в базе данных и в объектах Java. Типичная проблема при работе с базами данных состоит в том, что значения в объектах Java должны отличаться от хранящихся в базе данных. Например, значения 1 и 0 часто используются в базах данных для обозначения `true` и `false` соответственно. В языке Java имеется превосходный логический тип, поэтому значения `true` и `false` желательно было бы использовать непосредственно;
- JPA Criteria API теперь может выполнять массовые изменения и удаления;
- в JPA 2.1 появилась поддержка хранимых процедур;
- добавлена аннотация `@ConstructorResult`, позволяющая возвращать Java-классы (но не сущности JPA) из запросов SQL.

Java API для веб-служб RESTful (JAX-RS) 2.0

JAX-RS – это Java API для разработки веб-служб RESTful. Веб-службы RESTful используют архитектуру передачи репрезентативного состояния (**Representational State Transfer, REST**). JAX-RS была принята в состав официальной спецификации Java EE в версии 6.

Служба сообщений Java Message Service (JMS) 2.0

Прикладной интерфейс Java Message Service (JMS) был полностью переработан в Java EE 7. Предыдущие версии JMS требовали писать массу типового кода; при использовании новой, переработанной версии JMS 2.0 API, этого больше не требуется.

Java API для обработки JSON (JSON-P) 1.0

JSON-P – это совершенно новый прикладной интерфейс, введенный в Java EE 7. Он позволяет генерировать и анализировать строки в формате **JSON (JavaScript Object Notation** – форма записи объектов JavaScript).

Java API для веб-сокетов 1.0

Традиционные веб-приложения используют модель запрос/ответ, то есть, клиент (обычно веб-браузер) запрашивает ресурсы, а сервер возвращает ответ. В этой модели взаимодействий инициатором всегда является клиент.

В спецификацию HTML5 была включена новая технология Web-Sockets (веб-сокеты). Она обеспечивает полноценные двусторонние взаимодействия между клиентом и сервером.

Преимущества GlassFish

Имеется много версий серверов приложений для Java EE, но почему чаще всего выбирают именно GlassFish? Помимо очевидных преимуществ бесплатного сервера, GlassFish предлагает множество других:

- **эталонная реализация Java EE**: сервер GlassFish является эталонной реализацией поддержки Java EE. Это означает, что разработчики других серверов приложений могут использовать GlassFish для сравнения, чтобы убедиться, что их продукты соответствуют спецификации. Теоретически GlassFish можно было бы использовать для отладки других серверов приложений. Если приложение, развернутое на другом сервере приложений, выполняется с ошибками, но после развертывания в GlassFish действует правильно, это скорее всего свидетельствует о неправильной работе другого сервера приложений;
- **поддерживает самые последние версии спецификации Java EE**. Поскольку GlassFish является эталонной реализацией спецификации Java EE, он начинает поддерживать самые последние нововведения в спецификации раньше, чем любые другие серверы приложений на рынке. Действительно, на момент написания этой книги GlassFish являлся единственным Java EE-совместимым сервером приложений на рынке, который поддерживал спецификацию Java EE 7 в полном объеме.

Получение GlassFish

Сервер GlassFish можно загрузить по адресу: <http://glassfish.java.net/>.



Сервер GlassFish 4.0 также входит в состав дистрибутива NetBeans IDE версии 7.4 и выше.

После перехода по указанному адресу, в окне веб-браузера откроется страница, как показано на рис. 1.1.

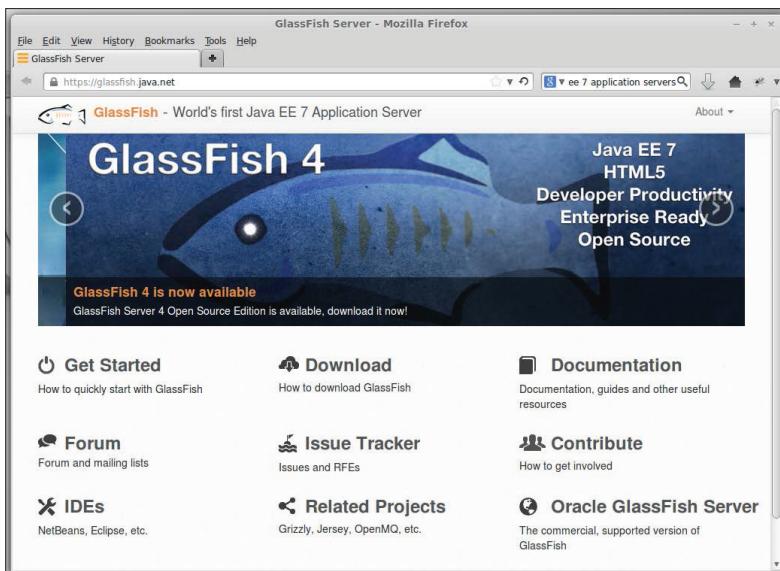


Рис. 1.1. Домашняя страница проекта GlassFish

Если щелкнуть на ссылке **Download** (Загрузить), откроется страница мастера, где будет предложено несколько вариантов для загрузки, как показано на рис. 1.2.

На странице загрузки предлагается на выбор несколько вариантов; можно загрузить полный комплект платформы Java EE (**Java EE 7 Full Platform**) или только комплект, ориентированный на разработку веб-приложений (**Java EE 7 Web Profile**). Можно также загрузить GlassFish в виде ZIP-архива или инсталлятора для выбранной операционной системы.

Чтобы иметь возможность опробовать все примеры из этой книги, нужно загрузить полный комплект платформы Java EE. Далее будет

описана установка из ZIP-архива, поскольку порядок установки в этом случае почти не отличается в разных операционных системах, но вы можете загрузить инсталлятор для своей платформы и выполнить установку с его помощью.

The screenshot shows the GlassFish Server Open Source Edition 4.0 Download page. At the top, there are tabs for 'GlassFish Open Source Edition', 'Oracle GlassFish Server', 'Java EE SDK', and 'Maven'. Below them are buttons for 'Work in progress' and 'Earlier Releases'. A message box says 'Please take the Java EE 8 Community Survey'. The main content area is divided into steps:

- Step 0. Prerequisite**: 'Java EE 7 requires JDK 7'.
- Step 1. Download**:
 - Zip (quick start)**: A link to 'glassfish-4.0.zip'.
 - or a specific installer**: Options for 'Java EE 7' (Full Java EE platform, Java EE Web Profile), 'Installer' (Zip, Native installer), 'Platform' (Windows, Linux, Solaris, Mac OS X), 'Localisation' (English, Multilingual), and a 'Download Link' to 'glassfish-4.0.zip' (97 MB).
- Step 2. Install**: Instructions to 'unzip glassfish-4.0.zip'.
- Note**: 'Note: This command will extract GlassFish with a preconfigured 'Domain1' domain.'
- Step 3. Start**: Command 'glassfish4/bin/asadmin start-domain'.
- Step 4. Load Console**: Instruction to 'Go to <http://localhost:4848>'.
- Step 5. Check the documentation**: Links to 'Quick Start Guide', 'Installation Guide', 'Release Notes', and 'All-in-one Documentation Bundle'. A note says 'Visit the [documentation page](#) for additional guides and documentations.'

Рис. 1.2. Страница мастера с несколькими вариантами для загрузки

Установка GlassFish

Процесс установки будет проиллюстрирован на примере установки из ZIP-архива. Этот процесс почти не отличается во всех основных операционных системах.

Процесс установки GlassFish достаточно прост; тем не менее GlassFish предполагает, что в системе должны присутствовать определенные элементы, от которых он зависит.

Зависимости GlassFish

Для установки GlassFish 4 на рабочей станции должна быть установлена свежая версия комплекта разработчика Java (**Java Development Kit, JDK**) – требуется версия JDK 1.7 или выше – а выполняемый файл Java должен находиться в одном из каталогов, перечисленных в системной переменной окружения `PATH`. Последнюю версию JDK можно загрузить с сайта: <http://www.oracle.com/technetwork/java/javase/downloads/index.html>. За инструкциями по установке JDK обращайтесь к документации для своей платформы по адресу: <http://docs.oracle.com/javase/7/docs/webnotes/install/index.html>.

Установка

После установки JDK можно установить GlassFish, просто распаковав загруженный ZIP-архив, как показано на рис. 1.3.

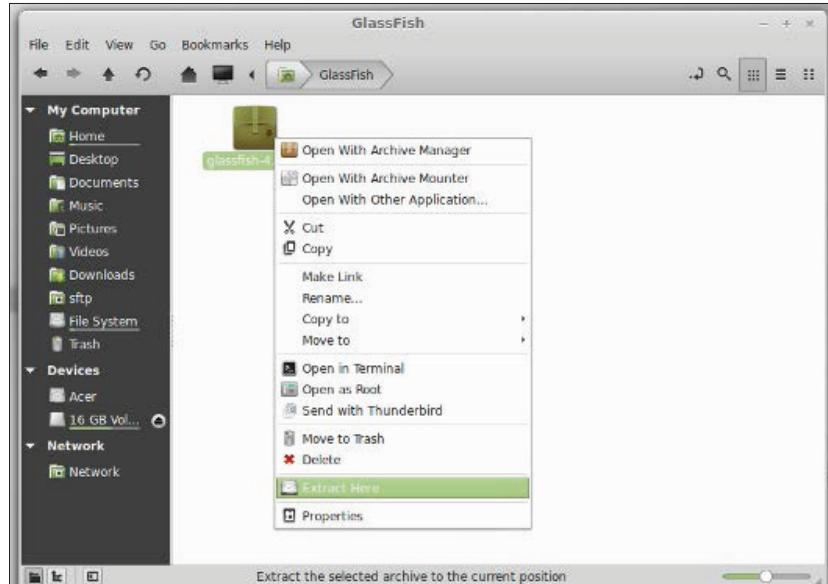


Рис. 1.3. Распаковка ZIP-архива



Все современные операционные системы, включая Linux, Windows и Mac OS X, имеют встроенную поддержку для распаковки ZIP-файлов. За подробностями обращайтесь к документации для своей операционной системы.

После распаковки появится новый каталог с именем glassfish4. Этот новый каталог содержит готовую к использованию версию GlassFish.

Запуск GlassFish

Чтобы запустить GlassFish, нужно перейти в [каталог установки GlassFish]/GlassFish4/bin и выполнить следующую команду:

```
./asadmin start-domain domain1
```



Эта команда, как и большинство команд, описываемых в этой главе, предполагает использование Unix или Unix-подобной операционной системы, такой как Linux или Mac OS. В Windows начальные символы ./ в команде не требуются.

Спустя несколько коротких мгновений после запуска команды, в окне терминала должно появиться следующее сообщение:

```
$ ~/GlassFish/glassfish4/bin $ ./asadmin start-domain
Waiting for domain1 to start .....
Successfully started the domain : domain1
domain Location: /home/heffel/GlassFish/glassfish4/glassfish/
domains/domain1
Log File: /home/heffel/GlassFish/glassfish4/glassfish/domains/
domain1/
logs/server.log
Admin Port: 4848
Command start-domain executed successfully.
```



Загрузите примеры кода. Загрузить файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.dmk.ru в разделе «Читателям – Файлы к книгам».

Затем можно открыть окно браузера и ввести в адресной строке строку URL:

`http://localhost:8080.`

Если все в порядке, должна появиться страница, как показано на рис. 1.4.

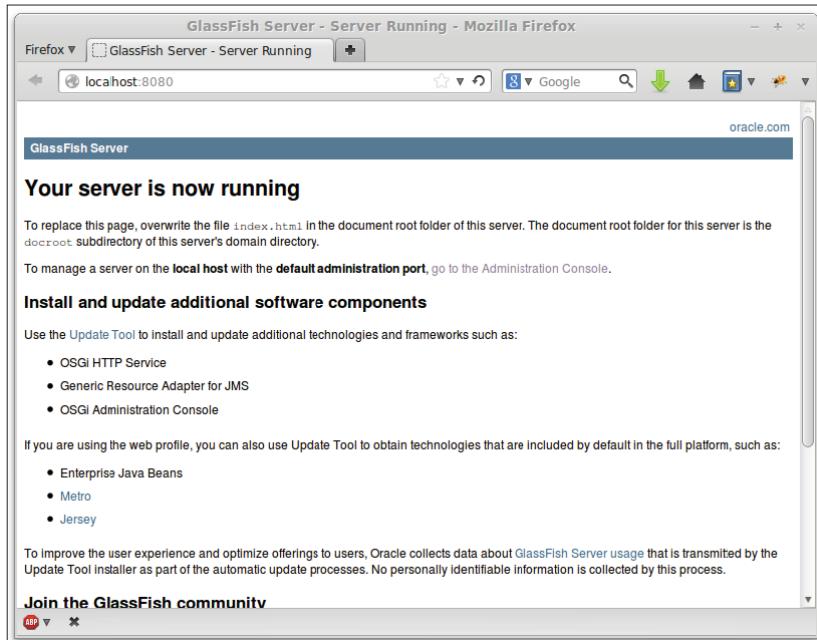


Рис. 1.4. Начальная страница сервера GlassFish



Получение справки. Если какой-либо из предыдущих шагов завершился неудачей либо вам требуется общая справочная информация о сервере GlassFish, обращайтесь к замечательному информационному ресурсу – форуму GlassFish, по адресу: <http://www.java.net/forums/glassfish/glassfish>.

Развертывание первого приложения Java EE

Чтобы убедиться в работоспособности сервера GlassFish, развернем WAR-файл (Web ARchive – веб-архив) и убедимся, что он развер-

тывается и выполняется без ошибок. Прежде чем двинуться дальше, загрузите файл simpleapp.war с веб-сайта www.dmkpress.com.

Развертывание приложения через веб-консоль

Чтобы развернуть simpleapp.war, запустите браузер и откройте в нем страницу с URL: <http://localhost:4848>. В браузере должна появиться страница администрирования, как показано на рис. 1.5.



Рис. 1.5. Страница администрирования сервера GlassFish

По умолчанию сервер GlassFish настроен на работу в режиме разработки. В этом режиме нет необходимости вводить имя пользователя и пароль для доступа к веб-консоли GlassFish. В эксплуатационном окружении обязательно следует защитить доступ к веб-консоли паролем.

Теперь щелкните на пункте **Deploy an Application** (Развернуть приложение) в разделе **Deployment** (Развертывание), в центральной панели.

Чтобы развернуть приложение выберите переключатель **Local packaged file or directory that is accessible from the Application Server** (Локальный упакованный файл или каталог, доступный из сервера приложений) и введите путь к WAR-файлу или выберите его,

щелкнув на кнопке **Browse Files...** (Обзор файлов...). В последнем случае появится окно, как показано на рис. 1.6.

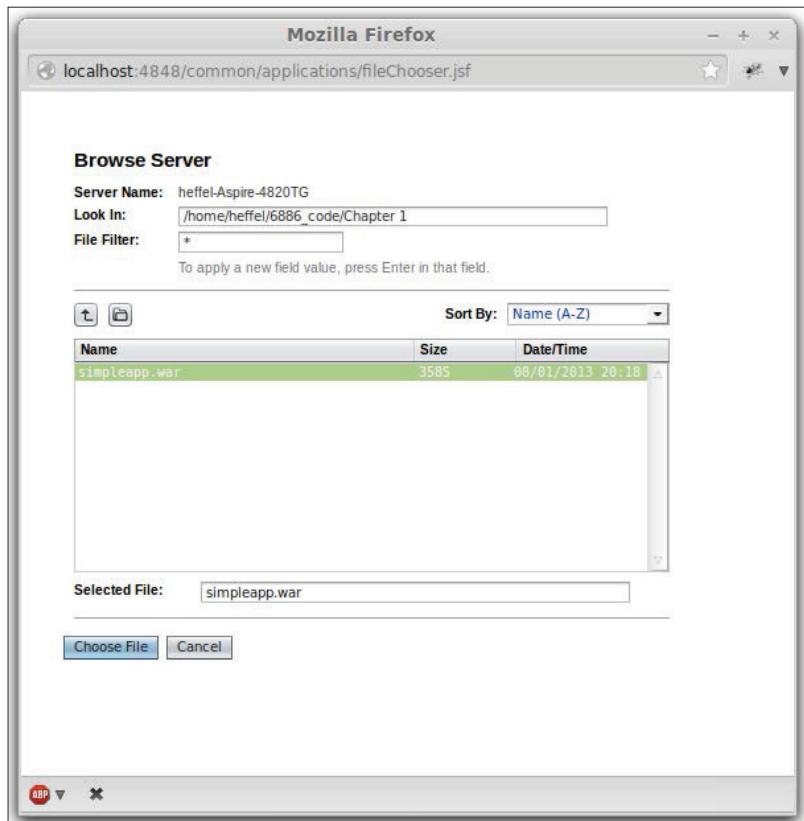


Рис. 1.6. Окно выбора файла для развертывания

После выбора WAR-файла будет предоставлена возможность уточнить некоторые параметры (см. рис. 1.7). Для нашей цели вполне подходят значения по умолчанию, поэтому можно просто щелкнуть на кнопке **OK** справа вверху, чтобы развернуть файл.

После развертывания в веб-консоли сервера GlassFish отобразится страница **Applications** (Приложения), где в списке будет присутствовать только что развернутое приложение (см. рис. 1.8).

Чтобы запустить приложение `simpleapp`, введите в адресной строке браузера строку URL:

```
http://localhost:8080/simpleapp/simple servlet.
```

Deploy Applications or Modules

Specify the location of the application or module to deploy. An application can be in a packaged file or specified as a directory.

* Indicates required field

Location: **Packaged File to Be Uploaded to the Server**
 No file selected.

Local Packaged File or Directory That Is Accessible from GlassFish Server

Type: *

Context Root:
 Path relative to server's base URL.

Application Name: *

Virtual Servers: 
 Associates an Internet domain name with a physical server.

Status: **Enabled**
 Allows users to access the application.

Precompile JSPs:
 Precompiles JSP pages during deployment.

Run Verifier:
 Verifies the syntax and semantics of the deployment descriptor.Verifier packages must be installed.

Force Redeploy:
 Forces redeployment even if this application has already been deployed or already exists.

Keep State:
 Retains web sessions, SFSB instances, and persistently created EJB timers between redeployments.

Deployment Order:
 A number that determines the loading order of the application at server startup. Lower numbers are loaded first. The default is 100.

Libraries:
 A comma-separated list of library JAR files. Specify the library JAR files by their relative or absolute paths. Specify relative paths relative to `instance-root/lib/applications`. The libraries are made available to the application in the order specified.

Description:

OK **Cancel**

Рис. 1.7. Окончание развертывания WAR-файла

Applications

Applications can be enterprise or web applications, or various kinds of modules. Restart an application or module by clicking on the reload link, this action will apply only to the targets that the application or module is enabled on.

Deployed Applications (1)

Select	Name	Deployment Order	Enabled	Engines	Action
<input type="checkbox"/>	simpleapp	100	<input checked="" type="checkbox"/>	web	Launch Redeploy Reload

Рис. 1.8. Вновь развернутое приложение появится в списке приложений

В окне браузера должна открыться страница, как показано на рис. 1.9.

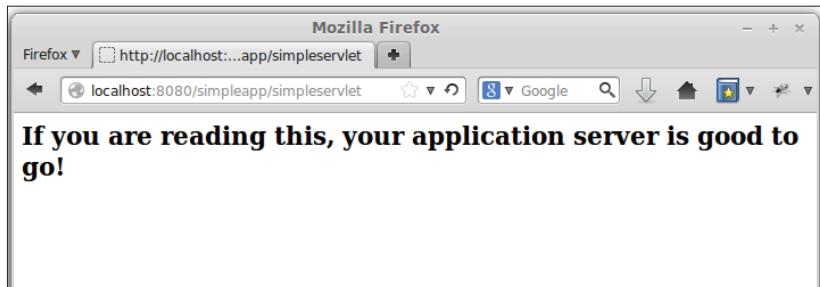


Рис. 1.9. Начальная страница приложения simpleapp

Вот и все! Мы успешно развернули первое приложение Java EE.

Удаление приложения через веб-консоль GlassFish

Чтобы удалить приложение, развернутое в предыдущем разделе, откройте консоль администрирования GlassFish, введя следующий URL в адресной строке браузера:

`http://localhost:4848.`

Затем щелкните на пункте **Applications** (Приложения) в панели навигации слева или на элементе **List Deployed Applications** (Список развернутых приложений) в основной панели консоли администрирования.

В любом случае должна открыться страница управления приложениями (см. рис. 1.10).

 A screenshot of the 'Deployed Applications' section of the GlassFish Admin Console. The table lists one application: 'simpleapp'. The 'Enabled' column shows a checked checkbox, and the 'Action' column contains three buttons: 'Launch', 'Redeploy', and 'Reload'.

Select	Name	Deployment Order	Enabled	Engines	Action
<input type="checkbox"/>	simpleapp	100	<input checked="" type="checkbox"/>	web	Launch Redeploy Reload

Рис. 1.10. Страница управления приложениями

Приложение можно удалить, просто выбрав его в списке развернутых приложений и щелкнув на кнопке **Undeploy** (Удалить). После этого страница управления приложениями будет выглядеть, как показано на рис. 1.11.

The screenshot shows the 'Deployed Applications (0)' section of the GlassFish Admin Console. At the top, there are buttons for Deploy..., Undeploy, Enable, Disable, and Filter. Below is a table with columns: Select, Name, Deployment Order, Enabled, Engines, and Action. A message 'No items found.' is displayed below the table.

Рис. 1.11. Страница управления приложениями после удаления приложения

Развертывание приложения из командной строки

Имеется два способа развертывания приложения из командной строки. Это можно сделать путем копирования в каталог `autodeploy` артефакта, который требуется развернуть, или с помощью утилиты командной строки `asadmin`, входящей в состав сервера GlassFish

Каталог `autodeploy`

Теперь, когда мы удалили развернутый файл `simpleapp.war`, можно попробовать развернуть его с использованием командной строки. Для этого просто скопируйте файл `simpleapp.war` в [каталог установки GlassFish]/glassfish4/glassfish/domains/domain1/autodeploy. Приложение будет автоматически развернуто сразу после создания его копии в этом каталоге.

Чтобы убедиться, что приложение было успешно развернуто, зайдите в журнал сервера. Журнал можно найти в файле [каталог установки GlassFish]/glassfish4/glassfish/domains/domain1/logs/server.log. Последние несколько строк в этом файле должны выглядеть примерно так:

```
[2013-08-02T10:57:45.387-0400] [glassfish 4.0] [INFO] [NCLSDEPLOYMENT-00027] [javax.enterprise.system.tools.deployment.autodeploy] [tid: _ThreadID=91 _ThreadName=AutoDeployer] [timeMillis: 1375455465387] [levelValue: 800] [[
```

```
Selecting file /home/heffel/GlassFish/glassfish4/glassfish/domains/domain1/autodeploy/simpleapp.war for autodeployment]]
```

```
[2013-08-02T10:57:45.490-0400] [glassfish 4.0] [INFO] [] [javax.enterprise.system.tools.deployment.common] [tid: _ThreadID=91 _ThreadName=AutoDeployer] [timeMillis: 1375455465490] [levelValue: 800] [[
```

```
visiting unvisited references]]
```

```
[2013-08-02T10:57:45.628-0400] [glassfish 4.0] [INFO] [AS-WEB-GLUE-00172]
```

```
[javax.enterprise.web] [tid: _ThreadID=91_ThreadName=AutoDeployer]
[timeMillis: 1375455465628] [levelValue: 800] [[

    Loading application [simpleapp] at [/simpleapp]]]

[2013-08-02T10:57:45.714-0400] [glassfish 4.0] [INFO] [] [javax.
enterprise.system.core] [tid: _ThreadID=91_ThreadName=AutoDeployer]
[timeMillis: 1375455465714] [levelValue: 800] [[

    simpleapp was successfully deployed in 302 milliseconds.]]]

[2013-08-02T10:57:45.723-0400] [glassfish 4.0] [INFO] [NCLSDEPLOYMENT-
00035] [javax.enterprise.system.tools.deployment.autodeploy]
[tid: _ThreadID=91_ThreadName=AutoDeployer] [timeMillis: 1375455465723]
[levelValue: 800] [[

    [AutoDeploy] Successfully autodeployed : /home/heffel/GlassFish/
glassfish4/glassfish/domains/domain1/autodeploy/simpleapp.war.]]
```

Конечно, можно выполнить дополнительную проверку, открыв URL приложения, который будет таким же, как использованный выше, когда приложение развертывалось через веб-консоль: `http://localhost:8080/simpleapp/simple servlet`.

Приложение должно действовать, как ожидается.

Чтобы удалить приложение, развернутое таким способом, достаточно просто удалить артефакт (в нашем случае WAR-файл) из каталога `autodeploy`. После удаления файла в журнале сервера появится примерно такое сообщение:

```
[2013-08-02T11:01:57.410-0400] [glassfish 4.0] [INFO] [NCLS-
DEPLOYMENT-00026] [javax.enterprise.system.tools.deployment.autodeploy]
[tid: _ThreadID=91_ThreadName=AutoDeployer] [timeMillis: 1375455717410]
[levelValue: 800] [[

    Autoundeploying application: simpleapp]]]

[2013-08-02T11:01:57.475-0400] [glassfish 4.0] [INFO] [NCLS-
DEPLOYMENT-00035] [javax.enterprise.system.tools.deployment.autodeploy]
[tid: _ThreadID=91_ThreadName=AutoDeployer] [timeMillis: 1375455717475]
[levelValue: 800] [[

    [AutoDeploy] Successfully autoundeployed : /home/heffel/GlassFish/
glassfish4/glassfish/domains/domain1/autodeploy/simpleapp.war.]]
```

Утилита asadmin

Альтернативный способ развертывания приложений из командной строки заключается в использовании команды:

```
asadmin deploy [путь к файлу]/simpleapp.war
```



Предыдущая команда должна выполняться в каталоге [каталог установки GlassFish]/glassfish4/bin.

В окне терминала должно появиться следующее подтверждение успешного развертывания приложения:

Application deployed with name simpleapp.

Command deploy executed successfully.

А в журнале сервера должно появиться примерно такое сообщение:

```
[2013-08-02T11:05:34.583-0400] [glassfish 4.0] [INFO] [AS-WEB-GLUE-00172]
[javax.enterprise.web] [tid: _ThreadID=37 _ThreadName=admin-listener(5)]
[timeMillis: 1375455934583] [levelValue: 800] [[
```

```
    Loading application [simpleapp] at [/simpleapp]]]
```

```
[2013-08-02T11:05:34.608-0400] [glassfish 4.0] [INFO] [] [javax.
enterprise.system.core] [tid: _ThreadID=37 _ThreadName=admin-listener(5)]
[timeMillis: 1375455934608] [levelValue: 800] [[
```

```
    simpleapp was successfully deployed in 202 milliseconds.]]
```

С помощью утилиты asadmin можно также удалить развернутое приложение:

asadmin undeploy simpleapp

В этом случае в окне терминала появится следующее сообщение:

Command undeploy executed successfully.

Обратите внимание, что при удалении приложения расширение файла не указывается – аргументом для asadmin undeploy должно быть имя приложения, роль которого по умолчанию играет имя WAR-файла без расширения.

Домены GlassFish

Внимательный читатель, возможно, заметил, что каталог autodeploy находится в подкаталоге domains/domain1. GlassFish использует понятие доменов (domains). Домены позволяют совместно развертывать наборы связанных между собой приложений. Одновременно может выполняться несколько доменов; при этом они будут вести себя как отдельные экземпляры сервера GlassFish. Домен по умолчанию, с именем domain1, создается во время установки сервера GlassFish.

Создание доменов

Дополнительные домены можно создавать из командной строки, с помощью следующей команды:

```
asadmin create-domain имя_домена
```

Эта команда принимает несколько параметров, определяющих номера портов, через которые домен будет предоставлять услуги (HTTP, администрирование, JMS, IIOP, защищенный HTTP и т. д.). Чтобы увидеть все эти параметры, введите команду:

```
asadmin create-domain --help
```

Если потребуется, чтобы несколько доменов работали одновременно на одном сервере, выбор портов для них должен осуществляться с известной степенью осторожности, потому что при назначении одних и тех же портов для разных служб (или даже для одной и той же службы в разных доменах) возникнут проблемы, препятствующие нормальной работе одного из доменов.

Номера портов по умолчанию для домена domain1 перечислены в табл. 1.1.

Таблица 1.1. Номера портов по умолчанию для домена domain1

Служба	Порт
Администратор	4848
HTTP	8080
Система обмена сообщениями Java (JMS)	7676
Интернет-протокол Inter-ORB (IIOP)	3700
Защищенный HTTP (HTTPS)	8181
Защищенный IIOP	3820
Взаимная авторизация IIOP	3920
Администрирование расширений управления Java (JMX)	8686

Обратите внимание, что при создании домена единственным портом, который должен быть указан явно, является порт службы администратора. Если номера других портов не указаны, для них будут использоваться значения портов по умолчанию, перечисленные в табл. 1.1. При создании домена необходимо соблюдать осторожность, поскольку, как уже говорилось выше, два домена не смогут работать одновременно на одном и том же сервере, если какие-либо из служб будут прослушивать один и тот же порт.

Как вариант, с помощью следующей команды можно создать домен, не указывая порты для каждой службы:

```
asadmin create-domain --portbase [номер порта] имя_домена
```

Значение параметра `--portbase` определяет номер базового порта для домена. Номера портов для других служб домена будут вычисляться путем добавления смещения к данному номеру. В табл. 1.2 перечислены номера портов, назначаемые в таком случае всем прочим службам.

Таблица 1.2. Номера портов, назначаемые командой

```
asadmin create-domain --portbase
```

Служба	Порт
Администратор	portbase + 48
HTTP	portbase + 80
Система обмена сообщениями Java (JMS)	portbase + 76
Интернет-протокол Inter-ORB (IIOP)	portbase + 37
Защищенный HTTP (HTTPS)	portbase + 81
Защищенный IIOP	portbase + 38
Взаимная авторизация IIOP	portbase + 39
Администрирование расширений управления Java (JMX)	portbase + 86

Конечно, необходимо соблюдать осторожность при выборе значения для `--portbase`, и убедиться, что ни один из присвоенных номеров портов не пересекается ни с каким другим доменом.



Как показывает опыт, при использовании номера базового порта выше 8000 и кратного 1000, создаются домены, которые не конфликтуют друг с другом. Например, безопасно создать один домен с номером базового порта 9000, а другой – с номером базового порта 10000 и т. д.

Удаление доменов

Удалить домен очень просто – достаточно выполнить следующую команду:

```
asadmin delete-domain имя_домена
```

После этого в окне терминала появится сообщение:

```
Command delete-domain executed successfully.
```



Используйте эту команду с осторожностью, поскольку после удаления домена, восстановить его будет нелегко (будут уничтожены все развернутые приложения, а также пулы соединений, источники данных и другие инфраструктурные элементы).

Остановка домена

Действующий домен можно остановить командой:

```
asadmin stop-domain имя_домена
```

Эта команда остановит домен с именем *имя_домена*.

Если на сервере действует только один домен, аргумент *имя_домена* можно опустить.



В этой книге предполагается, что читатель работает с доменом по умолчанию `domain1`, использующим номера портов по умолчанию. Если дело обстоит иначе, приведенные выше команды необходимо изменить, чтобы они соответствовали домену и порту.

Настройка подключения к базе данных

Любое нетривиальное приложение Java EE устанавливает соединение с **системой управления реляционной базой данных**, сокращенно **СУРБД (Relational Database Management System, RDBMS)**. К числу СУРБД, поддерживаемых сервером GlassFish, относятся Java DB, Oracle, Derby, Sybase, DB2, PointBase, MySQL, PostgreSQL, Informix, Cloudscape и SQL Server. В этом разделе мы покажем, как настроить GlassFish для работы с базой данных MySQL. Для других СУРБД процедура будет аналогичной.



GlassFish поставляется в комплекте с СУРБД JavaDB. Эта СУРБД основана на проекте Apache Derby. Чтобы минимизировать объем загружаемых файлов и настроек, необходимых для опробования примеров в этой книге, все примеры, которым требуется СУРБД, будут использовать встроенную СУРБД – JavaDB. В этом разделе также описывается, как связать GlassFish с СУРБД сторонних производителей.

Создание пулов соединений

Открытие и закрытие соединений с базой данных – это довольно медленная операция. Из соображений производительности GlassFish и другие серверы приложений Java EE хранят пул открытых соединений с базой данных. Когда развернутому приложению требуется соединение с базой данных, одно из них извлекается из пула и передается приложению. Когда надобность в соединении отпадает, соединение возвращается в пул.

Первый шаг в процессе создания пула соединений – копирование JAR-файла с драйвером JDBC для используемой СУРБД в каталог `lib` домена (информацию, где взять JAR-файл, ищите в документации по СУРБД). Если домен GlassFish, куда будет добавляться пул соединений, выполняется в этот момент, после копирования драйвера JDBC его нужно перезапустить, чтобы изменения вступили в силу. Перезапустить домен можно командой:

```
asadmin restart-domain имя_домена
```

После того как драйвер JDBC будет скопирован в нужное место и сервер приложений перезапущен, войдите в консоль администрирования по адресу: `http://localhost:4848`.

Щелкните на узле **Resources | JDBC | Connection Pools** (Ресурсы | JDBC | Пулы соединений) в панели навигации слева. В браузере должна открыться страница, как показано на рис. 1.12.

Select	Pool Name	Resource Type	Classname	Description
<input type="checkbox"/>	DerbyPool	javax.sql.DataSource	org.apache.derby.jdbc.ClientDataSource	
<input type="checkbox"/>	_Time Pool	javax.sql.XADataSource	org.apache.derby.jdbc.EmbeddedXADataSource	

Рис. 1.12. Список пулов соединений

Щелкните на **New...** (Новый...). После ввода настроек соединения, соответствующих базе данных, основная область страницы должна выглядеть, как показано на рис. 1.13.

После щелчка на кнопке **Next** (Далее) откроется страница, как показано на рис. 1.14.

New JDBC Connection Pool (Step 1 of 2)

Identify the general settings for the connection pool.

* Indicates required field

General Settings

Pool Name:	<input type="text" value="FlightStatsDB"/>
Resource Type:	<input type="text" value="javax.sql.DataSource"/>
Database Driver Vendor:	<input type="text" value="MySql"/>
Introspect:	<input type="checkbox"/> Enabled

Must be specified if the datasource class implements more than 1 of the interface.

Select or enter a database driver vendor

If enabled, data source or driver implementation class names will enable introspection.

Next **Cancel**

Рис. 1.13. Настройки пула соединений с базой данных

New JDBC Connection Pool (Step 2 of 2)

Identify the general settings for the connection pool. Datasource Classname or Driver Classname must be specified for the connection pool.

* Indicates required field

General Settings

Pool Name:	FlightStatsDB
Resource Type:	javax.sql.DataSource
Database Driver Vendor:	MySQL
Datasource Classname:	<input type="text" value="com.mysql.jdbc.jdbc2.optional.MysqlDataSource"/>
Driver Classname:	<input type="text"/>
Ping:	<input type="checkbox"/> Enabled
Description:	<input type="text"/>

Select or enter vendor-specific classname that implements the DataSource and/or XADataSource APIs

Select or enter vendor-specific classname that implements the java.sql.Driver interface.

When enabled, the pool is pinged during creation or reconfiguration to identify and warn of any erroneous values for its attributes

Pool Settings

Initial and Minimum Pool Size:	<input type="text" value="8"/>	Connections
--------------------------------	--------------------------------	-------------

Previous **Finish** **Cancel**

Рис. 1.14. Дополнительные настройки пула соединений с базой данных

Большинство значений по умолчанию в верхней части страницы вполне приемлемы и не требуют изменений. Прокрутив страницу вниз и определив дополнительные настройки, щелкните на кнопке **Finish** (Завершить) справа вверху.

Имена свойств могут отличаться для разных СУРБД, но обычно у всех есть свойство URL, куда следует ввести JDBC URL базы данных,

а также поля ввода имени пользователя и пароля, где должны быть указаны учетные данные для аутентификации в базе данных.

Созданный новый пул соединений теперь должен появиться в списке пулов соединений, как показано на рис. 1.15.

JDBC Connection Pools

To store, organize, and retrieve data, most applications use relational databases. Java EE applications access relational databases through the JDBC API. Before an application can access a database, it must get a connection.

Pools (3)					
Select	Pool Name	Resource Type	Classname	Description	
<input type="checkbox"/>	DerbyPool	javax.sql.DataSource	org.apache.derby.jdbc.ClientDataSource		
<input type="checkbox"/>	FlightStatsDB	javax.sql.DataSource	com.mysql.jdbc.jdbc2.optional.MysqlDataSource		
<input type="checkbox"/>	TimerPool	javax.sql.XADataSource	org.apache.derby.jdbc.EmbeddedXADataSource		

Рис. 1.15. Вновь созданный пул соединений

Иногда бывает необходимо перезапустить GlassFish после создания нового пула соединений.

Убедиться в правильности настройки пула соединений можно, щелкнув на его имени и затем – на кнопке **Ping**, как показано на рис. 1.16.

Edit JDBC Connection Pool

Modify an existing JDBC connection pool. A JDBC connection pool is a group of reusable connections for a particular database.

General Settings

Pool Name:	FlightStatsDB
Resource Type:	javax.sql.DataSource
Data source Classname:	com.mysql.jdbc.jdbc2.optional.MysqlDataSource
Driver Classname:	
Ping:	<input type="checkbox"/> Enabled
Deployment Order:	100
Description:	

Ping Succeeded

Save Cancel

* Indicates required field

Рис. 1.16. Проверка пула соединений

Теперь пул соединений готов к использованию приложениями.

Создание источников данных

Приложения Java EE не используют пул соединений напрямую, а обращаются к источнику данных, который ссылается на пул соединений. Чтобы создать новый источник данных, нужно щелкнуть на узле **JDBC Resources** (Ресурсы JDBC) в панели навигации слева, а затем на кнопке **New...** (Новый...). После ввода настроек для нового источника данных основная часть страницы должна выглядеть, как показано на рис. 1.17.

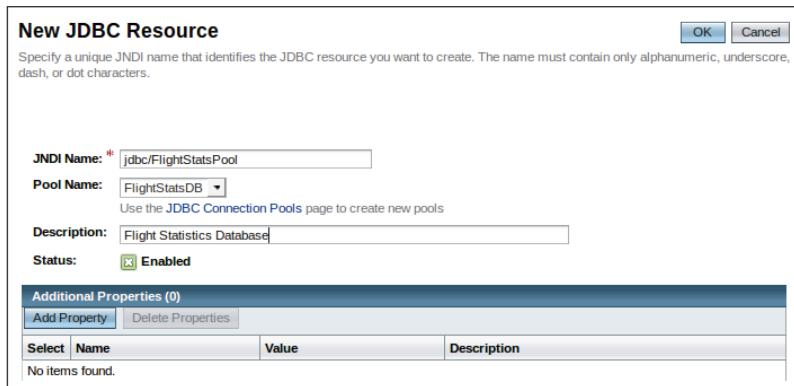


Рис. 1.17. Настройки источника данных

После щелчка на кнопке **OK** появится новый источник данных, как показано на рис. 1.18.

JDBC Resources					
JDBC resources provide applications with a means to connect to a database.					
Resources (3)					
Select	JNDI Name	Logical JNDI Name	Enabled	Connection Pool	Description
<input type="checkbox"/>	jdbc/FlightStatsPool		<input checked="" type="checkbox"/>	FlightStatsDB	Flight Statistics Database
<input type="checkbox"/>	jdbc/_TimerPool		<input checked="" type="checkbox"/>	_TimerPool	
<input type="checkbox"/>	jdbc/_default	java:comp/DefaultDataSource	<input checked="" type="checkbox"/>	DerbyPool	

Рис. 1.18. Вновь созданный источник данных

Резюме

В этой главе мы обсудили, как загрузить и установить GlassFish. Мы также обсудили несколько способов развертывания приложений Java EE: через веб-консоль GlassFish, из командной строки и путем копи-

рования файла в каталог `autodeploy`. Кроме того, мы обсудили основные задачи администрирования GlassFish, такие как создание доменов и настройка соединений с базами данных, а также добавление пулов соединений и источников данных. В следующей главе мы посмотрим, как разрабатывать веб-приложения с использованием JSF.



ГЛАВА 2.

JavaServer Faces

В этой главе рассказывается о JavaServer Faces (JSF) – фреймворке стандартных компонентов платформы Java EE. Java EE 7 включает версию JSF 2.2, самую последнюю на момент написания книги. Примечательно, что JSF опирается на множество соглашений о настройках. Следование этим соглашениям избавляет от необходимости писать большие объемы конфигурационной информации. В большинстве случаев можно вообще не написать ни строчки конфигурации. Если учесть, что файл дескриптора развертывания `web.xml` перестал быть обязательным, начиная с версии Java EE 6, это значит, что часто можно создать завершенное веб-приложение, не написав ни строчки XML-конфигурации.

Введение в JSF

В версии JSF 2.2 появился ряд улучшений для упрощения разработки приложений. В следующих разделах некоторые из этих новых функций будут рассмотрены подробнее.



Читатели, не знакомые с более ранними версиями JSF, возможно, не до конца поймут следующие несколько разделов. Однако нет повода для беспокойства, поскольку к концу этой главы все встанет на свои места.

Фейслеты

Одно из заметных отличий современных версий JSF от более ранних обусловлено преобладающей ролью фейслетов (Facelets), как технологии представления. Более ранние версии JSF по умолчанию использовали для этого JSP. Поскольку технология JSP является предшественницей JSF, использование JSP с JSF иногда выглядело

неестественным или создавало проблемы. Например, жизненный цикл JSP-страниц отличается от жизненного цикла JSF. Это несопоставимо с некоторыми трудностями для разработчиков JSF 1.x-приложений.

Фреймворк JSF с самого начала создавался для поддержки нескольких технологий представления. Чтобы воспользоваться этой возможностью, Джейкоб Хуком (Jacob Hookom) написал новую технологию представления специально для JSF. Он назвал ее фейслетами (Facelets). Фейслеты оказались настолько удачными, что де-факто стали стандартом для JSF. Экспертная группа JSF приняла во внимание популярность фейслетов и сделала их официальной технологией представления в версии JSF 2.0.

Необязательный файл `faces-config.xml`

В свое время приложения J2EE пострадали от того, что некоторые считали их чрезмерно «XML-законфицированными».

Стандарт Java EE 5 способствовал значительному уменьшению объема файлов XML с настройками. В Java EE 6 объем конфигурационной информации уменьшился еще больше, а конфигурационный файл JSF `faces-config.xml` вообще стал необязательным в версии JSF 2.0.

В версиях JSF 2.0 и выше управляемые компоненты JSF могут настраиваться с помощью новой аннотации `@ManagedBean`, что устраняет потребность в их настройке с помощью дескриптора `faces-config.xml`. В Java EE 6 появился механизм **контекстов и внедрения зависимостей (Contexts and Dependency Injection, CDI)**, дающий альтернативный способ реализации функциональности, которая прежде обычно возлагалась на управляемые компоненты JSF. Начиная с версии JSF 2.2, предпочтение должно отдаваться именованным компонентам CDI вместо управляемых компонентов JSF.

Кроме прочего, имеется соглашение по навигации JSF. Если значение атрибута `action` ссылки или кнопки JSF 2 соответствует имени фейслета (минус расширение `.xhtml`), в соответствии с соглашением приложение перейдет к фейслету, который определяется значением атрибута `action`. Это соглашение избавляет от необходимости определять пути навигации для приложений в файле дескриптора `faces-config.xml`.

Для большинства JSF-приложений файл `faces-config.xml` совершенно не нужен, при условии, что неуклонно соблюдаются принятые соглашения.

Стандартное расположение ресурсов

В JSF 2.0 было определено стандартное местоположение ресурсов. Ресурсы считаются артефактами, которые страница или компонент JSF должны отобразить. Примерами ресурсов являются таблицы стилей CSS, файлы JavaScript и изображения.

В JSF 2.0 ресурсы могут быть помещены в каталог `resources`, который в свою очередь помещается в корень WAR-файла, или в его подкаталог `META-INF`. В соответствии с соглашением компоненты JSF «знают», что требуемые им ресурсы могут находиться в одном из этих двух местоположений.

Чтобы избежать загромождения каталога `resources`, ресурсы обычно помещаются в подкаталог. Этот подкаталог упоминается в атрибуте `library` компонентов JSF.

Например, таблицу стилей CSS с именем `styles.css` можно поместить в каталог `/resources/css/`.

Получить этот CSS-файл в странице JSF можно с помощью тега `<h:outputStylesheet>`:

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

Аналогично можно сохранить файлы JavaScript в каталоге `/resources/scripts/`:

```
<h:outputScript library="scripts" name="somescript.js"/>
```

А изображение `logo.png` сохранить в каталоге `/resources/images/` и обращаться к этому ресурсу, как показано ниже:

```
<h:graphicImage library="images" name="logo.png"/>
```

Обратите внимание, что в каждом случае значение атрибута `library` совпадает с именем соответствующего подкаталога в каталоге `resources`, а значение атрибута `name` совпадает с именем файла ресурса.

Разработка первого JSF-приложения

Чтобы проиллюстрировать основные понятия JSF, напишем простое приложение, состоящее из двух страниц фейслетов и одного именованного компонента CDI.

Фейслеты

Как отмечалось во введении к этой главе, технологией представления по умолчанию для JSF 2 являются фейслеты. Фейслеты должны быть написаны с использованием стандартного XML. Самый популярный способ разработки страниц фейслетов – использование XHTML в сочетании с определенными пространствами имен JSF. Следующий пример показывает, на что похожа типичная страница фейслета:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!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://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<h:head>
    <title>Enter Customer Data</title>
</h:head>
<h:body>
    <h:outputStylesheet library="css" name="styles.css"/>
    <h:form id="customerForm">
        <h:messages></h:messages>
        <h:panelGrid columns="2"
                     columnClasses="rightAlign,leftAlign">
            <h:outputLabel for="firstName" value="First Name:>
            </h:outputLabel>
            <h:inputText id="firstName"
                         label="First Name"
                         value="#{customer.firstName}"
                         required="true">
                <f:validateLength minimum="2" maximum="30">
                </f:validateLength>
            </h:inputText>
            <h:outputLabel for="lastName" value="Last Name:>
            </h:outputLabel>
            <h:inputText id="lastName"
                         label="Last Name"
                         value="#{customer.lastName}"
                         required="true">
                <f:validateLength minimum="2" maximum="30">
                </f:validateLength>
            </h:inputText>
            <h:outputLabel for="email" value="Email:>
            </h:outputLabel>
            <h:inputText id="email"
                         label="Email"
                         value="#{customer.email}">
                <f:validateLength minimum="3" maximum="30">
                </f:validateLength>
```

```

</h:inputText>
<h:panelGroup></h:panelGroup>
<h:commandButton action="confirmation"
    value="Save">
</h:commandButton>
</h:panelGrid>
</h:form>
</h:body>
</html>

```

Снимок экрана на рис. 2.1 показывает, как эта страница отображается в браузере.

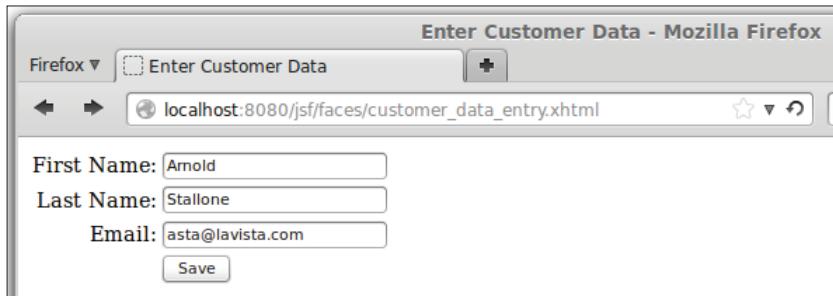


Рис. 2.1. Страница первого JSF-приложения в браузере

Конечно, снимок экрана на рис. 2.1 был получен после ввода некоторых данных в текстовые поля; первоначально все текстовые поля были пустыми.

Любая фейслет-страница JSF включает два пространства имен, показанные в примере. Первое пространство (`xmlns:h="http://java.sun.com/jsf/html"`) включает теги, которые отображают HTML-компоненты. В соответствии с соглашением при использовании этой библиотеки тегов применяется префикс `h` (означающий «HTML»).

Второе пространство имен (`xmlns:f="http://java.sun.com/jsf/core"`) является библиотекой базовых тегов JSF. В соответствии с соглашением при использовании этой библиотеки тегов применяется префикс `f` (означающий «faces»).

Первыми JSF-тегами в предыдущем примере являются теги `<h:head>` и `<h:body>`. Они напоминают стандартные HTML-теги `<head>` и `<body>` и отображаются точно так же, когда страница выводится в браузере.

Тег `<h:outputStylesheet>` используется для загрузки таблицы стилей CSS из известного местоположения (в JSF определено стандартное местоположение ресурсов, таких как таблицы стилей CSS и

JavaScript-файлы; это будет подробно обсуждаться ниже). Значение атрибута `library` должно соответствовать каталогу, где находится файл CSS (этот каталог должен находиться в каталоге `resources`). Атрибут `name` должен соответствовать имени таблицы стилей CSS, которую нужно загрузить.

Следующий тег – `<h:form>`. Этот тег генерирует HTML-форму при отображении страницы. Как видно из примера, нет никакой необходимости определять атрибуты `action` и `method` в этом теге. Фактически он вообще не имеет атрибутов `action` и `method`. Атрибут `action` для отображаемой HTML-формы будет сгенерирован автоматически, а атрибут `method` всегда будет иметь значение "`post`". Атрибут `id` тега `<h:form>` является необязательным, тем не менее, на практике предпочтительнее всегда добавлять его, потому что это упростит отладку JSF-приложений.

Следующий тег в примере – `<h:messages>`. Как явствует из его имени, он используется для вывода сообщений. Как будет показано чуть ниже, JSF может автоматически генерировать сообщения проверки допустимости, которые будут выводиться в этом теге. Кроме того, произвольные сообщения можно добавлять программно, с помощью метода `addMessage()`, определенного в `javax.faces.context.FacesContext`.

Еще один тег JSF в рассматриваемом примере – `<h:panelGrid>`. Этот тег является примерным эквивалентом HTML-таблицы, но работает немного по-другому. Он не требует объявлять строки и столбцы, а предлагает атрибут `columns`. Значение этого атрибута определяет число столбцов в таблице, представленной данным тегом. По мере добавления компонентов в тег, они будут располагаться последовательно в одну строку, пока их число не достигнет значения атрибута `columns`, после этого компоненты начнут располагаться в следующей строке. В приведенном примере атрибут `columns` имеет значение, равное двум. Поэтому первые два вложенных тега будут помещены в первую строку, следующие два – во вторую строку и т. д.

Другим интересным атрибутом тега `<h:panelGrid>` является `columnClasses`. Этот атрибут назначает класс CSS каждому столбцу в отображаемой таблице. В данном примере в качестве значения этого атрибута используются два класса CSS (разделенные запятой). В результате к элементам в первом столбце применяется первый класс CSS, а к элементам во втором столбце – второй класс. Если бы столбцов было три или больше, к элементам в третьем столбце был бы применен первый класс CSS, в четвертом – второй класс CSS и т. д., с

чредованием первого и второго. Чтобы прояснить, как это работает, ниже приводится фрагмент разметки HTML, сгенерированной предыдущей страницей.

```
<table>
  <tbody>
    <tr>
      <td class="rightAlign">
        <label for="customerForm:firstName">
          First Name:
        </label>
      </td>
      <td class="leftAlign">
        <input id="customerForm:firstName" type="text"
               name="customerForm:firstName" />
      </td>
    </tr>
    <tr>
      <td class="rightAlign">
        <label for="customerForm:lastName">
          Last Name:
        </label>
      </td>
      <td class="leftAlign">
        <input id="customerForm:lastName" type="text"
               name="customerForm:lastName" />
      </td>
    </tr>
    <tr>
      <td class="rightAlign">
        <label for="customerForm:email">
          Email:
        </label>
      </td>
      <td class="leftAlign">
        <input id="customerForm:email" type="text"
               name="customerForm:email" />
      </td>
    </tr>
    <tr>
      <td class="rightAlign"></td>
      <td class="leftAlign">
        <input type="submit" name="customerForm:j_idt12"
               value="Save" />
      </td>
    </tr>
  </tbody>
</table>
```

Обратите внимание, что в тегах `<td>` чередуются классы CSS `"rightAlign"` и `"leftAlign"`. Это достигнуто присваиванием значения

"rightAlign, leftAlign" атрибуту `columnClasses` в теге `<h:panelGrid>`. Следует отметить, что классы CSS, используемые в примере, определены в таблице стилей CSS, загруженной с помощью тега `<h:outputStylesheet>`, описанного выше. Атрибуты `id` в генерированной разметке получили значения, являющиеся комбинацией значения атрибута `id` в теге `<h:form>` и атрибута `id` каждого отдельного компонента. В теге `<h:commandButton>`, ближе к концу страницы, мы опустили атрибут `id`, поэтому механизм JSF времени автоматически выбрал для него свое значение.

С этого места в примере начинают добавляться компоненты в тег `<h:panelGrid>`. Эти компоненты будут отображаться в таблице `<h:panelGrid>`. Как упоминалось выше, число столбцов в таблице определяется атрибутом `columns` в теге `<h:panelGrid>`. Поэтому нет нужды беспокоиться о столбцах или строках – можно просто добавлять компоненты, а они автоматически будут распределяться по ячейкам.

Следующий тег – `<h:outputLabel>`. Этот тег отображается в HTML-элемент `label` (метка). Метки связываются с другими компонентами посредством атрибута `for`, в котором должно указываться значение атрибута `id` компонента, с которым связана метка.

Далее следует тег `<h:inputText>`. Он генерирует текстовое поле в отображаемой странице. Его атрибут `label` используется в сообщениях проверки допустимости, где указывает пользователю, к какому полю относится сообщение.



Значение атрибута `label` в теге `<h:inputText>` не обязательно должно соответствовать метке, отображаемой на странице, однако я настоятельно рекомендую использовать это значение. В случае ошибки пользователю будет видно, к какому полю относится сообщение.

Особый интерес представляет атрибут `value`. В качестве значения данного атрибута в данном примере выступает **выражение связывания значения (value binding expression)**. Это означает, что значение атрибута будет получено из свойства одного из именованных компонентов в приложении. В данном примере это конкретное текстовое поле связано со свойством `firstName` именованного компонента `customer`. Когда пользователь введет значение в это текстовое поле и отправит форму, соответствующее свойство именованного компонента получит это значение. Атрибут `required` является необязательным; он может принимать значения `true` и `false`. Если присвоить атрибу-

ту значение `true`, контейнер не даст пользователю отправить форму, пока тот не введет некоторые данные в текстовое поле. Если пользователь попытается отправить форму с незаполненными обязательными полями, страница будет передана браузеру повторно с сообщением об ошибке в теге `<h:messages>`. Это можно увидеть на рис. 2.2.



Рис. 2.2. Сообщение об ошибке после попытки отправить пустые обязательные поля

На рис. 2.2 можно видеть сообщение об ошибке по умолчанию, которое появляется, когда пользователь пытается сохранить форму в примере, оставив пустым поле с именем заказчика. Первая часть сообщения («First Name») получена из значения атрибута `label` соответствующего тега `<h:inputTextField>`. Текст сообщения можно изменить так же, как и его стиль (шрифт, цвет и т. д.). Мы рассмотрим, как это сделать, немного ниже.

Этапы проекта

Наличие тегов `<h:messages>` на каждой странице JSF – хорошая практика программирования. Без них пользователь не сможет увидеть сообщения проверки допустимости и будет недоумевать, почему не выполняется отправка формы. По умолчанию сообщения проверки допустимости JSF не выводятся в журнал сервера GlassFish. Распространенная ошибка, которую делают новички, заключается в том, что они не добавляют теги `<h:messages>` в свои страницы. В результате, если форма не прошла проверку допустимости, кажется, что навигация, перестала работать без всякой причины (на экране отображается та же самая страница, и в отсутствие тега `<h:messages>` никакие сообщения об ошибках не выводятся).

Чтобы избежать подобной ситуации, в JSF 2.0 было введено понятие **этапов проекта (project stages)**.

В JSF 2 определяются следующие этапы проекта:

- промышленная эксплуатация (Production);
- разработка (Development);
- модульное тестирование (Unit Test);
- системное тестирование (System Test).

Этап проекта можно определить как параметр инициализации фейс-сервлета в файле `web.xml` или как пользовательский ресурс JNDI. Поскольку `web.xml` теперь является необязательным, а при его наличии относительно легко использовать неправильную стадию проекта, если забыть исправить ее при переносе кода из одной среды в другую, то предпочтительнее определять этап проекта в виде пользовательского ресурса JNDI.

В GlassFish это можно сделать, войдя в веб-консоль и переместившись к **JNDI | Custom Resources** (**JNDI | Пользовательские ресурсы**) и затем щелкнув на кнопке **New...** (**Новый...**). В результате откроется страница, как показано на рис. 2.3.

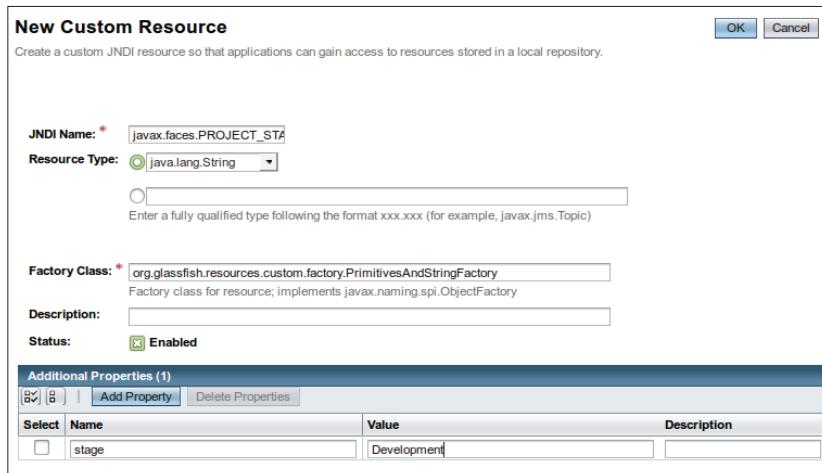


Рис. 2.3. Страница создания нового пользовательского ресурса

На этой странице нужно ввести следующую информацию:

JNDI Name (Имя JNDI) javax.faces.PROJECT_STAGE

Resource Type (Тип ресурса) java.lang.String

После ввода двух предыдущих значений поле **Factory Class** (Класс фабрики) будет автоматически заполнено значением `org.glassfish.resources.custom.factory.PrimitivesAndStringFactory`.

После ввода указанных значений нужно добавить новое свойство с именем `stage` (этап) и значением, соответствующим этапу проекта.

Установка этапа проекта позволяет выполнять некоторую логику только при выполнении в рамках конкретного этапа. Например, в одном из именованных компонентов может иметься следующий код:

```
FacesContext facesContext =  
    FacesContext.getCurrentInstance();  
Application application = facesContext.getApplication();  
  
if (application.getProjectStage().equals(ProjectStage.Production))  
{  
    // код для выполнения на этапе промышленной эксплуатации  
}  
else if (application.getProjectStage().equals(ProjectStage.Development))  
{  
    // код для выполнения на этапе разработки  
}  
else if (application.getProjectStage().equals(ProjectStage.UnitTest))  
{  
    // код для выполнения на этапе модульного тестирования  
}  
else if (application.getProjectStage().equals(ProjectStage.SystemTest))  
{  
    // код для выполнения на этапе системного тестирования  
}
```

Как видите, этапы проекта позволяют изменять поведение кода в разных окружениях. Более того, установка этапа проекта позволяет механизму JSF вести себя немного по-разному. Применительно к обсуждаемому случаю, установка этапа проекта в значение `ProjectStage.Development` приведет к появлению дополнительных операторов журналирования событий в журнале сервера приложений. Поэтому, если забыть добавить тег `<h:messages>` в страницу, а этапом проекта будет `ProjectStage.Development` и возникнут ошибки при проверке допустимости, сообщение об ошибке все равно появится на странице, как показано на рис. 2.4.

На этапе промышленной эксплуатации (`ProjectStage.Production`) это сообщение не появится на странице, и пользователь будет недоумевать, почему не работает навигация.

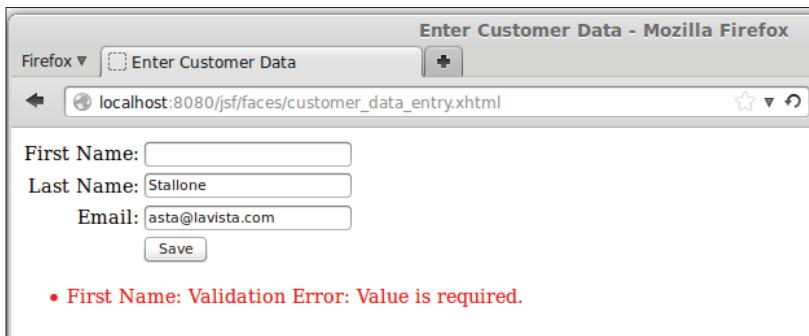


Рис. 2.4. Сообщение об ошибке появится даже в отсутствие тега `<h:messages>`, если выбран этап проекта `ProjectStage.Development`

Проверка допустимости

Фреймворк JSF включает средства проверки допустимости введенных значений.

Обратите внимание, что каждый тег `<h:inputField>` в примере имеет вложенный тег `<f:validateLength>`. Как явствует из его имени, он проверяет значение, введенное в текстовое поле, на соответствие его длины минимальному и максимальному значениям. Минимальное и максимальное значения определяются атрибутами `minimum` и `maximum`. Тег `<f:validateLength>` является одним из стандартных валидаторов (элементов проверки допустимости), включенных в JSF. Точно так же, как атрибут `required` в теге `<h:inputText>`, JSF автоматически выведет сообщение об ошибке по умолчанию, если пользователь попытается отправить форму с недопустимым значением (см. рис. 2.5).

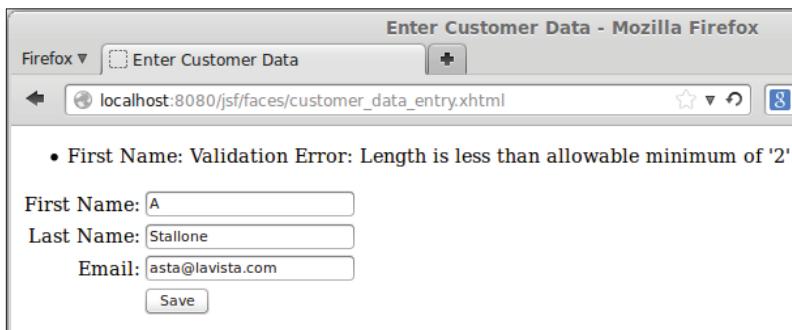


Рис. 2.5. JSF автоматически выведет сообщение об ошибке по умолчанию

Текст сообщения по умолчанию и его стиль можно переопределить. Как это сделать, будет показано в следующем разделе «Настройка сообщений JSF по умолчанию».

Помимо `<f:validateLength>` в JSF имеются и другие валидаторы. Они перечислены в табл. 2.1.

Таблица 2.1. Стандартные валидаторы JSF

Тег валидатора	Описание
<code><f:validateBean></code>	Проверка допустимости на стороне компонента позволяет проверять значения в именованных компонентах с использованием аннотаций, без необходимости добавлять валидаторы в теги JSF. Этот тег позволяет выполнить тонкую настройку проверки допустимости на стороне компонента, если потребуется.
<code><f:validateDoubleRange></code>	Проверяет введенное вещественное число двойной точности (<code>Double</code>) на вхождение в диапазон, определяемый атрибутами <code>minimum</code> и <code>maximum</code> .
<code><f:validateLength></code>	Проверяет длину введенного текста на вхождение в диапазон, определяемый атрибутами <code>minimum</code> и <code>maximum</code> .
<code><f:validateLongRange></code>	Проверяет введенное целое число двойной длины (<code>Long</code>) на вхождение в диапазон, определяемый атрибутами <code>minimum</code> и <code>maximum</code> .
<code><f:validateRegex></code>	Проверяет введенное значение на соответствие регулярному выражению, указанному в атрибуте <code>pattern</code> .
<code><f:validateRequired></code>	Проверяет заполнение обязательного поля. Действует подобно атрибуту <code>required</code> со значением <code>true</code> в родительском поле ввода.

Обратите внимание, что в описании тега `<f:validateBean>` упоминается проверка допустимости на стороне компонента. Запрос на спецификацию «Bean Validation JSR» ставит целью стандартизировать проверку допустимости компонентами JavaBean. Компоненты JavaBean используются в нескольких других API, которые до недавнего времени должны были реализовывать свою логику проверки допустимости. Включение поддержки стандарта Bean Validation в JSF 2.0 помогает организовать проверку свойств именованных компонентов.

Чтобы воспользоваться преимуществами проверки допустимости на стороне компонента, достаточно просто декорировать требуемое

поле соответствующей аннотацией без явного использования валидатора JSF.



Полный список аннотаций проверки допустимости на стороне компонента можно найти в пакете `javax.validation.constraints`, в описании Java EE 7 API: <http://docs.oracle.com/javaee/7/api/>.

Группировка компонентов

Следующий новый тег в примере выше – `<h:panelGroup>`. Как правило, он используется для группировки нескольких компонентов, чтобы они заняли одну ячейку в `<h:panelGrid>`. Этого можно добиться, добавив компоненты в тег `<h:panelGroup>` и включив его в тег `<h:panelGrid>`. Как показано в примере, экземпляр `<h:panelGroup>` не имеет дочерних компонентов. В данном случае тег `<h:panelGroup>` предназначен для получения «пустой» ячейки и, чтобы заставить компонент `<h:commandButton>` выровняться со всеми другими полями ввода в форме.

Отправка формы

Тег `<h:commandButton>` отображается в HTML-кнопку отправки формы (submit). Так же, как в случае со стандартной HTML-кнопкой, ее целью является отправка формы. Атрибут `value` просто определяет надпись на кнопке. Атрибут `action` используется для навигации. Следующая страница для перехода определяется из значения этого атрибута. Атрибут `action` может быть строковой константой или **выражением связывания метода (method binding expression)**, ссылающимся на метод в именованном компоненте, возвращающий строку.

Если в приложении имеется страница, базовое имя которой совпадает со значением атрибута `action` в теге `<h:commandButton>`, при нажатии кнопки выполняется переход к этой странице. Данная возможность JSF освобождает от необходимости определять правила навигации, что обязательно требовалось сделать в старых версиях JSF. В нашем примере страница подтверждения хранится в файле `confirmation.xhtml`. Поэтому, в соответствии с соглашением, после щелчка на кнопке будет выполнен переход на эту страницу, поскольку значение атрибута `action` кнопки совпадает с базовым именем страницы (`"confirmation"`).



Даже при том, что на кнопке имеется надпись **Save** (Сохранить), в нашем простом примере щелчок на кнопке фактически ничего не сохраняет.

Именованные компоненты

Существует два типа компонентов JavaBeans, способных взаимодействовать со страницами JSF: управляемые компоненты JSF и именованные компоненты CDI. Управляемые компоненты JSF появились давно, с первой версии спецификации JSF, и могли использоваться только в контексте а JSF. Именованные компоненты CDI были введены в Java EE 6 и могли взаимодействовать с другими Java EE API, такими как компоненты Enterprise JavaBeans. По этой причине предпочтение следует отдавать именованным компонентам CDI.

Чтобы создать Java-класс именованного компонента CDI, достаточно определить в этом классе общедоступный (`public`) конструктор без аргументов (он будет создан неявно, если в классе отсутствуют любые другие конструкторы, что как раз относится к нашему примеру) и декорировать класс аннотацией `@Named`. Ниже приводится фрагмент с определением именованного компонента CDI для нашего примера:

```
package net.ensode.glassfishbook.jsf;
import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

@Named
@RequestScoped
public class Customer {
    private String firstName;
    private String lastName;
    private String email;

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
```

```
    this.firstName = firstName;
}

public String getLastName() {
    return lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}
}
```

Аннотация `@Named` превращает данный класс в именованный компонент CDI. Эта аннотация имеет необязательный атрибут `value`, в котором можно определить логическое имя компонента для использования в JSF-страницах. Однако, в соответствии с соглашениями, по умолчанию этому атрибуту присваивается значение, совпадающее с именем класса (в данном случае `Customer`), первый символ которого преобразован в нижний регистр. В данном примере используется это поведение по умолчанию; поэтому доступ к свойствам компонента может осуществляться с использованием логического имени `customer`. Обратите внимание на атрибут `value` любого из полей ввода в странице примера, и вы увидите это логическое имя в действии.

Отметьте также, что кроме аннотаций `@Named` и `@RequestScoped` в этом компоненте нет ничего особенного. Это стандартный компонент JavaBean с приватными свойствами и соответствующими методами чтения/записи (`getter/setter`). Аннотация `@RequestScoped` указывает, что жизненный цикл компонента совпадает с жизненным циклом запроса.

Именованные компоненты всегда имеют определенный контекст. Контекст именованного компонента определяет продолжительность его жизни и задается аннотацией на уровне класса. В табл. 2.2 перечислены все допустимые контексты именованных компонентов.

Таблица 2.2. Контексты именованных компонентов

Аннотация контекста именованного компонента	Описание
<code>@ApplicationScoped</code>	Один и тот же экземпляр именованного компонента в контексте приложения доступен всем клиентам приложения. Если один из клиентов изменит значение свойства именованного компонента в контексте приложения, это изменение отразится на всех клиентах.

Аннотация контекста именованного компонента	Описание
@SessionScoped	В контексте сеанса для каждого клиента приложения создается свой экземпляр именованного компонента. Именованные компоненты в контексте сеанса используются для сохранения данных конкретного клиента между запросами.
@RequestScoped	Именованный компонент в контексте запроса «живет» только в течение одного HTTP-запроса.
@Dependent	Именованный компонент в зависимом контексте получает контекст родительского компонента, в который он внедряется. Этот контекст используется по умолчанию, если явно не задан никакой другой.
@ConversationScoped	Именованный компонент в контексте диалога может существовать на протяжении нескольких запросов, но этот период обычно короче периода, который охватывает контекст сеанса.

Навигация

Как видите, когда выполняется щелчок на кнопке **Save** (Сохранить), на странице `customer_data_entry.xhtml` приложение выполняет переход к странице `confirmation.xhtml`. Это происходит потому, что используется соглашение по конфигурации в JSF 2, в соответствии с которым, если значение атрибута `action` кнопки или ссылки соответствует базовому имени другой страницы, переход осуществляется к этой странице.



Щелчок на кнопке или ссылке, которая должна вызвать переход к другой странице, отображает ту же самую страницу? Если JSF не распознает значение атрибута `action` кнопки или ссылки, по умолчанию выполняется переход к той же самой странице, которая отображалась в браузере в момент щелчка на кнопке или ссылке, предназначеннной для перехода к другой странице.

Если кажется, что навигация не работает должным образом, скорее всего, в значение этого атрибута вкрадлась опечатка. Помните, что по соглашению JSF будет искать страницу, базовое имя которой совпадает со значением атрибута `action` кнопки или ссылки.

Ниже приводится исходный код страницы `confirmation.xhtml`:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!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://java.sun.com/jsf/html">
<h:head>
    <title>Customer Data Entered</title>
</h:head>
<h:body>
    <h:panelGrid columns="2" columnClasses="rightAlign, leftAlign">
        <h:outputText value="First Name:></h:outputText>
        <h:outputText value="#{customer.firstName}"></h:outputText>
        <h:outputText value="Last Name:></h:outputText>
        <h:outputText value="#{customer.lastName}"></h:outputText>
        <h:outputText value="Email:></h:outputText>
        <h:outputText value="#{customer.email}"></h:outputText>
    </h:panelGrid>
</h:body>
</html>
```

Тег `<h:outputText>` – единственный, который не был описан выше. Этот тег просто отображает значение своего атрибута `value`, которое может быть простой строкой или выражением привязки значения. Поскольку здесь используются те же выражения, что и в предыдущей странице, в тегах `<h:inputText>`, их значения будут соответствовать данным, которые ввел пользователь (см. рис. 2.6).

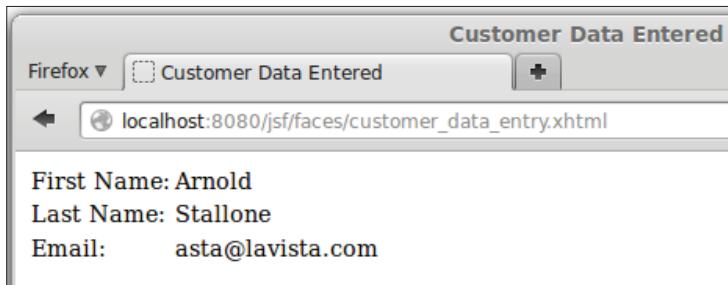


Рис. 2.6. Теги `<h:outputText>` отображают значения, введенные пользователем

В традиционных веб-приложениях на Java (не использующих JSF) определяются шаблоны URL, которые будут обрабатываться определенными сервлетами. Для JSF часто использовались суффиксы `.jsf` и `.faces`. Также часто для отображения адресов URL при использовании JSF применяется префикс `/faces`. По умолчанию GlassFish автоматически добавляет префикс `/faces` к фейс-сервлету. Поэтому можно вообще отказаться от определения отображений адресов URL. Если по какой-либо причине потребуется указать другое отображе-

ние, в приложение следует добавить конфигурационный файл `web.xml`. Однако соглашений по умолчанию вполне достаточно в большинстве случаев.

Адрес URL, использовавшийся для страниц приложения в примере, представляет собой имя страницы фейслета с префиксом `/faces`.

Пользовательская проверка допустимости данных

JSF не только предоставляет стандартные валидаторы для проверки допустимости, но и позволяет создавать свои, нестандартные валидаторы. Сделать это можно двумя способами: создать класс нестандартного валидатора или добавить методы проверки в именованные компоненты.

Создание нестандартных валидаторов

В дополнение к стандартным валидаторам JSF позволяет создавать нестандартные (пользовательские) валидаторы путем создания класса Java, реализующего интерфейс `javax.faces.validator.Validator`.

Следующий класс реализует проверку допустимости адреса электронной почты. Мы будем использовать его для проверки содержимого поля текстового ввода в странице ввода информации о клиентах:

```
package net.ensode.glassfishbook.jsfcustomval;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.html.HtmlInputText;
import javax.faces.context.FacesContext;
import javax.faces.validator.FacesValidator;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;
import org.apache.commons.lang.StringUtils;

@FacesValidator(value = "emailValidator")
public class EmailValidator implements Validator {

    @Override
    public void validate(FacesContext facesContext,
        UIComponent uiComponent,
        Object value) throws ValidatorException {
        org.apache.commons.validator.EmailValidator emailValidator =
            org.apache.commons.validator.EmailValidator.getInstance();
```

```
HtmlInputText htmlInputText = (HtmlInputText) uiComponent;

String email = (String) value;

if (!StringUtils.isEmpty(email)) {
    if (!emailValidator.isValid(email)) {
        FacesMessage facesMessage =
            new FacesMessage(htmlInputText.getLabel()
                + ": email format is not valid");
        throw new ValidatorException(facesMessage);
    }
}
}
```

Аннотация `@FacesValidator` регистрирует класс как нестандартный валидатор JSF. Значением атрибута `value` является логическое имя, которое страницы JSF могут использовать для обращения к валидатору.

Как видно из приведенного примера, единственным методом интерфейса `Validator`, который нужно реализовать, является метод `validate()`. Он принимает три параметра: экземпляр `javax.faces.context.FacesContext`, экземпляр `javax.faces.component.UIComponent` и объект. Как правило, разработчики приложений должны беспокоиться только о последних двух. Второй параметр является компонентом, данные которого проверяются; третий – фактическое значение. В примере `uiComponent` приводится к типу `javax.faces.component.html.HtmlInputText`. Благодаря этому открывается доступ к методу `getLabel()`, который можно использовать как часть сообщения об ошибке.

Если введенное значение не является допустимым адресом электронной почты, создается новый экземпляр `javax.faces.application.FacesMessage` для передачи сообщения об ошибке в параметре его конструктора, которое будет выведено в окне браузера. Затем возбуждается исключение `javax.faces.validator.ValidatorException`. После этого сообщение об ошибке будет отображено браузером.



Apache Commons Validator. Для фактической проверки допустимости предыдущий валидатор использует библиотеку Apache Commons Validator. Эта библиотека включает множество типовых функций проверки допустимости данных, таких как даты, номера кредитных карт, ISBN и адреса электронной почты. При реализации нестандартного валидатора вначале стоит посмотреть, нет ли в этой библиотеке подходящей функции.

Чтобы задействовать валидатор в странице, следует воспользоваться JSF-тегом `<f:validator>`. Ниже приводится страница фрейслета – модифицированная версия формы ввода данных о клиентах. Эта версия проверяет содержимое поля ввода адреса электронной почты с помощью тега `<f:validator>`:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!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://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<h:head>
    <title>Enter Customer Data</title>
</h:head>
<h:body>
    <h:outputStylesheet library="css" name="styles.css"/>
    <h:form>
        <h:messages></h:messages>
        <h:panelGrid columns="2"
                     columnClasses="rightAlign, leftAlign">
            <h:outputText value="First Name:>">
            </h:outputText>
            <h:inputText label="First Name"
                         value="#{customer.firstName}"
                         required="true">
                <f:validateLength minimum="2" maximum="30">
                </f:validateLength>
            </h:inputText>
            <h:outputText value="Last Name:>"></h:outputText>
            <h:inputText label="Last Name"
                         value="#{customer.lastName}"
                         required="true">
                <f:validateLength minimum="2" maximum="30">
                </f:validateLength>
            </h:inputText>
            <h:outputText value="Email:>">
            </h:outputText>
            <h:inputText label="Email" value="#{customer.email}">
                <f:validator validatorId="emailValidator" />
            </h:inputText>
            <h:panelGroup></h:panelGroup>
            <h:commandButton action="confirmation" value="Save">
            </h:commandButton>
        </h:panelGrid>
    </h:form>
</h:body>
</html>
```

Обратите внимание, что значение атрибута `validatorId` в теге `<f:validator>` совпадает со значением атрибута `value` в аннотации `@FacesValidator`, декорирующей класс нестандартного валидатора.

После реализации нестандартного валидатора и изменения страницы для использования его возможностей можно посмотреть, как действует наш валидатор (см. рис. 2.7).

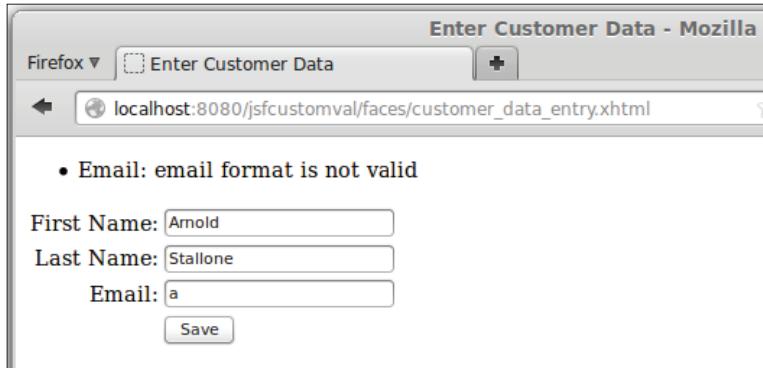


Рис. 2.7. Нестандартный валидатор в действии

Методы валидатора

Другой способ реализовать пользовательскую проверку допустимости – добавить методы проверки в один или более именованных компонентов приложения. Следующий Java-класс иллюстрирует реализацию таких методов:

```
package net.ensode.glassfishbook.jsfcustomval;

import javax.enterprise.context.RequestScoped;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.html.HtmlInputText;
import javax.faces.context.FacesContext;
import javax.faces.validator.ValidatorException;
import javax.inject.Named;

import org.apache.commons.lang.StringUtils;

@Named
@RequestScoped
public class AlphaValidator {

    public void validateAlpha(FacesContext facesContext,
        UIComponent uiComponent,
        Object value) throws ValidatorException {
        if (!StringUtils.isAlphaSpace((String) value)) {
            HtmlInputText htmlInputText = (HtmlInputText) uiComponent;
```

```
        FacesMessage facesMessage =
            new FacesMessage(htmlInputText.getLabel()
                + ": only alphabetic characters are allowed.");
        throw new ValidatorException(facesMessage);
    }
}
}
```

Класс в этом примере содержит только метод проверки. Этому методу можно дать любое имя, по своему усмотрению, однако он должен возвращать пустое значение. Кроме того, три входных параметра, показанные в примере, должны передаваться в указанном порядке. Иными словами, кроме имени, сигнатура метода должна быть идентична сигнатуре метода `validate()`, объявленного в интерфейсе `javax.faces.validator.Validator`.

Тело этого метода почти идентично телу метода `validate()` в нестандартном валидаторе, демонстрировавшемся выше. В нем проверяется значение, введенное пользователем, чтобы убедиться, что оно содержит только буквенные символы и/или пробелы. Если это не так, возбуждается исключение `ValidatorException`, с передачей экземпляра `FacesMessage`, содержащего строку сообщения об ошибке.



StringUtils. Фактическая проверка в примере выполняется валидатором org.apache.commons.lang.StringUtils. В дополнение к методу, использованному в примере, данный класс содержит еще несколько методов, позволяющих проверить: содержит ли строка только цифровые или алфавитно-цифровые символы. Этот класс, являющийся частью библиотеки Apache commons-lang, очень пригодится вам при создании собственных валидаторов.

Поскольку каждый метод проверки должен находиться в именованном компоненте, класс с эти методом следует декорировать аннотацией `@Named`, как в данном примере.

Последнее, что необходимо сделать для использования метода проверки, – связать его с компонентом посредством атрибута `validator`, как показано ниже:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!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://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<h:head>
    <title>Enter Customer Data</title>
```

```
</h:head>

<h:body>
    <h:outputStylesheet library="css" name="styles.css"/>
    <h:form>
        <h:messages></h:messages>
        <h:panelGrid columns="2"
            columnClasses="rightAlign, leftAlign">
            <h:outputText value="First Name:>
            </h:outputText>
            <h:inputText label="First Name"
                value="#{customer.firstName}"
                required="true"
                validator="#{alphaValidator.validateAlpha}">
                <f:validateLength minimum="2" maximum="30">
                </f:validateLength>
            </h:inputText>
            <h:outputText value="Last Name:></h:outputText>
            <h:inputText label="Last Name"
                value="#{customer.lastName}"
                required="true"
                validator="#{alphaValidator.validateAlpha}">
                <f:validateLength minimum="2" maximum="30">
                </f:validateLength>
            </h:inputText>
            <h:outputText value="Email:>
            </h:outputText>
            <h:inputText label="Email" value="#{customer.email}">
                <f:validateLength minimum="3" maximum="30">
                </f:validateLength>
                <f:validator validatorId="emailValidator" />
            </h:inputText>
            <h:panelGroup></h:panelGroup>
            <h:commandButton action="confirmation" value="Save">
            </h:commandButton>
        </h:panelGrid>
    </h:form>
</h:body>
</html>
```

Ни поле **First Name** (Имя), ни поле **Last Name** (Фамилия) не должны принимать что-либо кроме буквенных символов или пробелов, поскольку к ним был добавлен наш метод проверки.

Обратите внимание, что значение атрибута `validator` в теге `<h:inputText>` содержит инструкцию на языке выражений JSF, использующим имя по умолчанию именованного компонента, содержащего метод проверки. Компонент имеет имя `alphaValidator`, а метод называется `validateAlpha`.

После изменения страницы и включения в нее нестандартного метода проверки, можно посмотреть, как он действует (см. рис. 2.8).

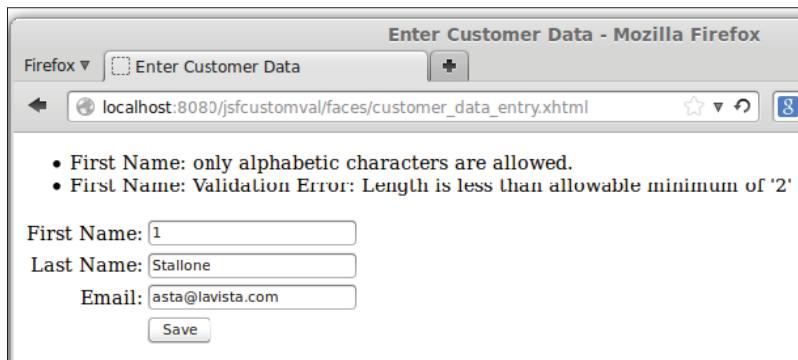


Рис. 2.8. Нестандартный метод проверки в действии

Обратите внимание, что для поля **First Name** (Имя) были выведены: и сообщение от нестандартного валидатора, и сообщение от валидатора, проверяющего допустимость длины.

Преимущество методов проверки заключается в отсутствии издержек, возникающих при создании целого класса только ради единственного метода (в данном примере делается только это, но во многих случаях методы проверки добавляются в существующие именованные компоненты, содержащие другие методы). Тем не менее такой подход имеет свой недостаток: для проверки любого компонента можно использовать только один метод проверки. При использовании классов валидаторов можно вложить несколько тегов `<f:validator>` в тег, который требует проверки, поэтому для поля может быть выполнено несколько проверок допустимости – как пользовательских, так и стандартных.

Настройка сообщений JSF по умолчанию

Как упоминалось в предыдущем разделе, есть возможность настроить стиль (шрифт, цвет и т. д.) сообщений JSF, используемых по умолчанию. Кроме того, можно изменить текст таких сообщений. В следующих разделах мы посмотрим, как изменить текст сообщения об ошибке и его форматирование.

Настройка стилей сообщения

Настройку стилей сообщения можно выполнить через каскадные таблицы стилей (Cascading Style Sheets, CSS), посредством атрибутов `style` или `styleClass` тега `<h:message>`. Атрибут `style` используется, когда нужно определить встроенный стиль CSS, а атрибут `styleClass` – когда используется предопределенный стиль в таблице стилей CSS или в теге `<style>`.

Следующая разметка поясняет использование атрибута `styleClass` для изменения стиля сообщений об ошибках. Это модифицированная версия формы, представленной в предыдущем разделе:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!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://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core">
<h:head>
  <title>Enter Customer Data</title>
</h:head>

<h:body>
  <h:outputStylesheet library="css" name="styles.css" />
  <h:form>
    <h:messages styleClass="errorMsg"></h:messages>
    <h:panelGrid columns="2"
      columnClasses="rightAlign, leftAlign">
      <h:outputText value="First Name:>">
      </h:outputText>
      <h:inputText label="First Name"
        value="#{customer.firstName}"
        required="true"
        validator="#{alphaValidator.validateAlpha}">
        <f:validateLength minimum="2" maximum="30">
        </f:validateLength>
      </h:inputText>
      <h:outputText value="Last Name:>"></h:outputText>
      <h:inputText label="Last Name"
        value="#{customer.lastName}"
        required="true"
        validator="#{alphaValidator.validateAlpha}">
        <f:validateLength minimum="2" maximum="30">
        </f:validateLength>
      </h:inputText>
      <h:outputText value="Email:>">
      </h:outputText>
      <h:inputText label="Email" value="#{customer.email}">
```

```

<f:validator validatorId="emailValidator" />
</h:inputText>
<h:panelGroup></h:panelGroup>
<h:commandButton action="confirmation" value="Save">
</h:commandButton>
</h:panelGrid>
</h:form>
</h:body>
</html>

```

Единственное отличие этой страницы от предыдущей состоит в использовании атрибута `styleClass` в теге `<h:messages>`. Как отмечалось выше, значение атрибута `styleClass` должно совпадать с именем стиля CSS в каскадной таблице стилей, к которой обращается страница.

В нашем случае мы определили стиль CSS в файле `style.css`, как показано ниже:

```

.errorMsg
{
    color: red;
}

```

Затем использовали этот стиль в качестве значения атрибута `styleClass` в теге `<h:messages>`.

На рис. 2.9 показано, как выглядят сообщения об ошибках после реализации этого изменения.

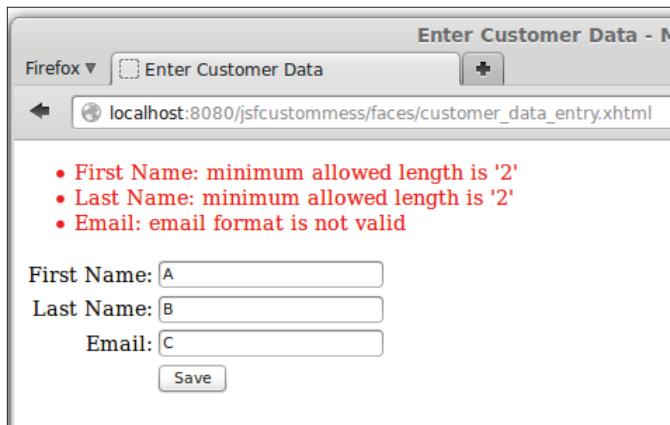


Рис. 2.9. Сообщения об ошибках с измененным оформлением

В данном случае изменился только цвет текста, с черного на красный, но на самом деле в оформлении сообщений мы ограничены только возможностями CSS.

Изменение текста сообщения

Иногда желательно переопределить текст сообщений об ошибках, выдаваемых по умолчанию. Текст этих сообщений определяется в пакете ресурса `Messages.properties`. Данный файл можно найти в архиве `javax.faces.jar`, который в свою очередь хранится в каталоге [каталог установки Glassfish]/glassfish/modules. Файл `Messages.properties` находится внутри JAR-файла `javax.faces.jar`, в его каталоге `javax/faces`. Он содержит несколько сообщений, но нас интересуют только сообщения об ошибках проверки допустимости. По умолчанию они определяются, как показано ниже:

```
javax.faces.validator.DoubleRangeValidator.MAXIMUM={1}: Validation
    Error: Value is greater than allowable maximum of "{0}"
javax.faces.validator.DoubleRangeValidator.MINIMUM={1}: Validation
    Error: Value is less than allowable minimum of "{0}"
javax.faces.validator.DoubleRangeValidator.NOT_IN_RANGE={2}:
    Validation Error: Specified attribute is not between the
        expected values of {0} and {1}.
javax.faces.validator.DoubleRangeValidator.TYPE={0}: Validation
    Error: Value is not of the correct type
javax.faces.validator.LengthValidator.MAXIMUM={1}: Validation
    Error: Value is greater than allowable maximum of "{0}"
javax.faces.validator.LengthValidator.MINIMUM={1}: Validation
    Error: Value is less than allowable minimum of "{0}"
javax.faces.validator.LongRangeValidator.MAXIMUM={1}: Validation
    Error: Value is greater than allowable maximum of "{0}"
javax.faces.validator.LongRangeValidator.MINIMUM={1}: Validation
    Error: Value is less than allowable minimum of "{0}"
javax.faces.validator.LongRangeValidator.NOT_IN_RANGE={2}:
    Validation Error: Specified attribute is not between the
        expected values of {0} and {1}.
javax.faces.validator.LongRangeValidator.TYPE={0}: Validation
    Error: Value is not of the correct type.
javax.faces.validator.NOT_IN_RANGE=Validation Error: Specified
    attribute is not between the expected values of {0} and {1}.
javax.faces.validator.RegexValidator.PATTERN_NOT_SET=Regex pattern
    must be set.
javax.faces.validator.RegexValidator.PATTERN_NOT_SET_detail=Regex
    pattern must be set to non-empty value.
javax.faces.validator.RegexValidator.NOT_MATCHED=Regex Pattern not
    matched
javax.faces.validator.RegexValidator.NOT_MATCHED_detail=Regex
    pattern of '{0}' not matched
javax.faces.validator.RegexValidator.MATCH_EXCEPTION=Error in
    regular expression.
javax.faces.validator.RegexValidator.MATCH_EXCEPTION_detail=Error
    in regular expression, "{0}"
javax.faces.validator.BeanValidator.MESSAGE={0}
```

Чтобы переопределить сообщения об ошибках по умолчанию, следует создать собственный пакет ресурса, используя те же ключи, что используются по умолчанию, но при этом изменить значения на такие, которые удовлетворяют нашим потребностям. Далее показан очень простой специализированный пакет ресурса для нашего приложения:

```
javax.faces.validator.LengthValidator.MINIMUM={1}: minimum allowed  
length is "{0}"
```

В этом пакете ресурса переопределяется сообщение об ошибке для случая, когда значение, проверяемое тегом <f:validateLength>, окажется короче допустимого минимума. Чтобы сообщить приложению, что у нас есть пользовательский пакет ресурса для свойств сообщений, нужно отредактировать файл конфигурации приложения faces-config.xml:

```
<?xml version='1.0' encoding='UTF-8'?>  
<faces-config version="2.0"  
    xmlns="http://java.sun.com/xml/ns/javaee"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  
        http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd">  
    <application>  
        <message-bundle>net.ensode.Messages</message-bundle>  
    </application>  
</faces-config>
```

Как видите, достаточно всего лишь добавить в файл конфигурации приложения faces-config.xml элемент <message-bundle>, определяющий название и местоположение пакета ресурса, содержащего пользовательские сообщения.



Примечание
Определение пользовательских сообщений об ошибках – один из немногих случаев, когда все еще необходимо создавать файл faces-config.xml для JSF-приложения. Однако заметьте, насколько просто файл faces-config.xml. Он сильно отличается от типично го файла faces-config.xml для JSF 1.x, который обычно содержит определения именованных компонентов, правила навигации, определения валидаторов JSF и т. д.

После добавления пользовательского пакета ресурса сообщений и изменения конфигурационного файла faces-config.xml можно посмотреть, как действуют пользовательские сообщения об ошибках (см. рис. 2.10).

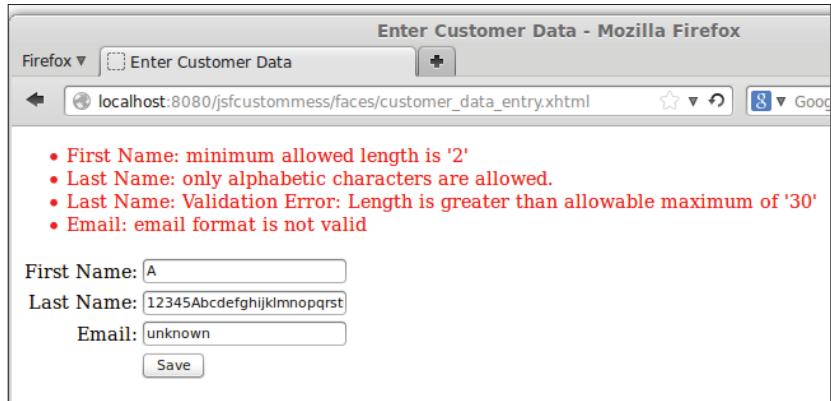


Рис. 2.10. Пользовательские сообщения об ошибках

Как видно на рис. 2.10, для сообщений, которые не были переопределены, выводится текст по умолчанию. В нашем пакете ресурса было переопределено только сообщение об ошибке проверки минимальной длины, поэтому для поля **First Name** (Имя) показано пользовательское сообщение. Поскольку никаких других сообщений не было переопределено, для всех других ошибок были выведены сообщения с текстом по умолчанию. Валидатор адресов электронной почты – это наш нестандартный валидатор, созданный нами выше в этой главе. Поскольку это пользовательский валидатор, на его сообщения об ошибке наш пакет ресурса влияния не оказывает.

Поддержка Ajax в JSF-приложениях

В ранних версиях JSF отсутствовала собственная поддержка Ajax. Производители библиотек для JSF были вынуждены реализовать поддержку Ajax по-своему. К сожалению, такое положение дел приводило к несовместимости между библиотеками компонентов JSF. В JSF 2.0 поддержка Ajax была стандартизована введением нового тега `<f:ajax>`.

Следующий код демонстрирует типичное использование тега `<f:ajax>`:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!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://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<h:head>
    <title>JSF Ajax Demo</title>
</h:head>
<h:body>
    <h2>JSF Ajax Demo</h2>
    <h:form>
        <h:messages/>
        <h:panelGrid columns="2">

            <h:outputText value="Echo input:"/>
            <h:inputText id="textInput" value="#{controller.text}">
                <f:ajax render="textVal" event="keyup"/>
            </h:inputText>

            <h:outputText value="Echo output:"/>
            <h:outputText id="textVal" value="#{controller.text}" />
        </h:panelGrid>
        <hr/>
        <h:panelGrid columns="2">
            <h:panelGroup/>
            <h:panelGroup/>
            <h:outputText value="First Operand:"/>
            <h:inputText id="first" value="#{controller.firstOperand}"
                         size="3"/>
            <h:outputText value="Second Operand:"/>
            <h:inputText id="second"
                         value="#{controller.secondOperand}"
                         size="3"/>
            <h:outputText value="Total:"/>
            <h:outputText id="sum" value="#{controller.total}" />
            <h:commandButton
                actionListener="#{controller.calculateTotal}"
                value="Calculate Sum">
                <f:ajax execute="first second" render="sum"/>
            </h:commandButton>
        </h:panelGrid>
    </h:form>
</h:body>
</html>

```

После развертывания приложения этот код будет отображаться в окне браузера, как показано на рис. 2.11.

Этот пример страницы иллюстрирует два случая использования тега `<f:ajax>`. В начале страницы этот тег используется для реализации типичного примера «Ajax-эхо», в котором имеется компонент

<h:outputText>, обновляющий себя значением из поля ввода. Всякий раз, когда в поле ввода вводится символ, значение компонента <h:outputText> автоматически обновляется.

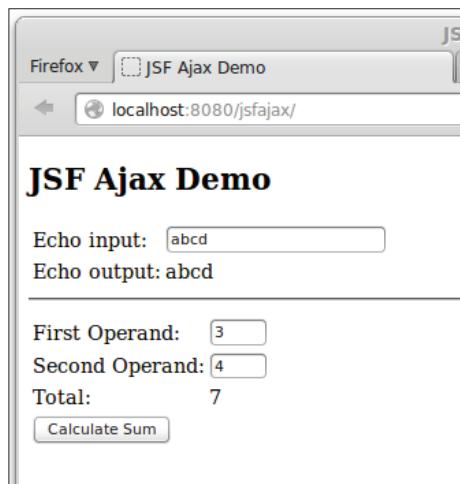


Рис. 2.11. Страница с поддержкой Ajax

Чтобы реализовать описанную функциональность, мы поместили тег <f:ajax> в тег <h:inputText>. Значение атрибута render в теге <f:ajax> должно соответствовать атрибуту id компонента, который требуется обновить после завершения запроса Ajax. В данном конкретном примере требуется обновить компонент <h:outputText> с идентификатором "textVal", поэтому данное значение используется в атрибуте render тега <f:ajax>.



В некоторых случаях может понадобиться отобразить данные более чем в одном компоненте JSF. Для этого можно добавить в атрибут render несколько идентификаторов, перечислив их через пробел.

Второй атрибут тега <f:ajax>, использованный в этом экземпляре, – это атрибут event. Он определяет событие JavaScript, которое влечет за собой событие Ajax. В данном случае событие Ajax должно инициироваться каждым событием отпускания клавиши во время ввода данных. Поэтому было выбрано событие keyup.

В табл. 2.3 перечислены все поддерживаемые события JavaScript.

Таблица 2.3. Поддерживаемые события JavaScript

Событие	Описание
blur	Компонент теряет фокус ввода.
change	Компонент теряет фокус ввода и его значение изменяется.
click	Щелчок мышью на компоненте.
dblclick	Двойной щелчок мышью на компоненте.
focus	Компонент получил фокус ввода.
keydown	Нажата клавиша, когда компонент владел фокусом ввода.
keypress	Нажата и удерживается клавиша, когда компонент владел фокусом ввода.
keyup	Отпущена клавиша, когда компонент владел фокусом ввода.
mousedown	Нажата кнопка мыши, когда компонент владел фокусом ввода.
mousemove	Перемещение указателя мыши над компонентом.
mouseout	Указатель мыши вышел за границы компонента.
mouseover	Указатель мыши находится над компонентом.
mouseup	Отпущена кнопка мыши, когда компонент владел фокусом ввода.
select	Выделен текст в компоненте.
valueChange	Эквивалентно событию change, компонент теряет фокус ввода и его значение изменяется.

Второй раз тег `<f:ajax>` используется ниже, с целью включения поддержки Ajax для компонента кнопки. В этом случае требуется повторно вычислить значение на основе значений двух компонентов ввода. Чтобы получить значение, обновленное на сервере последними данными, введенными пользователем, мы использовали атрибут `execute` тела `<f:ajax>`. Этот атрибут принимает список идентификаторов компонентов ввода, разделенных пробелами. Также мы использовали атрибут `render`, как и выше, чтобы определить, какие компоненты должны быть повторно перерисованы после завершения запроса Ajax.

Обратите внимание, что мы использовали атрибут `actionListener` в теге `<h:commandButton>`. Этот атрибут обычно используется, когда нужно избежать перемещения к другой странице после нажатия кнопки. Значением для этого атрибута является метод-обработчик, который мы реализовали в одном из именованных компонентов. Методы-обработчики не должны ничего возвращать (имеют тип `void`)

и принимают экземпляр javax.faces.event.ActionEvent в единственном параметре.

Ниже приводится реализация именованного компонента для нашего приложения:

```
package net.ensode.glassfishbook.jsfajax;

import javax.faces.event.ActionEvent;
import javax.faces.view.ViewScoped;
import javax.inject.Named;

@Named
@ViewScoped
public class Controller {

    private String text;
    private int firstOperand;
    private int secondOperand;
    private int total;

    public Controller() {
    }

    public void calculateTotal(ActionEvent actionEvent) {
        total = firstOperand + secondOperand;
    }

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }

    public int getFirstOperand() {
        return firstOperand;
    }

    public void setFirstOperand(int firstOperand) {
        this.firstOperand = firstOperand;
    }

    public int getSecondOperand() {
        return secondOperand;
    }

    public void setSecondOperand(int secondOperand) {
        this.secondOperand = secondOperand;
    }
}
```

```
}

public int getTotal() {
    return total;
}

public void setTotal(int total) {
    this.total = total;
}
}
```

Обратите внимание, что в именованном компоненте не нужно делать ничего особенного для поддержки Ajax в приложении. Все это выполняется тегами <f:ajax>, размещенными на странице.

Как показано в данном примере, поддержка Ajax в приложениях JSF не требует реализации сложных действий. Достаточно лишь использовать единственный тег для включения поддержки Ajax в странице, без необходимости писать какой-либо код на JavaScript, JSON или XML.

Поддержка HTML5 в JSF 2.2

HTML 5 – новейшая версия спецификации HTML. Она включает множество улучшений, по сравнению с предыдущей версией HTML. JSF 2.2 также включает ряд усовершенствований, обеспечивающих работу JSF-страниц с разметкой HTML5.

HTML5-совместимая разметка

Используя сквозные элементы (pass-through elements), можно создавать страницы с применением тегов HTML 5 и интерпретировать их как компоненты JSF. Для этого нужно определить хотя бы один атрибут из пространства имен `http://xmlns.jcp.org/jsf`, как показано в следующем примере:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:jsf="http://xmlns.jcp.org/jsf">
<head jsf:id="head">
    <title>JSF Page with HTML5 Markup</title>
    <link jsf:library="css" jsf:name="styles.css"
          rel="stylesheet"
          type="text/css"
          href="resources/css/styles.css"/>
</head>
```

```
<body jsf:id="body">
    <form jsf:prependId="false">
        <table style="border-spacing: 0; border-collapse: collapse">
            <tr>
                <td class="rightAlign">
                    <label jsf:for="firstName">First Name</label>
                </td>
                <td class="leftAlign">
                    <input type="text" jsf:id="firstName"
                           jsf:value="#{customer.firstName}"/>
                </td>
            </tr>
            <tr>
                <td class="rightAlign">
                    <label jsf:for="lastName">Last Name</label>
                </td>
                <td class="leftAlign">
                    <input type="text" jsf:id="lastName"
                           jsf:value="#{customer.lastName}"/>
                </td>
            </tr>
            <tr>
                <td class="rightAlign">
                    <label jsf:for="email">Email Address</label>
                </td>
                <td class="leftAlign">
                    <input type="email" jsf:id="email"
                           jsf:value="#{customer.email}"/></td>
                </td>
            </tr>
            <tr>
                <td></td>
                <td>
                    <input type="submit"
                           jsf:action="confirmation"
                           value="Submit"/>
                </td>
            </tr>
        </table>
    </form>
</body>
</html>
```

Первое, на что следует обратить внимание в этом примере, – определение префикса `jsf` для пространства имен XML. Это пространство имен позволяет использовать атрибуты JSF в страницах с разметкой HTML 5. Когда механизм времени выполнения JSF встречает атрибут с префиксом `jsf` в любом из тегов в странице, он автоматически преобразует тег HTML5 в эквивалентный компонент JSF. Теги JSF остались прежними, что и в обычных JSF-страницах, за исключени-

ем того, что теперь они снабжаются префиксом `jsf`. С этого момента назначение таких атрибутов должно быть очевидно, поэтому они не будут обсуждаться далее во всех деталях. Страница в примере выше выглядит и действует точно так же, как первый пример в этой главе.

Прием, описываемый в этом разделе, пригодится тем, кто имеет большой опыт верстки HTML-страниц и предпочитает иметь полный контроль над внешним видом страницы. Страницы разрабатываются с использованием стандартного языка разметки HTML5 и атрибутов JSF, чтобы механизм времени выполнения JSF мог управлять пользовательским вводом.

Если в вашем коллективе подобрались программисты на Java с ограниченными навыками использования CSS/HTML, тогда для вас предпочтительнее создавать веб-страницы с использованием обычных компонентов JSF. В HTML 5 появилось несколько новых атрибутов. По этой причине в JSF 2.2 была введена возможность добавлять произвольные атрибуты в компоненты JSF. Этот прием интеграции JSF/HTML5 обсуждается в следующем разделе.

Сквозные элементы

JSF 2.2 позволяет определять произвольные атрибуты (не обрабатываемые механизмом JSF). Эти атрибуты просто переносятся «как есть» в генерированную разметку HTML и обрабатываются браузером. Ниже приводится новая версия примера, созданного в начале главы, где используются сквозные элементы HTML5:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!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://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:p="http://xmlns.jcp.org/jsf/passthrough">
  <h:head>
    <title>Enter Customer Data</title>
  </h:head>
  <h:body>
    <h:outputStylesheet library="css" name="styles.css"/>
    <h:form id="customerForm">
      <h:messages/>
      <h:panelGrid columns="2"
                   columnClasses="rightAlign, leftAlign">
        <h:outputLabel for="firstName" value="First Name:>
        </h:outputLabel>
        <h:inputText id="firstName"
```

```
        label="First Name"
        value="#{customer.firstName}"
        required="true"
        p:placeholder="First Name">
    <f:validateLength minimum="2" maximum="30">
    </f:validateLength>
</h:inputText>
<h:outputLabel for="lastName" value="Last Name :">
</h:outputLabel>
<h:inputText id="lastName"
            label="Last Name"
            value="#{customer.lastName}"
            required="true"
            p:placeholder="Last Name">
    <f:validateLength minimum="2" maximum="30">
    </f:validateLength>
</h:inputText>
<h:outputLabel for="email" value="Email :">
</h:outputLabel>
<h:inputText id="email"
            label="Email"
            value="#{customer.email}"
            p:placeholder="Email Address">
    <f:validateLength minimum="3" maximum="30">
    </f:validateLength>
</h:inputText>
<h:panelGroup></h:panelGroup>
<h:commandButton action="confirmation" value="Save">
</h:commandButton>
</h:panelGrid>
</h:form>
</h:body>
</html>
```

Прежде всего обратите внимание на подключение нового для нас пространства имен `xmlns:p="http://xmlns.jcp.org/jsf/passthrough"` – это позволяет добавлять любые атрибуты в компоненты JSF.

В этом примере во все текстовые поля ввода был добавлен HTML5-атрибут `placeholder`. Как показано в примере, его необходимо предварять префиксом для пространства имен, который был определен в начале страницы (в данном случае `p`). Атрибут `placeholder` просто добавляет некоторый текст-заполнитель в поле ввода, который автоматически стирается, как только пользователь начинает ввод (до появления поддержки в HTML5, этот прием часто реализовывался «вручную» на JavaScript).

На рис. 2.12 показано, как выглядит новая страница.

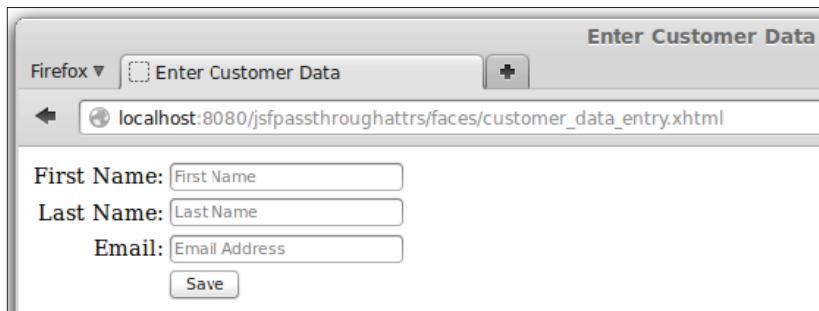


Рис. 2.12. Новая страница приложения, использующая сквозные элементы

JSF 2.2 Faces Flows

Faces Flows – новая особенность JSF 2.2, позволяющая определит контекст, распространяющийся на несколько страниц. Когда пользователь входит в последовательность Faces Flows¹, создается компонент с контекстом потока, который уничтожается сразу после выхода из последовательности.

В Faces Flows используется тот же принцип программирования по соглашениям, что и в JSF. Ниже перечислены основные соглашения, которые обычно используются при разработке последовательностей Faces Flows:

- все страницы, относящиеся к последовательности, должны находиться в каталоге с именем, совпадающим с именем последовательности;
- в каталоге со страницами последовательности должен находиться конфигурационный файл XML с именем, состоящим из имени каталога и окончания `-flow` (файл может быть пустым, но должен существовать);
- первая страница в последовательности должна иметь имя, совпадающее с именем вмещающего каталога;
- последняя страница в последовательности *не* должна находиться в каталоге с последовательностью и должна иметь имя, составленное из имени каталога и окончания `-return`.

¹ «Flow» переводится в данном случае как поток/последовательность операций/действий/этапов/страниц, выполняемых/отображаемых в определенном порядке. Примером такого потока (или последовательности) может служить мастер настройки чего-либо, выполняющийся в несколько этапов. – *Прим. перев.*

Все эти соглашения иллюстрирует скриншот на рис. 2.13.

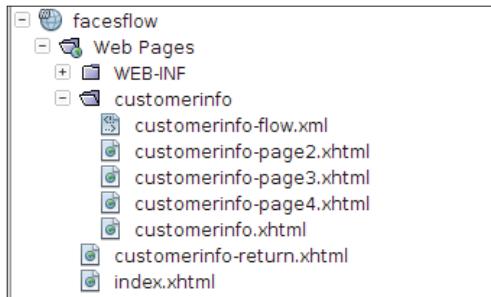


Рис. 2.13. Иллюстрация соглашений
о последовательностях Faces Flows

В данном примере определена последовательность с именем `customerinfo`. По соглашениям файлы находятся в каталоге с именем `customerinfo`, а первая страница последовательности названа как `customerinfo.xhtml` (на имена других страниц в последовательности не накладывается никаких ограничений). Когда происходит выход из последовательности, осуществляется переход к странице `customerinfo-return.xhtml`, которая также названа в соответствии с соглашениями.

В разметке страниц нет ничего такого, чего бы мы не видели прежде, поэтому я не буду показывать ее здесь. Все примеры кода можно найти в пакете с примерами для книги.

Все страницы в примере сохраняют данные в именованном компоненте `Customer`, имеющем контекст потока.

```
@Named
@FlowScoped("customerinfo")
public class Customer implements Serializable {
    // реализация класса опущена
}
```

Аннотация `@FlowScoped` имеет атрибут `value`, значение которого должно совпадать с именем последовательности, обслуживаемой данным компонентом (`customerinfo` в данном примере).

Этот пример реализует мастера, состоящего из нескольких страниц, в которых пользователь последовательно вводит свои данные.

На первой странице вводится имя клиента, как показано на рис. 2.14.

The screenshot shows a Firefox browser window with the title bar "Customer Information". The address bar displays the URL "localhost:8080/facesflow/faces/index.xhtml?jfwid=f589457". The main content area has a title "Enter Customer Information (Page 1 of 4)". It contains three text input fields: "First Name" with value "Piper", "Middle Name" with value "Nicole", and "Last Name" with value "Vause". Below these fields is a "Next" button.

Рис. 2.14. Первая страница последовательности реализует ввод имени

Вторая страница предоставляет возможность ввести адрес, как показано на рис. 2.15.

The screenshot shows a Firefox browser window with the title bar "Customer Information". The address bar displays the URL "localhost:8080/facesflow/faces/customerinfo/customerinfo.xhtml". The main content area has a title "Enter Customer Information (Page 2 of 4)". It contains five text input fields: "Line 1" with value "123 Basketball Ct", "Line 2" (empty), "City" with value "Litchfield", "State" with value "California" (selected from a dropdown menu), and "Zip" with value "12345". Below these fields are "Previous" and "Next" buttons.

Рис. 2.15. Вторая страница последовательности реализует ввод адреса

Следующая страница позволяет ввести номер телефона, как показано на рис. 2.16.

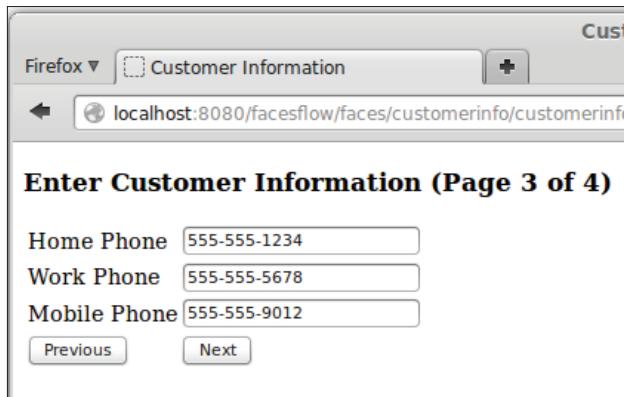


Рис. 2.16. Третья страница последовательности реализует ввод номера телефона

И в заключение отображается страница подтверждения, как показано на рис. 2.17.

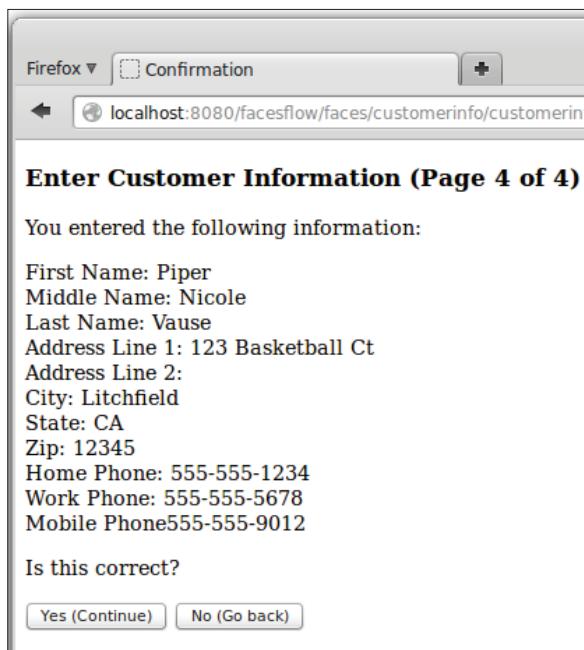


Рис. 2.17. Заключительная страница последовательности запрашивает подтверждение

Библиотеки дополнительных компонентов JSF

В дополнение к библиотекам стандартных компонентов JSF существует множество сторонних библиотек тегов JSF. В табл. 2.4 перечислены наиболее популярные из них.

Таблица 2.4. Библиотеки дополнительных компонентов JSF

Библиотека	Производитель	Лицензия	URL
ICEfaces	ICEsoft	MPL 1.1	http://www.icefaces.org
RichFaces	Red Hat/JBoss	LGPL	http://www.jboss.org/richfaces
Primefaces	Prime Technology	Apache 2.0	http://www.primefaces.org

Резюме

В этой главе мы узнали, как разрабатывать веб-приложения с использованием технологии JavaServer Faces, фреймворка стандартных компонентов для платформы Java EE. Было показано, как написать простое приложение путем создания страниц с применением фейслетов, как технологии отображения, и именованных компонентов CDI. Мы также обсудили, как проверить введенные пользователем данные, используя стандартные или наши собственные валидаторы JSF, а также как писать методы проверки допустимости. Речь шла и о том, как настроить стандартные сообщения об ошибках JSF – в частности, изменить текст и стиль сообщений (шрифт, цвет и т. д.). Наконец, мы рассмотрели, как писать JSF-страницы с поддержкой Ajax и как интегрировать JSF и HTML5.

В следующей главе рассказывается об организации взаимодействий с реляционными базами данных посредством Java Persistence API.



ГЛАВА 3.

Объектно-реляционное отображение в JPA

Любое нетривиальное приложение Java EE сохраняет данные в реляционной базе данных. В этой главе мы рассмотрим, как установить соединение с базой данных и выполнить операции CRUD (Create (Создать), Read (Прочитать), Update (Изменить), Delete (Удалить)).

Java Persistence API (JPA) – это стандартный инструмент **объектно реляционного отображения** для Java EE (**Object Relational Mapping, ORM**). В данной главе мы подробно обсудим этот API.

Вот неполный перечень тем, которые будут затронуты в этой главе:

- ❖ извлечение данных из базы данных с помощью JPA;
- ❖ вставка данных в базу данных с помощью JPA;
- ❖ изменение данных в базе данных с помощью JPA;
- ❖ удаление данных из базы данных с помощью JPA;
- ❖ создание программных запросов с помощью API критериев JPA (JPA Criteria API);
- ❖ автоматизация проверки данных с помощью JPA 2.0, с поддержкой проверки на стороне компонента.

База данных CUSTOMERDB

В примерах этой главы будет использоваться база данных CUSTOMERDB. Она содержит таблицы для хранения информации о заказчиках и заказах вымышленного магазина. В качестве своей СУРБД (RDBMS) используется JavaDB, поскольку она поставляется в комплекте с сервером GlassFish.

Сценарий создания базы данных включен в пакет примеров для этой книги; он создает базу данных и производит предварительное заполнение некоторых ее таблиц. Помимо этого, в пакет примеров включены инструкции по выполнению сценария, а также по добавле-

нию пула соединений и источника данных, обеспечивающих доступ к базе данных. Схема базы данных CUSTOMERDB изображена на рис. 3.1.

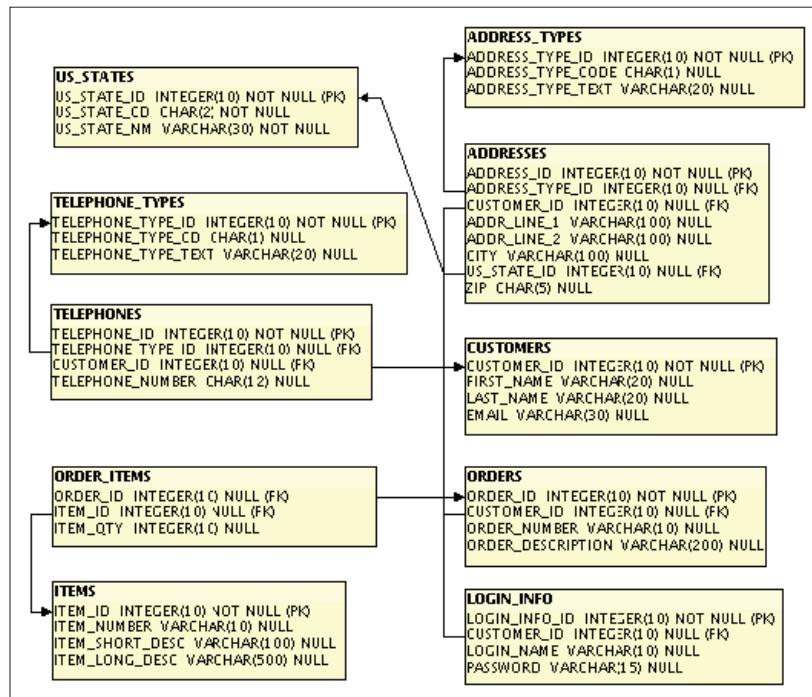


Рис. 3.1. Схема базы данных CUSTOMERDB

Как видно на рис. 3.1, база данных содержит таблицы для хранения информации о клиентах, такой как имя, почтовый адрес и адрес электронной почты. В базе данных также имеются таблицы для хранения информации о заказах и товарах, включенных в них.

Таблица **ADDRESS_TYPES** (типы адресов) хранит классификационные значения, такие как «Домашний», «Почтовый» и «Доставки», определяющие разные типы адресов в таблице **ADDRESSES**. Аналогично таблица **TELEPHONE_TYPES** (типы телефонов) хранит классификационные значения «Сотовый», «Домашний» и «Рабочий». Эти две таблицы заполняются предварительными данными при создании базы данных, так же как таблица **US_STATES**.



Для простоты примера база данных предполагает хранение только адресов в США.

Введение в Java Persistence API

Java Persistence API (JPA) был введен в спецификацию Java EE в версии 5. Как явствует из названия¹, этот прикладной интерфейс используется для организации хранения данных в системах управления реляционными базами данных (СУРБД). JPA заменил собой объектные компоненты (Entity Beans), использовавшиеся в J2EE. Сущности – это обычные классы Java, которые контейнер Java EE распознает как сущности JPA. Давайте рассмотрим отображение сущности в таблицу CUSTOMER в базе данных CUSTOMERDB:

```
package net.ensode.glassfishbook.jpaintro.entity;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "CUSTOMERS")
public class Customer implements Serializable
{
    @Id
    @Column(name = "CUSTOMER_ID")
    private Long customerId;

    @Column(name = "FIRST_NAME")
    private String firstName;

    @Column(name = "LAST_NAME")
    private String lastName;

    private String email;

    public Long getCustomerId()
    {
        return customerId;
    }

    public void setCustomerId(Long customerId)
    {
        this.customerId = customerId;
    }

    public String getEmail()
```

¹ Название «Java Persistence API» можно перевести как «Прикладной программный интерфейс к хранилищам данных». – Прим. перев.

```
{  
    return email;  
}  
  
public void setEmail(String email)  
{  
    this.email = email;  
}  
  
public String getFirstName()  
{  
    return firstName;  
}  
  
public void setFirstName(String firstName)  
{  
    this.firstName = firstName;  
}  
  
public String getLastName()  
{  
    return lastName;  
}  
  
public void setLastName(String lastName)  
{  
    this.lastName = lastName;  
}  
}
```

В данном примере аннотация `@Entity` позволяет серверу GlassFish (или любому другому Java EE-совместимому серверу приложений) узнать, что этот класс является сущностью.

Аннотация `@Table(name = "CUSTOMERS")` сообщает серверу приложений, в какую таблицу отображается сущность. Значение элемента `name` определяет имя таблицы в базе данных, в которую отображается сущность. Эта аннотация является необязательной – если имя класса совпадает с именем таблицы, нет необходимости указывать, в какую таблицу отображается сущность.

Аннотация `@Id` указывает, что поле `customerId` отображается в первичный ключ (уникальный идентификатор) сущности.

Аннотация `@Column` отображает каждое поле в столбец таблицы. Если имя поля соответствует имени столбца, эту аннотацию можно опустить. По этой причине поле `email` в примере выше не было аннотировано.

Класс `EntityManager` (в действительности `entityManager` – это интерфейс; все Java EE-совместимые серверы приложений предостав-

ляют свою реализацию этого интерфейса) используется для сохранения сущности в базе данных. Следующий пример поясняет его использование:

```
package net.enseode.glassfishbook.jpaintro.namedbean;

import javax.annotation.Resource;

import javax.enterprise.context.RequestScoped;
import javax.inject.Named;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.transaction.HeuristicMixedException;
import javax.transaction.HeuristicRollbackException;
import javax.transaction.NotSupportedException;
import javax.transaction.RollbackException;
import javax.transaction.SystemException;
import javax.transaction.UserTransaction;
import net.enseode.glassfishbook.jpaintro.entity.Customer;

@Named
@RequestScoped
public class JpaDemoBean {

    @PersistenceContext
    private EntityManager entityManager;

    @Resource
    private UserTransaction userTransaction;

    public String updateDatabase() {

        String retVal = "confirmation";

        Customer customer = new Customer();
        Customer customer2 = new Customer();
        Customer customer3;

        customer.setCustomerId(3L);
        customer.setFirstName("James");
        customer.setLastName("McKenzie");
        customer.setEmail("jamesm@notreal.com");

        customer2.setCustomerId(4L);
        customer2.setFirstName("Charles");
        customer2.setLastName("Jonson");
        customer2.setEmail("cjohnson@phony.org");

        try {
            userTransaction.begin();
```

```

entityManager.persist(customer);
entityManager.persist(customer2);
customer3 = entityManager.find(Customer.class, 4L);
customer3.setLastName("Johnson");
entityManager.persist(customer3);
entityManager.remove(customer);

userTransaction.commit();
} catch (HeuristicMixedException |
        HeuristicRollbackException |
        IllegalStateException |
        NotSupportedException |
        RollbackException |
        SecurityException |
        SystemException e) {
    retVal = "error";
    e.printStackTrace();
}

return retVal;
}
}

```

Именованный компонент CDI, представленный выше, получает экземпляр класса, реализующего интерфейс `javax.persistence.EntityManager` через внедрение зависимости. Это делается путем декорирования переменной `EntityManager` аннотацией `@PersistenceContext`.

Затем, с помощью аннотации `@Resource`, внедряется экземпляр класса, реализующий интерфейс `javax.transaction.UserTransaction`. Этот объект совершенно необходим, потому что без него попытки сохранить сущности в базе данных будут возбуждать исключение `javax.persistence.TransactionRequiredException`.

Класс `EntityManager` решает множество задач, связанных со взаимодействиями с базой данных, таких как поиск сущностей в базе данных, их изменение или удаление.

Поскольку сущности JPA представляют собой **простые старые объекты Java (Plain Old Java Objects, POJO)**, их экземпляры можно создавать с помощью оператора `new`.



POJO – это объекты Java, от которых не требуется, чтобы они наследовали какой-либо родительский класс или реализовали какой-либо интерфейс.

Вызов метода `setCustomerId()` пользуется возможностями автоматической упаковки, механизма, добавленного в язык Java в JDK 1.5. Обратите внимание, что метод принимает экземпляр

`java.lang.Long`, однако мы используем примитивный тип `long`. Благодаря этой особенности код компилируется и выполняется должным образом.

Вызовы метода `persist()` объекта `EntityManager` должны выполняться в рамках транзакции, поэтому ее нужно начинать до обращения к методу `persist()`, вызывая метод `begin()` объекта `UserTransaction`.

Далее в таблицу `CUSTOMERS` вставляются две новые строки, для чего дважды вызывается метод `persist()` объекта `entityManager` с экземплярами класса `Customer`, которые мы заполнили данными ранее в коде.

После сохранения данных в объектах `customer` и `customer2`, выполняется поиск в базе данных записи с первичным ключом 4 в таблице `CUSTOMERS`. Делается это вызовом метода `find()` объекта `entityManager`. Этот метод принимает в первом параметре класс искомой сущности, а во втором – первичный ключ записи, соответствующей объекту, который требуется получить. Этот метод приблизительно эквивалентен методу `findByPrimaryKey()` домашнего интерфейса объектного компонента.

Первичный ключ, установленный для объекта `customer2`, был равен 4; по этой нам возвращается копия данного объекта. В фамилии этого заказчика была допущена орфографическая ошибка, когда сведения о нем заносились в базу. Теперь мы исправляем фамилию г-на Джонсона, вызывая метод `setLastName()` объекта `customer3`, а затем изменяем информацию в базе данных, вызывая метод `entityManager.persist()`.

После этого производится удаление информации для объекта `customer` вызовом метода `entityManager.remove()`, с передачей ему объекта `customer` в качестве параметра.

Наконец, мы подтверждаем транзакцию вызовом метода `commit()` объекта `userTransaction`.

Чтобы предыдущий код работал как ожидается, необходимо развернуть конфигурационный XML-файл с именем `persistence.xml` в WAR-файле, содержащем компонент `JPADemoBean`. Этот файл должен находиться в каталоге `WEB-INF/classes/META-INF/` внутри WAR-файла. Содержимое этого файла для предыдущего кода показано ниже:

```
<?xml version='1.0' encoding='UTF-8'?>
<persistence version="2.1"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://
```

```
java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
<persistence-unit name="customerPersistenceUnit">
  <jta-data-source>jdbc/__CustomerDBPool</jta-data-source>
</persistence-unit>
</persistence>
```

Файл persistence.xml должен содержать по крайней мере один элемент `<persistence-unit>`. Каждый такой элемент должен содержать атрибут `name` и дочерний элемент `<jta-data-source>`, значение которого является JNDI-именем источника данных, который будет использоваться для определения модуля хранения.

Причина, по которой разрешено иметь более чем один элемент `<persistence-unit>`, состоит в том, что приложению может потребоваться доступ к более чем одной базе данных, для каждой из которых должен быть определен свой элемент `<persistence-unit>`. Если приложение определяет несколько элементов `<persistence-unit>`, аннотация `@PersistenceContext`, используемая для внедрения интерфейса `EntityManager`, должна определить значение для своего элемента `unitName`. Значение этого элемента должно соответствовать атрибуту `name` элемента `<persistence-unit>` в файле `persistence.xml`.



Исключение: «Отсоединенный объект не может быть сохранен». Зачастую приложение получает JPA-сущность с помощью метода `EntityManager.find()`, а затем передает ее на уровень бизнес-логики или пользовательского интерфейса, где она потенциально может быть изменена и информация в базе данных, соответствующая сущности, позже будет обновлена. В случаях, подобных этому, вызов метода `EntityManager.persist()` возбудит исключение. Чтобы обновить JPA-сущность таким путем, нужно вызвать метод `EntityManager.merge()`. Этот метод принимает экземпляр JPA-сущности и обновляет соответствующую запись в базе данных.

Отношения между сущностями

В предыдущем разделе было показано, как получать отдельные сущности из базы данных, а также вставить их в базу данных, изменять их и удалять. Однако на практике сущности редко бывают изолированными; в подавляющем большинстве случаев они связаны с другими сущностями.

Сущности могут иметь следующие типы отношений: «один к одному», «один ко многим», «многие к одному» и «многие ко многим».

Например, в базе данных CustomerDB таблицы LOGIN_INFO (Учетные данные) и CUSTOMERS (Заказчики) связаны отношением «один к одному». Это означает, что каждому заказчику соответствует ровно одна запись в таблице LOGIN_INFO. Таблицы CUSTOMERS и ORDERS (Заказы) связаны отношением «один ко многим». Это обусловлено тем, что один заказчик может разместить много заказов. Кроме того, таблицы ORDERS и ITEMS (Товары) связаны отношением «многие ко многим». То есть каждый заказ может содержать много товаров, а один и тот же товар, в свою очередь, может входить в несколько заказов.

В следующих нескольких разделах мы обсудим, как устанавливать отношения между JPA-сущностями.

Отношение «один к одному»

Отношение «один к одному» возникает, когда экземпляру сущности может соответствовать нуль или один экземпляр другой сущности.

Отношения «один к одному» могут быть двунаправленными (каждая сущность знает об отношении) или односторонними (только одна из сущностей знает об отношении). В базе данных CustomerDB отношение «один к одному» между таблицами LOGIN_INFO и CUSTOMERS является односторонним, потому что в таблице LOGIN_INFO есть внешний ключ к таблице CUSTOMERS, а в таблице CUSTOMERS нет. Как мы скоро увидим, этот факт не мешает создать двунаправленное отношение «один к одному» между сущностями Customer и LoginInfo.

Ниже приводится исходный код сущности LoginInfo, которая отображается в таблицу LOGIN_INFO:

```
package net.ensode.glassfishbook.entityrelationship.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.Table;

@Entity
@Table(name = "LOGIN_INFO")
public class LoginInfo
{
    @Id
    @Column(name = "LOGIN_INFO_ID")
    private Long loginInfoId;

    @Column(name = "LOGIN_NAME")
```

```
private String loginName;  
  
private String password;  
  
{@OneToOne  
 @JoinColumn(name="CUSTOMER_ID")  
 private Customer customer;  
  
public Long getLoginInfoId()  
{  
    return loginInfoId;  
}  
  
public void setLoginInfoId(Long loginInfoId)  
{  
    this.loginInfoId = loginInfoId;  
}  
  
public String getPassword()  
{  
    return password;  
}  
  
public void setPassword(String password)  
{  
    this.password = password;  
}  
  
public String getLoginName()  
{  
    return loginName;  
}  
  
public void setLoginName(String userName)  
{  
    this.loginName = userName;  
}  
  
public Customer getCustomer()  
{  
    return customer;  
}  
  
public void setCustomer(Customer customer)  
{  
    this.customer = customer;  
}  
}
```

Реализация этой сущности очень похожа на реализацию сущности Customer. Здесь определяются поля, которые отображаются в столбцы

таблицы. Каждое поле, имя которого не соответствует имени столбца, декорируется аннотацией `@Column`. В дополнение к этому первичный ключ декорируется аннотацией `@Id`.

Самое интересное в этом коде – объявление поля `customer`. Поле `customer` декорировано аннотацией `@OneToOne`. Это позволяет серверу приложений (в данном случае GlassFish) узнать, что имеется отношение «один к одному» между этой сущностью и сущностью `Customer`. Кроме того, поле `customer` декорировано аннотацией `@JoinColumn`. Эта аннотация сообщает контейнеру, что соответствующий столбец в таблице `LOGIN_INFO` является внешним ключом, ссылающимся на первичный ключ в таблице `CUSTOMER`. Поскольку в таблице `LOGIN_INFO`, в которую отображается сущность `LoginInfo`, есть внешний ключ к таблице `CUSTOMER`, сущность `LoginInfo` является владельцем отношения. Если бы отношение было односторонним, нам не нужно было бы производить изменения в сущности `Customer`. Однако, поскольку мы хотим получить двунаправленное отношение между этими двумя сущностями, необходимо добавить в сущность `Customer` поле `LoginInfo`, а также соответствующие методы чтения/записи (`getter/setter`), как показано ниже:

```
package net.ensode.glassfishbook.entityrelationship.entity;

import java.io.Serializable;
import java.util.Set;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name = "CUSTOMERS")
public class Customer implements Serializable
{
    @Id
    @Column(name = "CUSTOMER_ID")
    private Long customerId;

    @Column(name = "FIRST_NAME")
    private String firstName;

    @Column(name = "LAST_NAME")
```

```
private String lastName;

private String email;

@OneToOne(mappedBy = "customer")
private LoginInfo loginInfo;

public Long getCustomerId()
{
    return customerId;
}

public void setCustomerId(Long customerId)
{
    this.customerId = customerId;
}

public String getEmail()
{
    return email;
}

public void setEmail(String email)
{
    this.email = email;
}

public String getFirstName()
{
    return firstName;
}

public void setFirstName(String firstName)
{
    this.firstName = firstName;
}

public String getLastName()
{
    return lastName;
}

public void setLastName(String lastName)
{
    this.lastName = lastName;
}

public LoginInfo getLoginInfo()
{
    return loginInfo;
```

```
}

public void setLoginInfo(LoginInfo loginInfo)
{
    this.loginInfo = loginInfo;
}
```

Чтобы сделать отношение «один к одному» двунаправленным, нужно всего лишь добавить в сущность `Customer` поле `LoginInfo` с соответствующими методами чтения/записи. Поле `LoginInfo` декорировано аннотацией `@OneToOne`. Поскольку сущность `Customer` не является владельцем отношения (таблица, в которую она отображается, не имеет внешнего ключа к соответствующей таблице), в аннотацию `@OneToOne` должен быть добавлен элемент `mappedBy`, определяющий поле, соответствующее сущности другом конце отношения. В данном случае поле `customer` в сущности `LoginInfo` соответствует другому концу этого отношения «один к одному».

Следующий класс Java иллюстрирует использование этой сущности:

```
package net.enseode.glassfishbook.entityrelationship.namedbean;

import javax.annotation.Resource;
import javax.enterprise.context.RequestScoped;
import javax.inject.Named;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.transaction.HeuristicMixedException;
import javax.transaction.HeuristicRollbackException;
import javax.transaction.NotSupportedException;
import javax.transaction.RollbackException;
import javax.transaction.SystemException;
import javax.transaction.UserTransaction;
import net.enseode.glassfishbook.entityrelationship.entity.Customer;
import net.enseode.glassfishbook.entityrelationship.entity.LoginInfo;

@Named
@RequestScoped
public class OneToOneRelationshipDemoBean {

    @PersistenceContext
    private EntityManager entityManager;

    @Resource
    private UserTransaction userTransaction;

    public String updateDatabase() {
        String retVal = "confirmation";
        Customer customer;
```

```
>LoginInfo loginInfo = new LoginInfo();

loginInfo.setLoginInfoId(1L);
loginInfo.setLoginName("charlesj");
loginInfo.setPassword("iwonttellyou");

try {
    userTransaction.begin();

    customer = entityManager.find(Customer.class, 4L);
    loginInfo.setCustomer(customer);

    entityManager.persist(loginInfo);

    userTransaction.commit();
} catch (NotSupportedException | 
        SystemException | 
        SecurityException | 
        IllegalStateException | 
        RollbackException | 
        HeuristicMixedException | 
        HeuristicRollbackException e) {
    retVal = "error";
    e.printStackTrace();
}

return retVal;
}
}
```

В этом примере сначала создается экземпляр сущности `LoginInfo` и заполняется некоторыми данными. Затем приобретается экземпляр сущности `Customer` из базы данных вызовом метода `find()` объекта `EntityManager` (данные для этой сущности были добавлены в таблицу `CUSTOMERS` в одном из предыдущих примеров). Затем вызывается метод `setCustomer()` сущности `LoginInfo`, которому передается сущность с информацией о заказчике. Наконец, вызывается метод `EntityManager.persist()`, чтобы сохранить данные в базе.

За кулисами происходит следующее: в столбец `CUSTOMER_ID` таблицы `LOGIN_INFO` записывается значение первичного ключа соответствующей записи в таблице `CUSTOMERS`. Это легко проверить путем запроса к базе данных `CUSTOMERDB`.



Обратите внимание, что вызов метода `EntityManager.find()` для получения сущности `Customer` производится в той же транзакции, в которой вызывается метод `EntityManager.persist()`. Это обязательное условие, иначе база данных не будет успешно обновлена.

Отношение «один ко многим»

Отношение «один ко многим» между JPA-сущностями могут быть двунаправленными (одна сущность владеет отношением «многие к одному», а сущность на другом конце отношения владеет обратным отношением «один ко многим»).

В SQL отношения «один ко многим» определяются внешними ключами в одной из таблиц. Сторона отношения «ко многим» имеет внешний ключ, ссылающийся на сторону «один». Обычно в базах данных отношения «один ко многим» определяются как односторонние, потому что создание двунаправленных отношений «один ко многим» приводит к денормализации данных.

Подобно тому как определяется одностороннее отношение «один ко многим» в СУРБД, в JPA сторона отношения «ко многим» имеет ссылку на сторону «один». Поэтому соответствующий метод записи декорируется аннотацией `@ManyToOne`.

В базе данных CUSTOMERDB односторонним отношением «один ко многим» связаны заказчики и заказы. Определим это отношение в сущности Order:

```
package net.ensode.glassfishbook.entityrelationship.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

@Entity
@Table(name = "ORDERS")
public class Order
{
    @Id
    @Column(name = "ORDER_ID")
    private Long orderId;

    @Column(name = "ORDER_NUMBER")
    private String orderNumber;

    @Column(name = "ORDER_DESCRIPTION")
    private String orderDescription;

    @ManyToOne
    @JoinColumn(name = "CUSTOMER_ID")
    private Customer customer;

    public Customer getCustomer()
```

```
{  
    return customer;  
}  
  
public void setCustomer(Customer customer)  
{  
    this.customer = customer;  
}  
  
public String getOrderDescription()  
{  
    return orderDescription;  
}  
  
public void setOrderDescription(String orderDescription)  
{  
    this.orderDescription = orderDescription;  
}  
  
public Long getOrderID()  
{  
    return orderId;  
}  
  
public void setOrderID(Long orderId)  
{  
    this.orderId = orderId;  
}  
  
public String getOrderNumber()  
{  
    return orderNumber;  
}  
  
public void setOrderNumber(String orderNumber)  
{  
    this.orderNumber = orderNumber;  
}  
}
```

Если потребуется определить одностороннее отношение «множество к одному» между сущностями Orders и Customer, нам не нужно будет производить изменения в сущности Customer. Чтобы определить двунаправленное отношение «один ко многим» между этими двумя сущностями, нужно добавить в сущность Customer новое поле с аннотацией @OneToMany:

```
package net.ensode.glassfishbook.entityrelationship.entity;  
  
import java.io.Serializable;
```

```
import java.util.Set;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
@Table(name = "CUSTOMERS")
public class Customer implements Serializable
{
    @Id
    @Column(name = "CUSTOMER_ID")
    private Long customerId;

    @Column(name = "FIRST_NAME")
    private String firstName;

    @Column(name = "LAST_NAME")
    private String lastName;

    private String email;

    @OneToOne(mappedBy = "customer")
    private LoginInfo loginInfo;

    @OneToMany(mappedBy="customer")
    private Set<Order> orders;

    public Long getCustomerId()
    {
        return customerId;
    }

    public void setCustomerId(Long customerId)
    {
        this.customerId = customerId;
    }

    public String getEmail()
    {
        return email;
    }

    public void setEmail(String email)
    {
        this.email = email;
    }

    public String getFirstName()
```

```
{  
    return firstName;  
}  
  
public void setFirstName(String firstName)  
{  
    this.firstName = firstName;  
}  
  
public String getLastName()  
{  
    return lastName;  
}  
  
public void setLastName(String lastName)  
{  
    this.lastName = lastName;  
}  
  
public LoginInfo getLoginInfo()  
{  
    return loginInfo;  
}  
  
public void setLoginInfo(LoginInfo loginInfo)  
{  
    this.loginInfo = loginInfo;  
}  
  
public Set<Order> getOrders()  
{  
    return orders;  
}  
  
public void setOrders(Set<Order> orders)  
{  
    this.orders = orders;  
}
```

Единственная разница между этой версией сущности `Customer` и предыдущей в дополнительном поле `orders` и связанных с ним методов чтения/записи. Особый интерес представляет аннотация `@OneToMany` перед этим полем. Значение атрибута `mappedBy` должно совпадать с именем соответствующего поля в сущности на стороне «ко многим» отношения. Проще говоря, значение атрибута `mappedBy` должно соответствовать имени поля, декорированного аннотацией `@ManyToOne` в компоненте на другой стороне отношения.

Следующий пример демонстрирует, как сохранить отношение «один ко многим» в базе данных:

```
package net.enseode.glassfishbook.entityrelationship.namedbean;

import javax.annotation.Resource;
import javax.enterprise.context.RequestScoped;
import javax.inject.Named;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.transaction.HeuristicMixedException;
import javax.transaction.HeuristicRollbackException;
import javax.transaction.NotSupportedException;
import javax.transaction.RollbackException;
import javax.transaction.SystemException;
import javax.transaction.UserTransaction;
import net.enseode.glassfishbook.entityrelationship.entity.Customer;
import net.enseode.glassfishbook.entityrelationship.entity.Order;

@Named
@RequestScoped
public class OneToManyRelationshipDemoBean {

    @PersistenceContext
    private EntityManager entityManager;

    @Resource
    private UserTransaction userTransaction;

    public String updateDatabase() {
        String retVal = "confirmation";

        Customer customer;
        Order order1;
        Order order2;

        order1 = new Order();
        order1.setOrderId(1L);
        order1.setOrderNumber("SFX12345");
        order1.setOrderDescription("Dummy order.");

        order2 = new Order();
        order2.setOrderId(2L);
        order2.setOrderNumber("SFX23456");
        order2.setOrderDescription("Another dummy order.");

        try {
            userTransaction.begin();

            customer = entityManager.find(Customer.class, 4L);

            order1.setCustomer(customer);
```

```

        order2.setCustomer(customer);

        entityManager.persist(order1);
        entityManager.persist(order2);

        userTransaction.commit();
    } catch (NotSupportedException | 
             SystemException | 
             SecurityException | 
             IllegalStateException | 
             RollbackException | 
             HeuristicMixedException | 
             HeuristicRollbackException e) {
        retVal = "error";
        e.printStackTrace();
    }

    return retVal;
}
}
}

```

Этот пример очень похож на предыдущий. Здесь создаются два экземпляра сущности `Order`, которые заполняются некоторыми данными, а затем в транзакцию включается экземпляр сущности `Customer` и используется как параметр метода `setCustomer()` для обоих экземпляров сущности `Order`. Далее оба экземпляра сущности `order` сохраняются вызовом метода `EntityManager.persist()` каждого из них.

Так же, как в случае с отношением «один к одному», за кулисами происходит следующее: в столбец `CUSTOMER_ID` таблицы `ORDERS` записывается первичный ключ, соответствующий связанной записи в таблице `CUSTOMERS`.

Поскольку отношение является двунаправленным, получить все заказы, сделанные определенным заказчиком, можно вызовом метода `getOrders()` сущности `Customer`.

Отношение «многие ко многим»

Отношением «многие ко многим» в базе данных `CUSTOMERDB` связаны таблицы `ORDERS` и `ITEMS`. Представить это отношение можно, добавив новое поле `Collection<Item>` в сущность `order` и декорировав его аннотацией `@ManyToMany`:

```

package net.enseode.glassfishbook.entityrelationship.entity;

import java.util.Collection;

import javax.persistence.Column;

```

```
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

@Entity
@Table(name = "ORDERS")
public class Order
{
    @Id
    @Column(name = "ORDER_ID")
    private Long orderId;

    @Column(name = "ORDER_NUMBER")
    private String orderNumber;

    @Column(name = "ORDER_DESCRIPTION")
    private String orderDescription;

    @ManyToOne
    @JoinColumn(name = "CUSTOMER_ID")
    private Customer customer;

    @ManyToMany
    @JoinTable(name = "ORDER_ITEMS",
        joinColumns = @JoinColumn(name = "ORDER_ID",
            referencedColumnName = "ORDER_ID"),
        inverseJoinColumns = @JoinColumn(name = "ITEM_ID",
            referencedColumnName = "ITEM_ID"))
    private Collection<Item> items;

    public Customer getCustomer()
    {
        return customer;
    }

    public void setCustomer(Customer customer)
    {
        this.customer = customer;
    }

    public String getOrderDescription()
    {
        return orderDescription;
    }

    public void setOrderDescription(String orderDescription)
```

```
{  
    this.orderDescription = orderDescription;  
}  
  
public Long getOrderID()  
{  
    return orderId;  
}  
  
public void setOrderID(Long orderId)  
{  
    this.orderId = orderId;  
}  
  
public String getOrderNumber()  
{  
    return orderNumber;  
}  
  
public void setOrderNumber(String orderNumber)  
{  
    this.orderNumber = orderNumber;  
}  
  
public Collection<Item> getItems()  
{  
    return items;  
}  
  
public void setItems(Collection<Item> items)  
{  
    this.items = items;  
}  
}
```

Здесь, в дополнение к аннотации @ManyToMany поле items декорируется также аннотацией @JoinTable. Как и следует из названия, эта аннотация сообщает серверу приложений, какая таблица используются в качестве объединяющей таблицы для создания отношения «многие ко многим» между этими двумя сущностями. Эта аннотации имеет три соответствующих элемента: элемент name определяет имя объединяющей таблицы, а элементы joinColumns и inverseJoinColumns определяют столбцы внешних ключей в объединяющей таблице, которые ссылаются на первичные ключи в сущностях. Значением элементов joinColumns и inverseJoinColumns является еще одна аннотация – @JoinColumn. Она имеет два взаимосвязанных элемента: элемент name определяет имя столбца в объединяющей таблице и элемент referencedColumnName определяет имя столбца в таблице сущности.

Сущность Item – простая сущность, отображающаяся в таблицу ITEMS:

```
package net.ensode.glassfishbook.entityrelationship.entity;

import java.util.Collection;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.Table;

@Entity
@Table(name = "ITEMS")
public class Item
{
    @Id
    @Column(name = "ITEM_ID")
    private Long itemId;

    @Column(name = "ITEM_NUMBER")
    private String itemNumber;

    @Column(name = "ITEM_SHORT_DESC")
    private String itemShortDesc;

    @Column(name = "ITEM_LONG_DESC")
    private String itemLongDesc;

    @ManyToMany(mappedBy="items")
    private Collection<Order> orders;

    public Long getItemId()
    {
        return itemId;
    }

    public void setItemId(Long itemId)
    {
        this.itemId = itemId;
    }

    public String getItemLongDesc()
    {
        return itemLongDesc;
    }

    public void setItemLongDesc(String itemLongDesc)
    {
```

```
    this.itemLongDesc = itemLongDesc;
}

public String getItemNumber()
{
    return itemNumber;
}

public void setItemNumber(String itemNumber)
{
    this.itemNumber = itemNumber;
}

public String getItemShortDesc()
{
    return itemShortDesc;
}

public void setItemShortDesc(String itemShortDesc)
{
    this.itemShortDesc = itemShortDesc;
}

public Collection<Order> getOrders()
{
    return orders;
}

public void setOrders(Collection<Order> orders)
{
    this.orders = orders;
}
```

Точно так же, как отношения «один к одному» и «один ко многим», отношение «многие ко многим» может быть односторонним или двунаправленным. Поскольку нам нужно двунаправленное отношение «многие ко многим» между сущностями Order и Item, было добавлено поле `Collection<Order>` с аннотацией `@ManyToMany`. Поскольку соответствующему полю в сущности Order объединяющая таблица уже была определена, нет необходимости делать это снова. Как принято говорить, отношение принадлежит сущности с аннотацией `@JoinTable`. Отношение «многие ко многим» может принадлежать любой сущности. В данном примере оно принадлежит сущности Order и представлено полем `Collection<Item>` с аннотацией `@JoinTable`.

Точно так же, как в отношениях «один к одному» и «один ко многим», аннотация `@ManyToMany` на стороне, не являющейся владельцем

дву направленного отношения «многие ко многим», должна содержать элемент `mappedBy`, указывающий, какое поле во владеющей сущности определяет это отношение.

Теперь, после знакомства с изменениями, необходимыми для образования дву направленного отношения «многие ко многим» между сущностями `Order` и `Item`, посмотрим, как действует это отношение:

```
package net.ensode.glassfishbook.entityrelationship.namedbean;

import java.util.ArrayList;
import java.util.Collection;
import javax.annotation.Resource;
import javax.enterprise.context.RequestScoped;
import javax.inject.Named;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.transaction.HeuristicMixedException;
import javax.transaction.HeuristicRollbackException;
import javax.transaction.NotSupportedException;
import javax.transaction.RollbackException;
import javax.transaction.SystemException;
import javax.transaction.UserTransaction;
import net.ensode.glassfishbook.entityrelationship.entity.Item;
import net.ensode.glassfishbook.entityrelationship.entity.Order;

@Named
@RequestScoped
public class ManyToManyRelationshipDemoBean {

    @PersistenceContext
    private EntityManager entityManager;

    @Resource
    private UserTransaction userTransaction;

    public String updateDatabase() {
        String retVal = "confirmation";

        Order order;
        Collection<Item> items = new ArrayList<Item>();
        Item item1 = new Item();
        Item item2 = new Item();

        item1.setItemId(1L);
        item1.setItemNumber("BCD1234");
        item1.setItemShortDesc("Notebook Computer");
        item1.setItemLongDesc("64 bit Quad core CPU, 4GB memory");

        item2.setItemId(2L);
        item2.setItemNumber("CDF2345");
```

```

item2.setItemShortDesc("Cordless Mouse");
item2.setItemLongDesc("Three button, infrared,
    + "vertical and horizontal scrollwheels");

items.add(item1);
items.add(item2);

try {
    userTransaction.begin();

    entityManager.persist(item1);
    entityManager.persist(item2);

    order = entityManager.find(Order.class, 1L);
    order.setItems(items);

    entityManager.persist(order);

    userTransaction.commit();
} catch (NotSupportedException | 
        SystemException | 
        SecurityException | 
        IllegalStateException | 
        RollbackException | 
        HeuristicMixedException | 
        HeuristicRollbackException e) {
    retVal = "error";
    e.printStackTrace();
}

return retVal;
}

```

Здесь создаются два экземпляра сущности `Item` и заполняются некоторыми данными. Затем эти два экземпляра добавляются в коллекцию. После этого запускается транзакция, и два экземпляра `Item` сохраняются в базе данных. Далее из базы данных извлекается экземпляр сущности `Order`. Потом вызывается метод `setItems()` сущности `Order`, которому передается коллекция с двумя экземплярами `Item`. После этого экземпляр `Customer` сохраняется в базе данных. За кулисами создаются две записи в таблице `ORDER_ITEMS`, объединяющей таблицы `ORDERS` и `ITEMS`.

Составные первичные ключи

Большинство таблиц в базе данных CUSTOMERDB имеет столбец, предназначенный исключительно для использования в качестве первичного

го ключа (этот тип первичного ключа иногда называют суррогатным или искусственным). Однако некоторые базы данных проектируются иначе: на роль первичного ключа выбирается столбец, который заранее будет иметь уникальные значения во всех записях. Если нет такого столбца, значение которого гарантированно будет уникальным, в качестве первичного ключа используется комбинация из двух или более столбцов. Эту разновидность первичного ключа можно отобразить в сущности JPA, путем использования класса первичного ключа.

Таблица ORDER_ITEMS в базе данных CUSTOMERDB не имеет суррогатного первичного ключа. Эта таблица является объединяющей для таблиц ORDERS и ITEMS. Помимо внешних ключей для этих двух таблиц в ORDER_ITEMS имеется дополнительный столбец под названием ITEM_QTY, который хранит количество каждого товара в заказе. Поскольку эта таблица не имеет суррогатного первичного ключа, сущность JPA, отображающаяся в нее, должна иметь пользовательский класс первичного ключа. Комбинация столбцов ORDER_ID и ITEM_ID в таблице ORDER_ITEMS всегда будет уникальной, поэтому комбинация этих столбцов является хорошим кандидатом на роль составного первичного ключа.

```
package net.ensode.glassfishbook.compositeprimarykeys.entity;

import java.io.Serializable;

public class OrderItemPK implements Serializable
{
    public Long orderId;
    public Long itemId;

    public OrderItemPK()
    {
    }

    public OrderItemPK(Long orderId, Long itemId)
    {
        this.orderId = orderId;
        this.itemId = itemId;
    }

    @Override
    public boolean equals(Object obj)
    {
        boolean returnVal = false;
        if (obj == null)
        {
            returnVal = false;
        }
        else
        {
            OrderItemPK tempObj = (OrderItemPK) obj;
            if (tempObj.orderId.equals(this.orderId) && tempObj.itemId.equals(this.itemId))
                returnVal = true;
        }
        return returnVal;
    }

    @Override
    public int hashCode()
    {
        int hash = 0;
        hash += orderId.hashCode();
        hash += itemId.hashCode();
        return hash;
    }
}
```

```

        }
        else if (!obj.getClass().equals(this.getClass()))
        {
            returnVal = false;
        }
        else
        {
            OrderItemPK other = (OrderItemPK) obj;
            if (this == other)
            {
                returnVal = true;
            }
            else if (orderId != null && other.orderId != null
                      && this.orderId.equals(other.orderId))
            {
                if (itemId != null && other.itemId != null
                    && itemId.equals(other.itemId))
                {
                    returnVal = true;
                }
            }
            else
            {
                returnVal = false;
            }
        }
        return returnVal;
    }

@Override
public int hashCode()
{
    if (orderId == null || itemId == null)
    {
        return 0;
    }
    else
    {
        return orderId.hashCode() ^ itemId.hashCode();
    }
}
}

```

Пользовательский класс первичного ключа должен удовлетворять следующим требованиям:

- имеет область видимости `public`;
- реализует интерфейс `java.io.Serializable`;
- имеет общедоступный конструктор без аргументов;
- поля класса имеют область видимости `public` или `protected`;

- имена полей и типы соответствуют именам полей и типам в сущности;
- переопределяет методы по умолчанию `hashCode()` и `equals()`, унаследованные от класса `java.lang.Object`.

Класс `OrderItemPK` из предыдущего примера соответствует всем перечисленным требованиям. У него есть также вспомогательный конструктор, принимающий два объекта типа `Long`, предназначенных для инициализации полей `orderId` и `itemId`. Этот конструктор был добавлен исключительно для удобства – он не является обязательным для класса, который будет использоваться в качестве первичного ключа.

Сущность, использующая пользовательский класс первичного ключа, должна быть декорирована аннотацией `@IdClass`. Поскольку класс `OrderItem` использует `OrderItemPK` как класс первичного ключа, он должен быть декорирован указанной аннотацией:

```
package net.ensode.glassfishbook.compositeprimarykeys.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.IdClass;
import javax.persistence.Table;

@Entity
@Table(name = "ORDER_ITEMS")
@IdClass(value = OrderItemPK.class)
public class OrderItem
{
    @Id
    @Column(name = "ORDER_ID")
    private Long orderId;

    @Id
    @Column(name = "ITEM_ID")
    private Long itemId;

    @Column(name = "ITEM_QTY")
    private Long itemQty;

    public Long getItemId()
    {
        return itemId;
    }

    public void setItemId(Long itemId)
```

```
{  
    this.itemId = itemId;  
}  
  
public Long getItemQty()  
{  
    return itemQty;  
}  
  
public void setItemQty(Long itemQty)  
{  
    this.itemQty = itemQty;  
}  
  
public Long getOrderId()  
{  
    return orderId;  
}  
  
public void setOrderId(Long orderId)  
{  
    this.orderId = orderId;  
}  
}
```

Эта и предыдущая сущности имеют два отличия. Первое отличие: данная сущность декорирована аннотацией `@IdClass`, в которой указан класс соответствующего ей первичного ключа. Второе отличие: данная сущность имеет более одного поля с аннотацией `@Id`. Поскольку у этой сущности есть составной первичный ключ, каждое поле, которое является частью первичного ключа, должно быть декорировано этой аннотацией.

Получение ссылки на сущность с составным первичным ключом не сильно отличается от получения ссылки на сущность с первичным ключом в виде единственного поля. Следующий пример демонстрирует, как это сделать:

```
package net.ensode.glassfishbook.compositeprimarykeys.namedbean;  
  
import javax.enterprise.context.RequestScoped;  
import javax.inject.Named;  
import javax.persistence.EntityManager;  
import javax.persistence.PersistenceContext;  
import net.ensode.glassfishbook.compositeprimarykeys.entity.OrderItem;  
import net.ensode.glassfishbook.compositeprimarykeys.entity.OrderItemPK;  
  
@Named  
@RequestScoped
```

```
public class CompositePrimaryKeyDemoBean {  
  
    @PersistenceContext  
    private EntityManager entityManager;  
  
    private OrderItem orderItem;  
  
    public String findOrderItem() {  
        String retVal = "confirmation";  
        try {  
            orderItem = entityManager.find(OrderItem.class,  
                new OrderItemPK(1L, 2L));  
        } catch (Exception e) {  
            retVal = "error";  
            e.printStackTrace();  
        }  
  
        return retVal;  
    }  
  
    public OrderItem getOrderItem() {  
        return orderItem;  
    }  
  
    public void setOrderItem(OrderItem orderItem) {  
        this.orderItem = orderItem;  
    }  
}
```

Как видите, единственная разница между поиском сущности с составным первичным ключом и сущности с первичным ключом в виде единственного поля заключается в том, что методу EntityManager.find() нужно передать во втором параметре экземпляр класса первичного ключа. Поля этого экземпляра должны быть заполнены соответствующими значениями всех полей, составляющих первичный ключ.

Введение в язык запросов JPA

Во всех примерах извлечения сущностей из базы данных, демонстрировавшихся до сих пор, для простоты предполагалось, что первичный ключ сущности известен заранее. Однако все мы знаем, что зачастую дело обстоит далеко не так. Всякий раз, когда требуется найти сущность по полю, не являющемуся первичным ключом, следует использовать **язык запросов JPA (Java Persistence Query Language, JPQL)**.

JPQL – это SQL-подобный язык, используемый для извлечения, изменения и удаления сущностей в базе данных. Следующий пример

показывает, как с помощью JPQL получить подмножество штатов из таблицы US_STATES в базе данных CUSTOMERDB:

```
package net.enseode.glassfishbook.jpql.namedbean;

import java.util.List;
import javax.enterprise.context.RequestScoped;
import javax.inject.Named;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
import net.enseode.glassfishbook.jpql.entity.UsState;

@Named
@RequestScoped
public class SelectQueryDemoBean {

    @PersistenceContext
    private EntityManager entityManager;

    private List<UsState> matchingStatesList;

    public String findStates() {
        String retVal = "confirmation";

        try {
            Query query = entityManager.createQuery(
                "SELECT s FROM UsState s WHERE s.usStateNm "
                + "LIKE :name");
            query.setParameter("name", "New%");
            matchingStatesList = query.getResultList();
        } catch (Exception e) {
            retVal = "error";
            e.printStackTrace();
        }
        return retVal;
    }

    public List<UsState> getMatchingStatesList() {
        return matchingStatesList;
    }

    public void setMatchingStatesList(List<UsState>
                                         matchingStatesList)
    {
        this.matchingStatesList = matchingStatesList;
    }
}
```

Здесь вызывается метод EntityManager.createQuery(), которому передается строка с запросом JPQL. Этот метод возвращает эк-

земпляр `javax.persistence.Query`. Запрос извлекает все сущности `UsState`, названия которых начинаются со слова «New».

Как видите, язык JPQL очень похож на SQL. Однако есть и ряд отличий, которые могут смутить читателей, знакомых с SQL. Эквивалентный SQL-запрос выглядит так:

```
SELECT * from US_STATES s where s.US_STATE_NM like 'New%'
```

Первое отличие: запросы JPQL всегда ссылаются на имена сущностей, тогда как запросы SQL – на имена таблиц. Символ `s` после имени сущности в запросе JPQL является псевдонимом сущности. Псевдонимы таблиц в SQL являются необязательными, но псевдонимы сущностей в JPQL обязательны. Если принять во внимание эти различия, запрос JPQL уже не столь обескураживает.

Элемент `:name` в запросе является именованным параметром. Именованные параметры предназначены для их замены действительными значениями. Такая замена производится вызовом метода `setParameter()` экземпляра `javax.persistence.Query`, возвращаемого методом `EntityManager.createQuery()`. Запрос JPQL может иметь несколько именованных параметров.

Чтобы фактически выполнить запрос и извлечь сущности из базы данных, нужно вызвать метод `get resultList()` экземпляра `javax.persistence.Query`. Этот метод возвращает экземпляр класса, реализующего интерфейс `java.util.List`. Возвращаемый им список будет содержать сущности, соответствующие критериям запроса. Если никакие сущности не соответствуют критерию, будет возвращен пустой список.

Если есть уверенность, что запрос вернет единственную сущность, можно воспользоваться методом `getSingleResult()` экземпляра `Query`. Этот метод возвращает объект `Object`, который следует привести к типу соответствующей сущности.

Чтобы найти сущности с именами, начинающимися со слова «New», в предыдущем примере используется оператор `LIKE`. Для этого именованный параметр `:name` в запросе замещается значением `"New%"`. Знак процента в конце означает, что любое число символов после слова «New» будет соответствовать выражению. Знак процента может использоваться в значении параметра где угодно. Например, значению `"%Dakota"` соответствовали бы любые сущности с названиями, заканчивающимися на «Dakota»; значение `"A%a"` будет соответствовать любым штатам, названия которых начинаются с заглавной буквы «A» и заканчивается строчной буквой «a». В значении параметра

тра может быть более одного знака процента. Знак подчеркивания (_) соответствует точно одному символу. Все правила для знака процента также применимы и к знаку подчеркивания.

Кроме `LIKE` существуют также другие операторы, которые можно использовать для извлечения сущностей из базы данных:

- = извлекает сущности со значением поля, указанным слева от оператора, совпадающим со значением справа;
- > извлекает сущности со значением поля, указанным слева от оператора, больше значения справа;
- < извлекает сущности со значением поля, указанным слева от оператора, меньше значения справа;
- >= извлекает сущности со значением поля, указанным слева от оператора, больше или равно значения справа;
- <= извлекает сущности со значением поля, указанным слева от оператора, меньше или равно значения справа.

Все эти операторы действуют подобно их аналогам в SQL. Так же как в SQL, эти операторы могут комбинироваться с помощью операторов `AND` и `OR`. Условия, объединенные с помощью `AND`, соответствуют оператору, когда оба условия являются истинными, а условия, объединенные с помощью `OR`, соответствуют оператору, когда хотя бы одно из условий является истинным.

Если предполагается использовать запрос много раз, его можно сохранить в именованном запросе (named query). Именованный запрос можно определить декорированием соответствующего класса сущности аннотацией `@NamedQuery`. Эта аннотация имеет два элемента: элемент `name` определяет имя запроса, а элемент `query` – сам запрос. Чтобы выполнить именованный запрос, нужно вызвать метод `createNamedQuery()` экземпляра `EntityManager`. Этот метод принимает строку с именем запроса и возвращает экземпляр `javax.persistence.Query`.

Помимо извлечения сущностей, JPQL может использоваться для их изменения или удаления. Однако операции изменения и удаления сущностей могут выполняться программно, через интерфейс `EntityManager`. Реализация таких операций в коде выглядит более читабельной, чем при использовании JPQL. В связи с этим мы не будем описывать изменение и удаление сущностей с помощью JPQL. Для читателей, кому интересна тема создания запросов JPQL для изменения и удаления сущностей, равно как и для читателей, желающих больше узнать о JPQL, можно порекомендовать изучить спецификацию «Java Persistence 2.1», доступную по адресу: <http://jcp.org/en/jsr/detail?id=338>.

Введение в Criteria API

Одним из основных дополнений к спецификации JPA 2.0 было введение API критерииев (Criteria API). Этот прикладной интерфейс задумывался как дополнение к языку запросов JPQL.

Несмотря на большую гибкость, JPQL имеет некоторые проблемы, затрудняющие его использование. Во-первых, запросы JPQL сохраняются как строки, и компилятор не имеет возможности проверить синтаксис JPQL. Во-вторых, JPQL небезопасен с точки зрения типов. Можно написать запрос JPQL, в котором предложение `where` будет сопоставлять строковое значение с числовым свойством, и наш код будет нормально скомпилирован и развернут.

Чтобы обойти недостатки JPQL, описанные в предыдущем абзаце, в спецификации JPA версии 2.0 был введен Criteria API, позволяющий писать программные запросы JPA вообще без использования JPQL.

Следующий пример показывает, как использовать Criteria API в приложениях Java EE:

```
package net.ensode.glassfishbook.criteriaapi.namedbean;

import java.util.List;
import javax.enterprise.context.RequestScoped;
import javax.inject.Named;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Path;
import javax.persistence.criteria.Predicate;
import javax.persistence.criteria.Root;
import javax.persistence.metamodel.EntityType;
import javax.persistence.metamodel.Metamodel;
import javax.persistence.metamodel.SingularAttribute;
import net.ensode.glassfishbook.criteriaapi.entity.UsState;

@Named
@RequestScoped
public class CriteriaApiDemoBean {

    @PersistenceContext
    private EntityManager entityManager;

    private List<UsState> matchingStatesList;

    public String findStates() {
```

```

String retVal = "confirmation";
try {
    CriteriaBuilder criteriaBuilder =
        entityManager.getCriteriaBuilder();
    CriteriaQuery<UsState> criteriaQuery =
        criteriaBuilder.createQuery(UsState.class);
    Root<UsState> root =
        criteriaQuery.from(UsState.class);

    Metamodel metamodel = entityManager.getMetamodel();
    EntityType<UsState> usStateEntityType =
        metamodel.entity(UsState.class);
    SingularAttribute<UsState, String> usStateAttribute =
        usStateEntityType.getDeclaredSingularAttribute(
            "usStateNm", String.class);
    Path<String> path = root.get(usStateAttribute);
    Predicate predicate = criteriaBuilder.like(
        path, "New%");
    criteriaQuery = criteriaQuery.where(predicate);

    TypedQuery typedQuery = entityManager.createQuery(
        criteriaQuery);

    matchingStatesList = typedQuery.getResultList();
} catch (Exception e) {
    retVal = "error";
    e.printStackTrace();
}

return retVal;
}

public List<UsState> getMatchingStatesList() {
    return matchingStatesList;
}

public void setMatchingStatesList(List<UsState> matchingStatesList)
{
    this.matchingStatesList = matchingStatesList;
}
}

```

Этот пример эквивалентен примеру JPQL, приводившемуся выше. Однако здесь вместо запроса на языке JPQL используются преимущества Criteria API.

При использовании Criteria API прежде всего следует получить экземпляр класса, реализующего интерфейс `javax.persistence.criteria.CriteriaBuilder`. Как видно в предыдущем примере, получить его можно вызовом метода `getCriteriaBuilder()` объекта `EntityManager`.

Из реализации `CriteriaBuilder` следует получить экземпляр класса, реализующего интерфейс `javax.persistence.criteria.CriteriaQuery`. Делается это вызовом метода `createQuery()` реализации `CriteriaBuilder`. Обратите внимание, что `CriteriaQuery` имеет обобщенный тип. Аргумент обобщенного типа определяет тип результата, который реализация `CriteriaQuery` возвращает после выполнения. Используя преимущества обобщения таким способом, Criteria API позволяет писать безопасный с точки зрения типов код.

После получения реализации `CriteriaQuery`, из нее можно получить экземпляр класса, реализующего интерфейс `javax.persistence.criteria.Root`. Реализация `Root` определяет, какие JPA-сущности будут запрашиваться. Она походит на предложение `FROM` в языке JPQL (и SQL).

Следующие две строки используют преимущества другого нововведения в спецификацию JPA – API метамодели (**Metamodel API**). Чтобы использовать возможности Metamodel API, следует получить реализацию интерфейса `javax.persistence.metamodel.Metamodel` вызовом метода `getMetamodel()` экземпляра `EntityManager`.

Из реализации `Metamodel` можно получить экземпляр интерфейса обобщенного типа `javax.persistence.metamodel.EntityType`. Обобщенный тип аргумента определяет JPA-сущность, которой соответствует реализация `EntityType`. `EntityType` позволяет просматривать сохраняемые атрибуты JPA-сущностей во время выполнения. Именно это и делается в следующей строке. В данном случае мы получаем экземпляр `SingularAttribute`, который отображается в простой обособленный атрибут JPA-сущности. Интерфейс `EntityType` имеет методы для получения атрибутов, которые отображаются в коллекции, множества, списки и ассоциативные массивы. Получение атрибутов этих типов очень похоже на получение `SingularAttribute`, поэтому мы не будем рассматривать их непосредственно. За дополнительной информацией обращайтесь к документации по API Java EE 7 (<http://docs.oracle.com/javaee/7/api/>).

Как показано в предыдущем примере, `singularAttribute` имеет два аргумента обобщенного типа. Первый аргумент определяет JPA-сущность, а второй – тип атрибута. Мы получаем реализацию `SingularAttribute`, вызывая метод `getDeclaredSingularAttribute()` реализации `EntityType` и передавая имя атрибута (объявленного в JPA-сущности) как строку.

После получения реализации `SingularAttribute` необходимо получить реализацию `javax.persistence.criteria.Path`, вызвав метод

`get()` экземпляра `Root` реализацией `SingularAttribute` в качестве параметра.

В данном примере мы получим список всех «новых» штатов в США (т. е. всех штатов, названия которых начинаются с «New»). Конечно, это работа для `like`-условия. То же самое можно сделать с помощью Criteria API, вызывая метод `like()` реализации `CriteriaBuilder`, который принимает реализацию `Path` в первом параметре и значение для поиска – во втором.

Интерфейс `CriteriaBuilder` определяет множество методов, действующих подобно предложениям SQL и JPQL, такие как `equals()`, `greaterThan()`, `lessThan()`, `and()`, `or()` и др. (полный список можно найти в документации Java EE 7 по адресу: <http://docs.oracle.com/javaee/7/api/>). Эти методы можно объединять для создания сложных запросов.

Метод `like()` в `CriteriaBuilder` возвращает реализацию интерфейса `javax.persistence.criteria.Predicate`, который должен передаваться методу `where()` реализации `CriteriaQuery`. Этот метод возвращает новый экземпляр `CriteriaQuery`, который присваивается переменной `criteriaQuery`.

Теперь мы готовы создать запрос. Работая с Criteria API, мы имеем дело с интерфейсом `javax.persistence.TypedQuery`, который может считаться безопасной (с точки зрения типов) версией интерфейса `Query`, который используется в JPQL. Экземпляр `TypedQuery` приобретается вызовом метода `createQuery()` экземпляра `EntityManager` и передается реализации `CriteriaQuery` в качестве параметра.

Чтобы получить результаты запроса в виде списка, достаточно просто вызвать метод `getResultList()` реализации `TypedQuery`. Следует еще раз подчеркнуть, что Criteria API безопасен с точки зрения типов. Поэтому попытка присвоить результат `getResultList()` списку неправильного привела бы к ошибке компиляции.

Изменение данных с помощью Criteria API

Когда Criteria API впервые был добавлен в JPA 2.0, он поддерживал только возможность выборки данных из базы – возможность изменения существующих данных не поддерживалась.

В версии JPA 2.1, введенной в Java EE 7, появилась возможность изменять данные в базе данных посредством интерфейса `CriteriaUpdate`. Как это сделать показано в следующем примере:

```
package net.ensode.glassfishbook.criteriaupdate.namedbean;  
  
// инструкции импортирования опущены  
  
@Named
```

```
@RequestScoped
public class CriteriaUpdateDemoBean {

    @PersistenceContext
    private EntityManager entityManager;

    @Resource
    private UserTransaction userTransaction;

    private int updatedRows;

    public String updateData() {
        String retVal = "confirmation";

        try {

            userTransaction.begin();
            insertTempData();

            CriteriaBuilder criteriaBuilder =
                entityManager.getCriteriaBuilder();
            CriteriaUpdate<Address> criteriaUpdate =
                criteriaBuilder.createCriteriaUpdate(Address.class);
            Root<Address> root =
                criteriaUpdate.from(Address.class);
            criteriaUpdate.set("city", "New York");
            criteriaUpdate.where(criteriaBuilder.equal(
                root.get("city"), "New Yorc"));

            Query query =
                entityManager.createQuery(criteriaUpdate);

            updatedRows = query.executeUpdate();
            userTransaction.commit();
        } catch (Exception e) {
            retVal = "error";
            e.printStackTrace();
        }

        return retVal;
    }

    public int getUpdatedRows() {
        return updatedRows;
    }

    public void setUpdatedRows(int updatedRows) {
        this.updatedRows = updatedRows;
    }

    private void insertTempData() throws NotSupportedException,
```

```

SystemException, RollbackException,
HeuristicMixedException,
HeuristicRollbackException
{
    // тело метода опущено, так как не имеет отношения к обсуждению
    // полный исходный код можно найти в пакете примеров для книги
}

```

Этот пример находит все записи в базе данных с названием города «New Yorc» (опечатка) и заменяет его правильным названием «New York».

Так же как в предыдущем примере сначала приобретается экземпляр класса, реализующего интерфейс CriteriaBuilder вызовом метода getCriteriaBuilder() экземпляра EntityManager.

Затем, вызовом метода createCriteriaUpdate() экземпляра CriteriaBuilder приобретается экземпляр класса, реализующего интерфейс CriteriaUpdate.

Следующий шаг – получение экземпляра класса, реализующего интерфейс Root вызовом метода from() экземпляра CriteriaUpdate.

Затем вызывается метод set() экземпляра CriteriaUpdate, чтобы определить новые значения для записей. В первом параметре методу set() должна передаваться строка с именем свойства в классе Entity, а во втором – новое значение.

Далее вызовом метода where() экземпляра CriteriaUpdate конструируется предложение where. В вызов метода передается экземпляр Predicate, полученный вызовом метода equal() экземпляра CriteriaBuilder.

Затем приобретается реализация Query вызовом метода createQuery() экземпляра EntityManager, которому передается экземпляр CriteriaUpdate.

Наконец запрос выполняется как обычно, вызовом executeUpdate() реализации Query.

Удаление данных с помощью Criteria API

Помимо возможности изменять данные с помощью Criteria API, в JPA 2.1 была также добавлена возможность массового удаления записей из базы данных в виде интерфейса CriteriaDelete. Как это делается, иллюстрирует следующий фрагмент:

```

package net.ensode.glassfishbook.criteria.delete.namedbean;

// инструкции импортирования опущены

@Named

```

```
@RequestScoped
public class CriteriaDeleteDemoBean {

    @PersistenceContext
    private EntityManager entityManager;

    @Resource
    private UserTransaction userTransaction;

    private int deletedRows;

    public String deleteData() {
        String retVal = "confirmation";

        try {

            userTransaction.begin();

            CriteriaBuilder criteriaBuilder =
                entityManager.getCriteriaBuilder();
            CriteriaDelete<Address> criteriaDelete =
                criteriaBuilder.createCriteriaDelete(Address.class);
            Root<Address> root =
                criteriaDelete.from(Address.class);
            criteriaDelete.where(criteriaBuilder.or(
                criteriaBuilder.equal(root.get("city"), "New York"),
                criteriaBuilder.equal(root.get("city"), "New York")));

            Query query = entityManager.createQuery(criteriaDelete);

            deletedRows = query.executeUpdate();
            userTransaction.commit();
        } catch (Exception e) {
            retVal = "error";
            e.printStackTrace();
        }

        return retVal;
    }

    public int getDeletedRows() {
        return deletedRows;
    }

    public void setDeletedRows(int updatedRows) {
        this.deletedRows = updatedRows;
    }
}
```

Чтобы воспользоваться интерфейсом CriteriaDelete, необходимо сначала получить экземпляр CriteriaBuilder, как обычно, и затем вы-

звать его метод `createCriteriaDelete()`, чтобы получить реализацию `CriteriaDelete`.

После получения экземпляра `CriteriaDelete` конструируется предложение `where`, как это обычно принято при использовании `Criteria API`.

Затем приобретается реализация интерфейса `Query` и вызывается ее метод `executeUpdate()`.

Поддержка проверки допустимости на стороне компонентов

В JPA 2.0 была введена еще одна функция – поддержка стандартом JSR 303 проверки допустимости на стороне компонентов. Эта поддержка позволяет декорировать JPA-сущности аннотациями проверки допустимости, организовать проверку введенных пользователем данных и выполнять их «санитарную обработку».

Пользоваться проверкой допустимости на стороне компонентов очень просто. Для этого нужно лишь декорировать поля или методы чтения в JPA-сущности любой из аннотаций, определенных в пакете `javax.validation.constraints`. После декорирования полей `EntityManager` будет препятствовать сохранению данных, не прошедших проверку допустимости.

Следующий пример представляет модифицированную версию JPA-сущности `Customer`, которую мы рассматривали выше в этой главе. Она использует преимущества проверки допустимости на стороне компонентов в некоторых ее полях.

```
package net.ensode.glassfishbook.beanvalidation.entity;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

@Entity
@Table(name = "CUSTOMERS")
public class Customer implements Serializable
{
    @Id
    @Column(name = "CUSTOMER_ID")
```

```
private Long customerId;

@Column(name = "FIRST_NAME")
@NotNull
@Size(min=2, max=20)
private String firstName;

@Column(name = "LAST_NAME")
@NotNull
@Size(min=2, max=20)
private String lastName;

private String email;

public Long getCustomerId()
{
    return customerId;
}

public void setCustomerId(Long customerId)
{
    this.customerId = customerId;
}

public String getEmail()
{
    return email;
}

public void setEmail(String email)
{
    this.email = email;
}

public String getFirstName()
{
    return firstName;
}

public void setFirstName(String firstName)
{
    this.firstName = firstName;
}

public String getLastName()
{
    return lastName;
}

public void setLastName(String lastName)
```

```
{  
    this.lastName = lastName;  
}  
}
```

В этом примере использована аннотация `@NotNull`, чтобы предотвратить сохранение сущности с пустыми полями `firstName` и `lastName`. Также была использована аннотация `@Size` для ограничения минимальной и максимальной длины этих полей.

Это все, что необходимо для того, чтобы задействовать механизм проверки на стороне компонента в JPA. Если код попытается сохранить или изменить экземпляр сущности, не соответствующий критериям проверки допустимости, будет возбуждено исключение типа `javax.validation.ConstraintViolationException` и сущность не будет сохранена.

Как видите, проверка допустимости на стороне компонентов в значительной степени автоматизирует подтверждение правильности данных, освобождая от необходимости вручную писать код проверки.

Кроме двух аннотаций, представленных в предыдущем примере, пакет `javax.validation.constraints` включает еще несколько аннотаций, которые можно использовать для автоматизации проверки JPA-сущностей. Полный список аннотаций можно найти в документации по Java EE 7 API: <http://docs.oracle.com/javaee/7/api/>.

Заключительные замечания

В примерах для этой главы было показано, как организовать доступ к базе данных непосредственно из именованных компонентов CDI, выполняющих роль контроллеров. Это было сделано, чтобы пояснить суть излагаемого материала, не вдаваясь в ненужные детали. Однако вообще такая практика считается неправильной – код доступа к базе данных должен быть инкапсулирован в **объектах доступа к данным (Data Access Objects, DAO)**.



Дополнительную информацию о шаблоне проектирования DAO можно найти по адресу: <http://www.oracle.com/technetwork/java/dao-138824.html>.

Именованные компоненты CDI обычно выполняют роль контроллеров и/или моделей в шаблоне проектирования Модель-Представление-Контроллер (Model-View-Controller, MVC). Этот шаблон так

часто используется на практике, что стал стандартом де-факто для разработки приложений Java EE.



Дополнительную информацию о шаблоне проектирования MVC можно найти по адресу: <http://www.oracle.com/technetwork/java/mvc-140477.html>.

Кроме того, было решено не добавлять в примеры какой-либо код пользовательского интерфейса, поскольку он не имеет отношения к данной теме. Однако в загружаемых примерах для этой главы вы найдете страницы JSF, вызывающие именованные компоненты и отображающие страницы подтверждения после вызовов именованных компонентов.

Резюме

В этой было показано, как получить доступ к данным в базе данных через JPA.

Мы узнали, как превратить класс Java в сущность, декорировав его аннотацией `@Entity`. Также было показано, как отобразить сущность в таблицу базы данных с помощью аннотации `@Table`, поля сущности в столбцы таблицы с помощью аннотации `@Column`, и как объявить первичный ключ сущности с помощью аннотации `@Id`.

Речь также шла об использовании интерфейса `javax.persistence.EntityManager` для поиска, сохранения и изменения JPA-сущностей.

Было рассмотрено определение односторонних и двунаправленных отношений «один к одному», «один ко многим» и «многие ко многим» между JPA-сущностями.

Дополнительно мы обсудили, как использовать составные первичные ключи путем разработки пользовательских классов.

Также мы показали, как извлекать сущности из базы данных с использованием языка запросов JPQL.

Затем были рассмотрены дополнительные средства JPA, такие как Criteria API, позволяющий создавать программные запросы JPA, Metamodel API, обеспечивающий безопасность типов Java при работе с JPA, и механизм проверки допустимости на стороне компонентов, упрощающий проверку введенных данных с помощью собственных аннотаций для полей JPA-сущности.

В следующей главе мы займемся рассмотрением компонентов Enterprise JavaBeans (EJB).



ГЛАВА 4.

Enterprise JavaBeans

Enterprise JavaBeans – это серверные компоненты, инкапсулирующие бизнес-логику приложения. Компоненты Enterprise JavaBeans упрощают разработку приложений, автоматически заботясь об управлении транзакциями и безопасности. Существует два типа компонентов Enterprise JavaBeans: сеансовые компоненты, которые выполняют бизнес-логику, и компоненты, управляемые сообщениями, которые действуют как приемники сообщений.

Читатели, знакомые с J2EE, заметят, что мы сейчас не упомянули объектные компоненты (Entity Beans). Они были признаны устаревшими в пользу Java Persistence API (JPA). Объектные компоненты все еще поддерживаются для обратной совместимости, однако в новых приложениях следует отдавать предпочтение механизму объектно-реляционного отображения в JPA.

В этой главе мы затронем следующие темы:

- ❖ сеансовые компоненты:
 - простой сеансовый компонент;
 - более реалистический пример;
 - использование сеансового компонента для реализации шаблона проектирования DAO;
 - сеансовый компонент-одиночка (singleton);
- ❖ компоненты, управляемые сообщениями;
- ❖ транзакции в компонентах Enterprise JavaBeans:
 - транзакции, управляемые контейнером;
 - транзакции, управляемые компонентом;
- ❖ жизненный цикл компонентов Enterprise JavaBeans:
 - жизненный цикл сеансового компонента с сохранением состояния;
 - жизненный цикл сеансового компонента без сохранения состояния;
 - жизненный цикл компонента, управляемого сообщениями;

- служба таймеров EJB;
- безопасность EJB.

Сеансовые компоненты

Как упоминалось выше, сеансовые компоненты обычно инкапсулируют бизнес-логику. Для создания сеансового компонента в Java EE достаточно создать только два артефакта: собственно компонент и необязательный бизнес-интерфейс. Эти артефакты должны быть декорированы соответствующими аннотациями, сообщающими контейнеру EJB, что они образуют сеансовый компонент.



В J2EE от разработчика требовалось создать несколько артефактов, чтобы получить сеансовый компонент. Эти артефакты включали: собственно компонент, локальный или удаленный интерфейс (или оба), локальный домашний или удаленный домашний интерфейс (или оба) и XML-дескриптор развертывания. Как мы увидим в этой главе, разработка EJB была значительно упрощена в Java EE.

Простой сеансовый компонент

Ниже приводится реализация очень простого сеансового компонента:

```
package net.ensode.glassfishbook;

import javax.ejb.Stateless;

@Stateless
public class SimpleSessionBean implements SimpleSession
{
    private String message =
        "If you don't see this, it didn't work!";

    public String getMessage()
    {
        return message;
    }
}
```

Аннотация `@Stateless` сообщает контейнеру EJB, что этот класс является сеансовым компонентом без сохранения состояния (stateless session bean). Существует два типа сеансовых компонентов: без сохранения состояния и с сохранением состояния. Прежде чем исследовать различия между этими двумя типами, следует пояснить, как экземпляр EJB предоставляется клиентским EJB-приложениям.

Когда развертывается сеансовый компонент с сохранением или без сохранения состояния, контейнер EJB создает серию экземпляров каждого сеансового компонента, обычно называемую пулом EJB (EJB pool). Когда клиентское приложение пытается получить экземпляр EJB, сервер приложений (в данном случае GlassFish) возвращает ему экземпляр из пула.

Разница между сеансовыми компонентами с сохранением и без сохранения состояния, заключается в том, что первые поддерживают состояние диалога (conversational state) с клиентом, тогда как вторые этого не делают. Это означает, что когда клиентское приложение EJB получает экземпляр сеансового компонента с сохранением состояния, гарантируется, что значения любых переменных экземпляра компонента сохраняются после вызова метода. Поэтому можно безопасно изменять любые переменные экземпляра сеансового компонента с сохранением состояния: они сохранят свои значения до следующего вызова метода. Контейнер EJB сохраняет состояние диалога при пассивации компонентов с сохранением состояния и извлекает это состояние при активации компонента. Из-за необходимости сохранения состояния диалога сеансовые компоненты с сохранением состояния имеют более сложный жизненный цикл, чем сеансовые компоненты без сохранения состояния или компоненты, управляемые сообщениями (жизненный цикл EJB будет обсуждаться ниже).

Когда клиентское приложение запрашивает экземпляр сеансового компонента без сохранения состояния, контейнер EJB может вернуть любой экземпляр EJB из пула. Поскольку не гарантируется получение того же экземпляра компонента, значения, устанавливаемые в свойствах компонента, могут быть «потеряны» (в действительности они не теряются, а попросту могут находиться в другом экземпляре EJB из пула).

Кроме того, класс, декорированный аннотацией `@Stateless`, не имеет ничего особенного, в сравнении с предыдущим классом. Обратите внимание, что он реализует интерфейс под названием `SimpleSession`, являющийся бизнес-интерфейсом компонента. Этот интерфейс показан в следующем примере:

```
package net.ensode.glassfishbook;

import javax.ejb.Remote;

@Remote
public interface SimpleSession
```

```
{  
    public String getMessage();  
}
```

Единственная особенность этого интерфейса в том, что он декорирован аннотацией `@Remote`. Она указывает, что перед нами удаленный бизнес-интерфейс (remote business interface). Это означает, что реализация интерфейса может находиться в другой виртуальной машине JVM, отличной от JVM, где выполняется клиентское приложение. Удаленные бизнес-интерфейсы могут вызываться даже по сети.

Также бизнес-интерфейсы могут декорироваться аннотацией `@Local`. Эта аннотация указывает, что бизнес-интерфейс является локальным бизнес-интерфейсом (local business interface). Реализация локального бизнес-интерфейса должна выполняться в той же JVM, что и клиентское приложение, вызывающее его методы.

Удаленный бизнес-интерфейс может вызываться из той же JVM, где выполняется клиентское приложение, или из другой JVM. На первый взгляд кажется, что было бы здорово, если бы все бизнес-интерфейсы были удаленными. Однако следует учитывать, что гибкость, предоставляемая удаленными бизнес-интерфейсами, приводит к потере производительности, поскольку вызовы методов производятся с учетом того, что они могут выполняться по сети. В действительности самое типичное приложение Java EE состоит из веб-приложений, выступающих в качестве клиентов компонентов EJB. В этом случае клиентские приложения и компоненты EJB выполняются в одной и той же JVM. Поэтому локальные бизнес-интерфейсы используются намного чаще, чем удаленные.

Скомпилировав сеансовые компоненты и соответствующие им бизнес-интерфейсы, необходимо поместить их в JAR-файл и затем развернуть. Как и в случае с WAR-файлами, самый простой способ развертывания EJB JAR-файла – скопировать его в [каталог установки glassfish]/glassfish/domains/domain1/autodeploy.

Теперь, когда мы рассмотрели сеансовый компонент и соответствующий ему бизнес-интерфейс, перейдем к примеру клиентского приложения:

```
package net.ensode.glassfishbook;  
  
import javax.ejb.EJB;  
  
public class SessionBeanClient  
{
```

```
@EJB  
private static SimpleSession simpleSession;  
  
private void invokeSessionBeanMethods()  
{  
    System.out.println(simpleSession.getMessage());  
    System.out.println("\nSimpleSession is of type: "  
        + simpleSession.getClass().getName());  
}  
  
public static void main(String[] args)  
{  
    new SessionBeanClient().invokeSessionBeanMethods();  
}
```

Здесь просто объявляется переменная экземпляра типа `net.ensode.SimpleSession`. Этот тип является бизнес-интерфейсом сеансового компонента. Переменная экземпляра декорирована аннотацией `@EJB`, сообщающей контейнеру EJB, что переменная является бизнес-интерфейсом сеансового компонента. Контейнер EJB внедряет реализацию бизнес-интерфейса в эту переменную, чтобы клиентский код мог воспользоваться ею.

Поскольку клиент является автономным приложением (в противоположность артефакту EJB, такому как WAR-файл, или другому EJB JAR-файлу), чтобы получить доступ к коду, развернутому на сервере, он должен быть помещен в JAR-файл и запущен с помощью утилиты `appclient`. Эта утилита входит в состав сервера приложений GlassFish, она позволяет автономным Java-приложениям обращаться к ресурсам, развернутым на сервере приложений, и находится в `[каталог установки Glassfish]/glassfish/bin/`. Предполагая, что этот путь включен в переменную окружения `PATH` и клиентский код помещен в JAR-файл `simplesessionbeanclient.jar`, предыдущее приложение можно запустить следующей командой:

```
appclient -client simplesessionbeanclient.jar
```

В результате ее выполнения в консоли появится следующее сообщение:

```
If you don't see this, it didn't work!
```

```
SimpleSession is of type: net.ensode.glassfishbook._SimpleSession_Wrapper
```

Этот вывод сгенерирован классом `SessionBeanClient`.



Для сборки приложений мы обычно используем *Maven*. Сборка JAR-файла для этого примера выполнялась с применением расширения *Maven Assembly* (<http://maven.apache.org/plugins/maven-assembly-plugin/>), чтобы включить в него все зависимости. Это освобождает от необходимости указывать все зависимые JAR-файлы в параметре командной строки `-classpath` утилиты `appclient`. Чтобы создать такой JAR-файл, достаточно вызвать из командной строки команду `mvn assembly:assembly`.

Первая строка в выводе является всего лишь возвращаемым значением метода `getMessage()`, реализованным в сеансовом компоненте. Вторая строка содержит полностью определенное (квалифицированное) имя класса, реализующего бизнес-интерфейс. Обратите внимание, что имя класса не является полностью определенным именем сеансового компонента, который мы написали. То, что фактически отображается, является реализацией бизнес-интерфейса, созданного контейнером EJB за кулисами.

Более реалистический пример

В предыдущем разделе мы видели очень простой пример в духе «Привет, мир». В этом разделе мы рассмотрим более реалистический пример. Сеансовые компоненты часто используются в качестве **объектов доступа к данным (Data Access Objects, DAO)**. Иногда они служат обертками для вызовов JDBC, а иногда – обертками вызовов, извлекающих или изменяющих сущности JPA. В этом разделе будет реализован последний подход.

Следующий пример показывает, как реализовать шаблон проектирования DAO в сеансовом компоненте. Прежде чем переходить к реализации компонента, рассмотрим соответствующий ему бизнес-интерфейс:

```
package net.ensode.glassfishbook;

import javax.ejb.Remote;

@Remote
public interface CustomerDao
{
    public void saveCustomer(Customer customer);

    public Customer getCustomer(Long customerId);

    public void deleteCustomer(Customer customer);
}
```

Как видите, этот удаленный интерфейс определяет три метода. Метод `saveCustomer()` сохраняет информацию о заказчиках в базе данных, метод `getCustomer()` получает информацию о заказчиках из базы данных и метод `deleteCustomer()` удаляет информацию о заказчиках из базы данных. Два из них – первый и третий – принимают экземпляр сущности `Customer`, который мы разработали в главе 3, «Объектно-реляционное отображение в JPA», а второй (`GetCustomer()`) принимает значение типа `Long` поля `id` объекта `Customer`, который требуется извлечь из базы данных.

Теперь давайте рассмотрим сеансовый компонент, реализующий предыдущий бизнес-интерфейс. Как будет показано в следующем примере, имеются некоторые различия между способами реализации кода JPA в сеансовом компоненте в простом старом объекте Java:

```
package net.ensode.glassfishbook;

import javax.ejb.Stateful;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateful
public class CustomerDaoBean implements CustomerDao {

    @PersistenceContext
    private EntityManager entityManager;

    public void saveCustomer(Customer customer) {
        if (customer.getCustomerId() == null) {
            saveNewCustomer(customer);
        } else {
            updateCustomer(customer);
        }
    }

    private void saveNewCustomer(Customer customer) {
        entityManager.persist(customer);
    }

    private void updateCustomer(Customer customer) {
        entityManager.merge(customer);
    }

    public Customer getCustomer(Long customerId) {
        Customer customer;
        customer = entityManager.find(Customer.class, customerId);
        return customer;
    }
}
```

```
}

public void deleteCustomer(Customer customer) {
    entityManager.remove(customer);
}

}
```

Основное отличие данного сеансового компонента от предыдущих примеров использования JPA состоит в том, что прежде вызовы методов JPA выполнялись между вызовами `UserTransaction.begin()` и `UserTransaction.commit()`. Это требовалось потому, что вызовы JPA должны выполняться в рамках транзакции – при попытке выполнить вызов за рамками транзакции, большинство методов вызовет исключение `TransactionRequiredException`. В данном примере не требуется явно оберывать вызовы JPA транзакцией, как в предыдущих примерах, потому что методы сеансового компонента являются неявно транзакционными. Поэтому нам нет смысла поступать иначе. Данное поведение является поведением по умолчанию; оно также известно как **транзакции, управляемые контейнером (Container-Managed Transactions)**. Транзакции, управляемые контейнером, будут подробно обсуждаться ниже.



Как рассказывалось в главе 3, «Объектно-реляционное отображение в JPA», когда сущность JPA извлекается в одной транзакции и изменяется в другой, должен вызываться метод `EntityManager.merge()`, чтобы изменить данные в базе данных. Вызов `EntityManager.persist()` в данном случае вызовет исключение: «Невозможно сохранить, отсоединенный объект».

Вызов сеансовых компонентов в веб-приложениях

Часто приложения Java EE состоят из веб-приложений, играющих роль клиентов компонентов EJB. До выхода Java EE 6 наиболее распространенный способ развертывания приложений Java EE, состоящих из веб-приложения и одного или нескольких сеансовых компонентов, заключался в упаковке веб-приложения в WAR-файл, EJB JAR-файлов – в файл EAR (Enterprise Archive).

Упаковка и развертывание приложений, включающих EJB и веб-компоненты, в Java EE 6 были упрощены.

В этом напишем приложение JSF с именованным компонентом CDI, действующим как клиент сеансового компонента DAO, демонстрировавшегося в предыдущем разделе.

Чтобы это приложение могло действовать как клиент EJB, создадим именованный компонент `CustomerController`, делегирующий логику сохранения нового заказчика в базе данных сессионному компоненту `CustomerDaoBean`, разработанному в предыдущем разделе. Начнем с именованного компонента `CustomerController`:

```
package net.ensode.glassfishbook.jsfjpa;

// инструкции импортирования опущены

@Named
@RequestScoped
public class CustomerController implements Serializable {

    @EJB
    private CustomerDaoBean customerDaoBean;

    private Customer customer;

    private String firstName;
    private String lastName;
    private String email;

    public CustomerController() {
        customer = new Customer();
    }

    public String saveCustomer() {
        String returnValue = "customer_saved";

        try {
            populateCustomer();
            customerDaoBean.saveCustomer(customer);
        } catch (Exception e) {
            e.printStackTrace();
            returnValue = "error_saving_customer";
        }

        return returnValue;
    }

    private void populateCustomer() {
        if (customer == null) {
            customer = new Customer();
        }
        customer.setFirstName(getFirstName());
        customer.setLastName(getLastName());
        customer.setEmail(getEmail());
    }
}
```

```
}

// методы чтения/записи опущены

}
```

Как видите, чтобы внедрить экземпляр соответствующего компонента EJB достаточно объявить переменную-член типа сеансового компонента *CustomerDaoBean* и декорировать ее аннотацией @EJB, после чего можно пользоваться методом *saveCustomer()* этого компонента.

Обратите внимание, что экземпляр сеансового компонента внедряется прямо в клиентский код. Это позволяет сделать функциональность, появившаяся в Java EE 6. Используя версию Java EE 6 или выше можно покончить с локальными интерфейсами и использовать экземпляры сеансовых компонентов прямо в клиентском коде.

Теперь, после создания веб-приложения, выступающего в роли клиента сеансового компонента, необходимо упаковать его в WAR-файл (веб-архив) и развернуть для использования.

Сеансовые компоненты-одиночки (*Singleton*)

В Java EE 6 появился новый тип сеансовых компонентов – сеансовые компоненты-одиночки (*singleton session bean*). На сервере приложений может существовать только один экземпляр такого компонента.

Сеансовые компоненты-одиночки полезны для кэширования баз данных. Кэширование часто используемых данных в сеансовом компоненте-одиночке увеличивает производительность, за счет значительного снижения количества обращений к базе данных. Обычно для этого создается компонент с методом, декорированным аннотацией @PostConstruct. Этот метод будет получать данные для кэширования. Также нужно реализовать метод записи для вызова компонента клиентами. Следующий пример иллюстрирует эту технологию:

```
package net.ensode.glassfishbook.singletonsession;

import java.util.List;
import javax.annotation.PostConstruct;
import javax.ejb.Singleton;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
import net.ensode.glassfishbook.entity.UsStates;

@Singleton
```

```
public class SingletonSessionBean implements
    SingletonSessionBeanRemote
{
    @PersistenceContext
    private EntityManager entityManager;
    private List<UsStates> stateList;

    @PostConstruct
    public void init() {
        Query query = entityManager.createQuery(
            "Select us from UsStates us");
        stateList = query.getResultList();
    }

    @Override
    public List<UsStates> getStateList() {
        return stateList;
    }
}
```

Поскольку наш компонент является одиночкой, все клиенты получают доступ к одному и тому же экземпляру, что позволяет избежать многократного выполнения одинаковых запросов к базе данных. По этой же причине безопасно иметь переменную экземпляра, поскольку все клиенты обращаются к одному и тому же экземпляру компонента и, соответственно, к одной и той же переменной.

Асинхронные вызовы методов

Иногда полезно иметь возможность выполнять обработку асинхронно, т. е. когда клиент вызывает метод и тут же получает управление обратно, не ожидая завершения работы метода.

В ранних версиях Java EE единственным способом асинхронного вызова метода EJB было использование компонентов, управляемых сообщениями (они будут обсуждаться в следующем разделе). Хотя такие компоненты легко написать, в действительности они требуют некоторой настройки – такой как создание очереди или темы сообщений JMS – прежде чем их можно будет использовать.

В EJB 3.1 появилась аннотация `@asynchronous`, которой можно отметить метод сеансового компонента как асинхронный. Когда клиент EJB вызывает асинхронный метод, он сразу получает управление обратно, не ожидая завершения работы метода.

Асинхронные методы могут возвращать тип `void` или реализацию интерфейса `java.util.concurrent.Future`. Интерфейс `Future` был вве-

ден в Java 5 и представляет результат асинхронных вычислений. Следующий пример демонстрирует оба сценария:

```
package net.ensode.glassfishbook.asynchronousmethods;

import java.util.concurrent.Future;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.ejb.AsyncResult;
import javax.ejb.Asynchronous;
import javax.ejb.Stateless;

@Stateless
public class AsynchronousSessionBean implements
    AsynchronousSessionBeanRemote
{

    private static Logger logger = Logger.getLogger(
        AsynchronousSessionBean.class.getName());

    @Asynchronous
    @Override
    public void slowMethod() {
        long startTime = System.currentTimeMillis();
        logger.info("entering " + this.getClass().getCanonicalName()
            + ".slowMethod()");
        try {
            Thread.sleep(10000); // имитировать работу в течение 10 секунд
        } catch (InterruptedException ex) {
            Logger.getLogger(AsynchronousSessionBean.class.getName()).log(Level.SEVERE, null, ex);
        }
        logger.info("leaving " + this.getClass().getCanonicalName()
            + ".slowMethod()");
        long endTime = System.currentTimeMillis();
        logger.info("execution took " + (endTime - startTime)
            + " milliseconds");
    }

    @Asynchronous
    @Override
    public Future<Long> slowMethodWithValue() {
        try {
            Thread.sleep(15000); // имитировать работу в течение 15 секунд
        } catch (InterruptedException ex) {
            Logger.getLogger(AsynchronousSessionBean.class.getName()).log(Level.SEVERE, null, ex);
        }
        return new AsyncResult<Long>(42L);
    }
}
```

Если асинхронный метод ничего не возвращает (имеет тип `void` возвращаемого значения), достаточно просто декорировать его аннотацией `@Asynchronous`, после чего его можно будет вызывать как обычно.

Если нужно, чтобы метод что-то возвращал, возвращаемое значение должно быть обернуто реализацией интерфейса `java.util.concurrent.Future`. Для удобства Java EE API предоставляет реализацию в форме класса `javax.ejb.AsyncResult`. Оба они – интерфейс `Future` и класс `AsyncResult` – используют обобщения, поэтому тип возвращаемого значения следует указать как параметр типа этих артефактов.

Интерфейс `Future` имеет несколько методов, которые можно использовать для отмены выполнения асинхронного метода. При этом необходимо придерживаться следующего порядка действий: выяснить, выполнился ли метод, и получить возвращаемое значение или выяснить, не был ли метод отменен. Все эти методы перечислены в табл. 4.1.

Таблица 4.1. Методы, используемые для отмены вызовов асинхронных методов

Метод	Описание
<code>cancel(boolean mayInterruptIfRunning)</code>	Отменяет выполнение метода. Если параметр имеет значение <code>true</code> , этот метод будет пытаться отменить выполнение асинхронного метода, даже если тот уже запущен.
<code>get()</code>	Возвращает «распакованное» значение, возвращаемое асинхронным методом. Возвращаемое значение будет иметь тип параметра реализации интерфейса <code>Future</code> , возвращаемого методом.
<code>get(long timeout, TimeUnit unit)</code>	Будет пытаться вернуть «распакованное» значение, возвращаемое асинхронным методом. Возвращаемое значение будет иметь тип параметра реализации интерфейса <code>Future</code> , возвращаемого методом. Этот метод может заблокироваться на интервал времени, указанный в первом параметре. Единица измерения времени определяется вторым параметром. Перечисление <code>TimeUnit</code> включает константы: <code>NANOSECONDS</code> (наносекунды), <code>MILLISECONDS</code> (миллисекунды), <code>SECONDS</code> (секунды), <code>MINUTES</code> (минуты) и т. д. Подробностисмотрите в документации JavaDoc.
<code>isCancelled()</code>	Возвращает <code>true</code> , если асинхронный метод был отменен; в противном случае возвращает <code>false</code> .

Метод	Описание
isDone()	Возвращает true, если асинхронный метод закончил работу; в противном случае возвращает false.

Как видите, аннотация @Asynchronous облегчает вызов асинхронных вызовов, устранивая издержки, связанные с созданием очередей или тем сообщений. Безусловно, это долгожданное дополнение к спецификации EJB.

Компоненты, управляемые сообщениями

Служба сообщений Java (Java Message Service, JMS) – это Java EE API, используемый для организации асинхронных взаимодействий между разными приложениями. Сообщения JMS хранятся либо в очередях сообщений, либо в темах.

Назначение компонентов, управляемых сообщениями, обеспечить использование сообщений из очередей или тем JMS – в зависимости от используемого способа обмена сообщениями (за подробностями обращайтесь к главе 8, «Служба обмена сообщениями Java»). Компонент, управляемый сообщениями, должен декорироваться аннотацией @MessageDriven. Атрибут mappedName этой аннотации должен содержать JNDI-имя очереди или темы сообщений JMS, из которой компонент будет извлекать сообщения. Следующий пример иллюстрирует простой компонент, управляемый сообщениями:

```
package net.ensode.glassfishbook;

import javax.ejb.MessageDriven;
import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

@MessageDriven(mappedName = "jms/GlassFishBookQueue")
public class ExampleMessageDrivenBean implements MessageListener
{
    public void onMessage(Message message)
    {
        TextMessage textMessage = (TextMessage) message;
        try
        {
```

```
        System.out.print("Received the following message: ");
        System.out.println(textMessage.getText());
        System.out.println();
    }
    catch (JMSEException e)
    {
        e.printStackTrace();
    }
}
```

Рекомендуется, но не требуется, чтобы компоненты, управляемые сообщениями, реализовали интерфейс `javax.jms.MessageListener`. Кроме того, компоненты, управляемые сообщениями, должны иметь метод `onMessage()` с сигнатурой, как показано в предыдущем примере.

Клиентские приложения никогда не вызывают методы компонентов, управляемых сообщениями, непосредственно. Вместо этого они помещают сообщения в очередь или тему сообщений, а компонент извлекает их и обрабатывает. Предыдущий пример просто выводит сообщение в стандартный вывод. Поскольку компоненты, управляемые сообщениями, выполняются в контейнере EJB, стандартный вывод перенаправляется в файл журнала сервера. Чтобы просмотреть сообщения в журнале GlassFish, откройте файл [каталог установки GlassFish]/glassfish/domains/domain1/logs/server.log.¹

Транзакции в Enterprise JavaBeans

Как отмечалось выше, любые методы компонентов EJB по умолчанию автоматически выполняются в рамках транзакции. Это поведение по умолчанию известно как **транзакции, управляемые контейнером (Container-Managed Transactions)**, поскольку транзакциями управляет контейнер EJB. У разработчиков приложений также может возникнуть потребность управлять транзакциями вручную. Это может быть реализовано с помощью механизма транзакций, управляемых компонентом (Bean-Managed Transactions). Оба эти подхода мы рассмотрим в следующих разделах.

¹ Предполагается, что используется домен по умолчанию. Если вы используете другой домен, в приведенном пути domain1 нужно заменить именем вашего домена. – Прим. перев.

Транзакции, управляемые контейнером

Поскольку методы EJB по умолчанию выполняются в пределах транзакции, возникает интересная дилемма, когда сеансовый компонент вызывается из клиентского кода, который уже запустил транзакцию: как контейнер EJB должен вести себя в этом случае? Может быть, он должен приостановить клиентскую транзакцию, выполнить метод в новой транзакции, а затем возобновить клиентскую транзакцию? Или он должен выполнить метод в рамках клиентской транзакции? Или, может быть, он должен возбудить исключение?

По умолчанию, если метод EJB вызван клиентским кодом, который уже выполняется в рамках транзакции, контейнер EJB просто выполнит метод сеансового компонента как часть клиентской транзакции. Если это не то поведение, которое нужно, можно изменить его, декорировав метод аннотацией `@TransactionAttribute`. Атрибут `value` этой аннотации определяет, как будет вести себя контейнер EJB при вызове метода сеансового компонента в пределах существующей, а также при вызове вне любых транзакций. В качестве значения атрибута `value` обычно выбирают одну из констант, определенных в перечислении `javax.ejb.TransactionAttributeType`.

В табл. 4.2 перечислены возможные значения атрибута `value` в аннотации `@TransactionAttribute`.

Таблица 4.2. Возможные значения атрибута `value` в аннотации `@TransactionAttribute`

Значение атрибута <code>value</code> в аннотации <code>@TransactionAttribute</code>	Описание
<code>TransactionAttributeType.MANDATORY</code>	Вызов метода должен выполняться в пределах транзакции. Попытка вызвать метод вне каких-либо транзакций приведет к исключению <code>TransactionRequiredException</code> .
<code>TransactionAttributeType.NEVER</code>	Вызов метода должен выполняться за пределами любых транзакций. Попытка вызвать метод в рамках клиентской транзакции вызовет исключение <code>RemoteException</code> . Если метод вызывается вне клиентской транзакции, новая транзакция не запускается.
<code>TransactionAttributeType.NOT_SUPPORTED</code>	Если метод вызывается внутри клиентской транзакции, эта транзакция приостанавливается и метод выполняется вне любой транзакции. После завершения метода клиентская транзакция возобновляется. Транзакция не запускается, если метод вызывается вне клиентской транзакции.

Значение атрибута value в аннотации @TransactionAttribute	Описание
TransactionAttributeType.REQUIRED	Если метод вызывается внутри клиентской транзакции, он выполняется как часть этой транзакции. Если метод вызывается вне любой транзакции, для него запускается новая транзакция. Это поведение по умолчанию.
TransactionAttributeType.REQUIRES_NEW	Если метод вызывается внутри клиентской транзакции, эта транзакция приостанавливается и запускается новая транзакция. Когда метод завершится, клиентская транзакция возобновляется. Если метод вызывается вне любых транзакций, для него запускается новая транзакция.
TransactionAttributeType.SUPPORTS	Если метод вызывается внутри клиентской транзакции, он продолжает выполняться как часть этой транзакции. Если метод вызывается вне транзакции, новая транзакция не запускается.

Значение по умолчанию для атрибута `value` транзакции подходит для большинства случаев, все же желательно иметь возможность переопределить его в случае необходимости. Например, поскольку транзакции оказывают влияние на производительность, удобно было бы иметь возможность выключать транзакции для методов, которые в них не нуждаются. В таких случаях можно декорировать метод, как показано в следующем фрагменте кода:

```
@TransactionAttribute(value=TransactionAttributeType.NEVER)
public void doitAsFastAsPossible()
{
    // Критичный к производительности код.
}
```

Путем аннотирования соответствующей константой перечисления `TransactionAttributeType` можно каждому методу определить свое поведение в отношении транзакций.

Если потребуется определить поведение для всех методов в сеансовом компоненте, сделать это можно, декорировав аннотацией `@TransactionAttribute` весь класс сеансового компонента. Значение атрибута `value` в этом случае будет применено к каждому методу компонента.

Транзакции, управляемые контейнером, автоматически откатываются всякий раз, когда в методе EJB возникает исключение. Кроме

того, есть возможность программно откатывать транзакцию, управляемую контейнером, вызывая метод `setRollbackOnly()` экземпляра `javax.ejb.EJBContext`, соответствующего рассматриваемому сеансовому компоненту. В следующем примере представлена новая версия сеансового компонента, который мы видели ранее в этой главе. Теперь он откатывает транзакции в случае необходимости:

```
package net.ensode.glassfishbook;

// инструкции импортирования опущены

@Stateless
public class CustomerDaoRollbackBean implements
    CustomerDaoRollback
{
    @Resource
    private EJBContext ejbContext;

    @PersistenceContext
    private EntityManager entityManager;

    @Resource(name = "jdbc/_CustomerDBPool")
    private DataSource dataSource;

    public void saveNewCustomer(Customer customer)
    {
        if (customer == null || customer.getCustomerId() != null)
        {
            ejbContext.setRollbackOnly();
        }
        else
        {
            customer.setCustomerId(getNewCustomerId());
            entityManager.persist(customer);
        }
    }

    public void updateCustomer(Customer customer)
    {
        if (customer == null || customer.getCustomerId() == null)
        {
            ejbContext.setRollbackOnly();
        }
        else
        {
            entityManager.merge(customer);
        }
    }

    // Другие методы опущены.
}
```

В этой версии сеансового компонента DAO был убран метод `saveCustomer()`, а методы `saveNewCustomer()` и `updateCustomer()` сделаны общедоступными (`public`). Каждый из этих методов теперь выясняет, установлено ли значение поля `customerId` корректно для выполняемой операции (`null` для вставки и `not null` – для изменения). Они также проверяют существование объекта для сохранения, сравнивая ссылку на него со значением `null`. Если какая-либо из проверок не будет пройдена, методы просто откатывают транзакцию, вызывая `setRollbackOnly()` внедренного экземпляра `EJBContext`, и база данных не изменяется.

Транзакции, управляемые компонентом

Как было показано выше, транзакции, управляемые контейнером, делают смеюточно простым создание кода, который обертывается в транзакцию. Попросту говоря, для поддержки транзакций не нужно делать ничего особенного. Некоторые разработчики иногда даже не знают, что разрабатывая сеансовый компонент, они пишут код, который в действительности будет выполняться в рамках транзакции. Транзакции, управляемые контейнером, охватывают большинство типичных случаев, встречающихся на практике. Однако они имеют серьезное ограничение: вызов любого метода может быть обернут не более чем одной транзакцией. С помощью транзакций, управляемых контейнером, невозможно реализовать метод, запускающий больше одной транзакции. Но это можно реализовать с использованием транзакций, управляемых компонентом (*bean-managed transactions*), как показано ниже:

```
package net.enseode.glassfishbook;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;
import javax.annotation.Resource;
import javax.ejb.Stateless;
import javax.ejb.TransactionManagement;
import javax.ejb.TransactionManagementType;
import javax.persistence.EntityManager;

// остальные инструкции импортирования опущены

@Stateless
@TransactionManagement(value = TransactionManagementType.BEAN)
```

```
public class CustomerDaoBmtBean implements CustomerDaoBmt
{
    @Resource
    private UserTransaction userTransaction;

    @PersistenceContext
    private EntityManager entityManager;

    @Resource(name = "jdbc/_CustomerDBPool")
    private DataSource dataSource;

    public void saveMultipleNewCustomers(
        List<Customer> customerList)
    {
        for (Customer customer : customerList)
        {
            try
            {
                userTransaction.begin();
                customer.setCustomerId(getNewCustomerId());
                entityManager.persist(customer);
                userTransaction.commit();
            }
            catch (Exception e)
            {
                e.printStackTrace();
            }
        }
    }

    private Long getNewCustomerId()
    {
        Connection connection;
        Long newCustomerId = null;
        try
        {
            connection = dataSource.getConnection();
            PreparedStatement preparedStatement =
                connection.prepareStatement("select " +
                    "max(customer_id)+1 as new_customer_id " +
                    "from customers");
            ResultSet resultSet = preparedStatement.executeQuery();
            if (resultSet != null && resultSet.next())
            {
                newCustomerId = resultSet.getLong("new_customer_id");
            }
            connection.close();
        }
        catch (SQLException e)
        {
            e.printStackTrace();
        }
    }
}
```

```
        return newCustomerId;
    }
}
```

В этом примере реализован метод `saveMultipleNewCustomers()`, принимающий список `List` заказчиков в единственном параметре. Цель этого метода – сохранить столько элементов из списка, сколько окажется возможным. Исключение, возникающее при сохранении одной из сущностей, не должно мешать попыткам метода сохранить оставшиеся элементы. Такое поведение невозможно реализовать при использовании транзакций, управляемых контейнером, поскольку исключение, возникшее при сохранении одной из сущностей, откатывает транзакцию целиком. Единственный способ достичь требуемого поведения – использовать транзакции, управляемые компонентом.

В примере объявляется, что сеансовый компонент использует транзакции, управляемые компонентом, путем декорирования класса аннотацией `@TransactionManagement` со значением `TransactionManagementType.BEAN` в атрибуте `value` (другое допустимое значение этого атрибута – `TransactionManagementType.CONTAINER`, но так как оно является значением по умолчанию, указывать его явно нет необходимости).

Чтобы иметь возможность программно управлять транзакциями, в компонент внедряется экземпляр `javax.transaction.UserTransaction`, который затем используется в цикле `for` в методе `saveMultipleNewCustomers()` для запуска и подтверждения транзакции в каждой итерации.

Если потребуется транзакцию, управляемую компонентом, сделать это можно простым вызовом метода `rollback()` экземпляра `javax.transaction.UserTransaction`.

Перед тем как двинуться дальше, отметим следующее: несмотря на то, что все приведенные в этом разделе примеры были сеансовыми компонентами, описанные здесь понятия также применяются к компонентам, управляемым сообщениями.

Жизненные циклы компонентов Enterprise JavaBeans

Компоненты Enterprise JavaBeans в течение жизни проходят через разные состояния. Каждый тип EJB имеет различные состояния.

В следующих разделах мы обсудим состояния, определенные для каждого типа EJB.

Жизненный цикл сеансового компонента с сохранением состояния

Читатели, имеющие опыт работы с предыдущими версиями J2EE, помнят, что в предыдущих версиях спецификации сеансовые компоненты были обязаны реализовать интерфейс `javax.ejb.SessionBean`. Он определяет методы, которые будут выполняться в определенные моменты жизненного цикла сеансового компонента. В число методов интерфейса `SessionBean` входят:

- `ejbActivate();`
- `ejbPassivate();`
- `ejbRemove();`
- `setSessionContext (SessionContext ctx);`

Первые три метода предназначены для выполнения в определенные моменты жизненного цикла компонента. В большинстве случаев в реализациях этих методов не нужно делать ничего (отсюда огромное число сеансовых компонентов, реализующих пустые версии этих методов). К счастью, начиная с Java EE 5, больше нет необходимости реализовывать интерфейс `SessionBean`. Однако при необходимости все же можно написать эти методы, которые будут вызываться в определенные моменты жизненного цикла компонента. Такие методы должны декорироваться определенными аннотациями.

Прежде чем перейти к описанию аннотаций, доступных для реализации методов жизненного цикла, вкратце поясним жизненный цикл сеансового компонента. Жизненный цикл сеансового компонента с сохранением состояния отличается от жизненного цикла сеансового компонента без сохранения состояния.

Жизненный цикл сеансового компонента с сохранением состояния (stateful session bean) включает три состояния: **Не существует (Does Not Exist)**, **Готов (Ready)** и **Пассивен (Passive)**, как показано на рис. 4.1.

Прежде чем сеансовый компонент с сохранением состояния будет развернут, он попадает в состояние «Не существует». После успешного

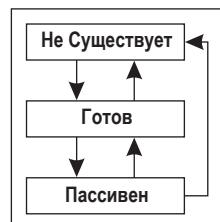


Рис. 4.1.
Жизненный цикл сеансового компонента с сохранением состояния

развертывания контейнер EJB выполняет внедрение необходимых зависимостей в компонент и он переходит в состояние «Готов». На данном этапе компонент готов предоставить свои методы для вызова клиентским приложением.

Когда сеансовый компонент с сохранением состояния находится в состоянии «Готов», контейнер EJB может решить, что нужно его пассивировать, т. е. переместить из оперативной памяти во внешнюю. Когда это происходит, компонент входит в состояние «Пассивен». Если к экземпляру сеансового компонента с сохранением состояния не обращались в течение определенного интервала времени, контейнер EJB установит компонент в состояние «Не существует». По умолчанию GlassFish переводит сеансовые компоненты с сохранением состояния в состояние «Не существует» после 90 минут отсутствия активности. Это значение по умолчанию можно изменить, для чего следует выполнить следующие шаги:

1. Перейти в консоль администрирования GlassFish.
2. Распахнуть узел **Configuration** (Конфигурация) в панели навигации слева.
3. Распахнуть узел **server-config**.
4. Щелкнуть на узле **EJB Container** (Контейнер EJB).
5. Прокрутить страницу вниз до конца и изменить значение в поле **Removal Timeout** (Тайм-аут удаления).
6. Для подтверждения изменений щелкнуть на кнопке **Save** (Сохранить), как показано на рис. 4.2.

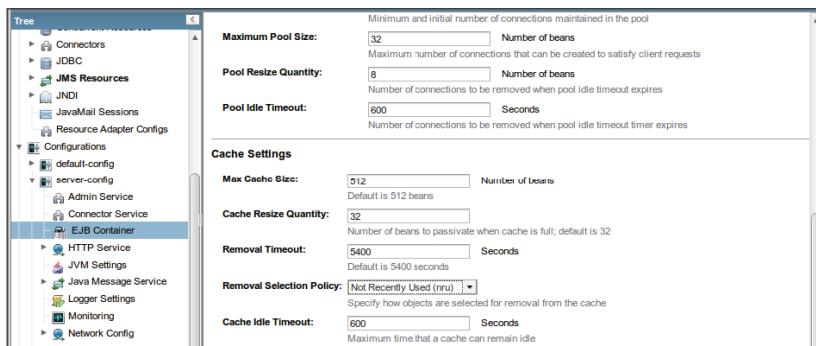


Рис. 4.2. Настройка таймаута удаления неактивных компонентов

Однако таким способом устанавливается значение таймаута для всех сеансовых компонентов с сохранением состояния. Если нужно изменить значение таймаута для конкретного компонента, необходимо

мо включить дескриптор развертывания `glassfish-ejb-jar.xml` в файл JAR с компонентом. В этом дескрипторе можно определить таймаут как значение элемента `<removal-timeout-in-seconds>`:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE glassfish-ejb-jar PUBLIC "-//GlassFish.org//DTD
    GlassFish Application Server 3.1 EJB 3.1//EN"
    "http://glassfish.org/dtds/glassfish-ejb-jar_3_1-1.dtd">
<glassfish-ejb-jar>
    <enterprise-beans>
        <ejb>
            <ejb-name>MyStatefulSessionBean</ejb-name>
            <bean-cache>
                <removal-timeout-in-seconds>
                    600
                </removal-timeout-in-seconds>
            </bean-cache>
        </ejb>
    </enterprise-beans>
</glassfish-ejb-jar>
```

Даже при том, что больше нет необходимости создавать дескриптор развертывания `ejb-jar.xml` для сеансовых компонентов (это требовалось раньше, в предыдущих версиях спецификации J2EE), его все еще можно добавлять, если это необходимо. Элемент `<ejb-name>` в дескрипторе развертывания `glassfish-ejb-jar.xml` должен соответствовать значению одноименного элемента в файле `ejb-jar.xml`. Если нет желания создавать дескриптор развертывания `ejb-jar.xml`, это значение должно соответствовать имени класса EJB. Значение таймаута для сеансового компонента с сохранением состояния должно определяться элементом `<removal-timeout-in-seconds>`. Как можно заключить из имени элемента, время измеряется в секундах. В предыдущем примере значение таймаута установлено равным 600 секундам (или 10 минутам).

Любые методы в сеансовом компоненте с сохранением состояния, декорированные аннотацией `@PostActivate`, могут быть вызваны сразу после активации сеансового компонента. Это эквивалентно реализации метода `ejbActivate()` в предыдущих версиях J2EE. Аналогично любой метод, декорированный аннотацией `@PrePassivate`, будет вызван непосредственно перед тем, как сеансовый компонент с сохранением состояния будет пассивирован. Это эквивалентно реализации метода `ejbPassivate()` в предыдущих версиях J2EE.

Когда сеансовый компонент с сохранением состояния, находящийся в состоянии «Готов», переводится в состояние «Не существует»

ет», вызываются любые его методы, декорированные аннотацией `@PreDestroy`. Если сеансовый компонент из состояния «Пассивен» переводится в состояние «Не существует», методы с аннотацией `@PreDestroy`, не вызываются. Кроме того, если клиент сеансового компонента с сохранением состояния вызывает какой-либо метод, с аннотацией `@Remove`, выполняются любые методы с аннотацией `@PreDestroy` и компонент помечается, как подлежащий удалению. Декорирование метода аннотацией `@Remove` эквивалентно реализации метода `ejbRemove()` в предыдущих версиях спецификации J2EE.

Аннотации `@PostActivate`, `@PrePassivate` и `@Remove` допустимы только для сеансовых компонентов с сохранением состояния. Аннотации `@PreDestroy` и `@PostConstruct` допустимы для сеансовых компонентов с сохранением и без сохранения состояния, а также компонентов, управляемых сообщениями.

Жизненный цикл сеансового компонента без сохранения состояния

Жизненный цикл сеансового компонента без сохранения состояния включает только два состояния: **Не существует (Does Not Exist)** и **Готов (Ready)**, как показано на рис. 4.3.

Сеансовые компоненты без сохранения состояния никогда не пассивируются. Методы сеансового компонента без сохранения состояния могут декорироваться аннотациями `@PostConstruct` и `@PreDestroy`. Так же, как в сеансовых компонентах с сохранением состояния, любые методы с аннотацией `@PostConstruct` будут вызываться при переходе компонента из состояния «Не существует» в состояние «Готов»; любые методы с аннотацией `@PreDestroy` будут вызываться при переходе компонента из состояния «Готов» в состояние «Не существует». Сеансовые компоненты без сохранения состояния никогда не пассивируются, поэтому любые аннотации `@PrePassivate` и `@PostActivate` в сеансовых компонентах без сохранения состояния просто игнорируются контейнером EJB.

Как и в случае с сеансовыми компонентами с сохранением состояния, есть возможность определить, как GlassFish будет управлять жизненным циклом сеансовых компонентов без сохранения состояния (а равно и компонентов, управляемых сообщениями, которые

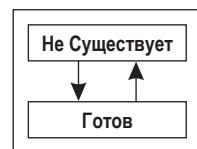


Рис. 4.3.
Жизненный
цикл сеансового
компоненты
без сохранения
состояния

будут обсуждаться в следующем разделе), через веб-консоль администрирования (см. рис. 4.4).

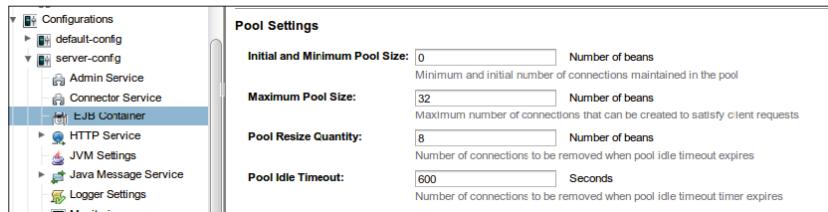


Рис. 4.4. Параметры управления жизненным циклом компонентов

- **Initial and Minimum Pool Size** (Начальный и минимальный размер пула) – минимальное число компонентов в пуле;
- **Maximum Pool Size** (Максимальный размер пула) – максимальное число компонентов в пуле;
- **Pool Resize Quantity** (Величина изменения размера пула) – число компонентов, которые будут удалены из пула по истечении таймаута простоя;
- **Pool Idle Timeout** (Таймаут простоя пула) – время бездействия (в секундах), по истечении которого компоненты будут удалены из пула.

Эти настройки влияют на все компоненты EJB, помещаемые в пул (и сеансовые компоненты без сохранения состояния, и компоненты, управляемые сообщениями). Как и в случае с сеансовыми компонентами с сохранением состояния, эти настройки могут быть переопределены в каждом конкретном случае на индивидуальной основе – путем добавления дескриптора развертывания `glassfish-ejb-jar.xml`:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE glassfish-ejb-jar PUBLIC "-//GlassFish.org//DTD
GlassFish Application Server 3.1 EJB 3.1//EN"
"http://glassfish.org/dtds/glassfish-ejb-jar_3_1-1.dtd">
<glassfish-ejb-jar>
<enterprise-beans>
<ejb>
    <ejb-name>MyStatelessSessionBean</ejb-name>
    <bean-pool>
        <steady-pool-size>10</steady-pool-size>
        <max-pool-size>60</max-pool-size>
        <resize-quantity>5</resize-quantity>
        <pool-idle-timeout-in-seconds>
            900
        </pool-idle-timeout-in-seconds>
    </bean-pool>
</ejb>
</enterprise-beans>
</glassfish-ejb-jar>
```

```
</ejb>
</enterprise-beans>
</glassfish-ejb-jar>
```

- элемент `<steady-pool-size>` соответствует параметру **Initial and Minimum Pool Size** (Начальный и минимальный размер пула) в веб-консоли GlassFish;
- элемент `<max-pool-size>` соответствует параметру **Maximum Pool Size** (Максимальный размер пула) в веб-консоли сервера GlassFish;
- элемент `<resize-quantity>` соответствует параметру **Pool Resize Quantity** (Величина изменения размера пула) в веб-консоли сервера GlassFish;
- элемент `<pool-idle-timeout-in-seconds>` соответствует **Pool Idle Timeout** (Таймаут простоя пула) в веб-консоли сервера GlassFish.

Жизненный цикл компонентов, управляемых сообщениями

Подобно сеансовые компонентам без сохранения состояния, компоненты, управляемые сообщениями имеют только два состояния – **Не существует** (Does Not Exist) и **Готов** (Ready), как показано на рис. 4.5.

Рисунок 4.5 в точности совпадает с рис. 4.3, откуда можно сделать вывод, что компоненты, управляемые сообщениями, имеют точно такой же жизненный цикл, что и компоненты без сохранения состояния.

Компоненты, управляемые сообщениями, могут иметь методы с аннотациями `@PostConstruct` и `@PreDestroy`. Методы, декорированные аннотацией `@PostConstruct`, вызываются непосредственно перед переходом компонента в состояние «Готов». Методы, декорированные аннотацией `@PreDestroy`, выполняются непосредственно перед переходом компонента в состояние «Не существует».

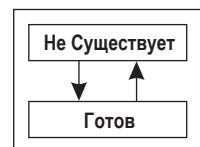


Рис. 4.5.
Жизненный
цикл сеансового
компонента
без сохранения
состояния

Служба таймеров EJB

Сеансовые компоненты без сохранения состояния и компоненты, управляемые сообщениями, могут иметь метод, который должен

выполняться через регулярные интервалы времени. Это можно реализовать при помощи **службы таймеров EJB (EJB Timer Service)**. Следующий пример демонстрирует, как использовать преимущества этой службы:

```
package net.ensode.glassfishbook;

// инструкции импортирования опущены

@Stateless
public class EjbTimerExampleBean implements EjbTimerExample
{
    private static Logger logger =
        Logger.getLogger(EjbTimerExampleBean.class.getName());

    @Resource
    TimerService timerService;

    public void startTimer(Serializable info)
    {
        Timer timer = timerService.createTimer(new Date(), 5000, info);
    }

    public void stopTimer(Serializable info)
    {
        Timer timer;
        Collection timers = timerService.getTimers();
        for (Object object : timers)
        {
            timer = ((Timer) object);
            if (timer.getInfo().equals(info))
            {
                timer.cancel();
                break;
            }
        }
    }

    @Timeout
    public void logMessage(Timer timer)
    {
        logger.info("This message was triggered by :" +
            timer.getInfo() + " at «
            + System.currentTimeMillis());
    }
}
```

Здесь выполняется внедрение интерфейса `javax.ejb.TimerService`, путем декорирования переменной экземпляра этого типа аннота-

цией `@Resource`. Затем, вызовом метода `createTimer()` экземпляра `TimerService` создается таймер.

Метод `createTimer()` имеет несколько перегруженных версий. Версия метода, которая используется в примере, принимает экземпляр `java.util.Date` в первом параметре. Этот параметр указывает время, через которое таймер должен сработать в первый раз (до его отключения). В примере используется совершенно новый экземпляр класса `Date`, который в действительности заставляет таймер остановиться сразу же. Во втором параметре методу `createTimer()` передается продолжительность ожидания (в миллисекундах), прежде чем таймер остановится снова. В предыдущем примере время таймера истекает каждые пять секунд. В третьем параметре методу `createTimer()` можно передать экземпляр любого класса, реализующего интерфейс `java.io.Serializable`. Поскольку один компонент EJB может иметь несколько таймеров, выполняемых одновременно, этот третий параметр используется, чтобы однозначно определить каждый из таймеров. Если таймеры идентифицировать не требуется, в третьем параметре можно передать `null`.



Вызов метода `TimerService.createTimer()` должен выполняться из клиента. Если поместить вызов этого метода в метод с аннотацией `@PostConstruct`, чтобы запустить таймер автоматически, при переходе компонента в состояние «Готов», это приведет к исключению `IllegalStateException`.

Остановить таймер можно вызовом его метода `cancel()`. Не существует способа получить конкретный таймер, связанный с EJB. Для этого нужно должны вызвать метод `getTimers()` экземпляра `TimerService`, связанного с компонентом с EJB. Этот метод вернет коллекцию со всеми таймерами, связанными с EJB. Затем можно выполнить обход этой коллекции и корректно отменить конкретный таймер, вызвав его метод `getInfo()`. Этот метод вернет сериализуемый объект, переданный в вызов метода `createTimer()`.

Наконец, любой метод компонента EJB, декорированный аннотацией `@Timeout`, будет автоматически вызываться по истечении времени таймера. Методы с этой аннотацией должны возвращать `void` и принимать единственный параметр типа `javax.ejb.Timer`. В данном примере метод просто выводит сообщение в журнал сервера.

Следующий класс – это автономный клиент для предыдущего EJB:

```
package net.ensode.glassfishbook;

import javax.ejb.EJB;

public class Client
{
    @EJB
    private static EjbTimerExample ejbTimerExample;

    public static void main(String[] args)
    {
        try
        {
            System.out.println("Starting timer 1...");
            ejbTimerExample.startTimer("Timer 1");
            System.out.println("Sleeping for 2 seconds...");
            Thread.sleep(2000);
            System.out.println("Starting timer 2...");
            ejbTimerExample.startTimer("Timer 2");
            System.out.println("Sleeping for 30 seconds...");
            Thread.sleep(30000);
            System.out.println("Stopping timer 1...");
            ejbTimerExample.stopTimer("Timer 1");
            System.out.println("Stopping timer 2...");
            ejbTimerExample.stopTimer("Timer 2");
            System.out.println("Done.");
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
```

Этот клиент просто запускает таймер, после чего ожидает в течение нескольких секунд и запускает второй таймер. Затем он бездействует в течение 30 секунд и останавливает оба таймера. После развертывания EJB и запуска клиента в журнале сервера должны появиться примерно такие записи:

```
[2013-08-26T20:44:55.180-0400] [glassfish 4.0] [INFO] []
[net.ensode.glassfishbook.EjbTimerExampleBean] [tid:
- ThreadID=147 _ThreadName=__ejb-thread-pool1] [timeMillis:
- 1377564295180] [levelValue: 800] [{

This message was triggered by :Timer 1 at 1377564295180]

[2013-08-26T20:44:57.203-0400] [glassfish 4.0] [INFO] []
[net.ensode.glassfishbook.EjbTimerExampleBean] [tid:
- ThreadID=148 _ThreadName=__ejb-thread-pool2] [timeMillis:
```

```

1377564297203] [levelValue: 800] [

This message was triggered by :Timer 2 at 1377564297203]

[2013-08-26T20:44:58.888-0400] [glassfish 4.0] [INFO] []
[net.ensode.glassfishbook.EjbTimerExampleBean] [tid:
_ThreadID=149 _ThreadName=__ejb-thread-pool3] [timeMillis:
1377564298888] [levelValue: 800] [

This message was triggered by :Timer 1 at 1377564298888]

[2013-08-26T20:45:01.156-0400] [glassfish 4.0] [INFO] []
[net.ensode.glassfishbook.EjbTimerExampleBean] [tid:
_ThreadID=150 _ThreadName=__ejb-thread-pool4] [timeMillis:
1377564301156] [levelValue: 800] [

This message was triggered by :Timer 2 at 1377564301156]

```

Эти записи создаются каждый раз, когда истекает время одного из таймеров.

Таймеры EJB на основе календаря

Пример в предыдущем разделе имеет один недостаток. Метод `startTimer()` в сеансовом компоненте, запускающий таймер, должен вызываться из клиента. Это ограничение мешает созданию таймеров, запускающихся сразу после развертывания компонента.

В Java EE 6 появилась поддержка таймеров на основе календаря. Эти таймеры позволяют организовать автоматический вызов одного или более методов сеансового компонента в определенные дни и в определенное время. Например, можно настроить один из методов так, чтобы он вызывался каждый вечер в 20:10:

```

package com.ensode.glassfishbook.calendarbasedtimer;

import java.util.logging.Logger;

import javax.ejb.Stateless;
import javax.ejb.LocalBean;
import javax.ejb.Schedule;

@Stateless
@LocalBean
public class CalendarBasedTimerEjbExampleBean {

    private static Logger logger = Logger.getLogger(
        CalendarBasedTimerEjbExampleBean.class.getName());

    @Schedule(hour = "20", minute = "10")

```

```

public void logMessage() {
    logger.info("This message was triggered at:"
        + System.currentTimeMillis());
}
}

```

Как видите, время, когда метод будет вызываться, определяется с помощью аннотации `javax.ejb.Schedule`. В данном случае вызов метода настроен на 20:10 установкой атрибута `hour` аннотации `@Schedule` в значение "20" и атрибута `minute` – в значение "10" (значение атрибута `hour` задается в 24-часовом формате).

Аннотация `@Schedule` еще несколько атрибутов, обеспечивающих большую гибкость в определении времени вызова метода, например: в третью пятницу каждого месяца, в последний день месяца и т. д.

В табл. 4.3 перечислены все атрибуты аннотации `@Schedule`, которые позволяют управлять временем вызова аннотированного метода.

Таблица 4.3. Атрибуты аннотации `@Schedule`

Атрибут	Описание	Примеры	Значение по умолчанию
<code>dayOfMonth</code>	День месяца	"3": третий день месяца. "Last": последний день месяца. "-2": за два дня до конца месяца. "1st Tue": первый вторник месяца.	"*"
<code>dayOfWeek</code>	День недели	"3": каждая среда. "Thu": каждый четверг	"*"
<code>hour</code>	Час дня (в 24-часовом формате)	"14": в 2 часа дня.	"0"
<code>minute</code>	Минута часа	"10": через десять минут после начала часа.	"0"
<code>month</code>	Месяц года	"2": февраль. "March": март.	"*"
<code>second</code>	Секунда минуты	"5": через пять секунд после начала минуты.	"0"
<code>timezone</code>	Идентификатор часового пояса	"America/New York"	" "
<code>year</code>	Четыре цифры года	"2010"	"*"

В дополнение к одиночным значениям большинство атрибутов принимает звездочку ("*") как шаблонный символ, означающий, что аннотируемый метод будет выполняться каждую единицу времени (каждый день, час и т. д.).

Кроме того, есть возможность указать несколько значений, разделяя их запятыми. Например, если нужно, чтобы метод выполнялся каждый вторник и четверг, его можно отметить аннотацией `@Schedule(dayOfWeek="Tue, Thu")`.

Допускается указывать диапазон значений, в котором первое и последнее значения разделяются дефисом (-). Так, чтобы метод выполнялся каждый день с понедельника по пятницу, его можно отметить аннотацией `@Schedule(dayOfWeek="Mon-Fri")`.

Кроме того, можно также указать, что метод должен выполняться каждую *n*-ю единицу времени (например, каждый день, каждые два часа, каждые десять минут и т. д.). Для этого можно использовать формат записи `@Schedule(hour="*/12")`: это значит, что метод будет выполняться каждые 12 часов.

Как видите, аннотация `@Schedule` дает большую гибкость определения времени выполнения методов. Еще один ее плюс заключается в том, что отпадает необходимость в вызове со стороны клиента. Наконец, данная аннотация имеет преимущество использования сноп-подобного синтаксиса, благодаря чему разработчики, знакомые с этим инструментом в Unix, не будут чувствовать ни малейших затруднений при использовании `@Schedule`.

Безопасность EJB

Компоненты Enterprise JavaBeans позволяют декларативно решать, какие пользователи могут получить доступ к их методам. Например, некоторые методы могут быть доступны только пользователям с определенной ролью. Согласно самому распространенному сценарию, только пользователи с ролью администратора могут добавлять, удалять или редактировать информацию о других пользователях в системе.

Следующий пример представляет немного измененную версию сеансового компонента DAO, рассматривавшегося выше в этой главе. В этой версии некоторые методы, ранее являвшиеся частными (`private`), были сделаны открытыми (`public`). Кроме того, сеансовый компонент изменен так, что только пользователи с определенными ролями смогут получать доступ к его методам.

```
package net.ensode.glassfishbook;

// инструкции импортирования опущены

@Stateless
@RolesAllowed("appadmin")
public class CustomerDaoBean implements CustomerDao
{
    @PersistenceContext
    private EntityManager entityManager;

    @Resource(name = "jdbc/_CustomerDBPool")
    private DataSource dataSource;

    public void saveCustomer(Customer customer)
    {
        if (customer.getCustomerId() == null)
        {
            saveNewCustomer(customer);
        }
        else
        {
            updateCustomer(customer);
        }
    }

    public Long saveNewCustomer(Customer customer)
    {
        entityManager.persist(customer);

        return customer.getCustomerId();
    }

    public void updateCustomer(Customer customer)
    {
        entityManager.merge(customer);
    }

    @RolesAllowed(
    { "appuser", "appadmin" })
    public Customer getCustomer(Long customerId)
    {
        Customer customer;

        customer = entityManager.find(Customer.class, customerId);

        return customer;
    }

    public void deleteCustomer(Customer customer)
```

```
{  
    entityManager.remove(customer);  
}  
}
```

Как видите, здесь, при помощи аннотации `@RolesAllowed`, явно объявляется, какие роли имеют право доступа к методам. Эта аннотация может принимать в качестве параметра одну строку или массив строк. Когда в качестве параметра используется одна строка, доступ к методу могут получить только пользователи с ролью, указанной в параметре. Если же в параметре указывается массив строк, доступ к методу разрешается для пользователей с любой из ролей, указанных в массиве.

Аннотацией `@RolesAllowed` можно декорировать весь класс EJB, если она должна применяться ко всем методам в EJB, либо один или несколько методов. В последнем случае аннотация применяются только к конкретному методу. Если, как в данном примере, аннотацией `@RolesAllowed` декорирован и класс EJB, и один или несколько его методов, аннотация на уровне метода пользуется преимуществом.

Роли приложения должны быть отображены на названия группы области безопасности (подробности смотрите в главе 9, «Безопасность приложений Java EE»). Это отображение, а также используемые в нем области должны определяться в дескрипторе развертывания `glassfish-ejb-jar.xml`:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE glassfish-ejb-jar PUBLIC "-//GlassFish.org//DTD
    GlassFish Application Server 3.1 EJB 3.1//EN"
    "http://glassfish.org/dtds/glassfish-ejb-jar_3_1-1.dtd">
<glassfish-ejb-jar>
    <security-role-mapping>
        <role-name>appuser</role-name>
        <group-name>appuser</group-name>
    </security-role-mapping>
    <security-role-mapping>
        <role-name>appadmin</role-name>
        <group-name>appadmin</group-name>
    </security-role-mapping>
    <enterprise-beans>
        <ejb>
            <ejb-name>CustomerDaoBean</ejb-name>
            <ior-security-config>
                <as-context>
                    <auth-method>username_password</auth-method>
                    <realm>file</realm>
                    <required>true</required>
                </as-context>
            </ior-security-config>
        </ejb>
    </enterprise-beans>
</glassfish-ejb-jar>
```

```
</as-context>
</ior-security-config>
</ejb>
</enterprise-beans>
</glassfish-ejb-jar>
```

Элемент `<security-role-mapping>` дескриптора развертывания `glassfish-ejb-jar.xml` определяет отображение между ролями приложения и группой области безопасности. Значение подэлемента `<role-name>` должно содержать роль приложения и соответствовать значению, используемому в аннотации `@RolesAllowed`. Значение подэлемента `<group-name>` должно содержать имя группы безопасности в области безопасности, используемой EJB. В предыдущем примере определяется отображение двух ролей приложения в соответствующие группы в области безопасности. Хотя в данном конкретном примере имя роли приложения и группы безопасности совпадают, на практике это не является обязательным требованием.



Автоматическое соответствие ролей группам безопасности. Можно установить автоматическое соответствие любых ролей приложения именам групп в области безопасности. Для этого в веб-консоли GlassFish следует щелкнуть на узле **Configuration** (Конфигурация), затем на узле **Security** (Безопасность), выбрать флашок **Default Principal to Role Mapping** (Участник по умолчанию для отображения роли) и сохранить настройки.

Как показано в предыдущем примере, область безопасности для аутентификации определяется в подэлементе `<realm>` элемента `<as-context>`. Значение этого подэлемента должно соответствовать имени области безопасности на сервере приложений. Другими подэлементами элемента `<as-context>` являются: `<auth-method>`, значением которого может быть только `username_password`, и `<required>`, значением которого в свою очередь могут быть только `true` и `false`.

Аутентификация клиента

Если клиентский код, получающий доступ к защищенному компоненту EJB, является частью веб-приложения, где пользователь уже прошел аутентификацию, для определения прав доступа пользователя будут использоваться его учетные данные.

Автономные клиенты должны запускаться утилитой `appclient`. Ниже представлен типичный клиент для предыдущего защищенного сеансового компонента:

```

package net.ensode.glassfishbook;

import javax.ejb.EJB;

public class Client
{
    @EJB
    private static CustomerDao customerDao;

    public static void main(String[] args)
    {
        Long newCustomerId;

        Customer customer = new Customer();
        customer.setFirstName("Mark");
        customer.setLastName("Butcher");
        customer.setEmail("butcher@phony.org");

        System.out.println("Saving New Customer...");
        newCustomerId = customerDao.saveNewCustomer(customer);

        System.out.println("Retrieving customer...");
        customer = customerDao.getCustomer(newCustomerId);
        System.out.println(customer);
    }
}

```

Как видите, здесь ничего не делается для аутентификации пользователя. Сеансовый компонент просто внедряется в код аннотацией `@EJB` и используется как обычно. Работоспособность этого примера объясняется тем, что все хлопоты об аутентификации пользователя берет на себя утилита `appclient`, если клиент запускается с ее помощью:

```
appclient -client ejbsecurityclient.jar
```

Утилита `appclient` запросит имя пользователя и пароль через диалог, как показано на рис. 4.6, если код клиента попробует вызвать защищенный метод компонента EJB.



Рис. 4.6. Утилита `appclient` запросит имя пользователя и пароль

Если предположить, что учетные данные верны и пользователь имеет соответствующие полномочия, код EJB выполнится, и мы должны увидеть вывод класса Client:

```
Saving New Customer...
Retrieving customer...
customerId = 29
firstName = Mark
lastName = Butcher
email = butcher@phony.org
```

Резюме

В этой главе мы узнали, как реализовать бизнес-логику с помощью сеансовых компонентов с сохранением и без сохранения состояния, а также познакомились с особенностями реализации компонентов, управляемых сообщениями.

Здесь мы также увидели, как использовать преимущества транзакционной природы EJB для упрощения реализации шаблона проектирования – «Объект доступа к данным» (DAO).

Дополнительно мы обсудили транзакции, управляемые контейнером, и показали, как ими управлять с помощью соответствующих аннотаций. Мы также объяснили, как реализовать транзакции, управляемые компонентом, на случай, если транзакций, управляемых контейнером, оказывается недостаточно для удовлетворения наших требований.

Были рассмотрены жизненные циклы различных типов компонентов Enterprise JavaBeans; в частности, было показано, как автоматически вызывать методы компонентов EJB контейнером в определенные моменты жизненного цикла.

Также мы рассмотрели, как периодически вызывать методы компонентов EJB с помощью контейнера, используя возможности службы таймеров EJB.

Наконец, мы узнали, как обеспечить доступность методов компонентов EJB только для авторизованных пользователей, аннотируя классы EJB и/или методы и добавляя соответствующие записи в файл дескриптора развертывания `glassfish-ejb-jar.xml`.

В следующей главе мы познакомимся с механизмом контекстов и внедрения зависимостей (Contexts and Dependency Injection, CDI).



ГЛАВА 5.

Контексты и внедрение зависимостей

Контексты и внедрение зависимостей (**Contexts and Dependency Injection, CDI**) – новое дополнение к спецификации Java EE, появившееся в Java EE 6. Оно предоставляет несколько возможностей, которых прежде не было у разработчиков Java EE – например, возможность использовать любой компонент JavaBean в качестве управляемого JSF-компонентта, включая сеансовые компоненты с сохранением и без сохранения состояния. Как следует из названия, CDI упрощает внедрение зависимостей в приложениях Java EE.

В этой главе мы затронем следующие темы:

- ❖ именованные компоненты;
- ❖ внедрение зависимостей;
- ❖ контексты;
- ❖ квалификаторы.

Именованные компоненты

Механизм CDI предоставляет возможность присваивать имена компонентам с помощью аннотации `@Named`. Поддержка именования позволяет легко внедрять компоненты в другие классы, которые от них зависят (см. раздел «Внедрение зависимостей») и ссылаться на них из JSF-страниц с помощью унифицированного языка выражений.

Следующий пример демонстрирует аннотацию `@Named` в действии:

```
package net.ensode.cdidependencyinjection.beans;

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

@Named
@RequestScoped
```

```
public class Customer {  
  
    private String firstName;  
    private String lastName;  
  
    public String getFirstName() {  
        return firstName;  
    }  
  
    public void setFirstName(String firstName) {  
        this.firstName = firstName;  
    }  
  
    public String getLastName() {  
        return lastName;  
    }  
  
    public void setLastName(String lastName) {  
        this.lastName = lastName;  
    }  
}
```

Как видите, для создания классов именованных компонентов нужно всего лишь декорировать их аннотацией `@Named`. По умолчанию имя компонента совпадает с именем класса (с той лишь разницей, что первая буква преобразована в нижний регистр). В данном примере имя компонента было бы `customer`. Если потребуется использовать другое имя, можно воспользоваться атрибутом `value` аннотации `@Named`. Например, чтобы назначить предыдущему компоненту имя `customerBean`, потребуется изменить аннотацию `@Named` следующим образом:

```
@Named(value="customerBean")
```

или просто:

```
@Named("customerBean")
```

Имя атрибута `value` указывать необязательно, можно просто указать имя компонента, а атрибут `value` будет подразумеваться по умолчанию.

Имя CDI может использоваться для доступа к компоненту из JSF-страницы с использованием унифицированного языка выражений:

```
<?xml version='1.0' encoding='UTF-8'?>  
<!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://java.sun.com/jsf/html">
<h:head>
    <title>Enter Customer Information</title>
</h:head>
<h:body>
    <h:form>
        <h:panelGrid columns="2">
            <h:outputLabel for="firstName" value="First Name"/>
            <h:inputText id="firstName"
                value="#{customer.firstName}"/>
            <h:outputLabel for="lastName" value="Last Name"/>
            <h:inputText id="lastName"
                value="#{customer.lastName}"/>
        <h:panelGroup/>
    </h:panelGrid>
</h:form>
</h:body>
</html>
```

Как видите, доступ к именованному компоненту можно получить из JSF-страницы так же, как к стандартным управляемым компонентам JSF. Это позволяет JSF получить доступ к любому именованному компоненту и отвязать код на Java от API JSF.

После развертывания и запуска этого простого приложения, оно будет выглядеть, как показано на рис. 5.1.

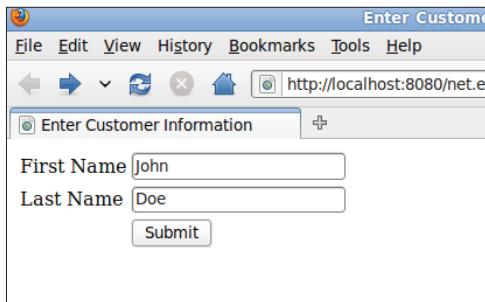


Рис. 5.1. Простое приложение, использующее именованный компонент

Внедрение зависимостей

Внедрение зависимостей (dependency injection) – это технология внедрения внешних зависимостей в класс Java. Поддержка внедрения зависимостей была введена в Java EE 5, в виде аннотации @Resource. Однако эта аннотация ограничивается внедрением ресурсов, таких

как соединения с базами данных, ресурсы JMS и т. д. Введение механизма CDI добавило аннотацию `@Inject`, которая может использоваться для внедрения экземпляров классов Java в любые зависящие от них объекты.

Приложения JSF обычно следуют шаблону проектирования «Модель-Представление-Контроллер» (**Model-View-Controller, MVC**), в котором одни управляемые компоненты JSF берут на себя роль контроллеров, а другие – роль моделей. Такой подход обычно необходим, когда управляемый компонент, выступающий в роли контроллера, должен иметь доступ к одному или нескольким управляемым компонентам-моделям.

Из-за частого применения на практике шаблона, описанного в предыдущем абзаце, разработчики часто задаются вопросом: «Как получить доступ к одному управляемому компоненту из другого?». Сделать это можно несколькими способами, но до CDI ни один из них нельзя было назвать простым. Самый простой способ состоял в том, чтобы объявить управляемое свойство в управляемом компоненте-контроллере. Это требовало вносить изменения в конфигурационный файл приложения `faces-config.xml`. Другой подход предусматривал использование кода, как показано ниже:

```
ELContext elc = FacesContext.getCurrentInstance().  
getELContext();  
SomeBean someBean =  
    (SomeBean) FacesContext.getCurrentInstance().  
getApplication()  
.getELResolver().getValue(elc, null, "someBean");
```

Здесь `someBean` – это имя компонента, указанное в конфигурационном файле приложения `faces-config.xml`. Как видите, ни один из описанных подходов не является простым или, по крайней мере, запоминающимся. К счастью, в настоящее время нет необходимости писать такой код, благодаря возможностям механизма внедрения зависимостей CDI, как показано ниже:

```
package net.ensode.cdidependencyinjection.ejb;  
  
import java.util.logging.Logger;  
import javax.inject.Inject;  
import javax.inject.Named;  
  
@Named  
@RequestScoped  
public class CustomerController {  
  
    private static final Logger logger = Logger.getLogger(
```

```
CustomerController.class.getName());  
  
@Inject  
private Customer customer;  
  
public String saveCustomer() {  
    logger.info("Saving the following information \n"  
        + customer.toString());  
  
    // В действующем приложении здесь был бы код  
    // выполняющий сохранение данных customer в базе данных.  
    return "confirmation";  
}  
}
```

Обратите внимание, что для инициализации экземпляра `customer` нужно всего лишь декорировать его аннотацией `@Inject`. Когда сервер приложений будет создавать этот компонент, он автоматически внедрит экземпляр компонента `Customer` в это поле. Заметьте, что внедренный компонент используется в методе `saveCustomer()`. Как видите, механизм CDI ускоряет доступ к одному компоненту из другого компонента, что существенно отличает его от вариантов кода, который приходилось использовать в предыдущих версиях спецификации Java EE.

Квалификаторы CDI

В некоторых случаях тип переменной для внедрения может быть интерфейсом или суперклассом Java, но нас может интересовать внедрение подкласса этого суперкласса либо класса, реализующего интерфейс. Для таких случаев в CDI имеются квалификаторы, которые можно использовать, чтобы указать конкретный тип, который нужно внедрить в код.

Квалификатор CDI – это аннотация, которая в свою очередь должна быть декорирована аннотацией `@Qualifier`. Эта аннотация затем может использоваться для декорирования конкретного подкласса или реализации интерфейса, которую требуется квалифицировать. Кроме того, поле внедрения в коде клиента также должно быть декорировано квалификатором.

Предположим, что в приложении могут различаться обычные и постоянные клиенты, которым может быть присвоен статус «премиум-клиентов». Чтобы предусмотреть иную обработку премиум-

клиентов, можно расширить компонент `Customer` и декорировать его следующим квалификатором:

```
package net.ensode.cdidependencyinjection.qualifiers;

import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface Premium { }
```

Как уже говорилось выше, квалификаторы – это стандартные аннотации. Обычно они сохраняются средой выполнения и могут применяться к методам, полям, параметрам или типам, как показано в предыдущем примере значением аннотации `@Retention`. Единственная разница между квалификатором и стандартной аннотацией – квалификаторы декорируются аннотацией `@Qualifier`.

После определения квалификатора, его можно использовать для декорирования конкретного подкласса или реализации интерфейса:

```
package net.ensode.cdidependencyinjection.beans;

import javax.enterprise.context.RequestScoped;
import javax.inject.Named;
import net.ensode.cdidependencyinjection.qualifiers.Premium;

@Named
@RequestScoped
@Premium
public class PremiumCustomer extends Customer {

    private Integer discountCode;

    public Integer getDiscountCode() {
        return discountCode;
    }

    public void setDiscountCode(Integer discountCode) {
        this.discountCode = discountCode;
    }
}
```

После декорирования квалификатором конкретного экземпляра, можно использовать квалификатор в клиентском коде, чтобы указать точный тип нужной зависимости:

```
package net.ensode.cdidependencyinjection.beans;

import java.util.Random;
import java.util.logging.Logger;
import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.inject.Named;
import net.ensode.cdidependencyinjection.qualifiers.Premium;

@Named
@RequestScoped
public class CustomerController {

    private static final Logger logger = Logger.getLogger(
        CustomerController.class.getName());
    @Inject
    @Premium
    private Customer customer;

    public String saveCustomer() {

        PremiumCustomer premiumCustomer = (PremiumCustomer)
customer;

        premiumCustomer.setDiscountCode(generateDiscountCode());

        logger.info("Saving the following information \n"
            + premiumCustomer.getFirstName() + " "
            + premiumCustomer.getLastName()
            + ", discount code = "
            + premiumCustomer.getDiscountCode());

        // В действующем приложении здесь был бы код
        // выполняющий сохранение данных customer в базе данных.
        return "confirmation";
    }

    public Integer generateDiscountCode() {
        return new Random().nextInt(100000);
    }
}
```

Здесь поле `customer`, куда будет внедряться экземпляр `PremiumCustomer`, декорировано квалификатором `@Premium`, потому что этот класс также декорирован квалификатором `@Premium`.

Что касается перехода к JSF-страницам, мы просто получаем доступ к нашему именованному компоненту, как обычно, используя его имя:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!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://java.sun.com/jsf/html">
  <h:head>
    <title>Enter Customer Information</title>
  </h:head>
  <h:body>
    <h:form>
      <h:panelGrid columns="2">
        <h:outputLabel for="firstName"
                      value="First Name"/>
        <h:inputText id="firstName"
                     value="#{premiumCustomer.firstName}" />
        <h:outputLabel for="lastName" value="Last Name"/>
        <h:inputText id="lastName"
                     value="#{premiumCustomer.lastName}" />
        <h:outputLabel for="discountCode"
                      value="Discount Code"/>
        <h:inputText id="discountCode"
                     value="#{premiumCustomer.discountCode}" />
      <h:panelGroup/>
      <h:commandButton value="Submit"
                       action="#{customerController.saveCustomer}" />
    </h:panelGrid>
  </h:form>
  </h:body>
</html>
```

В этом примере используется имя по умолчанию компонента, которое идентично имени класса (с первой буквой в нижнем регистре).

Это простое приложение отображается и функционирует точно так же, как и «обычное» (то есть, не использующее CDI) приложение JSF, как показано на рис. 5.2.

Контексты именованных компонентов

Подобно управляемым компонентам JSF, именованные компоненты CDI имеют некоторый контекст. Это означает, что они являются контекстно-зависимыми объектами. Когда возникает потребность в име-

нованном компоненте – для внедрения или потому, что на него имеется ссылка в JSF-странице, – механизм CDI отыскивает экземпляр компонента в контексте, которому он принадлежит, и внедряет его в зависимый код. Если не найдется ни одного подходящего экземпляра, он будет создан и сохранен в соответствующем контексте для последующего использования. Существуют разные контексты, в которых могут существовать компоненты.

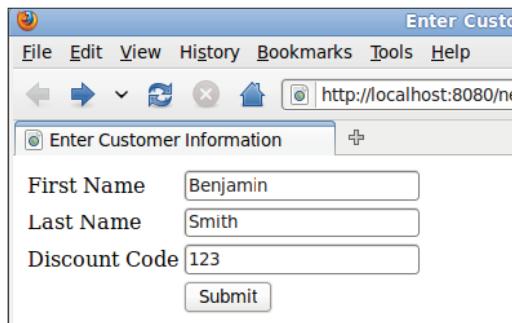


Рис. 5.2. Внешний вид приложения, использующего механизм CDI

В табл. 5.1 перечислены различные допустимые контексты, поддерживаемые механизмом CDI:

Таблица 5.1. Контексты именованных компонентов

Контекст	Аннотация	Описание
Запрос	@RequestScoped	Компоненты в контексте запроса существуют в течение единичного запроса. Под единичным запросом подразумевается HTTP-запрос, вызов метода EJB, вызов веб-службы или отправка JMS-сообщения.
Диалог	@ConversationScoped	Контекст диалога может охватить несколько запросов, но обычно короче, чем контекст сеанса.
Сеанс	@SessionScoped	Компоненты в контексте сеанса, действительны для всех запросов в пределах HTTP-сеанса. Каждый пользователь приложения получает свой собственный экземпляр компонента в контексте сеанса

Контекст	Аннотация	Описание
Приложение	@ApplicationScoped	Компоненты в контексте приложения, действительны на протяжении всего времени жизни приложения. Компоненты в этом контексте совместно используются всеми сессиями пользователей.
Зависимый	@Dependent	Компоненты в зависимом контексте, не используются совместно. Каждый раз, когда внедряется компонент с зависимым контекстом, создается новый экземпляр

Как видите, CDI включает все контексты, поддерживаемые JSF, и помимо этого добавляет ряд собственных. Контекст запроса (request scope) в CDI отличается от контекста запроса JSF; в нем запрос не обязательно является HTTP-запросом. Это может быть вызов метода EJB, вызов веб-службы или отправка сообщения JMS.

Контекст диалога (conversation scope) отсутствует в JSF. Этот контекст более продолжителен по времени, чем контекст запроса, но короче, чем контекст сеанса. Обычно он охватывает три страницы и более. В классы, желающие получить доступ к компоненту, определенному в контексте диалога, должен быть внедрен экземпляр `javax.enterprise.context.Conversation`. В той точке, где требуется начать диалог, нужно вызвать метод `begin()` данного объекта. В точке, где требуется завершить диалог, нужно вызвать метод `end()` данного объекта.

Контекст сеанса (session scope) CDI действует точно так же, как его «коллега» из JSF. Жизненный цикл компонентов в этом контексте связан со сроком жизни HTTP-сеанса.

Контекст приложения (application scope) CDI действует так же, как эквивалентный контекст в JSF. Компоненты в контексте приложения привязываются к сроку жизни приложения. Для приложения существует единственный экземпляр каждого компонента в контексте приложения. Это значит, что тот же самый экземпляр доступен во всех HTTP-сессиях.

Подобно контексту диалога, зависимый контекст (dependent scope) CDI отсутствует в JSF. Каждый раз, когда возникает необходимость получить компонент в зависимом контексте – обычно, когда он внедряется в класс, который от него зависит, – создается новый экземпляр.

Предположим, что нужно организовать накопление данных, вводимых пользователем, в единственном именованном компоненте. Однако компонент имеет настолько большое число полей, что желат-

тельно разделить ввод данных на несколько страниц. Это довольно распространенная ситуация, и с ней не так легко было справиться в предыдущих версиях JSF (в JSF 2.2 был добавлен механизм последовательностей Faces Flows, решающий эту проблему; см. главу 2, «JavaServer Faces»). Причина кроется в нетривиальности управления этими технологиями. Нетривиальность заключается в том, что класс можно поместить или в контекст запроса, когда экземпляры класса уничтожаются после каждого отдельного запроса и их данные теряются, или в контекст сеанса, когда экземпляры класса остаются в памяти еще долгое время после того, как необходимость в них отпала.

Для подобных случаев идеально подходит контекст диалога, как показано в следующем примере:

```
package net.ensode.conversationscope.model;

import java.io.Serializable;
import javax.enterprise.context.ConversationScoped;
import javax.inject.Named;
import org.apache.commons.lang.builder.
ReflectionToStringBuilder;

@Named
@ConversationScoped
public class Customer implements Serializable {

    private String firstName;
    private String middleName;
    private String lastName;
    private String addrLine1;
    private String addrLine2;
    private String addrCity;
    private String state;
    private String zip;
    private String phoneHome;
    private String phoneWork;
    private String phoneMobile;

    // методы чтения и записи опущены

    @Override
    public String toString() {
        return ReflectionToStringBuilder.reflectionToString(this);
    }
}
```

Здесь, с помощью аннотации `@ConversationScoped`, объявляется, что компонент находится в контексте диалога. Компоненты в контексте переговоров также должны реализовать интерфейс `java.io.Serializable`. Кроме реализации этих двух требований в нашем

коде нет ничего особенного. Это простой компонент JavaBean с закрытыми (`private`) свойствами и соответствующими им методами чтения и записи.



В этом примере используется библиотека commons-lang из проекта Apache, чтобы облегчить реализацию метода `toString()` компонента. Библиотека commons-lang имеет несколько вспомогательных методов вроде этого, которые реализуют часто востребованную функциональность. Библиотека commons-lang доступна в центральных репозиториях Maven, по адресу: <http://commons.apache.org/lang>.

Помимо поля для внедрения компонента в контексте диалога, клиентский код должен содержать поле для внедрения экземпляра `javax.enterprise.context.Conversation`, как показано в следующем примере:

```
package net.enseode.conversationscope.controller;

import java.io.Serializable;
import javax.enterprise.context.Conversation;
import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.inject.Named;
import net.enseode.conversationscope.model.Customer;

@Named
@RequestScoped
public class CustomerInfoController implements Serializable {

    @Inject
    private Conversation conversation;
    @Inject
    private Customer customer;

    public String customerInfoEntry() {
        conversation.begin();
        System.out.println(customer);
        return "page1";
    }

    public String navigateToPage1() {
        System.out.println(customer);
        return "page1";
    }

    public String navigateToPage2() {
        System.out.println(customer);
        return "page2";
    }
}
```

```
}

public String navigateToPage3() {
    System.out.println(customer);
    return "page3";
}

public String navigateToConfirmationPage() {
    System.out.println(customer);
    conversation.end();
    return "confirmation";
}
}
```

Диалоги могут быть длительными или скоротечными. Скоротечные диалоги заканчиваются в конце запроса, длительные же охватывают множество запросов. На практике в большинстве случаев используются длительные диалоги, хранящие ссылки на компонент в контексте диалога в течение множества HTTP-запросов в веб-приложении.

Длительные переговоры начинаются вызовом метода `begin()` внешнего экземпляра `Conversation` и заканчиваются вызовом метода `end()` того же объекта.

JSF-страницы получают доступ к компонентам CDI как обычно:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!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://java.sun.com/jsf/html">
    <h:head>
        <title>Customer Information</title>
    </h:head>
    <h:body>
        <h3>Enter Customer Information (Page 1 of 3)</h3>
        <h:form>
            <h:panelGrid columns="2">
                <h:outputLabel for="firstName" value="First Name"/>
                <h:inputText id="firstName"
                            value="#{customer.firstName}" />
                <h:outputLabel for="middleName" value="Middle Name"/>
                <h:inputText id="middleName"
                            value="#{customer.middleName}" />
                <h:outputLabel for="lastName" value="Last Name"/>
                <h:inputText id="lastName"
                            value="#{customer.lastName}" />
            <h:panelGroup/>
            <h:commandButton value="Next"
```

```

        action="#{customerInfoController.navigateToPage2}" />
    </h:panelGrid>
</h:form>
</h:body>
</html>

```

При переходе от одной страницы к другой сохраняется один и тот же экземпляр компонента в контексте диалога. Поэтому все введенные пользователем данные сохраняются до конца диалога. Когда будет вызван метод `end()` объекта диалога, диалог прекращается и компонент в контексте диалога уничтожается.

Поддержка контекста диалога значительно упрощает задачу реализации пользовательских интерфейсов в стиле «Мастер», где данные могут вводиться в череде последовательных страниц (см. рис. 5.3).



Рис. 5.3. Первая страница диалога

В данном примере, после щелчка на кнопке **Next** (Далее) на первой странице, можно увидеть частично заполненный компонент в журнале сервера GlassFish:

```

INFO: mailto:net.ensode.conversationscope.model.
Customer@6e1c51b4 [firstName=Daniel,middleName=,lastName=Jone
s,addrLine1=,addrLine2=,addrCity=,state=AL,zip=<null>,phoneHom
e=<null>,phoneWork=<null>,phoneMobile=<null>]

```

Далее в окне браузера выводится вторая страница нашего простого мастера (см. рис. 5.4).

После щелчка на кнопке **Next** (Далее) можно увидеть, что были заполнены дополнительные свойства компонента в контексте диалога:

```

INFO: net.ensode.conversationscope.model.Customer@6e1c51b4
[firstName=Daniel,middleName=,lastName=Jones,addrLine1=123
Basketball Ct,addrLine2=,addrCity=Montgomery,state=AL,zip=3610
1,phoneHome=<null>,phoneWork=<null>,phoneMobile=<null>]

```

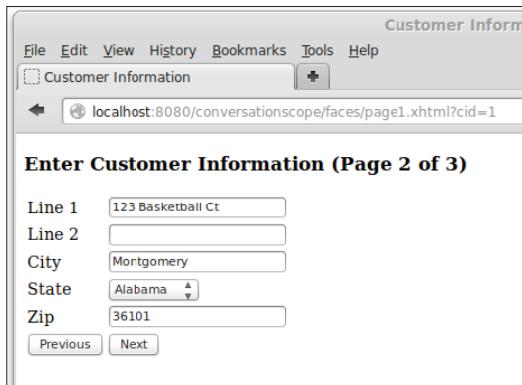


Рис. 5.4. Вторая страница диалога

После отправки третьей страницы мастера (см. рис. 5.5) заполняются дополнительные свойства компонента, соответствующие полям на этой странице.

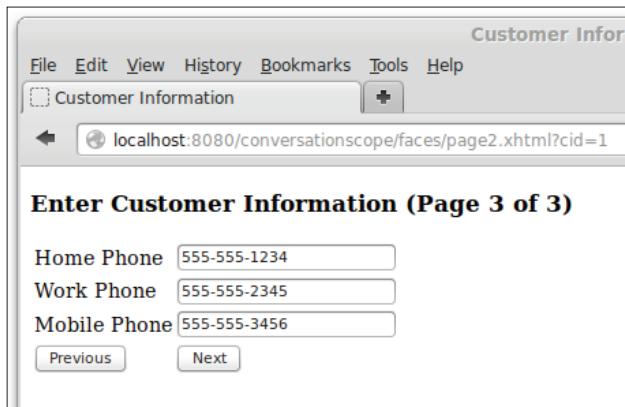


Рис. 5.5. Последняя страница диалога

В этой точке, где больше не нужно хранить в памяти информацию о клиентах, следует вызвать метод `end()` внедренного компонента диалога. Именно это мы и делаем, перед тем как отобразить страницу подтверждения:

```
public String navigateToConfirmationPage() {
    System.out.println(customer);
    conversation.end();
    return "confirmation";
}
```

После того как запрос на получение страницы подтверждения завершится, компонент в контексте диалога уничтожается вызовом метода `end()` внедренного экземпляра класса `Conversation`.

Резюме

В этой главе было представлено введение в механизм контекстов и внедрения зависимостей (Contexts and Dependency Injection, CDI). Мы рассмотрели, как JSF-страницы могут получить доступ к именованным компонентам CDI, когда для них это обращение ничем не отличается от обращения к управляемому компоненту JSF. Было показано также, как CDI облегчает внедрение зависимостей с помощью аннотации `@Inject`. Также мы объяснили, как использовать квалифициаторы для определения конкретной реализации внедряемой зависимости. Наконец, мы обсудили все контексты компонентов CDI. Они включают эквиваленты всех контекстов JSF, плюс еще два, отсутствующие в JSF, а именно контекст диалога и зависимый контекст.

В следующей главе рассказывается об обработке данных в форме записи объектов JavaScript (JavaScript Object Notation, JSON) с использованием новейшего JSON-P API.



ГЛАВА 6.

Обработка JSON с помощью JSON-P API

Форма записи объектов JavaScript (JavaScript Object Notation, JSON) – это доступный для восприятия человеком формат записи данных. Как следует из названия формата JSON, он родом из JavaScript. В Java EE 7 был введен новый JSON-P API – прикладной программный интерфейс Java для обработки JSON (Java API for JSON Processing) – в виде запроса на спецификацию Java Specification Request (JSR) 353.

Традиционно для обмена данными между разнородными системами использовался формат XML. Но, несмотря на огромную популярность XML, в последние годы неуклонно росла популярность JSON, как более простого формата обмена данными. Для Java было создано несколько библиотек, реализующих парсинг (синтаксический анализ) и создание данных в этом формате. И, наконец, данная функциональность была стандартизована в Java EE в виде Java API for JSON Processing (прикладной программный интерфейс Java для обработки JSON).

JSON-P включает два API для обработки JSON – Model API и Streaming API, и оба они рассматриваются в данной главе.

В этой главе охватываются следующие темы:

- ❖ JSON-P Model API:
 - создание данных в формате JSON с использованием Model API;
 - парсинг данных в формате JSON с использованием Model API;
- ❖ JSON-P Streaming API:
 - создание данных в формате JSON с использованием Streaming API;
 - парсинг данных в формате JSON с использованием Streaming API.

JSON-P Model API

JSON-P Model API (прикладной программный интерфейс обработки данных в формате JSON на основе объектной модели документа) позволяет создавать объекты JSON, хранящиеся в памяти и обеспечивающие возможность навигации по ним. Этот API более гибкий, чем Streaming API, обсуждаемый в разделе «JSON-P Streaming API», ниже, но он имеет более низкую производительность и более высокие требования к потреблению памяти, что может стать проблемой при необходимости обрабатывать большие объемы данных.

Создание данных в формате JSON с использованием Model API

Основой JSON-P Model API является класс `JsonObjectBuilder`. Он имеет несколько перегруженных методов `add()` для добавления свойств и их значений в данные JSON.

Ниже приводится фрагмент, который иллюстрирует, как генерировать данные в формате JSON с использованием Model API:

```
package net.ensode.glassfishbook.jsonpobject;

// другие инструкции импортирования опущены

import javax.inject.Named;
import javax.json.Json;
import javax.json JsonObject;
import javax.json.JsonReader;
import javax.json.JsonWriter;

@Named
@SessionScoped
public class JsonpBean implements Serializable{

    private String jsonStr;

    @Inject
    private Customer customer;

    public String buildJson() {
        JsonObjectBuilder jsonObjectBuilder =
            Json.createObjectBuilder();

        JsonObject jsonObject = jsonObjectBuilder.
            add("firstName", "Scott").
            add("lastName", "Gosling").
            add("email", "sgosling@example.com").
```

```
        build();

        StringWriter stringWriter = new StringWriter();

        try (JsonWriter jsonWriter = Json.createWriter(stringWriter))
        {
            jsonWriter.writeObject(jsonObject);
        }

        setJsonStr(stringWriter.toString());

        return "display_json";
    }

    // методы чтения и записи опущены
}
```



В примере используется именованный компонент CDI из более крупного JSF-приложения; здесь не будут показаны остальные части приложения, поскольку они не имеют отношения к обсуждаемой теме. Полные исходные коды приложения можно найти в загружаемом комплекте примеров для книги.

Как видите, экземпляр `JsonObject` создается в примере вызовом метода `add()` объекта `JsonObjectBuilder`. В данном случае метод `add()` добавляет в создаваемый объект `JsonObject` строковые значения. Первый параметр `add()` определяет имя свойства в будущем объекте JSON, второй – значение этого свойства. Возвращаемым значением метода `add()` является другой экземпляр `JsonObjectBuilder`. Благодаря этому имеется возможность составлять цепочки из вызовов метода `add()`, как показано в примере.

После добавления всех необходимых свойств необходимо вызвать метод `build()` экземпляра `JsonObjectBuilder`, чтобы получить экземпляр класса, реализующего интерфейс `JsonObject`.

Часто бывает необходимо получить строковое представление объекта JSON, созданного таким способом, для его обработки в другом процессе. Для этого следует создать экземпляр класса, реализующего интерфейс `JsonWriter`, – вызвать статический метод `createWriter()` класса `Json` и передать ему экземпляр `StringWriter`. После создания экземпляра `JsonWriter` нужно вызвать его метод `writeObject()` и передать ему экземпляр `JsonObject`.

В результате экземпляр `StringWriter` создаст строковое представление объекта JSON, которое можно получить вызовом метода `toString()`.

В данном конкретном примере будет сгенерирована строка в формате JSON, которая выглядит, как показано ниже:

```
{"firstName": "Scott", "lastName": "Gosling", "email":  
"sgosling@example.com"}
```

Несмотря на то, что в примере в объект JSON добавлялись только объекты типа `String`, существует поддержка и других типов данных. Класс `JsonObjectBuilder` имеет несколько перегруженных методов `add()`, позволяющих добавлять в объекты JSON значения разных типов.

В табл. 6.1 перечислены все доступные версии метода `add()`.

Таблица 6.1. Доступные версии метода `add()`

Методы <code>JsonObjectBuilder</code>	Описание
<code>add(String name, BigDecimal value)</code>	Добавляет в объект JSON значение типа <code>BigDecimal</code> .
<code>add(String name, BigInteger value)</code>	Добавляет в объект JSON значение типа <code>BigInteger</code> .
<code>add(String name, JSONArrayBuilder value)</code>	Добавляет в объект JSON массив. Реализация <code>JSONArrayBuilder</code> позволяет создавать массивы.
<code>add(String name, JsonObjectBuilder value)</code>	Добавляет в объект JSON другой объект JSON (значениями свойств объектов JSON могут быть другие объекты JSON). Добавляемая реализация <code>JsonObject</code> создается из параметра <code>JsonObjectBuilder</code> .
<code>add(String name, JsonValue value)</code>	Добавляет в объект JSON другой объект JSON (значениями свойств объектов JSON могут быть другие объекты JSON).
<code>add(String name, String value)</code>	Добавляет в объект JSON значение типа <code>String</code> .
<code>add(String name, boolean value)</code>	Добавляет в объект JSON значение типа <code>boolean</code> .
<code>add(String name, double value)</code>	Добавляет в объект JSON значение типа <code>double</code> .
<code>add(String name, int value)</code>	Добавляет в объект JSON значение типа <code>int</code> .
<code>add(String name, long value)</code>	Добавляет в объект JSON значение типа <code>long</code> .

Все версии метода `add()` принимают в первом параметре имя свойства для добавления в объект JSON, а во втором – значение этого свойства.

Парсинг данных в формате JSON с использованием Model API

В предыдущем разделе было показано, как генерировать данные в формате JSON в программах на языке Java, используя для этого Model API. В данном разделе мы увидим, как прочитать и проанализировать имеющиеся данные JSON. Следующий фрагмент иллюстрирует, как это сделать:

```
package net.ensode.glassfishbook.jsonobject;

// другие инструкции импортирования опущены

import javax.json.Json;
import javax.json.JsonObject;
import javax.json.JsonReader;
import javax.json.JsonWriter;

@Named
@SessionScoped
public class JsonpBean implements Serializable{

    private String jsonStr;

    @Inject
    private Customer customer;

    public String parseJson() {
        JsonObject jsonObject;

        try (JsonReader jsonReader = Json.createReader(
                new StringReader(jsonStr)))
        {
            jsonObject = jsonReader.readObject();
        }

        customer.setFirstName(
            jsonObject.getString("firstName"));
        customer.setLastName(
            jsonObject.getString("lastName"));
        customer.setEmail(jsonObject.getString("email"));

        return "display_parsed_json";
    }
}
```

```

    }

    // методы чтения и записи опущены
}

```

Чтобы выполнить парсинг строки в формате JSON, нужно создать объект `StringReader`, передав объект `String` с данными JSON. Затем, полученный объект `StringReader` следует передать статическому методу `createReader()` класса `Json`. Этот метод вернет экземпляр `JsonReader`, с помощью которого можно получить экземпляр `JsonObject`, вызвав метод `readObject()`.

В данном примере для получения значений всех свойств объектов JSON используется метод `getString()`; в первом и единственном аргументе этому методу передается имя свойства. Результатом, возвращаемым этим методом, является значение свойства.

Помимо метода `getString()` класс `JsonObject` имеет еще ряд похожих методов для чтения значений других типов. Все эти методы перечислены в табл. 6.2.

Таблица 6.2. Методы для извлечения значений свойств объекта JSON

Методы <code>JsonObject</code>	Описание
<code>get(Object key)</code>	Возвращает экземпляр класса, реализующего интерфейс <code>JsonValue</code> .
<code>getBoolean(String name)</code>	Возвращает значение типа <code>boolean</code> , соответствующее свойству с именем <code>name</code> .
<code>getInt(String name)</code>	Возвращает значение типа <code>int</code> , соответствующее свойству с именем <code>name</code> .
<code>getJSONArray(String name)</code>	Возвращает экземпляр класса, реализующего интерфейс <code>JSONArray</code> , соответствующий свойству с именем <code>name</code> .
<code>getJSONObject(String name)</code>	Возвращает экземпляр класса, реализующего интерфейс <code>JsonObject</code> , соответствующий свойству с именем <code>name</code> .
<code>getStringExtra(String name)</code>	Возвращает экземпляр класса, реализующего интерфейс <code>JsonString</code> , соответствующий свойству с именем <code>name</code> .
<code>getString(String Name)</code>	Возвращает значение типа <code>String</code> , соответствующее свойству с именем <code>name</code> .

Все эти методы принимают параметр типа `String` с именем свойства объекта JSON, значение которого требуется вернуть.

JSON-P Streaming API

JSON-P Streaming API (прикладной программный интерфейс обработки данных в формате JSON на основе потоковой модели) позволяет читать данные в формате JSON, которые поступают в виде потока (то есть, в виде экземпляра класса, наследующего `java.io.OutputStream` или `java.io.Writer`). Этот API имеет более высокую производительность в сравнении с Model API и более низкие требования к потреблению памяти. Однако, за это приходится платить более строгими ограничениями: данные JSON могут читаться только последовательно и отсутствует прямой доступ к свойствам JSON.

Создание данных в формате JSON с использованием Streaming API

JSON Streaming API включает класс `JsonGenerator`, с помощью которого можно создавать данные в формате JSON и записывать их в поток. Этот класс имеет несколько перегруженных методов `write()`, добавляющих свойства и их значения в данные JSON.

Следующий фрагмент иллюстрирует создание данных JSON с использованием Streaming API:

```
package net.ensode.glassfishbook.jsonpstreaming;

// остальные инструкции импортирования опущены

import javax.json.Json;
import javax.json.stream.JsonGenerator;
import javax.json.stream.JsonParser;
import javax.json.stream.JsonParser.Event;

@Named
@SessionScoped
public class JsonpBean implements Serializable {

    private String jsonStr;

    @Inject
    private Customer customer;

    public String buildJson() {
        StringWriter stringWriter = new StringWriter();
        try (JsonGenerator generator = Json.createGenerator(stringWriter)) {
            generator.writeStartObject();
            generator.write("customer", customer);
            generator.writeEndObject();
        }
        return stringWriter.toString();
    }
}
```

```

try (JsonGenerator jsonGenerator =
      Json.createGenerator(stringWriter))
{
    jsonGenerator.writeStartObject().
        write("firstName", "Larry").
        write("lastName", "Gates").
        write("email", "lgates@example.com").
        writeEnd();
}

setJsonStr(stringWriter.toString());

return "display_json";
}

// методы чтения и записи опущены
}

```

Здесь, вызовом статического метода `createGenerator()` класса `Json`, создается экземпляр `JsonGenerator`. JSON-P Streaming API имеет две перегруженные версии метода `createGenerator()`; одна принимает экземпляр класса, наследующего `java.io.Writer` (такой как `StringWriter`, используемый в данном примере), а другой принимает экземпляра класса, наследующего `java.io.OutputStream`.

Прежде чем начинать добавлять свойства в поток JSON, нужно вызывать метод `writeStartObject()` объекта `JsonGenerator`. Этот метод выводит начальный объект для формата JSON (в строках JSON – открывающая фигурная скобка `{}`) и возвращает другой экземпляр `JsonGenerator`, что позволяет составлять цепочки из вызовов `write()`, добавляющих свойства в поток JSON.

Метод `write()` класса `JsonGenerator` добавляет свойства в поток JSON. В первом параметре он принимает значение типа `String`, определяющее имя добавляемого свойства, а во втором – значение этого свойства.

Несмотря на то, что в примере в поток JSON добавлялись только объекты типа `String`, существует поддержка и других типов данных. Класс `JsonGenerator` имеет несколько перегруженных методов `write()`, позволяющих добавлять в объекты JSON значения разных типов. Все они перечислены в табл. 6.3.

Таблица 6.3. Доступные версии метода `write()`

Методы <code>JsonGenerator</code>	Описание
<code>write(String name, BigDecimal value)</code>	Выводит в поток JSON значение типа <code>BigDecimal</code> .

Методы JsonGenerator	Описание
write(String name, BigInteger value)	Выводит в поток JSON значение типа BigInteger.
write(String name, JsonValue value)	Выводит в поток JSON объект JSON (значениями свойств объектов JSON могут быть другие объекты JSON).
write(String name, String value)	Выводит в поток JSON значение типа String.
write(String name, boolean value)	Выводит в поток JSON значение типа boolean.
write(String name, double value)	Выводит в поток JSON значение типа double.
write(String name, int value)	Выводит в поток JSON значение типа int.
write(String name, long value)	Выводит в поток JSON значение типа long.

Все версии метода `write()` принимают в первом параметре имя свойства для добавления в поток JSON, а во втором – значение этого свойства.

После добавления всех свойств в поток JSON нужно вызвать метод `writeEnd()` экземпляра `JsonGenerator`; этот метод добавляет конечный объект для формата JSON (в строках JSON – закрывающая фигурная скобка `{}`).

После этого поток JSON будет заполнен фактическими данными. Что делать с ним дальше – зависит от целей приложения. В данном примере просто вызывается метод `toString()` класса `StringReader`, чтобы получить строковое представление данных в формате JSON.

Парсинг данных в формате JSON с использованием *Streaming API*

В предыдущем разделе было показано, как генерировать данные в формате JSON в программах на языке Java, используя для этого *Streaming API*. В данном разделе мы увидим, как прочитать и проанализировать данные JSON из потока. Следующий фрагмент иллюстрирует, как это сделать:

```
package net.ensode.glassfishbook.jsonpstreaming;

// другие инструкции импортирования опущены

import javax.json.Json;
```

```
import javax.json.stream.JsonGenerator;
import javax.json.stream.JsonParser;
import javax.json.stream.JsonParser.Event;

@Named
@SessionScoped
public class JsonpBean implements Serializable {

    private String jsonStr;

    @Inject
    private Customer customer;

    public String parseJson() {
        StringReader stringReader = new StringReader(jsonStr);
        JsonParser jsonParser = Json.createParser(stringReader);

        Map<String, String> keyValueMap = new HashMap<>();
        String key = null;
        String value = null;

        while (jsonParser.hasNext()) {
            JsonParser.Event event = jsonParser.next();

            if (event.equals(Event.KEY_NAME)) {
                key = jsonParser.getString();
            } else if (event.equals(Event.VALUE_STRING)) {
                value = jsonParser.getString();
            }

            keyValueMap.put(key, value);
        }

        customer.setFirstName(keyValueMap.get("firstName"));
        customer.setLastName(keyValueMap.get("lastName"));
        customer.setEmail(keyValueMap.get("email"));

        return "display_parsed_json";
    }

    // методы чтения и записи опущены
}
```

Первое, что нужно сделать, чтобы прочитать данные JSON с использованием Streaming API – создать экземпляр `JsonParser` вызовом статического метода `createJsonParser()` класса `Json`. Существует две перегруженные версии метода `createJsonParser()`; одна принимает экземпляр класса, наследующего `java.io.InputStream`, а

другая принимает экземпляр класса, наследующего `java.io.Reader`. В данном примере используется второй из них – экземпляр `java.io.StringReader`.

Следующий шаг – выполнить цикл по данным JSON, чтобы извлечь из них свойства. Цикл реализован на основе метода `hasNext()` класса `JsonParser`, который возвращает `true`, если в потоке имеются данные для чтения, и `false` – в противном случае.

Далее нужно прочитать очередную порцию данных из потока. Метод `JsonParser.next()` возвращает экземпляр `JsonParser.Event`, хранящий тип только что прочитанных данных. В данном примере проверяются только имена свойств (то есть, `"firstName"`, `"lastName"` и `"email"`) и соответствующие им строковые значения. Определить тип только что прочитанных данных можно, сравнив объект `Event`, полученный вызовом `JsonParser.next()`, с разными константами в перечислении `Event`, объявленном в классе `JsonParser`.

В табл. 6.4 перечислены все доступные константы, которые может вернуть `JsonParser.next()`.

Таблица 6.4. Константы в перечислении Event

Константы <code>JsonParser.Event</code>	Описание
<code>Event.START_OBJECT</code>	Соответствует начальному объекту в данных JSON.
<code>Event.END_OBJECT</code>	Соответствует конечному объекту в данных JSON.
<code>Event.START_ARRAY</code>	Соответствует началу массива в данных JSON.
<code>Event.END_ARRAY</code>	Соответствует концу массива в данных JSON.
<code>Event.KEY_NAME</code>	Указывает, что прочитано имя свойства JSON. Получить фактическое имя можно вызовом <code>getString()</code> объекта <code>JsonParser</code> .
<code>Event.VALUE_TRUE</code>	Указывает, что прочитано логическое значение <code>true</code> .
<code>Event.VALUE_FALSE</code>	Указывает, что прочитано логическое значение <code>false</code> .
<code>Event.VALUE_NULL</code>	Указывает, что прочитано значение <code>null</code> .
<code>Event.VALUE_NUMBER</code>	Указывает, что прочитано числовое значение.
<code>Event.VALUE_STRING</code>	Указывает, что прочитано строковое значение.

Как показано в примере, значения типа `String` извлекаются вызовом метода `getString()` экземпляра `JsonParser`. Числовые значения могут извлекаться несколькими способами; в табл. 6.5 перечислены методы класса `JsonParser`, возвращающие числовые значения:

Таблица 6.5. Методы класса JsonParser, возвращающие числовые значения

Методы JsonParser	Описание
getInt()	Возвращает числовое значение как int.
getLong()	Возвращает числовое значение как long.
getBigDecimal()	Возвращает числовое значение как экземпляр java.math.BigDecimal.

В классе JsonParser имеется также удобный метод `isIntegralNumber()`, возвращающий `true`, если числовое значение можно безопасно привести к типу `int` или `long`.

Что дальше делать с полученными значениями, зависит от логики приложения. В данном примере они просто помещаются в объект Map, который затем используется для заполнения Java-объекта.

Резюме

В этой главе мы познакомились с API для обработки JSON (Java API for JSON Processing, JSON-P). Мы рассмотрели два основных JSON-P API: Model API и Streaming API.

На примерах было показано, как генерировать данные в формате JSON с использованием JSON-P Model API, в частности с применением класса `JsonBuilder`, и как анализировать их с использованием JSON-P Model API, с применением класса `JsonReader`.

Также было объяснено, как генерировать данные в формате JSON с использованием JSON-P Streaming API, с применением класса `JsonGenerator`.

В заключение было показано, как анализировать данные в формате JSON с использованием JSON-P Streaming API, в частности с применением класса `JsonParser`.

В следующей главе будет рассказываться о еще одном новом API – Java API для веб-сокетов (Java API for WebSocket).



ГЛАВА 7.

Веб-сокеты

Традиционно, веб-приложения конструируются на основе модели запрос/ответ, реализованной в виде протокола HTTP. В этой модели запрос всегда инициируется клиентом, откликаясь на который сервер возвращает свой ответ.

До недавних пор сервер не имел возможности послать данные клиенту по собственной инициативе (без запроса со стороны клиента). С появлением протокола веб-сокетов (WebSocket) появилась возможность полноценного, двустороннего обмена между клиентом (браузером) и сервером.

В Java EE 7 был введен Java API для веб-сокетов (Java API for WebSocket), позволяющий создавать конечные точки веб-сокетов на Java. Java API для веб-сокетов – это совершенно новая технология в стандарте Java EE.



Сокет (socket) – это канал двунаправленного обмена данными, который продолжает оставаться открытым после обработки единственного запроса. Для HTML5-совместимых браузеров это означает возможность непрекращающихся взаимодействий с веб-сервером без необходимости загружать новую страницу (то есть, веб-сокеты чем-то подобны технологии AJAX).

В этой главе рассматриваются следующие темы:

- ❖ создание серверных конечных точек веб-сокетов;
- ❖ создание клиентов веб-сокетов на JavaScript;
- ❖ создание клиентов веб-сокетов на Java.

Создание серверных конечных точек веб-сокетов

Серверная конечная точка веб-сокета (WebSocket) – это Java-класс, развернутый в серверном приложении для обслуживания запросов к веб-сокетам.

Существует два способа реализации серверных конечных точек веб-сокетов с использованием Java API для веб-сокетов (Java API for WebSocket): программный, когда создается класс, наследующий `javax.websocket.Endpoint`, или путем декорирования **простых объектов Java (Plain Old Java Objects, POJO)** специализированными аннотациями WebSocket API. Эти два подхода очень похожи, поэтому мы подробно обсудим только способ на основе аннотаций и коротко коснемся другого подхода, основанного на явном определении серверной конечной точки веб-сокета.

В этой главе будет создано простое приложение веб-чата, использующее преимущества Java API для веб-сокетов.

Создание серверной конечной точки веб-сокета с применением аннотаций

Следующее определение класса Java наглядно показывает, как создать серверную конечную точку веб-сокета путем аннотирования класса:

```
package net.ensode.glassfishbook.websocketchat.serverendpoint;

import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.websocket.OnClose;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/websocketchat")
public class WebSocketChatEndpoint {

    private static final Logger LOG =
        Logger.getLogger(WebSocketChatEndpoint.class.getName());

    @OnOpen
    public void connectionOpened() {
        LOG.log(Level.INFO, "connection opened");
    }

    @OnMessage
    public synchronized void processMessage(Session session,
        String message) {
        LOG.log(Level.INFO, "received message: {}", message);

        try {
            ...
        }
    }
}
```

```
        for (Session sess : session.getOpenSessions()) {
            if (sess.isOpen()) {
                sess.getBasicRemote().sendText(message);
            }
        }
    } catch (IOException ioe) {
        LOG.log(Level.SEVERE, ioe.getMessage());
    }
}

@OnClose
public void connectionClosed() {
    LOG.log(Level.INFO, "connection closed");
}
```

Аннотация `@ServerEndpoint` на уровне класса указывает, что класс является серверной конечной точкой веб-сокета. Значение в круглых скобках ("`/websocketchat`"), следующее за аннотацией – это **уникальный идентификатор ресурса (Uniform Resource Identifier, URI)**. Этот URI будет использоваться клиентами для взаимодействий с конечной точкой.

Аннотация `@OnOpen` отмечает метод, который должен вызываться всякий раз, когда открывается соединение с новым клиентом. В данном примере метод просто выводит некоторую информацию в журнал сервера, однако он может содержать любой допустимый серверный код на Java.

Любые методы, отмеченные аннотацией `@OnMessage`, будут вызываться при приеме серверной конечной точкой сообщений от клиентов. Поскольку в этой главе создается чат, наш код просто рассыпает сообщение всем подключенным клиентам.

Метод `processMessage()`, отмеченный аннотацией `@OnMessage`, принимает два параметра: экземпляр класса, реализующего интерфейс `javax.websocket.Session`, и строку с принятым сообщением. В данном примере метод `processMessage()` просто рассыпает принятое сообщение всем клиентам.

Метод `getOpenSessions()` интерфейса `Session` возвращает множество объектов, представляющих все открытые сеансы. Мы выполняем обход всех членов этого множества и рассыпаем сообщение всем клиентам, вызывая сначала метод `getBasicRemote()` этих объектов, а затем метод `sendText()` реализации `RemoteEndpoint.Basic`, полученной вызовом предыдущего метода.

Метод `getOpenSessions()` интерфейса `Session` возвращает все сеансы, которые были открыты на момент его вызова. Соответственно

есть вероятность, что какие-то сеансы окажутся закрытыми уже после вызова этого метода; поэтому рекомендуется вызывать метод `isOpen()` реализации `Session` перед попыткой вернуть данные клиенту. Попытка обратиться к закрытому сеансу приведет к исключению.

Наконец, на случай, если потребуется обрабатывать событие закрытия сеанса клиентом, необходимо определить метод с аннотацией `@OnClose`. В данном примере этот метод просто выводит сообщение в журнал сервера.

Существует еще одна аннотация, не использованная в этом примере – `@OnError`. Она используется для декорирования метода, который должен вызываться в случае ошибки отправки или приема данных.

Как видите, реализовать серверную конечную точку веб-сокета очень просто. Нужно добавить всего несколько аннотаций и серверное приложение будет вызывать аннотированные методы по мере необходимости.

Чтобы реализовать серверную конечную точку программно, нужно определить класс Java, наследующий `javax.websocket.Endpoint`. Этот родительский класс имеет методы `onOpen()`, `onClose()` и `onError()`, которые автоматически вызываются в нужные моменты времени, в течение жизненного цикла конечной точки. Однако в этом классе нет метода, аналогичного методу с аннотацией `@OnMessage` для обработки клиентских сообщений. Поэтому придется вызвать метод `addMessageHandler()` в сеансе и передать ему экземпляр класса, реализующего интерфейс `javax.websocket.MessageHandler` (или один из его подинтерфейсов).



Вообще, подход на основе аннотаций выглядит проще и понятнее. Поэтому мы рекомендуем по мере возможности использовать этот подход.

Создание клиентов веб-сокетов

Подавляющее большинство клиентов веб-сокетов реализовано в виде веб-страниц HTML5, использующих JavaScript WebSocket API. Как таковые, они могут выполняться только в HTML5-совместимых веб-браузерах (большинство современных браузеров поддерживает стандарт HTML5).

Java API для веб-сокетов включает клиентский API, позволяющий создавать клиентов веб-сокетов в виде самостоятельных приложений

на Java. Но об этом мы расскажем позднее, в разделе «Создание клиентов веб-сокетов на Java».

Создание клиентов веб-сокетов на JavaScript

В этом разделе рассказывается, как создать клиента на JavaScript для взаимодействия с конечной точкой веб-сокета, реализованной в предыдущем разделе.

Клиентская страница для нашего примера использования веб-сокетов реализована как JSF-страница с HTML5-подобной разметкой (описывается в главе 2, «JavaServer Faces»).

Страница включает текстовую область, где выводятся сообщения других клиентов (в конце концов это – чат), и поле для ввода сообщений, посылаемых другим пользователям, как показано на рис. 7.1.

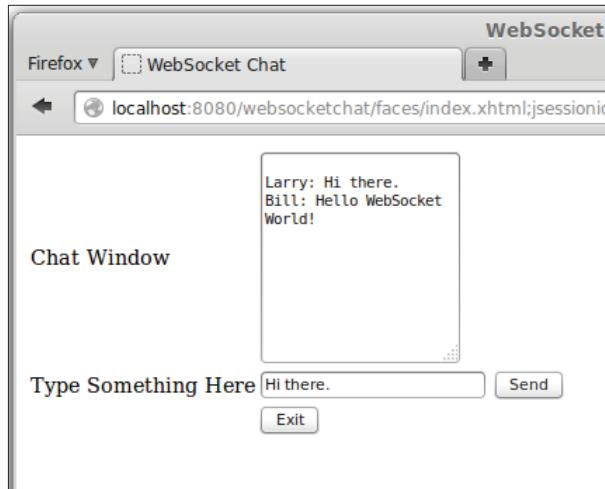


Рис. 7.1. Внешний вид клиентской страницы чата

Ниже приводится разметка страницы:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:jsf="http://
xmlns:jcp.org/jsf">
  <head>
    <title>WebSocket Chat</title>
    <meta name="viewport" content="width=device-width"/>
    <script type="text/javascript">
```

```
var websocket;
function init() {
    websocket = new WebSocket(
        'ws://localhost:8080/websocketchat/websocketchat');
    websocket.onopen = function(event) {
        websocketOpen(event)
    };
    websocket.onmessage = function(event) {
        websocketMessage(event)
    };
    websocket.onerror = function(event) {
        websocketError(event)
    };
}

function websocketOpen(event) {
    console.log("webSocketOpen invoked");
}

function websocketMessage(event) {
    console.log("websocketMessage invoked");
    document.getElementById('chatwindow').value += '\r' +
        event.data;
}

function websocketError(event) {
    console.log("websocketError invoked");
}

function sendMessage() {
    var userName =
        document.getElementById('userName').value;
    var msg =
        document.getElementById('chatinput').value;
    websocket.send(userName + " : " + msg);
}

function closeConnection(){
    websocket.close();
}

window.addEventListener("load", init);
</script>
</head>
<body>
<form jsf:prependId="false">
    <input type="hidden" id="userName"
           value="#{user.userName}"/>
    <table border="0">
        <tbody>
```

```
<tr>
    <td>
        <label for="chatwindow">
            Chat Window
        </label>
    </td>
    <td>
        <textArea id="chatwindow" rows="10"/>
    </td>
</tr>
<tr>
    <td>
        <label for="chatinput">
            Type Something Here
        </label>
    </td>
    <td>
        <input type="text" id="chatinput"/>
        <input id="sendBtn" type="button" value="Send"
               onclick="sendMessage()"/>
    </td>
</tr>
<tr>
    <td></td>
    <td>
        <input type="button" id="exitBtn" value="Exit"
               onclick="closeConnection()"/>
    </td>
</tr>
</tbody>
</table>
</form>
</body>
</html>
```

Последняя строка в коде на JavaScript (`window.addEventListener("load", init);`) устанавливает JavaScript-функцию `init()` как обработчик события окончания загрузки страницы.

Внутри метода `init()` инициализируется новый JavaScript-объект `websocket` с передачей URI серверной конечной точки в качестве параметра. Таким способом в JavaScript определяется местоположение серверной конечной точки.

Объект `websocket` позволяет определить множество обработчиков различных событий, таких как открытие соединения, прием сообщения и обработка ошибок. Чтобы получить возможность обрабатывать эти события, следует определить свои функции на JavaScript, что и делается в методе `init()` сразу после вызова конструктора объекта

websocket. В данном примере функции, присваиваемые свойствам объекта `websocket`, просто делегируют всю работу обычным функциям на JavaScript.

Функция `websocketOpen()` вызывается каждый раз, когда открывается соединение с веб-сокетом. В данном примере она просто выводит сообщение в JavaScript-консоль браузера.

Функция `webSocketMessage()` вызывается, когда браузер принимает сообщение от конечной точки веб-сокета. В данном примере функция обновляет содержимое текстовой области с атрибутом `id="chatwindow"`, добавляя в него полученное сообщение.

Функция `websocketError()` вызывается, когда возникает какая-нибудь ошибка, связанная с веб-сокетом. В данном примере она просто выводит сообщение в JavaScript-консоль браузера.

Функция `sendMessage()` посыпает сообщение серверной конечной точке веб-сокета, содержащее имя пользователя и текст, введенный в поле с атрибутом `id="chatinput"`. Она вызывается, когда пользователь щелкает на кнопке с атрибутом `id="sendBtn"`.

Функция `closeConnection()` закрывает соединение с конечной точкой. Она вызывается, когда пользователь щелкает на кнопке с атрибутом `id="exitBtn"`.

Как видите, реализация клиентской части на JavaScript, взаимодействующей с веб-сокетом, не содержит ничего сложного.

Создание клиентов веб-сокетов на Java

Несмотря на то, что подавляющее большинство клиентов веб-сокетов реализуется на JavaScript, в состав Java API для веб-сокетов включен клиентский API, который можно использовать для разработки клиентов веб-сокетов на Java.

В этом разделе будет создан простой клиент веб-сокета с использованием клиентского API из Java API для веб-сокетов. На рис. 7.2 показано, как выглядит законченная версия клиента.

Однако здесь мы не будем рассматривать код графического интерфейса (создан с использованием фреймворка Swing), потому что он не имеет отношения к обсуждаемой теме. Полный код примера, включая реализацию графического интерфейса, можно найти в загружающем пакете с примерами для книги.

Так же как серверные конечные точки веб-сокетов, клиенты на Java можно создавать программно или с использованием аннотаций. И снова мы подробно рассмотрим только подход на основе аннотаций: программная реализация клиентов выполняется почти так же,

как программная реализация серверных конечных точек, то есть, программные клиенты должны определять класс, наследующий `javax.websocket.Endpoint` и переопределять соответствующие методы.



Рис. 7.2. Внешний вид клиентского приложения на Java

А теперь обратимся к исходному коду клиента веб-сокета на Java:

```
package net.ensode.websocketjavaclient;

import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;
import javax.websocket.ClientEndpoint;
import javax.websocket.CloseReason;
import javax.websocket.ContainerProvider;
import javax.websocket.DeploymentException;
import javax.websocket.OnClose;
import javax.websocket.OnError;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.WebSocketContainer;

@ClientEndpoint
public class WebSocketClient {

    private String userName;
    private Session session;
    private final WebSocketJavaClientFrame webSocketJavaClientFrame;

    public WebSocketClient(WebSocketJavaClientFrame
            webSocketJavaClientFrame)
    {
        this.webSocketJavaClientFrame = webSocketJavaClientFrame;
        try {
            WebSocketContainer webSocketContainer =

```

```
    ContainerProvider.getWebSocketContainer();
    webSocketContainer.connectToServer(this, new URI(
        "ws://localhost:8080/websocketchat/websocketchat"));
}
catch (DeploymentException | IOException | URISyntaxException ex)
{
    ex.printStackTrace();
}
}

@OnOpen
public void onOpen(Session session) {
    System.out.println("onOpen() invoked");
    this.session = session;
}

@OnClose
public void onClose(CloseReason closeReason) {
    System.out.println("Connection closed, reason: "+
        closeReason.getReasonPhrase());
}

@OnError
public void onError(Throwable throwable) {
    System.out.println("onError() invoked");
    throwable.printStackTrace();
}

@OnMessage
public void onMessage(String message, Session session) {
    System.out.println("onMessage() invoked");
    webSocketJavaClientFrame.getChatWindowTextArea()
        .setText(webSocketJavaClientFrame.getChatWindowTextArea()
            .getText() + "\n" + message);
}

public void sendMessage(String message) {
    try {
        System.out.println("sendMessage() invoked, message = " +
            message);
        session.getBasicRemote().sendText(userName + ": " +
            message);
    }
    catch (IOException ex) {
        ex.printStackTrace();
    }
}

public String getUserName() {
    return userName;
```

```
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }
}
```

Аннотация `@ClientEndPoint` на уровне класса указывает, что класс реализует клиента веб-сокета – все клиенты веб-сокетов на Java должны отмечаться этой аннотацией.

Соединение с серверной конечной точкой здесь устанавливается в конструкторе класса. Сначала нужно вызвать `ContainerProvider.getWebSocketContainer()`, чтобы создать экземпляр `javax.websocket.WebSocketContainer`. Затем, вызовом метода `connectToServer()` экземпляра `WebSocketContainer`, устанавливается соединение с конечной точкой; этот метод принимает в первом параметре класс, отмеченный аннотацией `@ClientEndpoint` (в данном примере используется ссылка `this`, потому что код, устанавливающий соединение, определен в самом этом классе), а во втором – URI серверной конечной точки.

Когда соединение будет установлено, клиент сможет откликаться на события веб-сокетов. Внимательные читатели могли заметить, что в реализации клиента используются те же аннотации, что и в реализации серверной конечной точки.

Любой метод, отмеченный аннотацией `@OnOpen`, вызывается автоматически, когда будет открыто соединение с серверной конечной точкой. Метод должен возвращать `void` и может принимать необязательный параметр типа `javax.websocket.Session`. В данном примере он выводит некоторую информацию в консоль и инициализирует переменную класса экземпляром `Session`, полученным в параметре.

Методы с аннотацией `@OnClose` вызываются, когда закрывается сеанс работы с веб-сокетом. Аннотированный метод может принимать необязательные параметры типов `javax.websocket.Session` и `CloseReason`. В данном примере метод использует только параметр `CloseReason`, потому что этот класс имеет удобный метод `getReasonPhrase()`, возвращающий краткое описание причины завершения сеанса.

Аннотация `@OnError` используется для определения методов, которые должны вызываться в случае обнаружения ошибки. Методы с аннотацией `@OnError` должны принимать параметр типа `java.lang.Throwable` (родитель класса `java.lang.Exception`) и может принимать

еще один необязательный параметр типа `Session`. В данном примере метод просто выводит трассировку стека из параметра `Throwable` `B stderr`.

Методы с аннотацией `@OnMessage` вызываются при получении входящего сообщения. Методы `@OnMessage` могут принимать разные параметры, в зависимости от типа принимаемого сообщения и особенностей их обработки. В данном примере используется самый распространенный вариант: прием текстового сообщения. В данном конкретном случае метод имеет обязательный параметр типа `String` с содержимым сообщения и необязательный параметр `Session`.



Описание особенностей обработки сообщений других типов в методах с аннотацией `@OnMessage` ищите в документации по адресу: <http://docs.oracle.com/javaee/7/api/javax/websocket/OnMessage.html>.

В данном примере метод просто обновляет содержимое текстовой области **Chat Window** (Окно чата), добавляя в конец принятое сообщение.

Для отправки сообщения сначала вызывается метод `getBasicRemote()` экземпляра `Session`, затем метод `sendText()` экземпляра `RemoteEndpoint.Basic`, полученного вызовом предыдущего метода (если этот код покажется вам знакомым, вы правы – точно так же отправка сообщений реализована в серверной конечной точке). В данном примере эти операции выполняются в методе `sendMessage()`.

Дополнительная информация о Java API для веб-сокетов

В этой главе мы немало узнали о возможностях Java API для веб-сокетов. Дополнительную информацию можно найти в справочном руководстве по адресу: <https://tyrus.java.net/documentation/1.3.1/user-guide.html>.

Резюме

В этой главе мы охватили Java API для веб-сокетов (Java API for WebSocket) – новейший Java EE API для разработки серверных конечных точек и клиентов веб-сокетов.

Сначала мы посмотрели, как создаются серверные конечные точки с использованием Java API для веб-сокетов, сосредоточившись на приеме с применением аннотаций.

Затем была продемонстрирована реализация клиента веб-сокета на JavaScript и возможности JavaScript WebSocket API.

Наконец, было показано, как реализовать клиентское приложение на Java с применением аннотации `@clientEndpoint`.

В следующей главе мы рассмотрим **службу сообщений Java (Java Message Service, JMS)**.



ГЛАВА 8.

Служба обмена сообщениями Java

API службы обмена сообщениями Java (Java Messaging Service, JMS) реализует механизм обмена сообщениями для приложений Java EE. В Java EE 7 появилась новая версия JMS 2.0, существенно упрощающая разработку приложений, обменивающихся сообщениями.

Приложения JMS не взаимодействуют напрямую – они обмениваются сообщениями. Продюсеры отправляют сообщения в пункт назначения, а потребители получают (потребляют) сообщение из пункта назначения.

Пунктом назначения (destination) сообщения является очередь (queue) сообщений, когда используется модель обмена сообщениями «точка-точка» (point-to-point, PTP), или тема (topic) сообщения, когда используется модель обмена сообщениями «публикация/подписка» (publish/subscribe, pub/sub).

В этой главе мы рассмотрим следующие темы:

- ◊ настройка GlassFish для использования JMS;
- ◊ работа с очередями сообщений;
- ◊ работа с темами сообщений.

Настройка GlassFish для использования JMS

Прежде чем мы начнем писать код, использующий возможности JMS API, следует настроить некоторые ресурсы GlassFish. В частности, необходимо создать фабрику JMS-соединений, очереди сообщений и темы сообщений.



Java EE 7 требует от всех совместимых серверов приложений реализовать фабрику JMS-соединений по умолчанию. GlassFish, как сервер приложений, полностью совместимый с Java EE 7 (и являющийся эталонной реализацией Java EE 7), полностью соответствует этому требованию; поэтому, строго говоря, нам не нужно настраивать фабрику соединений. Однако, на практике это порой действительно бывает необходимо, поэтому в следующем разделе будет показано, как выполнить такую настройку.

Настройка фабрики JMS-соединений

Самый простой способ настроить фабрику JMS-соединений – воспользоваться веб-консолью сервера GlassFish. В главе 1, «Знакомство с сервером GlassFish», уже говорилось, что доступ к веб-консоли можно получить после запуска домена вводом следующей команды в командной строке:

```
asadmin start-domain domain1
```

Затем нужно открыть в браузере страницу с адресом URL `http://localhost:4848` и зарегистрироваться.

Чтобы добавить фабрику соединений распахните узел **Resources** (Ресурсы) в панели навигации слева, затем распахните узел **JMS Resources** (Ресурсы JMS), выберите пункт **Connection Factories** (Фабрика соединений) и щелкните на кнопке **New...** (Создать) в основной панели, как показано на рис. 8.1.



Рис. 8.1. Список фабрик соединений

В данном случае можно принять большинство значений по умолчанию; нам понадобится всего лишь ввести имя пула в поле **JNDI**

Name (Имя JNDI) и выбрать тип ресурса в поле **Resource Type** (Тип ресурса) для фабрики соединений (см. рис. 8.2).

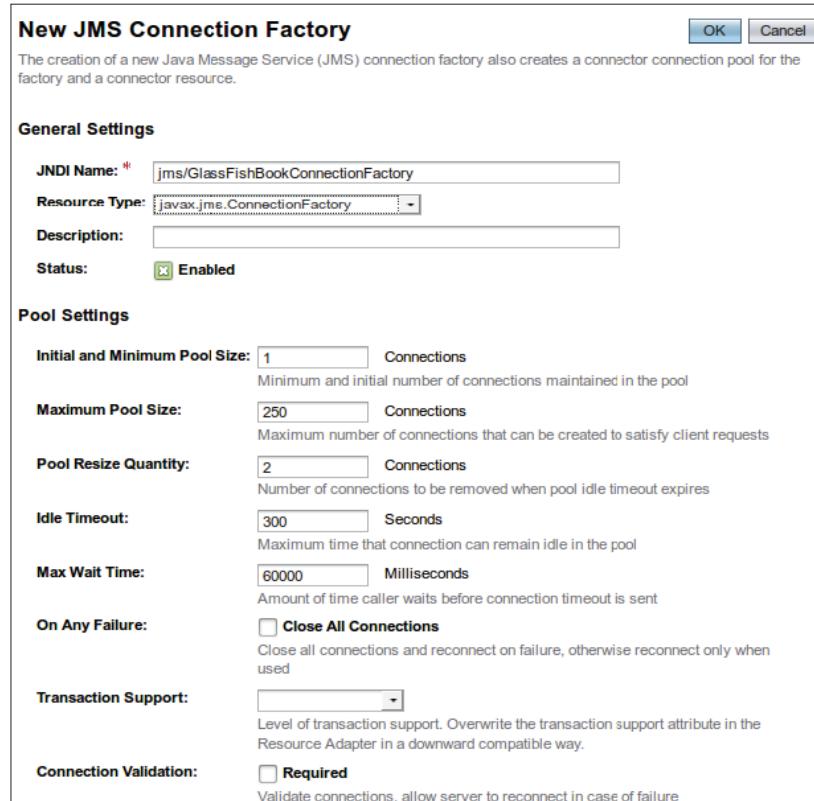


Рис. 8.2. Настройка новой фабрики соединений



При выборе имен ресурсов JMS предпочтительнее использовать имена, начинающиеся с `jms/`. Это упростит идентификацию ресурсов JMS при просмотре дерева JNDI.

В текстовом поле **JNDI Name** (Имя JNDI) введите `jms/GlassFish-BookConnectionFactory`. Примеры, рассматриваемые в этой главе, будут использовать это JNDI-имя для получения ссылки на данную фабрику соединений.

В раскрывающемся меню **Resource Type** (Тип ресурса) имеются три варианта для выбора:

- **javax.JMS.TopicConnectionFactory** – фабрика соединений, создающая темы JMS для клиентов JMS, использующих модель обмена сообщениями «публикация/подписка»;
- **javax.JMS.QueueConnectionFactory** – фабрика соединений, создающая очередь JMS для клиентов JMS, использующих модель обмена сообщениями PTP;
- **javax.JMS.ConnectionFactory** – фабрика соединений, создающая или темы JMS, или очереди JMS.

Для данного примера мы выберем `javax.jms.ConnectionFactory`. Благодаря этому мы сможем использовать одну и туже фабрику соединений для всех наших примеров – и для тех, что используют очереди сообщений, и для тех, что используют темы.

После ввода имени цула для фабрики соединений нужно выбрать тип фабрики и дополнительно ввести ее описание, после чего щелкнуть на кнопке **OK**, чтобы изменения вступили в силу.

После этого в основной области страницы веб-консоли GlassFish должна появиться вновь созданная фабрика соединений (`jms/GlassFishBookConnectionFactory`), как показано на рис. 8.3.

Connection Factories (2)						
Select	JNDI Name	Logical JNDI Name	Enabled	Resource Type	Descr	
<input type="checkbox"/>	jms/_defaultConnectionFactory	java:comp/DefaultJMSConnectionFactory	✓	javax.jms.ConnectionFactory		
<input type="checkbox"/>	jms/GlassFishBookConnectionFactory		✓	javax.jms.ConnectionFactory		

Рис. 8.3. В списке должна появиться новая фабрика соединений

Создание очереди JMS-сообщений

Чтобы добавить очередь сообщений, выполните следующие шаги (см. рис. 8.4):

1. Распахните узел **Resources** (Ресурсы) в панели навигации слева.
2. Распахните узел **Destination Resources** (Ресурсы пунктов назначения).
3. Щелкните на кнопке **New...** (Создать) в основной области страницы веб-консоли.
4. Введите имя очереди в поле **JNDI Name** (Имя JNDI).
5. Введите значение в поле **Physical Destination Name** (Имя физического пункта назначения).

6. Выберите в раскрывающемся списке **Resource Type** (Тип ресурса) пункт **javax.jms.Queue**.
7. Щелкните на кнопке **OK**.

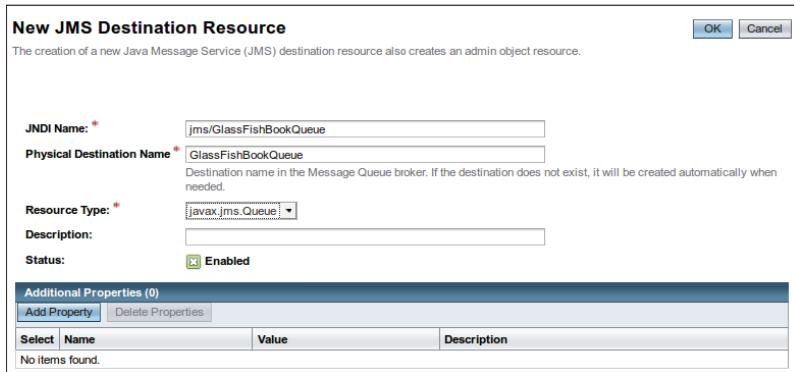


Рис. 8.4. Настройки для новой очереди

В наших примерах будет использоваться очередь сообщений с JNDI-именем `jms/GlassFishBookQueue`. В качестве типа ресурса следует выбрать `javax.jms.Queue`. Дополнительно нужно ввести имя физического пункта назначения. В данном примере физический пункт назначения будет называться `GlassFishBookQueue`.

После щелчка на кнопке **OK** в списке должна появиться новая очередь, как показано на рис. 8.5.

JMS Destination Resources			
JMS destinations serve as the repositories for messages. Click New to create a new destination resource. Click the name of a destination resource to modify its properties.			
Select	JNDI Name	Enabled	Resource Type
<input type="checkbox"/>	jms/GlassFishBookQueue	<input checked="" type="checkbox"/>	javax.jms.Queue

Рис. 8.5. В списке должна появиться новая очередь

Создание темы JMS-сообщений

Чтобы добавить тему сообщений, выполните следующие шаги (см. рис. 8.6):

1. Распахните узел **Resources** (Ресурсы) в панели навигации слева.
2. Распахните узел **Destination Resources** (Ресурсы пунктов назначения).

3. Щелкните на кнопке **New...** (Создать) в основной области страницы веб-консоли.
4. Введите имя очереди в поле **JNDI Name** (Имя JNDI).
5. Введите значение в поле **Physical Destination Name** (Имя физического пункта назначения).
6. Выберите в раскрывающемся списке **Resource Type** (Тип ресурса) пункт **javax.jms.Topic**.
7. Щелкните на кнопке **OK**.

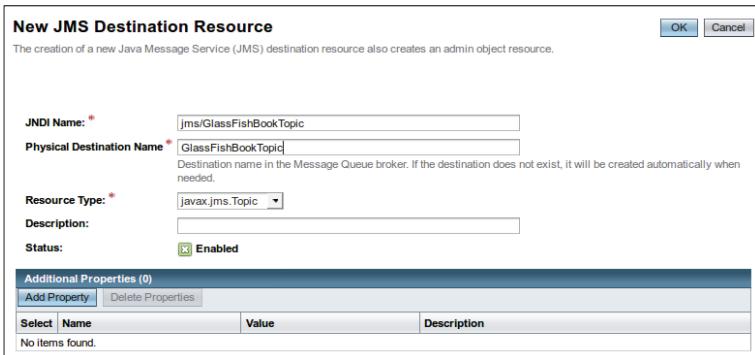


Рис. 8.6. Настройки для новой темы

В наших примерах будет использоваться тема с JNDI-именем `jms/GlassFishBookTopic`. Поскольку это тема сообщений, в поле **Resource Type** (Тип ресурса) следует выбрать `javax.jms.Topic`. Поле **Description** (Описание) заполнять не обязательно. В поле **Physical Destination Name** (Имя физического пункта назначения) требуется указать (для нашего примера) `GlassFishBookTopic`.

После щелчка на кнопке **OK** в списке должна появиться новая тема, как показано на рис. 8.7.

JMS Destination Resources

JMS destinations serve as the repositories for messages. Click New to create a new destination resource. Click the name of a destination resource to modify its properties.

Destination Resources (2)

New... Delete Enable Disable

Select	JNDI Name	Enabled	Resource Type	Description
<input type="checkbox"/>	jms/GlassFishBookQueue	<input checked="" type="checkbox"/>	javax.jms.Queue	
<input type="checkbox"/>	jms/GlassFishBookTopic	<input checked="" type="checkbox"/>	javax.jms.Topic	

Рис. 8.7. В списке должна появиться новая тема

Теперь, когда у нас есть фабрика соединений, очередь и тема сообщений, мы готовы начать писать код, использующий JMS API.

Очереди сообщений

Как уже упоминалось выше, очереди сообщений используются, когда приложение следует модели обмена сообщениями «точка-точка» (PTP). В этой модели обмена сообщениями обычно имеется один продюсер (message producer) и один потребитель (message consumer). Продюсер и потребитель не обязаны работать одновременно, чтобы взаимодействовать. Сообщение, помещенное в очередь сообщений продюсером, останется в очереди, пока потребитель не выполнит запрос к очереди.

Отправка сообщений в очередь

Следующий пример демонстрирует, как добавлять сообщения в очередь:

```
package net.ensode.glassfishbook;

import javax.annotation.Resource;
import javax.jms.ConnectionFactory;
import javax.jms.JMSContext;
import javax.jms.JMSProducer;
import javax.jms.Queue;

public class MessageSender {

    @Resource(mappedName = "jms/GlassFishBookConnectionFactory")
    private static ConnectionFactory connectionFactory;

    @Resource(mappedName = "jms/GlassFishBookQueue")
    private static Queue queue;

    public void produceMessages() {

        JMSContext jmsContext = connectionFactory.createContext();
        JMSProducer jmsProducer = jmsContext.createProducer();

        String msg1 = "Testing, 1, 2, 3. Can you hear me?";
        String msg2 = "Do you copy?";
        String msg3 = "Good bye!";

        System.out.println("Sending the following message: "
            + msg1);
        jmsProducer.send(queue, msg1);
        System.out.println("Sending the following message: "
            + msg2);
        jmsProducer.send(queue, msg2);
    }
}
```

```
System.out.println("Sending the following message: "
    + msg3);
jmsProducer.send(queue, msg3);
}

public static void main(String[] args) {
    new MessageSender().produceMessages();
}
}
```

Прежде чем углубляться в детали этого примера, отметим, что класс `MessageSender` относится к классу автономных приложений Java, так как содержит метод `main`. Поскольку это автономное приложение, оно выполняется вне сервера приложений. Тем не менее, мы видим, что в него внедряются некоторые ресурсы, в частности фабрика соединений и очередь. Причина, по которой это возможно – даже при том, что этот код выполняется вне сервера приложений, – состоит в том, что сервер GlassFish включает утилиту, называемую `appclient`.

Эта утилита позволяет «обернуть» выполняемый JAR-файл и дать ему доступ к ресурсам сервера приложений. Чтобы запустить предыдущий пример, принимая во внимание сказанное, его нужно упаковать в выполняемый JAR-файл `jmsptpproducer.jar` и ввести в командной строке следующую команду:

```
appclient -client jmsptpproducer.jar
```

После этого в консоли, среди нескольких журнальных сообщений GlassFish, появятся следующие строки:

```
Sending the following message: Testing, 1, 2, 3. Can you hear me?
```

```
Sending the following message: Do you copy?
```

```
Sending the following message: Good bye!
```

Выполняемый файл сценария `appclient` можно найти в каталоге [каталог установки GlassFish]/glassfish/bin. Предыдущий пример предполагает, что путь к этому каталогу включен в переменную окружения `PATH`. Если это не так, в командной строке следует ввести полный путь к сценарию `appclient`.

После этого небольшого отступления можем приступить к исследованию кода.

Все необходимые действия для отправки сообщения в очередь выполняет метод `produceMessages()`.

Прежде всего он получает объект `javax.jms.JMSContext`, вызывая метод `createConnection()` внедренного экземпляра `javax.jms.Connection`.

tionFactory. Обратите внимание, что значение атрибута mappedName в аннотации @Resource, декорирующей объект фабрики соединений, совпадает с JNDI-именем фабрики соединений, которое было установлено в веб-консоли сервера GlassFish. За кулисами выполняется поиск в каталоге JNDI по этому имени и возвращается найденный объект фабрики соединений.

Далее, вызовом метода `createProducer()` только что полученного экземпляра `JMSContext`, создается экземпляр `javax.jms.JMSProducer`.

После получения экземпляра `JMSProducer` код посыпает ряд текстовых сообщений, вызывая метод `send()` этого объекта. Данный метод принимает в первом параметре пункт назначения и строку с текстом сообщения – во втором.

В классе `JMSProducer` имеется несколько перегруженных версий метода `send()`; одну мы использовали в нашем примере – она создает экземпляр `javax.jms.TextMessage` и переписывает в него строку, полученную во втором параметре.

Хотя предыдущий пример отправляет только текстовые сообщения, мы не ограничены только этим типом сообщений. JMS API поддерживает несколько разных типов сообщений, которые могут быть отправлены и получены приложениями JMS. Все типы сообщений определяются как интерфейсы в пакете `javax.jms`.

В табл. 8.1 перечислены все доступные типы сообщений.

Таблица 8.1. Доступные типы сообщений

Тип сообщения	Описание
BytesMessage	Позволяет отправить в сообщении массив байтов. <code>JMSProducer</code> имеет вспомогательный метод <code>send()</code> , принимающий массив байтов; в момент отправки сообщения этот метод создает экземпляр <code>javax.jms.BytesMessage</code> «на лету».
MapMessage	Позволяет отправить в сообщении реализацию <code>java.util.Map</code> . <code>JMSProducer</code> имеет вспомогательный метод <code>send()</code> , принимающий экземпляр <code>Map</code> ; в момент отправки сообщения этот метод создает экземпляр <code>javax.jms.MapMessage</code> «на лету».
ObjectMessage	Позволяет отправить в сообщении реализацию <code>java.io.Serializable</code> . <code>JMSProducer</code> имеет вспомогательный метод <code>send()</code> , принимающий экземпляр <code>java.io.Serializable</code> ; в момент отправки сообщения этот метод создает экземпляр <code>javax.jms.ObjectMessage</code> «на лету».

Тип сообщения	Описание
StreamMessage	Позволяет отправить в сообщении массив байтов. Сообщения этого типа отличаются от сообщений BytesMessage тем, что хранят тип для каждого простого значения, добавленного в поток.
TextMessage	Позволяет отправить в сообщении реализацию java.lang.String. Как было показано в примере выше, JMSProducer имеет вспомогательный метод send(), принимающий String; в момент отправки сообщения этот метод создает экземпляр javax.jms.TextMessage «на лету».

За дополнительной информацией по всем этим типам сообщений обращайтесь к документации, доступной по адресу: <http://docs.oracle.com/javaee/7/api/>.

Извлечение сообщений из очереди

Нет никакого смысла отправлять сообщения в очередь, если никто не собирается их получать. Следующий пример демонстрирует, как извлекать сообщения из очереди JMS:

```
package net.ensode.glassfishbook;

import javax.annotation.Resource;
import javax.jms.ConnectionFactory;
import javax.jms.JMSConsumer;
import javax.jms.JMSContext;
import javax.jms.Queue;

public class MessageReceiver {

    @Resource(mappedName = "jms/GlassFishBookConnectionFactory")
    private static ConnectionFactory connectionFactory;

    @Resource(mappedName = "jms/GlassFishBookQueue")
    private static Queue queue;

    public void getMessages() {
        String message;
        boolean goodByeReceived = false;

        JMSContext jmsContext = connectionFactory.createContext();
        JMSConsumer jMSConsumer = jmsContext.createConsumer(queue);

        System.out.println("Waiting for messages...");
```

```

        while (!goodByeReceived) {
            message = jMSConsumer.receiveBody(String.class);

            if (message != null) {
                System.out.print("Received the following message: ");
                System.out.println(message);
                System.out.println();
                if (message.equals("Good bye!")) {
                    goodByeReceived = true;
                }
            }
        }

    public static void main(String[] args) {
        new MessageReceiver().getMessages();
    }
}

```

Как и в предыдущем примере, благодаря аннотации `@Resource`, происходит внедрение экземпляров `javax.jms.ConnectionFactory` и `javax.jms.Queue`.

Так же как в предыдущем примере, здесь сначала приобретается экземпляр `javax.jms.JMSSContext` вызовом метода `createContext()` объекта `ConnectionFactory`.

Затем, вызовом метода `createConsumer()` объекта `JMSSContext`, приобретается экземпляр `javax.jms.JMSConsumer`.

Прием сообщений из очереди сообщений осуществляется вызовом метода `receiveBody()` экземпляра `JMSConsumer`. Этот метод принимает ожидаемый тип сообщения (в данном примере `String.class`) и возвращает объект указанного типа (в данном примере – экземпляр `java.lang.String`).

В нашем конкретном случае мы поместили вызов метода в цикл `while`, чтобы убедиться, что никакие другие сообщения больше не поступают. В частности, мы ждем сообщение, содержащее текст «*Good bye!*». Как только это сообщение будет получено, цикл завершается и программа продолжает свою работу. В данном конкретном случае никаких других действий выполнять не требуется.

Так же как в предыдущем примере, использование утилиты `appclient` позволяет внедрить ресурсы в код и предотвратить необходимость добавления любых библиотек в `CLASSPATH`. После запуска приложения с помощью утилиты `appclient` в консоли должны появиться следующие строки:

```
appclient -client target/jmsptpconsumer.jar  
Waiting for messages...  
Received the following message: Testing, 1, 2, 3. Can you hear me?  
Received the following message: Do you copy?  
Received the following message: Good bye!
```

При этом, конечно, предполагается, что предыдущий пример уже запускался и сообщения были помещены в очередь. Если он не запускался, никаких сообщений принято не будет.

Асинхронный прием сообщений из очереди

Недостаток метода `JMSConsumer.receiveBody()` состоит в том, что он блокирует дальнейшее выполнение, пока сообщение не будет получено из очереди. Мы обошли это ограничение в предыдущем примере, прерывая цикл сразу после приема определенного сообщения (с текстом «Good bye!»).

Есть возможность избежать блокирования потребителя, если получать сообщения асинхронно, с помощью реализации интерфейса `javax.jms.MessageListener`.

Интерфейс `javax.jms.MessageListener` определяет единственный метод `onMessage()`. Он принимает экземпляр класса, реализующего интерфейс `javax.jms.Message`, в единственном параметре. Ниже приводится типичная реализация этого интерфейса:

```
package net.ensode.glassfishbook;  
  
import javax.jms.JMSEException;  
import javax.jms.Message;  
import javax.jms.MessageListener;  
import javax.jms.TextMessage;  
  
public class ExampleMessageListener implements MessageListener {  
  
    @Override  
    public void onMessage(Message message) {  
        TextMessage textMessage = (TextMessage) message;  
  
        try {  
            System.out.print("Received the following message: ");  
            System.out.println(textMessage.getText());  
        } catch (JMSEException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
        System.out.println(textMessage.getText());
        System.out.println();
    } catch (JMSEException e) {
        e.printStackTrace();
    }
}
```

В данном случае метод `onMessage()` просто выводит текст сообщения в консоль. Напомню, что когда вызывается метод `JMSProducer.send()` со вторым параметром типа `String`, JMS API создает экземпляр `javax.jms.TextMessage`; наша реализация `MessageListener` приводит принятый экземпляр `Message` к типу `TextMessage` и затем извлекает строку сообщения вызовом метода `getText()`.

Теперь наш основной код может делегировать извлечение сообщений пользовательской реализации `MessageListener`:

```
package net.ensode.glassfishbook;

import javax.annotation.Resource;
import javax.jms.ConnectionFactory;
import javax.jms.JMSConsumer;
import javax.jms.JMSContext;
import javax.jms.Queue;

public class AsynchMessReceiver {

    @Resource(mappedName = "jms/GlassFishBookConnectionFactory")
    private static ConnectionFactory connectionFactory;

    @Resource(mappedName = "jms/GlassFishBookQueue")
    private static Queue queue;

    public void getMessages() {
        try {
            JMSContext jmsContext = connectionFactory.createContext();
            JMSConsumer jMSConsumer = jmsContext.createConsumer(queue);

            jMSConsumer.setMessageListener(
                new ExampleMessageListener());

            System.out.println("The above line will allow the "
                + "MessageListener implementation to "
                + "receiving and processing messages"
                + " from the queue.");
            Thread.sleep(1000);
            System.out.println("Our code does not have to block "
                + "while messages are received.");
        }
    }
}
```

```
        Thread.sleep(1000);
        System.out.println("It can do other stuff "
            + "(hopefully something more useful than sending "
            + "silly output to the console. :)");
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) {
    new AsynchMessReceiver().getMessages();
}
```

Единственное существенно отличие этого примера от предыдущего состоит в том, что в данном случае вызывается метод `setMessageListener()` экземпляра `javax.jms.JMSConsumer`, полученным из контекста JMS. Мы передаем этому методу экземпляр пользовательской реализации `javax.jms.MessageListener`. Ее метод `onMessage()` автоматически вызывается всякий раз, когда в очереди появляется очередное сообщение. При таком подходе основной код не блокируется в ожидании сообщений.

Если запустить этот пример (разумеется, с использованием утилиты `appclient`), он выведет в консоль следующие строки:

```
appclient -client target/jmsptpasynchconsumer.jar
```

```
The above line will allow the MessageListener implementation to
receive and process messages from the queue.
```

```
Received the following message: Testing, 1, 2, 3. Can you hear me?
```

```
Received the following message: Do you copy?
```

```
Received the following message: Good bye!
```

```
Our code does not have to block while messages are received.
```

```
It can do other stuff (hopefully something more useful than
sending silly output to the console. :)
```

Обратите внимание, что сообщения были получены и обработаны в то время, как основной поток не прерывал своей работы. Это доказывает вывод метода `onMessage()` нашего класса `MessageListener`, который производится вызовами `System.out.println()` в основном классе.

Просмотр очередей сообщений

JMS предоставляет возможность просматривать очереди сообщений, не удаляя сообщения из очереди. Как это сделать, демонстрирует следующий пример:

```
package net.ensode.glassfishbook;

import java.util.Enumeration;
import javax.annotation.Resource;
import javax.jms.ConnectionFactory;
import javax.jms.JMSContext;
import javax.jms.JMSEException;
import javax.jms.Queue;
import javax.jms.QueueBrowser;
import javax.jms.TextMessage;

public class MessageQueueBrowser {

    @Resource(mappedName = "jms/GlassFishBookConnectionFactory")
    private static ConnectionFactory connectionFactory;

    @Resource(mappedName = "jms/GlassFishBookQueue")
    private static Queue queue;

    public void browseMessages() {
        try {
            Enumeration messageEnumeration;
            TextMessage textMessage;
            JMSContext jmsContext = connectionFactory.createContext();
            QueueBrowser browser = jmsContext.createBrowser(queue);

            messageEnumeration = browser.getEnumeration();

            if (messageEnumeration != null) {
                if (!messageEnumeration.hasMoreElements()) {
                    System.out.println("There are no messages "
                        + "in the queue.");
                } else {
                    System.out.println(
                        "The following messages are "
                        + "in the queue");
                    while (messageEnumeration.hasMoreElements()) {
                        textMessage =
                            (TextMessage) messageEnumeration.nextElement();
                        System.out.println(textMessage.getText());
                    }
                }
            }
        } catch (JMSEException e) {
```

```
        e.printStackTrace();
    }
}

public static void main(String[] args) {
    new MessageQueueBrowser().browseMessages();
}
}
```

Как видите, процедура просмотра сообщений в очереди элементарна. Для этого нужно получить фабрику соединений, очередь и контекст JMS, как обычно, а затем вызвать метод `createBrowser()` объекта контекста. Этот метод возвращает реализацию интерфейса `javax.jms.QueueBrowser`. Данный интерфейс содержит метод `getEnumeration()`, с помощью которого можно получить объект `Enumeration`, содержащий все сообщения в очереди. Чтобы исследовать сообщения в очереди, достаточно просто обойти их, одно за другим. В предыдущем примере мы просто вызываем метод `getText()` каждого сообщения в очереди.

Темы сообщений

Темы сообщений используются, когда обмен сообщениями организуется с применением модели «публикация/подписка». В этой модели одно и то же сообщение может быть отправлено всем подписчикам темы.

Отправка сообщений в тему

Следующий пример демонстрирует, как отправить сообщение в тему:

```
package net.ensode.glassfishbook;

import javax.annotation.Resource;
import javax.jms.ConnectionFactory;
import javax.jms.JMSCContext;
import javax.jms.JMSProducer;
import javax.jms.Topic;

public class MessageSender {

    @Resource(mappedName = "jms/GlassFishBookConnectionFactory")
    private static ConnectionFactory connectionFactory;

    @Resource(mappedName = "jms/GlassFishBookTopic")
```

```
private static Topic topic;

public void produceMessages() {
    JMSContext jmsContext = connectionFactory.createContext();
    JMSProducer jmsProducer = jmsContext.createProducer();

    String msg1 = "Testing, 1, 2, 3. Can you hear me?";
    String msg2 = "Do you copy?";
    String msg3 = "Good bye!";

    System.out.println("Sending the following message: "
        + msg1);
    jmsProducer.send(topic, msg1);
    System.out.println("Sending the following message: "
        + msg2);
    jmsProducer.send(topic, msg2);
    System.out.println("Sending the following message: "
        + msg3);
    jmsProducer.send(topic, msg3);
}

public static void main(String[] args) {
    new MessageSender().produceMessages();
}
```

Этот пример практически идентичен классу `MessageSender`, который рассматривался в обсуждении обмена сообщениями в модели «точка-точка». На самом деле отличаются только несколько выделенных строк кода. JMS API разрабатывался так, чтобы разработчикам приложений не нужно было изучать два разных API для двух моделей обмена сообщениями, PTP и «публикация/подписка».

Поскольку код практически идентичен примеру из раздела «Очереди сообщений», мы объясним только различия между этими двумя примерами. В данном примере вместо экземпляра класса, реализующего интерфейс `javax.jms.Queue`, объявляется экземпляр класса, реализующего интерфейс `javax.jms.Topic`. Затем объект `Topic` передается в первом параметре методу `send()` объекта `JMSProducer`, вместе с сообщением, подлежащим отправке.

Получение сообщений из темы

Подобно тому как отправка сообщений в тему практически идентична отправке сообщений в очередь, получение сообщений из темы практически идентично извлечению сообщений из очереди:

```
package net.ensode.glassfishbook;

import javax.annotation.Resource;
import javax.jms.ConnectionFactory;
import javax.jms.JMSConsumer;
import javax.jms.JMSCluster;
import javax.jms.Topic;

public class MessageReceiver {

    @Resource(mappedName = "jms/GlassFishBookConnectionFactory")
    private static ConnectionFactory connectionFactory;

    @Resource(mappedName = "jms/GlassFishBookTopic")
    private static Topic topic;

    public void getMessages() {
        String message;
        boolean goodByeReceived = false;

        JMSCluster jmsContext = connectionFactory.createContext();
        JMSConsumer jMSConsumer = jmsContext.createConsumer(topic);

        System.out.println("Waiting for messages...");
        while (!goodByeReceived) {
            message = jMSConsumer.receiveBody(String.class);

            if (message != null) {
                System.out.print("Received the following message: ");
                System.out.println(message);
                System.out.println();
                if (message.equals("Good bye!")) {
                    goodByeReceived = true;
                }
            }
        }
    }

    public static void main(String[] args) {
        new MessageReceiver().getMessages();
    }
}
```

И снова, различия между этим примером и соответствующим примером для модели РТР тривиальны. Вместо экземпляра класса, реализующего интерфейс javax.jms.Queue здесь объявляется класс, реализующий интерфейс javax.jms.Topic. Для внедрения экземпляра этого класса используется аннотация `@Resource`, которой передается JNDI-имя темы, указанное при ее создании в веб-консоли.

сервера GlassFish. Затем приобретаются экземпляры `JMSContext` и `JMSConsumer`, и, наконец, из темы извлекаются сообщения вызовом метода `receiveBody()` объекта `JMSConsumer`.

Модель обмена сообщениями «публикация/подписка» имеет то преимущество, что сообщения могут быть отправлены нескольким потребителям. Это легко можно проверить, одновременно запустив два экземпляра класса `MessageReceiver`, созданного в этом разделе, а затем запустив экземпляр класса `MessageSender`, созданный в предыдущем разделе. В результате оба экземпляра должны сообщить о получении всех отправленных сообщений.

Так же, как и в случае с очередями, сообщения из темы можно извлекать асинхронно. Соответствующая процедура настолько похожа на версию с очередью сообщений, что мы не будем приводить пример. Чтобы преобразовать асинхронный пример, представленный выше в этой главе, для работы с темой сообщений, просто замените переменную `javax.jms.Queue` экземпляром `javax.jms.Topic` и внедрите соответствующий экземпляр, указав имя темы "jms/GlassFishBookTopic" в атрибуте `mappedName` аннотации `@Resource`, декорирующей экземпляр `javax.jms.Topic`.

Создание долговременной подписки

Недостаток использования модели обмена сообщениями типа «публикация/подписка» состоит в том, что потребители сообщений должны выполнятся в момент отправки сообщений в тему. Если потребитель сообщения не будет выполняться в этот момент, он не получит сообщения, тогда как в модели PTP сообщения сохраняются в очереди, пока потребитель не извлечет их. К счастью, JMS API предоставляет возможность сохранять сообщения в теме при использовании модели «публикация/подписка», пока все потребители, подписанные на тему, не получат эти сообщения. Это может быть достигнуто путем создания долговременной подписки на тему.

Чтобы иметь возможность обслуживать долговременную подписку, необходимо установить свойство `clientId` фабрики соединений. У каждой долговременной подписки должен быть уникальный клиентский идентификатор и для каждого подписчика должна быть объявлена уникальная фабрика соединений.



Возникло исключение `InvalidClientIdException`? Только один клиент JMS с конкретным клиентским идентификатором сможет соединиться с темой. Если JMS-соединение попытаются получить

несколько клиентов, используя одну и ту же фабрику соединений, возникнет исключение `JMSEException`, уведомляющее, что такой клиентский идентификатор уже используется. Решить эту проблему можно, создав фабрику соединений для каждого потенциального клиента, который будет на долговременной основе получать сообщения из темы.

Как уже говорилось выше, самый простой способ добавить фабрику соединений – воспользоваться веб-консолью сервера GlassFish и выполнить следующие шаги:

1. Распахнуть узел **Resources** (Ресурсы) в панели навигации слева.
2. Распахнуть узел **JMS Resources** (Ресурсы JMS).
3. Щелкнуть на узле **Connection Factories** (Фабрики соединений).
4. Щелкнуть на кнопке **New...** (Создать) в основной области страницы.

В следующем примере будут использованы настройки, как показано на рис. 8.8.

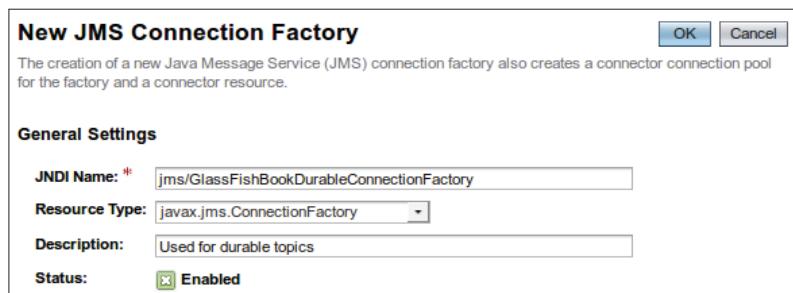


Рис. 8.8. Настройки фабрики соединений для долговременной подписки

Прежде чем щелкнуть на кнопке **OK**, прокрутите страницу до конца, щелкните на кнопке **Add Property** (Добавить свойство) и введите новое свойство с именем `clientId`. В данном примере этому свойству должно быть присвоено значение `ExampleId` (см. рис. 8.9).

Теперь, выполнив необходимые настройки GlassFish, можно попробовать организовать долговременную подписку:

```
package net.ensode.glassfishbook;  
  
import javax.annotation.Resource;
```

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSConsumer;
import javax.jms.JMSContext;
import javax.jms.JMSEException;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.jms.Topic;

public class MessageReceiver {

    @Resource(mappedName =
        "jms/GlassFishBookDurableConnectionFactory")
    private static ConnectionFactory connectionFactory;

    @Resource(mappedName = "jms/GlassFishBookTopic")
    private static Topic topic;

    public void getMessages() {
        String message;
        boolean goodByeReceived = false;

        JMSContext jmsContext = connectionFactory.createContext();
        JMSConsumer jMSConsumer =
            jmsContext.createDurableConsumer(topic, "Subscriber1");

        System.out.println("Waiting for messages...");
        while (!goodByeReceived) {
            message = jMSConsumer.receiveBody(String.class);

            if (message != null) {
                System.out.print("Received the following message: ");
                System.out.println(message);
                System.out.println();
                if (message.equals("Good bye!")) {
                    goodByeReceived = true;
                }
            }
        }
    }

    public static void main(String[] args) {
        new MessageReceiver().getMessages();
    }
}
```

Как видите, этот пример практически не отличается от предыдущих, целью которых было получение сообщений. Здесь имеется только два отличия: внедренный экземпляр `ConnectionFactory` – это

экземпляр, созданным нами ранее в этом разделе для организации долговременной подписки, а вместо метода `createConsumer()` объекта контекста JMS вызывается метод `createDurableSubscriber()`. Этот метод принимает два аргумента: объект темы JMS для извлечения сообщений и имя пункта назначения. Второй параметр должен быть уникальным для каждого из подписчиков темы.

Additional Properties (1)			
	Add Property	Delete Properties	
Select	Name	Value	Description
<input type="checkbox"/>	ClientId	ExampleId	

Рис. 8.9. Добавление свойства ClientId

Резюме

В этой главе мы рассмотрели приемы настройки фабрики соединений, очереди и темы сообщений на сервере GlassFish с использованием веб-консоли.

Мы также обсудили порядок отправки сообщений в очередь через интерфейс `javax.jms.JMSPublisher`.

Дополнительно было показано, как извлекать сообщения из очереди через интерфейс `javax.jms.JMSPublisher` и как то же самое делать асинхронно, путем реализации интерфейса `javax.jms.MessageListener`.

Мы также узнали, как использовать эти интерфейсы для отправки сообщений в темы и получения их из тем.

Выяснили, как просмотреть сообщения в очереди сообщений через интерфейс `javax.jms.QueueBrowser`, не удаляя их.

Наконец, мы показали, как настраивать долговременные подписки на темы JMS и взаимодействовать с ними.

В следующей главе мы поговорим о проблемах безопасности приложений Java EE.



ГЛАВА 9.

Безопасность приложений

Java EE

В этой главе мы посмотрим, как защитить приложение Java EE, используя возможности встроенных средств защиты GlassFish.

В основе безопасности Java EE лежит **API службы аутентификации и авторизации Java (Java Authentication and Authorization Service, JAAS)**. Как будет показано ниже, безопасность приложений Java EE требует написать совсем немного программного кода. По большей части безопасность приложения достигается путем создания пользователей и групп безопасности (security groups) в областях безопасности (security realms) на сервере приложений; затем выполняется настройка приложения с использованием конкретных областей безопасности для аутентификации и авторизации.

Вот некоторые из тем, которые мы затронем в этой главе:

- ❖ область администратора;
- ❖ область файла;
- ❖ область сертификата;
- ❖ создание самоподписанных сертификатов безопасности;
- ❖ область JDBC;
- ❖ пользовательские области.

Области безопасности

Область безопасности (security realm) представляет собой группу пользователей и связанную с ней группу безопасности. Пользователь может принадлежать одной или более группам безопасности. В зависимости от принадлежности к той или иной группе, система позволяет пользователям выполнять те или иные действия. Например, с приложением могут работать обычные пользователи, которым нужна только основная функциональность приложения, и могут быть ад-

министранторы, которые, помимо использования основной функциональности приложения, наделены правом добавлять в систему других пользователей.

Области безопасности хранят учетную информацию (имя пользователя, пароль и группы безопасности). Приложениям не нужно самим реализовать эту функциональность; их можно просто настроить для получения данной информации из области безопасности. Одна область безопасности может использоваться несколькими приложениями.

Предопределенные области безопасности

В GlassFish имеется три предварительно настроенные области безопасности: **область администратора (admin-realm)**, **область файла (file realm)** и **область сертификата (certificate realm)**. Область администратора используется для управления доступом пользователей к веб-консоли сервера GlassFish и не должна использоваться для других приложений. Область файла хранит пользовательскую информацию в файле. Область сертификата выполняет поиск клиентского сертификата для аутентификации пользователя.

На рис. 9.1 показан список предопределенных областей безопасности в веб-консоли GlassFish.



Рис. 9.1. Список предопределенных областей безопасности в веб-консоли GlassFish

В дополнение к предопределенным областям безопасности можно добавлять дополнительные области, прикладывая минимум усилий. Как это сделать, будет рассказываться чуть ниже, но сначала обсудим предопределенные области безопасности GlassFish.

Область администратора

Область администратора распространяется на предопределенного пользователя с именем **admin**, который принадлежит предопределенной группе **asadmin**.

Чтобы понять, как добавлять пользователей в область, давайте добавим нового пользователя в область администратора. Это позволит данному пользователю входить в веб-консоль сервера GlassFish. Для этого откройте веб-консоль сервера GlassFish, распахните узел **Configuration** (Конфигурация) в панели навигации слева, затем последовательно распахните узлы **server-config**, **Security** (Безопасность), **Realms** (Области) и щелкните на пункте **admin-realm**. В основной части страницы справа должна отобразиться форма настройки области, как показано на рис. 9.2.

Edit Realm

Save Cancel

Edit an existing security (authentication) realm.

Manage Users

* Indicates required field

Configuration Name: server-config

Realm Name: admin-realm

Class Name: com.sun.enterprise.security.auth.realm.file.FileRealm

Properties specific to this Class

JAAS Context: * fileRealm
Identifier for the login module to use for this realm

Key File: * \${com.sun.aas.instanceRoot}/config/admin-keyfile
Full path and name of the file where the server will store all user, group, and password information for this realm

Assign Groups:
Comma-separated list of group names

Additional Properties (0)

Add Property Delete Properties

Select	Name	Value	Description
No items found.			

Save Cancel

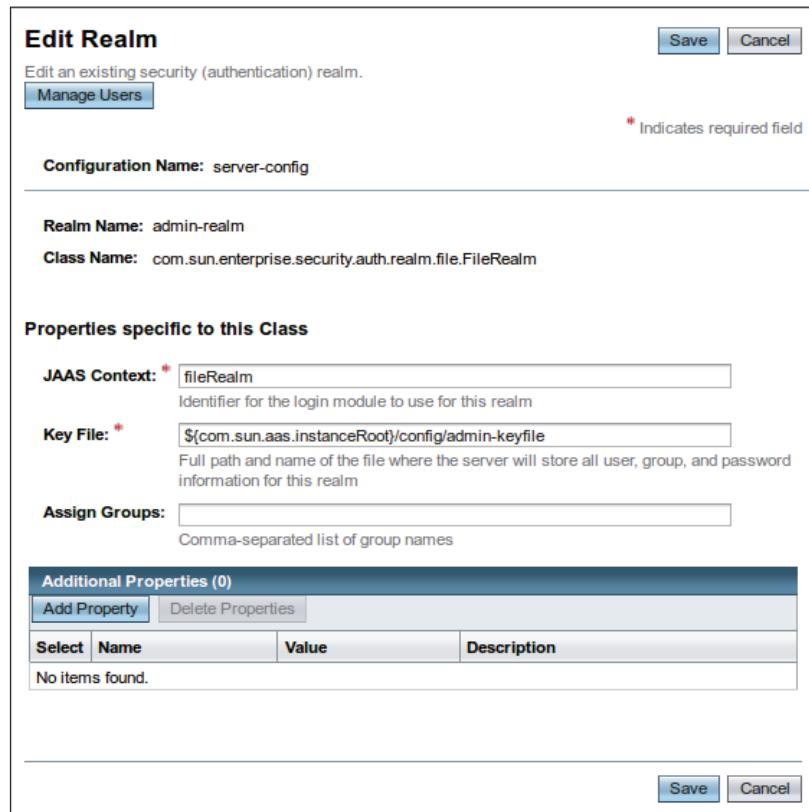


Рис. 9.2. Форма настройки области

Чтобы добавить пользователя в область, щелкните на кнопке **Manage Users** (Управление пользователями) вверху слева. В основной части страницы должна отобразиться форма управления пользователями, как показано на рис. 9.3.

The screenshot shows a web-based user management interface titled 'File Users'. At the top right is a 'Back' button. Below the title, a message says 'Manage user accounts for the currently selected security realm.' A configuration name 'server-config' is displayed. The 'Realm Name' is set to 'admin-realm'. A table titled 'File Users (1)' lists one user: 'admin' with 'User ID' and 'Group List' both set to 'asadmin'. There are 'New...' and 'Delete' buttons above the table.

Рис. 9.3. Форма управления пользователями

Чтобы добавить пользователя в область, просто щелкните на кнопке **New...** (Создать) вверху слева и введите данные нового пользователя, как показано на рис. 9.4.

The screenshot shows a 'New File Realm User' dialog. It has 'OK' and 'Cancel' buttons at the top right. A note says '* Indicates required field'. The 'Configuration Name' is set to 'server-config'. The 'Realm Name' is 'admin-realm'. The 'User ID' is 'root'. A note below it says 'Name can be up to 255 characters, must contain only letters, digits, underscore, dash, or dot characters'. The 'Group List' is 'asadmin'. The 'New Password' and 'Confirm New Password' fields both contain '*****'.

Рис. 9.4. Форма создания нового пользователя

Как показано на рис. 9.4, мы добавили нового пользователя с именем **root** в группу **asadmin**, а также ввели для него пароль.



Веб-консоль сервера GlassFish позволяет входить только пользователям из группы `asadmin`. Если ошибиться при добавлении пользователя в эту группу безопасности, он не сможет войти в консоль.

После этого вновь созданная учетная запись должна появиться в списке пользователей в области администратора, как показано на рис. 9.5.

The screenshot shows the 'File Users' configuration page. At the top, it says 'Manage user accounts for the currently selected security realm.' Below that, 'Configuration Name: server-config' and 'Realm Name: admin-realm' are specified. A table titled 'File Users (2)' lists the users:

Select	User ID	Group List:
<input type="checkbox"/>	root	asadmin
<input type="checkbox"/>	admin	asadmin

Рис. 9.5. Новая учетная запись в списке admin-realm

Теперь, после успешного добавления нового пользователя в область администратора, можно проверить новую учетную запись и попытаться войти в веб-консоль GlassFish с учетными данными добавленного пользователя.

Область файла

Вторая предопределенная область сервера GlassFish – это область файла. Она хранит зашифрованную пользовательскую информацию в текстовом файле. Добавление пользователей в эту область выполняется практически так же, как добавление пользователей в область администратора. Чтобы добавить пользователя, распахните последовательно узлы **Configuration** (Конфигурация) | **server-config** | **Security** (Безопасность) | **Realms** (Области), затем щелкните на пункте **file** (файл), на кнопке **Manage Users** (Управление пользователями) и, наконец, на кнопке **New...** (Создать). В основной части страницы должна отобразиться форма, как показано на рис. 9.6.

New File Realm User

Create new user accounts for the currently selected security realm.

OK **Cancel**

Configuration Name: server-config * Indicates required field

Realm Name:	file
User ID: *	<input type="text" value="peter"/>
Name can be up to 255 characters, must contain only letters, digits, underscore, dash, or dot characters	
Group List:	<input type="text" value="appuser,appadmin"/>
Separate multiple groups with colon	
New Password:	<input type="password" value="*****"/>
Confirm New Password:	<input type="password" value="*****"/>

Рис. 9.6. Форма создания нового пользователя в области файла

Поскольку эта область предназначается для использования приложениями, можно придумать собственные группы. Группы удобно использовать, чтобы дать одни и те же привилегии нескольким пользователям. Например, всех пользователей, нуждающихся в привилегиях администратора, можно добавить в группу администраторов (имя группы можно выбрать произвольно).

В этом примере мы добавили пользователя peter в группы appuser и appadmin.

Щелкните на кнопке **OK**, чтобы сохранить нового пользователя и вернуться к списку пользователей в данной области (см. рис. 9.7).

File Users

Manage user accounts for the currently selected security realm.

Back

Configuration Name: server-config

Realm Name: file

File Users (1)

Select	User ID	Group List:
<input type="checkbox"/>	peter	appuser appadmin

Рис. 9.7. Список пользователей в области файла

Щелкая на кнопке **New...** (Создать), можно добавлять других пользователей в область. Давайте добавим еще одного пользователя с именем `joe`, принадлежащего только группе `appuser`, как показано на рис. 9.8.

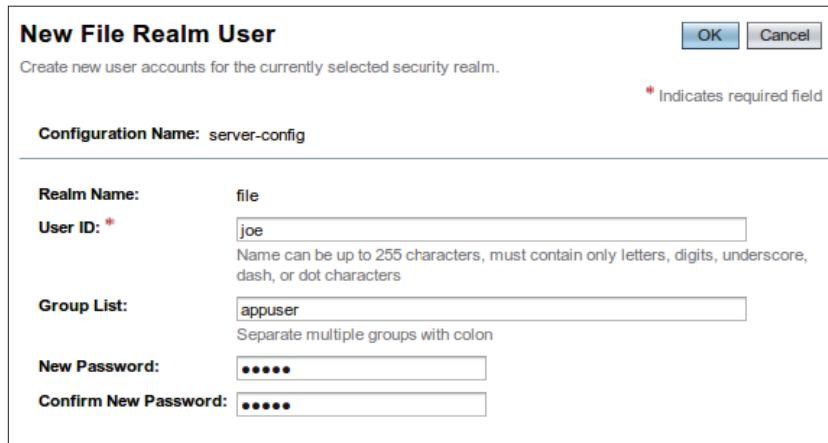


Рис. 9.8. Новый пользователь joe

Как видите, добавить пользователей в область файла очень просто. Теперь посмотрим, как выполняется аутентификация и авторизация пользователей с помощью области файла.

Стандартная аутентификация через область файла

В предыдущем разделе мы узнали, как добавлять пользователей в область файла и как включать этих пользователей в группы. В этом разделе будет показано, как защитить веб-приложение, чтобы только пользователи прошедшие аутентификацию и авторизацию могли получить к нему доступ. Для управления доступом пользователей это веб-приложение будет использовать область файла.

Приложение состоит из нескольких простых JSF-страниц. Всю логику аутентификации реализует сервер приложений. Поэтому единственное место, куда нам нужно внести изменения для обеспечения безопасности приложения, находится в дескрипторах развертывания `web.xml` и `glassfish-web.xml`. Начнем с `web.xml`, который показан ниже:

```
<?xml version='1.0' encoding='UTF-8'?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <context-param>
        <param-name>javax.faces.PROJECT_STAGE</param-name>
        <param-value>Development</param-value>
    </context-param>
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.jsf</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>index.jsf</welcome-file>
    </welcome-file-list>
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Admin Pages</web-resource-name>
            <url-pattern>/admin/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>admin</role-name>
        </auth-constraint>
    </security-constraint>
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>AllPages</web-resource-name>
            <url-pattern>/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>user</role-name>
        </auth-constraint>
    </security-constraint>
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>file</realm-name>
    </login-config>
</web-app>
```

Элемент `<security-constraint>` определяет, кто может получить доступ к страницам, соответствующим указанному шаблону URL. Шаблон URL страницы определяется элементом `<url-pattern>`, который, как показано в примере, должен быть вложен в элемент `<web-resource-collection>`. Роли, имеющие разрешение на доступ к стра-

нице, определяются элементом `<role-name>`, который должен быть вложен в элемент `<auth-constraint>`.

В этом примере определяются два набора страниц, которые должны быть защищены. В первый входят все страницы, URL которых начинается с `/admin`. Доступ к этим страницам смогут получить только пользователи, входящие в группу администраторов. Во второй набор входят страницы, определяемые шаблоном URL `/*`. Доступ к этим страницам смогут получить только пользователи с ролью `user`. Стоит отметить, что второй набор страниц является надмножеством первого, т. е. любая страница, URL которой соответствует шаблону `/admin/*`, также соответствует шаблону `/*`. В подобных случаях «побеждает» наиболее конкретный вариант. В данном примере пользователи с ролью `user` (не наделенные ролью `admin`) не смогут получать доступ к страницам, URL которых начинается с `/admin`.

Следующий элемент, который должен быть добавлен в файл `web.xml` для защиты страниц, – `<login-config>`. Он должен содержать элемент `<auth-method>`, определяющий метод аутентификации для приложения. Допустимые значения данного элемента: `BASIC`, `DIGEST`, `FORM` и `CLIENT-CERT`.

Значение `BASIC` (стандартная) указывает, что будет использоваться стандартная аутентификация. Когда используется этот тип аутентификации, при первой попытке пользователя получить доступ к защищенной странице веб-браузер выведет диалог с приглашением ввести имя пользователя и пароль. Если стандартная (`BASIC`) аутентификация выполняется не по протоколу HTTPS, учетные данные пользователя будут передаваться на сервер в кодировке Base64, в незашифрованном виде. Злоумышленнику не составит особого труда расшифровать эту информацию, поэтому использовать стандартную аутентификацию не рекомендуется.

Дайджест-аутентификация `DIGEST` подобна стандартной аутентификации, за исключением того, что используется дайджест (отпечаток) MD5 для шифрования учетных данных пользователя вместо их отправки в формате Base64.

Аутентификация с помощью формы (`FORM`) основана на использовании пользовательской HTML- или JSP-страницы с HTML-формой, содержащей поля имени пользователя и пароля. Значения, введенные в эту форму, затем проверяются в области безопасности для пользовательской аутентификации и авторизации. Если только не используется протокол HTTPS, учетные данные пользователя отправляются открытым текстом. В связи с этим рекомендуется ис-

пользовать именно протокол HTTPS, поскольку он шифрует данные. Мы рассмотрим настройку GlassFish для использования HTTPS далее в этой главе.

Аутентификация на основе клиентского сертификата (`CLIENT-CERT`) использует для аутентификации и авторизации пользователя клиентские сертификаты.

Элемент `<realm-name>`, вложенный в элемент `<login-config>`, указывает, какие области безопасности использовать для аутентификации и авторизации пользователя. В нашем примере мы используем область файла.

Все элементы файла `web.xml`, которые мы обсудили в этом разделе, могут использоваться с любой областью безопасности – они не привязаны к области файла. Единственное, что связывает наше приложение с областью файла, – значение элемента `<realm-name>`. Тем не менее, следует иметь в виду, что не все методы аутентификации поддерживаются всеми областями. Область файла поддерживает только стандартную (`BASIC`) аутентификацию и аутентификацию на основе формы (`FORM`).

Прежде чем приступать к аутентификации пользователей, нам нужно связать роли пользователей, определенные в файле `web.xml`, с группами, определенными в области. Это реализуется в файле дескриптора развертывания `glassfish-web.xml`:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE glassfish-web-app PUBLIC "-//GlassFish.org//DTD
    GlassFish Application Server 3.1 Servlet 3.0//EN"
    <http://glassfish.org/dtds/glassfish-web-app_3_0-1.dtd">
<glassfish-web-app error-url="">
    <context-root>/filerealmauth</context-root>
    <security-role-mapping>
        <role-name>admin</role-name>
        <group-name>appadmin</group-name>
    </security-role-mapping>
    <security-role-mapping>
        <role-name>user</role-name>
        <group-name>appuser</group-name>
    </security-role-mapping>
    <class-loader delegate="true"/>
</glassfish-web-app>
```

Как видите, дескриптор развертывания `glassfish-web.xml` может иметь один или более элементов `<security-role-mapping>`. Для каждой роли в файле `web.xml` (в тегах `<auth-constraint>`) необходим один такой элемент. Подэлемент `<role-name>` указывает роль для отобра-

жения. Его значение должно совпадать со значением соответствующего элемента `<role-name>` в файле `web.xml`. Подэлемент `<group-name>` должен совпадать со значением группы безопасности в области, которая используется для аутентификации пользователей в приложении.

В нашем примере первый элемент `<security-role-mapping>` отображает роль `admin`, определенную в дескрипторе развертывания приложения `web.xml` (в группе `appadmin`), которую мы создали ранее, добавляя пользователей в область файла. Второй элемент `<security-role-mapping>` отображает роль `user` в файле `web.xml` на группу `appuser` в области файла.

Как уже говорилось, нам не нужно писать какой-либо программный код для аутентификации и авторизации пользователей. Все, что мы должны сделать, – это изменить дескрипторы развертывания приложения, как описано в данном разделе. Поскольку наше приложение состоит только из нескольких простых страниц, мы не будем показывать их исходный код. Структура приложения показана на рис. 9.9.



Рис. 9.9. Структура приложения

В соответствии с дескриптором развертывания для нашего приложения, пользователи с ролью `user` смогут получить доступ к двум страницам в корне приложения (`index.xhtml` и `random.xhtml`). Только пользователи с ролью `admin` будут допущены к страницам в каталоге `admin`, где в данном случае находится единственная страница с названием `index.xhtml`.

После упаковки и развертывания приложения, введите в адресной строке браузера URL любой из страниц и вы должны увидеть всплывающее окно с предложением ввести имя пользователя и пароль, как показано на рис. 9.10.

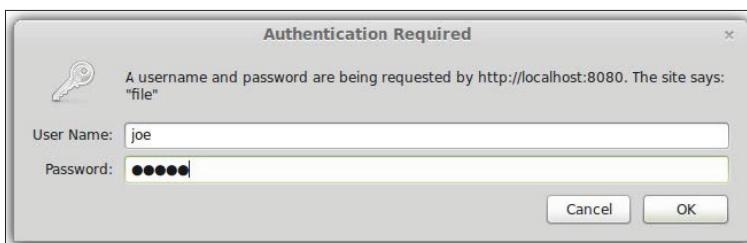


Рис. 9.10. Диалог ввода имени пользователя и пароля

После ввода корректного имени пользователя и пароля, откроется запрошенная страница, как показано на рис. 9.11.

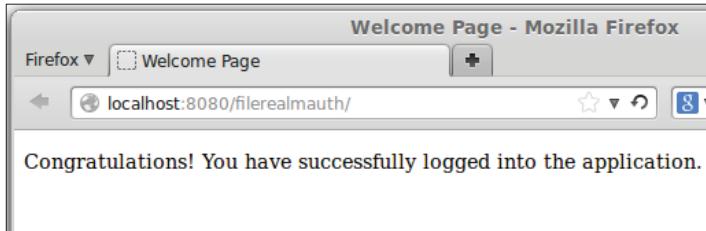


Рис. 9.11. После ввода имени пользователя и пароля откроется запрошенная страница

Теперь пользователь сможет перейти на любую страницу в приложении, к которой имеет доступ, – либо по ссылкам, либо вводя URL в адресной строке браузера, без повторного ввода имени пользователя и пароля.

Обратите внимание (см. рис. 9.10), что мы зарегистрировались как пользователь `joe` (Джо). Этот пользователь обладает только ролью `user` не имеет доступа к страницам, адреса URL которых начинаются, например, с `/admin`. Если Джо попытается перейти к одной из этих страниц, он увидит сообщение об ошибке, как показано на рис. 9.12.

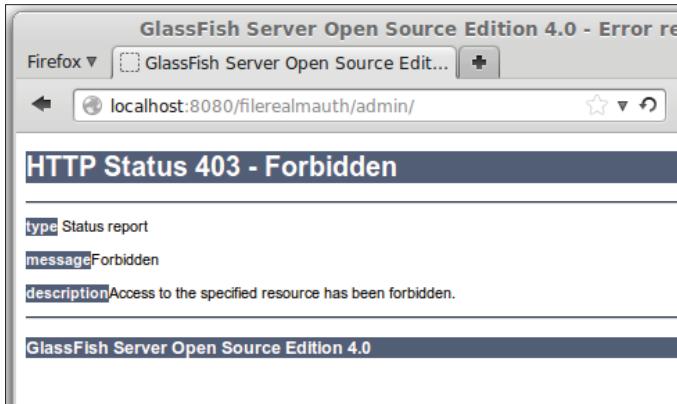


Рис. 9.12. Сообщение об ошибке при открыть страницу в отсутствие необходимых привилегий

Только пользователи с ролью `admin` смогут увидеть страницы, соответствующие данному URL. Добавляя пользователей в область файлов, мы включили туда пользователя `peter`, обладающего этой ролью.

Если зарегистрироваться как peter, можно увидеть требуемую страницу (см. рис. 9.13). В случае стандартной аутентификации, единственный способ выйти из приложения – закрыть браузер. Поэтому, чтобы войти в систему как peter, придется закрыть и вновь открыть браузер.



Рис. 9.13. Страница в административном разделе приложения

Как отмечалось выше, недостатком стандартного метода аутентификации, использованного в этом примере, является отсутствие шифрования учетной информации. Один из способов обойти эту проблему – использовать протокол HTTPS (HTTP через SSL). В этом случае вся информация, передаваемая между браузером и сервером, шифруется.

Чтобы задействовать HTTPS, достаточно изменить дескриптор развертывания приложения web.xml.

```
<?xml version='1.0' encoding='UTF-8'?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Admin Pages</web-resource-name>
            <url-pattern>/admin/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>admin</role-name>
        </auth-constraint>
        <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>AllPages</web-resource-name>
            <url-pattern>/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
```

```
<role-name>user</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>
</web-app>
```

Как видите, чтобы организовать доступ к приложению только через HTTPS, нужно всего лишь добавить элемент `<user-data-constraint>`, содержащий вложенный элемент `<transport-guarantee>` для каждого набора страниц, которые желательно шифровать при передаче по сети. Наборы страниц, которые должны быть защищены, объявляются в элементе `<security-constraint>`, в файле дескриптора развертывания `web.xml`.

Теперь, когда мы будем получать доступ к приложению через не-безопасный порт HTTP (по умолчанию – 8080), запрос будет автоматически переадресован к безопасному порту HTTPS (значение по умолчанию – 8181).

В данном примере в элементе `<transport-guarantee>` указывается значение `CONFIDENTIAL`. Оно требует шифрования всех данных, передаваемых между браузером и сервером. Кроме того, если запрос поступит на небезопасный порт HTTP, он автоматически будет переадресован на защищенный порт HTTPS.

Другим допустимым значением в элементе `<transport-guarantee>` является `INTEGRAL`. Оно гарантирует целостность данных, передаваемых между браузером и сервером. Другими словами, данные не могут быть изменены при их транспортировке. При использовании этого значения запросы, переданные по HTTP, не переадресуются автоматически в порт HTTPS. Если пользователь попытается получить доступ к защищенной странице через HTTP, когда используется это значение, браузер отклонит запрос и вернет ошибку 403 («Доступ запрещен»).

Третьим и последним допустимым значением в элементе `<transport-guarantee>` является `NONE`. При использовании этого значения не дается никаких гарантий в отношении целостности или конфиденциальности данных. `NONE` является значением по умолчанию, используемым, когда элемент `<transport-guarantee>` отсутствует в дескрипторе развертывания `web.xml`.

После внесения перечисленных изменений в дескриптор развертывания web.xml и повторного развертывания приложения откройте в браузере любую страницу, и вы увидите предупреждение при обращении к нашему приложению с помощью Firefox (см. рис. 9.14).

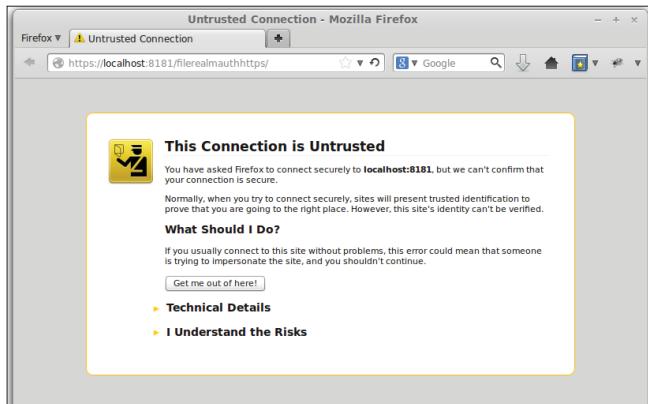


Рис. 9.14. Предупреждение в браузере Firefox

Если распахнуть узел **I Understand the Risks** (Я осознаю риски) и щелкнуть на кнопке **Add Exception...** (Добавить исключение...), откроется диалог, как показано на рис. 9.15.



Рис. 9.15. Диалог подтверждения исключения безопасности

После щелчка по кнопке **Confirm Security Exception** (Подтверждаю исключение безопасности) будет предложено ввести имя пользователя и пароль. После ввода учетных данных будет предоставлен доступ к требуемой странице, как показано на рис. 9.16.

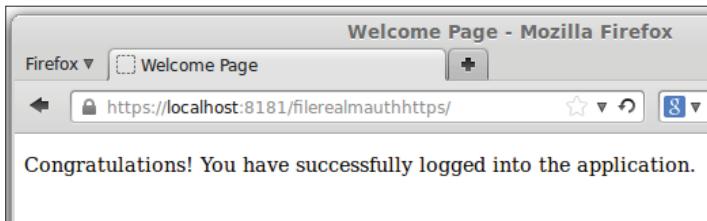


Рис. 9.16. После ввода учетных данных будет предоставлен доступ к странице

Это предупреждение появляется на экране потому, что для взаимодействия с сервером по протоколу HTTPS необходим сертификат SSL. Как правило сертификаты SSL выпускаются **центрами сертификации**, такими как Verisign или Thawte. Эти центры в цифровой форме подписывают сертификаты, удостоверяя тем самым, что сервер принадлежит той сущности, принадлежность к которой он декларирует.

Цифровой сертификат одного из этих центров сертификации обычно стоит около 400 долларов США и действует в течение одного года. Поскольку такие сертификаты могут быть слишком дороги для целей разработки или тестирования, GlassFish поставляется с предварительно созданным самоподписанным сертификатом SSL. А так как этот сертификат не был подписан центром сертификации, браузер выводит предупреждение при попытке получить доступ к защищенной странице по протоколу HTTPS.

Обратите внимание на адрес URL на рис. 9.15: в качестве протокола установлен HTTPS и используется порт 8181, при том, что в адресной строке браузера был указан адрес URL `http://localhost:8080/filerealmauthhttps`. Из-за изменений в дескрипторе развертывания приложения `web.xml` запрос был автоматически переадресован на данный адрес URL. Конечно, пользователи могут напрямую ввести безопасный URL и он будет работать без проблем.

Любые данные, передаваемые по протоколу HTTPS, шифруются, включая имя пользователя и пароль. Протокол HTTPS позволяет безопасно использовать стандартную аутентификацию, однако у нее есть еще один недостаток: выйти из приложения можно единствен-

ным способом – закрыв браузер. Если потребуется дать пользователям возможность выходить из приложения, не закрывая браузер, следует использовать аутентификацию на основе формы.

Аутентификация на основе формы

Чтобы использовать аутентификацию на основе формы, необходимо внести некоторые изменения в файле дескриптора развертывания приложения – web.xml:

```
<?xml version='1.0' encoding='UTF-8'?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
  </context-param>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>index.jsf</welcome-file>
  </welcome-file-list>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Admin Pages</web-resource-name>
      <url-pattern>/admin/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>admin</role-name>
    </auth-constraint>
    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
  </security-constraint>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>AllPages</web-resource-name>
```

```
<url-pattern>/*</url-pattern>
</web-resource-collection>
<auth-constraint>
    <role-name>user</role-name>
</auth-constraint>
<user-data-constraint>
    <description/>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
</security-constraint>
<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>file</realm-name>
    <form-login-config>
        <form-login-page>/login.jsf</form-login-page>
        <form-error-page>/loginerror.jsf</form-error-page>
    </form-login-config>
</login-config>
</web-app>
```

При использовании аутентификации на основе формы мы просто указываем значение FORM в элементе `<auth-method>`, в файле `web.xml`. Дополнительно необходимо предоставить страницу регистрации и страницу вывода сообщения в случае ошибки регистрации. Адреса URL этих страниц определяются как значения элементов `<form-login-page>` и `<form-error-page>` соответственно. Как было показано выше, эти элементы должны вкладываться в элемент `<form-login-config>`.

Ниже показана разметка страницы регистрации для нашего приложения:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!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">
  <head>
    <title>Login</title>
  </head>
  <body>
    <p>Please enter your username and password to access the
       application
    </p>
    <form method="POST" action="j_security_check">
      <table cellpadding="0" cellspacing="0" border="0">
        <tr>
          <td align="right">Username: </td>
          <td>
            <input type="text" name="j_username"/>
          </td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

```
</td>
</tr>
<tr>
    <td align="right">Password:&nbsp;</td>
    <td>
        <input type="password" name="j_password"/>
    </td>
</tr>
<tr>
    <td></td>
    <td>
        <input type="submit" value="Login"/>
    </td>
</tr>
</table>
</form>
</h:body>
</html>
```

Обратите внимание, хотя страница регистрации – это страница JSF, в ней используется стандартный тег `<form>`, а не его JSF-аналог `<h:form>`. Причина в том, что атрибут `action` формы должен иметь значение `j_security_check`, а это невозможно при использовании JSF-тега `<h:form>`. По той же причине используются стандартные HTML- поля ввода вместо их JSF-аналогов.

Страница регистрации для приложения, использующего аутентификацию на основе формы, должна содержать форму, которая отправляется методом `POST`, а ее атрибут `action` имеет значение `j_security_check`. Нам не нужно писать код аутентификации, потому что обработка формы выполняется сервером приложений.

Форма на странице регистрации должна содержать текстовое поле с именем `j_username`; оно предназначено для ввода имени пользователя. Также форма должна содержать поле с именем `j_password`, предназначенное для ввода пароля, и, конечно же, кнопку отправки формы для передачи данных серверу.

Единственное требование к странице регистрации – наличие формы с атрибутами, как определено выше, и полями ввода `j_username` и `j_password`.

К странице вывода сообщения об ошибке никаких специальных требований не предъявляется. Конечно, она должна отображать текст, сообщающий пользователю, что регистрация потерпела неудачу. Но в остальном она может содержать все, что вам заглагорассудится. Страница вывода сообщения об ошибке для нашего приложения просто сообщает пользователю, что произошла ошибка

регистрации, и включает ссылку для возврата на страницу регистрации, чтобы дать пользователю возможность попытаться зарегистрироваться еще раз.

В дополнение к странице регистрации и странице вывода сообщения об ошибке мы добавили в приложение именованный компонент CDI. Это позволяет нам реализовать функциональность выхода из системы, что было невозможно в случае использования стандартной аутентификации. Ниже приводится код, реализующий выход из приложения:

```
package net.ensode.glassfishbook;

import javax.enterprise.context.RequestScoped;
import javax.faces.context.ExternalContext;
import javax.faces.context.FacesContext;
import javax.inject.Named;
import javax.servlet.http.HttpSession;

@Named
@RequestScoped
public class LogoutManager {

    public String logout() {
        FacesContext facesContext = FacesContext.getCurrentInstance();
        ExternalContext externalContext =
            facesContext.getExternalContext();
        HttpSession session = (HttpSession)
            externalContext.getSession(true);

        session.invalidate();

        return "index?faces-redirect=true";
    }
}
```

В первых нескольких строках метода `logout` приобретается ссылка на объект сеанса `HttpSession`. Как только она будет получена, останется лишь сделать сеанс недействительным, вызвав метод `invalidate()` этого объекта. В завершение метод производит перенаправление на начальную страницу. Так как с этого момента сеанс недействителен, механизм безопасности автоматически направит пользователя на страницу регистрации.

Теперь мы готовы испытать аутентификацию на основе формы. После создания и развертывания приложения, попробуйте открыть в браузере любую из его страниц, и вы увидите страницу регистрации, как показано на рис. 9.17.



Рис. 9.17. Страница регистрации при использовании аутентификации на основе формы

Если ввести недопустимые учетные данные, автоматически будет выведена страница с общением об ошибке, как показано на рис. 9.18.

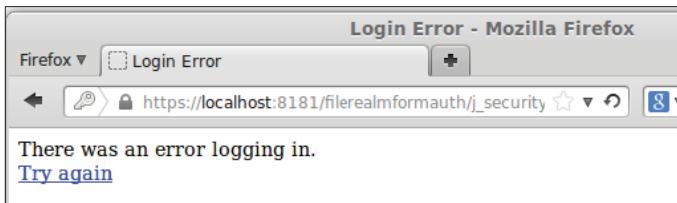


Рис. 9.18. Страница с сообщением об ошибке

Можно щелкнуть на ссылке **Try again** (Повторить попытку), чтобы попробовать еще раз. После ввода допустимых учетных данных будет разрешен доступ к приложению, как показано на рис. 9.19.

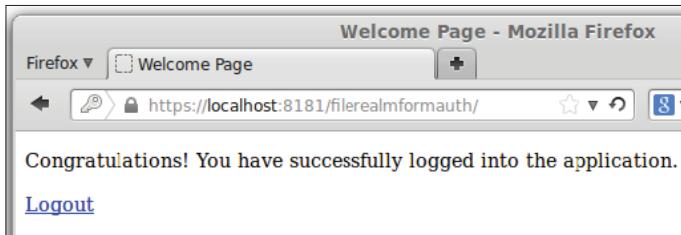


Рис. 9.19. Доступ к приложению разрешен

Как видно на рис. 9.19, мы добавили на страницу ссылку **Logout** (Выход). Щелчок на этой ссылке приведет к вызову метода `logout()` именованного компонента, который, как говорилось выше, просто делает недействительным сеанс. С точки зрения пользователя эта ссылка будет просто выполнять выход из приложения и направлять его на страницу регистрации.

Область сертификата

Для аутентификации в области сертификата используются клиентские сертификаты. Так же как серверные сертификаты, клиентские сертификаты обычно приобретаются в центре сертификации, например Verisign или Thawte. Эти центры сертификации удостоверяют аутентичность владельца сертификата.

Получение сертификата в центре сертификации стоит денег и занимает время. Непрактично будет получать сертификат одного из центров сертификации, только для нужд разработки и/или тестирования нашего приложения. К счастью, есть возможность создавать самоподписанные сертификаты для этих целей.

Создание самоподписанных сертификатов

Чтобы создать самоподписанный сертификат, требуется минимум усилий и умение пользоваться утилитой `keytool`, включенной в **комплект разработчика для Java (Java Development Kit, JDK)**.



Мы коротко рассмотрим только часть возможностей утилиты `keytool`, необходимых для создания и импорта самоподписанных сертификатов в GlassFish и в браузер. За дополнительной информацией об утилите `keytool` обращайтесь по адресу: <http://docs.oracle.com/javase/7/docs/technotes/tools/solaris/keytool.html>.

Сгенерировать самоподписанный сертификат можно следующей командой:

```
keytool -genkey -v -alias selfsignedkey -keyalg RSA -storetype PKCS12 -keystore client_keystore.p12 -storepass wonttellyou -keypass wonttellyou
```

Эта команда предполагает, что каталог с утилитой `keytool` включен в системную переменную `PATH`. Данный инструмент можно найти в подкаталоге `bin`, в каталоге установки JDK.

Замените значения параметров `-storepass` и `-keypass` своим собственным паролем. Оба эти пароля должны быть одинаковыми, чтобы можно было успешно использовать сертификат для аутентификации клиента. Вы вправе выбрать любое значение для параметра `-alias`. Любое значение можно также выбрать для параметра `-keystore`, одна-

ко оно должно заканчиваться на .p12, поскольку эта команда генерирует файл для импортирования в веб-браузер; данный файл не будет распознан, если у него будет другое расширение, отличное от .p12.

После ввода предыдущей команды, keytool запросит некоторую информацию:

```
What is your first and last name?
```

```
[Unknown]: David Heffelfinger
```

```
What is the name of your organizational unit?
```

```
[Unknown]: Book Writing Division
```

```
What is the name of your organization?
```

```
[Unknown]: Enkode Technology, LLC
```

```
What is the name of your City or Locality?
```

```
[Unknown]: Fairfax
```

```
What is the name of your State or Province?
```

```
[Unknown]: Virginia
```

```
What is the two-letter country code for this unit?
```

```
[Unknown]: US
```

```
Is CN=David Heffelfinger, OU=Book Writing Division, O="Enkode Technology, LLC", L=Fairfax, ST=Virginia, C=US correct?
```

```
[no]: y
```

После ввода ответов на все вопросы keytool сгенерирует сертификат. Он будет сохранен в текущем каталоге, в файле, имя которого указано в параметре -keystore (в нашем примере – client_keystore.p12).

Чтобы этот сертификат можно было использовать для аутентификации, его необходимо импортировать в браузер. Данная процедура схожа для всех браузеров, но все же каждый браузер имеет свои отличительные особенности. В Firefox следует открыть диалог **Preferences** (Настройки), затем щелкнуть на значке **Advanced** (Дополнительные) в верхней части диалога и выбрать вкладку **Certificates** (Сертификаты), как показано на рис. 9.20.

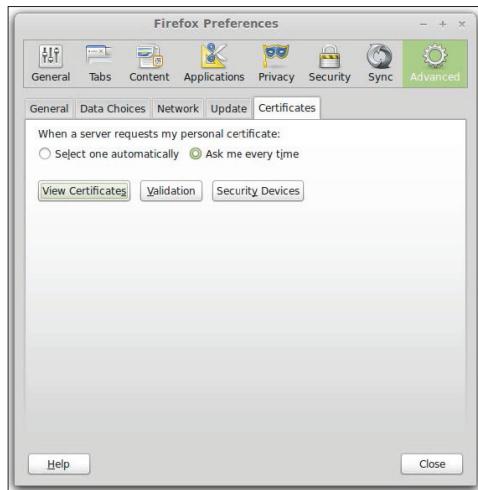


Рис. 9.20. Вкладка Certificates (Сертификаты)

Затем последовательно щелкните на **View Certificates** | **Your Certificates** | **Import** (Просмотр сертификатов | Ваши сертификаты | Импортировать). Найдите свой сертификат в каталоге, где он был создан и выберите его. В этом месте Firefox попросит ввести пароль, использовавшийся для шифрования сертификата. В нашем примере использовался пароль «wonttellyou». После ввода пароля должно появиться окно с подтверждением, что сертификат успешно импортирован, после чего он появится в списке сертификатов, как показано на рис. 9.21.

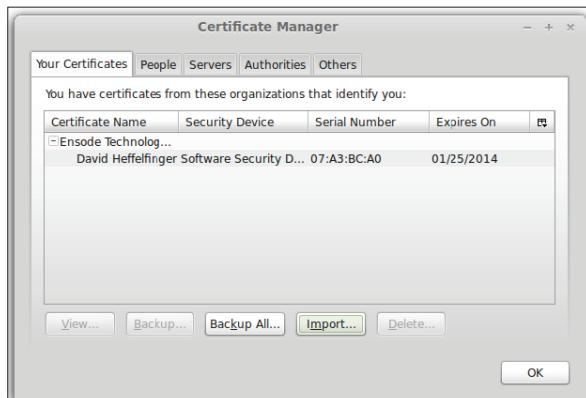


Рис. 9.21. Импортированный сертификат появился в списке

Итак, мы добавили свой сертификат в Firefox, чтобы его можно было использовать в для аутентификации. В других браузерах процедура импортирования сертификатов будет похожей (за подробностями обращайтесь к документации для своего браузера).

Сертификат, созданный на предыдущем шаге, должен быть экспортирован в формат, понятный GlassFish:

```
keytool -export -alias selfsignedkey -keystore client_keystore.p12 -storetype PKCS12 -storepass wonttellyou -rfc -file selfsigned.cer
```

Значения параметров `-alias`, `-keystore` и `-storepass` должны соответствовать значениям в предыдущей команде. Для параметра `-file` можно выбрать любое значение, но рекомендуется завершить его расширением `.cer`.

Поскольку наш сертификат не был выпущен центром сертификации, GlassFish по умолчанию не признает его допустимым. GlassFish знает, каким сертификатам можно доверять, опираясь на подпись центра сертификации. Это обеспечивается сохранением сертификатов различных органов сертификации в хранилище ключей под названием `cacerts.jks`. Это хранилище ключей можно найти в: [каталог установки GlassFish]/glassfish/domains/domain1/config/cacerts.jks.

Соответственно, чтобы GlassFish принял наш сертификат, его нужно импортировать в хранилище ключей `cacerts`. Сделать это можно следующей командой:

```
keytool -import -file selfsigned.cer -keystore [каталог установки GlassFish]/glassfish/domains/domain1/config/cacerts.jks -keypass changeit -storepass changeit
```

После этого `keytool` выведет информацию о сертификате и спросит, хотим ли мы доверять ему:

```
Owner: CN=David Heffelfinger, OU=Book Writing Division, O="Ensoode Technology, LLC", L=Fairfax, ST=Virginia, C=US
```

```
Issuer: CN=David Heffelfinger, OU=Book Writing Division, O="Ensoode Technology, LLC", L=Fairfax, ST=Virginia, C=US
```

```
Serial number: 7a3bca0
```

```
Valid from: Sun Oct 27 17:00:18 EDT 2013 until: Sat Jan 25 16:00:18 EST 2014
```

```
Certificate fingerprints:
```

```
MD5: 46:EA:41:ED:12:8A:EC:CE:8C:BE:F2:49:D5:71:00:ED
```

```
SHA1: 32:C2:D4:20:87:22:95:25:5D:B0:AC:35:43:0D:60:35:94:27:44:58
```

```
SHA256: 8C:2E:56:F4:98:45:AC:46:FD:20:27:38:D2:7D:BF:D8:2D:56:
```

```
D3:91:B7:78:AA:ED:FA:93:30:27:77:7F:F9:03
```

```
Signature algorithm name: SHA256withRSA
```

```
Version: 3
```

```
Extensions:
```

```
#1: ObjectId: 2.5.29.14 Criticality=false
```

```
SubjectKeyIdentifier [
```

```
KeyIdentifier [
```

```
0000: E8 75 1D 12 2F 18 D0 4B E5 84 C4 79 B6 C0 98 80 .u.../.K....y....
```

```
0010: 33 84 E7 C0
```

```
3...
```

```
]
```

```
]
```

```
Trust this certificate? [no]: y
```

```
Certificate was added to keystore
```

После добавления сертификата в хранилище ключей `cacerts.jks`, необходимо перезапустить домен, чтобы изменения вступили в силу.

Фактически здесь мы добавили себя в качестве центра сертификации, которому GlassFish будет доверять. Конечно, этого не нужно делать в промышленной системе.

Значение параметра `-file` должно соответствовать значению этого же параметра при экспортации сертификата.



Обратите внимание, что по умолчанию параметры `-keystorepass` и `-storepass` для хранилища ключей `cacerts.jks` имеют значение `changeit`. Это значение можно изменить следующей командой:

```
[каталог установки GlassFish]/glassfish/bin/asadmin
change-master-password --savemasterpassword=true
```

Эта команда запросит существующий основной пароль и новый основной пароль. Параметр `-savemasterpassword=true` можно опустить; он сохраняет основной пароль в файле с именем `master-password` в корневом каталоге домена. Если не использовать этот аргумент при изменении основного пароля, его придется вводить при каждом запуске домен.

Теперь, когда мы создали самоподписанный сертификат, импортировали его в браузер и установили себя в качестве центра сертификации, которому будет доверять GlassFish, мы готовы разработать приложение, которое будет использовать клиентские сертификаты для аутентификации.

Настройка приложений для использования области сертификата

Поскольку мы используем возможности средств защиты Java EE, нам вообще не нужно изменять программный код, чтобы использовать области безопасности. Все, что мы должны сделать, – изменить настройки приложения в дескрипторах развертывания `web.xml` и `glassfish-web.xml`:

```
<?xml version='1.0' encoding='UTF-8'?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>AllPages</web-resource-name>
            <url-pattern>/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>users</role-name>
        </auth-constraint>
        <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
    <login-config>
        <auth-method>CLIENT-CERT</auth-method>
        <realm-name>certificate</realm-name>
    </login-config>
</web-app>
```

Основное отличие этого дескриптора развертывания `web.xml` от дескриптора в предыдущем разделе заключено в элементе `<login-config>`. В данном случае объявлено, что используется метод авторизации `CLIENT-CERT` и область аутентификации `certificate`. В результате

GlassFish будет запрашивать у браузера клиентский сертификат прежде, чем позволит получить доступ к приложению.

При использовании аутентификации на основе клиентского сертификата запрос всегда должен выполняться через HTTPS. Поэтому в дескриптор развертывания web.xml целесообразно добавить элемент <transport-guarantee> со значением CONFIDENTIAL. Как отмечалось в предыдущем разделе, этим достигается эффект переадресации любых HTTP-запросов в порт HTTPS. Если не добавить это значение в дескриптор, любые запросы через порт HTTP перестанут работать, поскольку аутентификация на основе клиентского сертификата не может быть выполнена по протоколу HTTP.

Обратите внимание: добавив элемент <role-name> с ролью users в элемент <auth-constraint>, вложенный в элемент <security-constraint> в дескрипторе развертывания web.xml, мы объявили, что только пользователи с ролью users смогут получить доступ к любой странице в приложении. Чтобы предоставить доступ авторизованным пользователям, мы должны добавить их в данную роль. Это делается в дескрипторе развертывания glassfish-web.xml:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE glassfish-web-app PUBLIC "-//GlassFish.org//DTD
    GlassFish Application Server 3.1 Servlet 3.0//EN"
    "http://glassfish.org/dtds/glassfish-web-app_3_0-1.dtd">
<glassfish-web-app error-url="">
    <context-root>/certificaterealm</context-root>
    <security-role-mapping>
        <role-name>user</role-name>
        <principal-name>CN=David Heffelfinger, OU=Book Writing
            Division, O="Ensoe Technology, LLC", L=Fairfax,
            ST=Virginia, C=US</principal-name>
    </security-role-mapping>
    <class-loader delegate="true"/>
</glassfish-web-app>
```

Такое присваивание выполняется путем отображения принципала (участника) на роль в элементе <security-role-mapping> дескриптора развертывания glassfish-web.xml. Его подэлемент <role-name> должен содержать имя роли, а подэлемент <principal-name> – имя пользователя, которое берется из сертификата.

Если Вы сомневаетесь в том, какое имя использовать, это имя можно получить из сертификата с помощью утилиты keytool:

```
keytool -printcert -file selfsigned.cer
```

```
Owner: CN=David Heffelfinger, OU=Book Writing Division, O="Ensoe
```

```
Technology, LLC", L=Fairfax, ST=Virginia, C=US

Issuer: CN=David Heffelfinger, OU=Book Writing Division, O="Ensoode
Technology, LLC", L=Fairfax, ST=Virginia, C=US

Serial number: 7a3bca0

Valid from: Sun Oct 27 17:00:18 EDT 2013 until: Sat Jan 25 16:00:18 EST
2014

Certificate fingerprints:

MD5: 46:EA:41:ED:12:8A:EC:CE:8C:BE:F2:49:D5:71:00:ED

SHA1: 32:C2:D4:20:87:22:95:25:5D:B0:AC:35:43:0D:60:35:94:27:44:58

SHA256: 8C:2E:56:F4:98:45:AC:46:FD:20:27:38:D2:7D:BF:D8:2D:56
:D3:91:B7:78:AA:ED:FA:93:30:27:77:7F:F9:03

Signature algorithm name: SHA256withRSA

Version: 3

Extensions:

#1: ObjectId: 2.5.29.14 Criticality=false

SubjectKeyIdentifier [
    KeyIdentifier [
        0000: E8 75 1D 12 2F 18 D0 4B E5 84 C4 79 B6 C0 98 80 .u.../.K...y....
        0010: 33 84 E7 C0                                         3...
    ]
]
]
```

Значение для элемента <principal-name> находится в строке, следующей за строкой `Owner:`. Обратите внимание, что значение <principal-name> должно быть в одной строке с открывающим и закрывающим тегами (<principal-name> и </principal-name>). Если перед значением или после него будут присутствовать символы перевода строки или возврата каретки, они будут интерпретироваться как являющиеся частью значения и проверка перестанет работать.

Поскольку у нашего приложения всего один пользователь и одна роль, мы готовы развернуть его. Если бы у нас было больше пользователей,

телей, мы должны были бы добавить в дескриптор `glassfish-web.xml` дополнительные элементы `<security-role-mapping>`; по крайней мере по одному элементу на пользователя. Если бы у нас были пользователи, обладающие несколькими ролями, нам следовало бы добавить элемент `<security-role-mapping>` для каждой роли, которой принадлежит пользователь, используя значение `<principal-name>`, соответствующее сертификату пользователя.

Теперь можно протестировать приложение. После его развертывания и попытки открыть в браузере любую страницу приложения, должен появиться диалог, как показано на рис. 9.22 (предполагается, что браузер не был настроен на передачу сертификата по умолчанию при каждом его запросе сервером).

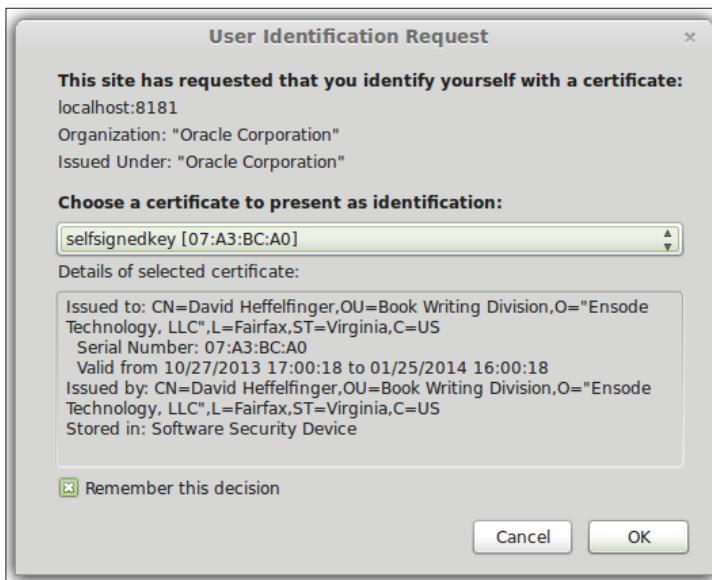


Рис. 9.22. Диалог выбора сертификата для передачи серверу

После щелчка на кнопке **OK** нам будет разрешено получить доступ к приложению (см. рис. 9.23).

Прежде чем разрешить доступ к приложению, GlassFish проверит центр сертификации, который выпустил сертификат и проверяет эти данные по списку доверенных центров сертификации. В предоставленном нами самоподписанном сертификате владелец сертификата и центр сертификации – это одно и то же, поскольку мы добавили себя как доверяемый орган, импортируя свой самоподписанный сертифи-

кат в хранилище ключей `cacerts.jks`. GlassFish признает центр сертификации допустимым; затем получит основное имя из сертификата и сравнил его с записями в дескрипторе развертывания приложения `glassfish-web.xml`. Поскольку мы добавили себя в этот дескриптор и назначили себе допустимую роль, нам будет разрешен доступ к приложению.

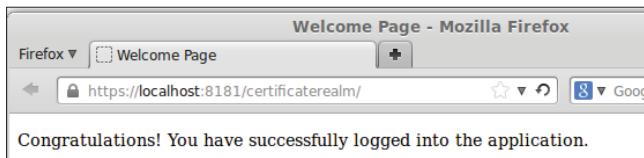


Рис. 9.23. Доступ к приложению открыт

Определение дополнительных областей

В дополнение к трем предварительно настроенным областям безопасности, обсуждавшимся в предыдущем разделе, можно создать дополнительные области для нужд аутентификации в приложении. Можно создать области, которые ведут себя так же, как область администратора или область файла, а можно создать области, которые ведут себя как область сертификата. Наконец, можно создать области, которые используют другие методы аутентификации. Например, можно организовать аутентификацию пользователей через базу данных LDAP или через реляционную базу данных. Если GlassFish устанавливается на сервере Solaris, можно использовать аутентификацию Solaris. Если же ни один из этих механизмов аутентификации не соответствует нашим потребностям, мы вправе реализовать собственный.

Определение дополнительных областей файла

В консоли администрирования распахните узел **Configuration** (Конфигурация), а затем последовательно узлы **server-config** и **Security** (Безопасность). Потом щелкните на узле **Realms** (Области) и на кнопке **New...** (Создать) в основной области веб-консоли. После этого должна открыться форма создания области, как показано на рис. 9.24.

Чтобы создать дополнительную область, нужно ввести ее уникальное имя в поле **Name** (Имя), выбрать пункт `com.sun.enterprise.security.auth.realm.file.FileRealm` в раскрывающемся списке **Class Name** (Имя класса) и ввести значения в поля **JAAS Context** (Контекст JAAS) и **Key File** (Файл ключей). Значение в поле **Key File**

(Файл ключей) должно быть абсолютным путем к файлу, где будет храниться пользовательская информация. Для областей файла в поле **JAAS Context** (Контекст JAAS) всегда следует вводить fileRealm.

The screenshot shows the 'New Realm' configuration dialog. At the top right are 'OK' and 'Cancel' buttons. Below them is a note: '* Indicates required field'. The main section is titled 'Configuration Name:' with the value 'server-config'. Under 'Properties specific to this Class', there are fields for 'Name:' (set to 'newFileRealm'), 'Class Name:' (set to 'com.sun.enterprise.security.auth.realm.file.FileRealm'), and 'Additional Properties (0)' which is currently empty. At the bottom right of the dialog are 'OK' and 'Cancel' buttons.

Рис. 9.24. Форма создания области

После ввода необходимой информации можно щелкнуть на кнопке **OK**, и новая область будет создана. Ее можно использовать ее точно так же, как предопределенную область файла. Для аутентификации через эту новую область, ее имя необходимо указать в виде значения элемента `<realm-name>` в дескрипторе развертывания приложения `web.xml`.

Как вариант, пользовательскую область файла можно добавить из командной строки, с помощью утилиты `asadmin`:

```
asadmin create-auth-realm --classname com.sun.enterprise.
security.auth.realm.file.FileRealm --property file=/home/heffel/
additionalFileRealmKeyFile:jaas-context=fileRealm newFileRealm
```

Аргумент `create-auth-realm` сообщает утилите `asadmin`, что требуется создать новую область безопасности. Значение параметра `--classname` соответствует имени класса области безопасности. Об-

ратите внимание, что оно совпадет со значением, выбранным нами ранее в веб-консоли (рис. 9.24). Параметр `--property` позволяет нам передавать свойства и их значения. В этом параметре должен передаваться список свойств и их значений, разделенных двоеточием (`:`). В последнем параметре этой команды передается имя области безопасности.



Несмотря на простоту создания области безопасности через веб-консоль, возможность выполнить эту процедуру с помощью утилиты `asadmin` имеет свои преимущества – ее можно использовать в сценариях и с их помощью легко настраивать множество экземпляров GlassFish.

Определение дополнительных областей сертификата

Чтобы определить дополнительную область сертификата, нужно просто ввести ее имя в поле **Name** (Имя) и выбрать `com.sun.enterprise.security.auth.realm.certificate.CertificateRealm` в списке **Class Name** (Имя класса), как показано на рис. 9.25, а затем щелкнуть на кнопке **OK**.

New Realm

Create a new security (authentication) realm. Valid realm types are PAM, OSGi, File, Certificate, LDAP, JDBC, Digest, Oracle Solaris, and Custom.

* Indicates required field

Configuration Name: server-config

Name: *

Class Name: com.sun.enterprise.security.auth.realm.certificate.CertificateRealm

Choose a realm class name from the drop-down list or specify a custom class

Properties specific to this Class

Assign Groups:

Comma-separated list of group names

Additional Properties (0)

Add Property	Delete Properties		
Select	Name	Value	Description

No items found.

Рис. 9.25. Создание пользовательской области сертификата

Для приложений, где предполагается использовать эту новую область для аутентификации, следует указать ее имя в элементе `<realm-name>`, в дескрипторе развертывания `web.xml`, и значение `CLIENT-CERT` в элементе `<auth-method>`. При этом должны быть созданы и настроены клиентские сертификаты, как было показано в разделе «Настройка приложений для использования области сертификата».

Пользовательскую область сертификата можно также создать с помощью утилиты `asadmin`:

```
asadmin create-auth-realm --classname com.sun.enterprise.security.auth.realm.certificate.CertificateRealm newCertificateRealm
```

В данном случае не требуется передавать дополнительных свойств, как при создании пользовательской области файла. Нужно лишь передать соответствующее значение в параметре `--classname` и указать имя новой области безопасности.

Определение области LDAP

Мы легко можем создать область для аутентификации через базу данных **LDAP (Lightweight Directory Access Protocol – облегченный протокол доступа к каталогам)**. Для этого нужно, в дополнение к очевидному шагу ввода имени области, выбрать `com.sun.enterprise.security.auth.realm.ldap.LDAPRealm` в списке **Class Name** (Имя класса), как показано на рис. 9.26.

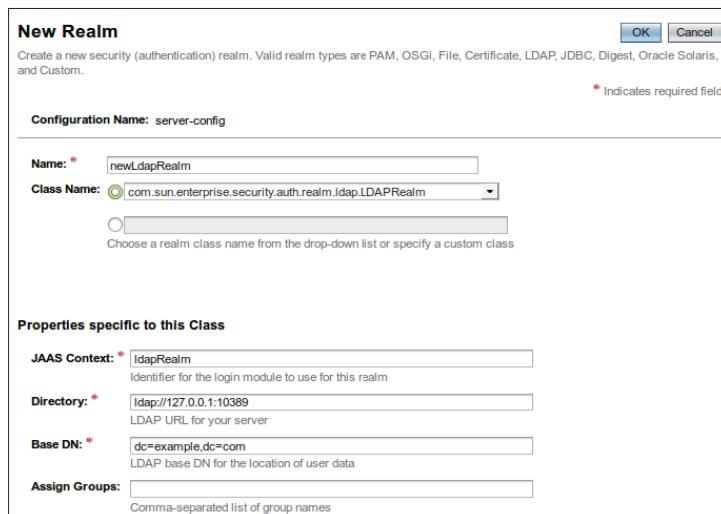


Рис. 9.26. Создание области LDAP



На момент написания этой книги в GlassFish имелась проблема, препятствовавшая созданию пользовательской области безопасности LDAP через веб-консоль администрирования. В этом разделе описывается, что должно происходить, а не что происходит на самом деле. Надеемся, что когда вы будете читать эти строки, проблема будет исправлена.

Однако, процедура добавления области из командной строки, демонстрируемая далее в этом разделе, работает именно так, как описывается.

После создания области LDAP приложения могут использовать ее для аутентификации через базу данных LDAP. Имя области должно быть указано в элементе `<realm-name>`, в дескрипторе развертывания `web.xml`. В элементе `<auth-method>` должно быть указано значение `BASIC` или `FORM`.

Пользователи и роли в базе данных LDAP можно отобразить в группы, в дескрипторе развертывания `glassfish-web.xml`, с использованием элементов `<principal-name>`, `<role-name>` и `<group-name>`, как было показано выше в этой главе.

Создать область LDAP из командной строки можно следующей командой:

```
asadmin create-auth-realm --classname com.sun.enterprise.security.auth.realm.ldap.LDAPRealm --property "jaas-context=ldapRealm:directory=ldap://127.0.0.1:1389:base-dn=dc\=ensode,dc\=com" newLdapRealm
```

Обратите внимание, что в данном случае значение параметра `--property` заключено в кавычки. Это необходимо, чтобы в его значение не были включены некоторые нежелательные символы, такие как двоеточия и знаки равенства. Во избежание подобной ошибки мы просто предваряем их обратным слешем (\).

Определение области Solaris

Если GlassFish устанавливается на сервер Solaris, появляется возможность использовать дополнительный механизм аутентификации операционной системы через область Solaris. Для этого типа области не требуется указывать никаких специфических свойств. Все, что мы должны сделать для ее создания, – указать имя, выбрать `com.sun.enterprise.security.auth.realm.solaris.SolarisRealm` в списке **Class Name** (Имя класса), и ввести значение `solarisRealm` в поле **JAAS Context** (Контекст JAAS), как показано на рис. 9.27.

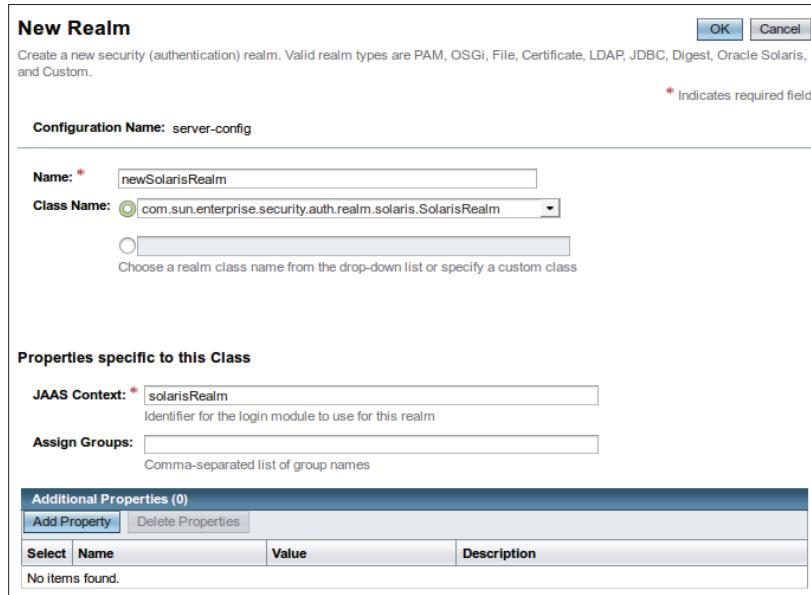


Рис. 9.27. Создание области Solaris

После добавления области приложения смогут выполнять аутентификацию через нее с использованием стандартной аутентификации или аутентификации на основе формы. Группы и пользователи операционной системы могут быть отображены в роли приложения, определенные в дескрипторе развертывания `web.xml`, с помощью элементов `<principal-name>` и `<role-name>`, а также элемента `<group-name>` в дескрипторе развертывания `glassfish-web.xml`.

Область Solaris можно также создать из командной строки:

```
asadmin create-auth-realm --classname com.sun.enterprise.
security.auth.realm.solaris.SolarisRealm --property jaas-context=
solarisRealm newSolarisRealm
```

Определение области JDBC

Еще один тип области, которую можно создать, – область JDBC. Для аутентификации пользователей этот тип области использует информацию в таблицах базы данных.

Чтобы показать, как можно аутентифицировать пользователей через области JDBC, создадим базу данных, содержащую информацию о пользователях (см. рис. 9.28).

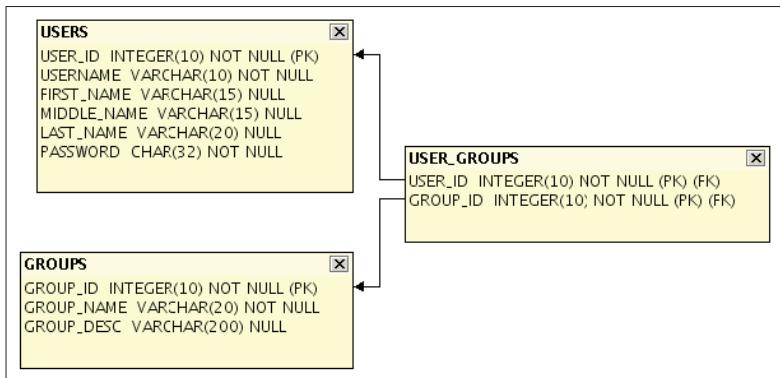


Рис. 9.28. Структура базы данных с информацией о пользователях

База данных содержит три таблицы. Таблица `USERS` хранит информацию о пользователях, а таблица `GROUPS` – о группах. Для реализации отношения «многие ко многим» между таблицами `USERS` и `GROUPS` необходимо добавить объединяющую таблицу, чтобы сохранить нормализацию данных. Имя этой таблицы – `USER_GROUPS`.

Обратите внимание, что столбец `PASSWORD` в таблице `USERS` имеет тип `CHAR(32)`. Причина, почему выбран этот тип вместо `VARCHAR`, заключается в том, что область JDBC ожидает, что по умолчанию пароли будут зашифрованы с применением алгоритма хэширования MD5, а эти хэши всегда имеют длину 32 символа.

Пароли легко можно зашифровать в ожидаемый формат с помощью класса `java.security.MessageDigest`, включенного в JDK. Следующий пример принимает пароль в текстовом виде и создает из него зашифрованный хеш MD5:

```

package net.ensode.glassfishbook;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class EncryptPassword {

    public static String encryptPassword(String password)
        throws NoSuchAlgorithmException {
        MessageDigest messageDigest =
            MessageDigest.getInstance("MD5");

        byte[] bs;

        messageDigest.reset();
    }
}
  
```

```
bs = messageDigest.digest(password.getBytes());  
  
StringBuilder stringBuilder = new StringBuilder();  
  
// шестнадцатиричный код дайджеста  
for (int i = 0; i < bs.length; i++) {  
    String hexVal = Integer.toHexString(0xFF & bs[i]);  
    if (hexVal.length() == 1) {  
        stringBuilder.append("0");  
    }  
    stringBuilder.append(hexVal);  
}  
  
return stringBuilder.toString();  
}  
  
public static void main(String[] args) {  
    String encryptedPassword = null;  
  
    try {  
        if (args.length == 0) {  
            System.err.println("Usage: java "+  
                "net.ensode.glassfishbook.EncryptPassword "+  
                "cleartext");  
        } else {  
            encryptedPassword = encryptPassword(args[0]);  
            System.out.println(encryptedPassword);  
        }  
    } catch (NoSuchAlgorithmException e) {  
        e.printStackTrace();  
    }  
}
```

Основой этого класса является его метод `encryptPassword()`. Он принимает текстовую строку и получает ее дайджест (отпечаток), используя алгоритм MD5 с помощью метода `digest()` экземпляра `java.security.MessageDigest`. Затем кодирует дайджест серией шестнадцатиричных цифр. Такое кодирование необходимо, поскольку GlassFish по умолчанию ожидает, что MD5 преобразует пароль в шестнадцатиричный код.

При использовании областей JDBC пользователи GlassFish и группы не добавляются в области через консоль GlassFish. Вместо этого они добавляются путем вставки данных в соответствующие таблицы.

После создания базы данных с учетными данными пользователей можно приступать к созданию новой области JDBC.

Чтобы создать область JDBC, нужно указав имя в поле **Name (Имя)**, в форме создания области, а затем выбрать `com.sun.enterprise.security.auth.realm.jdbc.JDBCRealm` в списке **Class Name (Имя класса)**, как показано на рис. 9.29.



Рис. 9.29. Создание области JDBC

При создании области JDBC требуется настроить множество дополнительных свойств, как показано на рис. 9.30.

Properties specific to this Class	
JAAS Context: *	<input type="text" value="jdbcRealm"/> Identifier for the login module to use for this realm
JNDI: *	<input type="text" value="jdbcuserauth"/> JNDI name of the JDBC resource used by this realm
User Table: *	<input type="text" value="V_USER_ROLE"/> Name of the database table that contains the list of authorized users for this realm
User Name Column: *	<input type="text" value="USERNAME"/> Name of the column in the user table that contains the list of user names
Password Column: *	<input type="text" value="PASSWORD"/> Name of the column in the user table that contains the user passwords
Group Table: *	<input type="text" value="V_USER_ROLE"/> Name of the database table that contains the list of groups for this realm
Group Table User Name Column:	<input type="text" value="USERNAME"/> Name of the column in the user group table that contains the list of groups for this realm
Group Name Column: *	<input type="text" value="GROUP_NAME"/> Name of the column in the group table that contains the list of group names
Password Encryption Algorithm: *	<input type="text" value="MD5"/> MD5 This denotes the algorithm for encrypting the passwords in the database. It is a security risk to leave this field empty.
Assign Groups:	<input type="text"/> Comma-separated list of group names
Database User:	<input type="text"/> Specify the database user name in the realm instead of the JDBC connection pool
Database Password:	<input type="text"/> Specify the database password in the realm instead of the JDBC connection pool
Digest Algorithm:	<input type="text" value="MD5"/> Digest algorithm (default is SHA-256); note that the default was MD5 in GlassFish versions prior to 3.1
Encoding:	<input type="text"/> Encoding (allowed values are Hex and Base64)
Charset:	<input type="text"/> Character set for the digest algorithm

Рис. 9.30. Настройка дополнительных при создании области JDBC

В поле **JAAS Context** (Контекст JAAS) нужно указать значение `jdbcRealm`. В поле **JNDI** – JNDI-имя источника данных, соответствующего базе данных, где хранится информация о пользователях и группах. В поле **User Table** (Таблица пользователей) – имя таблицы с именами и паролями пользователей.



*Обратите внимание, что на рис. 9.30 в этом поле указано имя `V_USER_ROLE`. Это – представление в базе данных, которое содержит информацию о пользователях и группах. Мы не использовали таблицу `USERS` напрямую, поскольку GlassFish предполагает, что обе таблицы – с информацией о пользователях и группах – содержат столбец с именем пользователя. Вследствие этого возникает дублирование данных. Чтобы этого избежать, мы создали представление, которое можно использовать в качестве значения обоих свойств: **User Table** (Таблица пользователей) и **Group Table** (Таблица групп).*

Свойство **User Name Column** (Столбец с именами пользователей) должно содержать имя столбца в **User Table** (Таблица пользователей), содержащего имена пользователей. Свойство **Password Column** (Столбец пароля) должно содержать имя столбца в **User Table** (Таблица пользователей), содержащего пароли пользователей. Свойство **Group Table** (Таблица групп) должно содержать имя таблицы с информацией о группах пользователей. Свойство **Group Name Column** (Столбец с именами групп) должно содержать имя столбца в **Group Table** (Таблица групп), содержащего имена групп.

Все другие свойства являются необязательными и в большинстве случаев остаются незаполненными. Особенно интересно свойство **Digest Algorithm** (Алгоритм дайджеста). Оно позволяет указать алгоритм шифрования паролей пользователей. Допустимые значения этого свойства включают все алгоритмы, поддерживаемые JDK, а именно: MD2, MD5, SHA-1, SHA-256, SHA-384 и SHA-512. Кроме того, если требуется запретить хранение пользовательских паролей в открытом виде, можно указать в данном свойстве значение `none` (ничтожной).



Алгоритмы MD2, MD5 и SHA-1 обеспечивают не очень высокий уровень безопасности, и в большинстве случаев их следует избегать.

После определения области JDBC необходимо настроить дескрипторы развертывания приложения `web.xml` и `glassfish-web.xml`. Использование области на основе JDBC для аутентификации и автори-

зации, настраивается точно так же, как использование любой другой области.

В дополнение к заявлению, что мы будем использовать область на основе JDBC для аутентификации и авторизации, как и при использовании других типов областей, мы должны отобразить роли, определенные в дескрипторе развертывания `web.xml`, в названия группы безопасности. Это выполняется в дескрипторе развертывания `glassfish-web.xml`.

Область JDBC можно также создать из командной строки:

```
asadmin create-auth-realm --classname com.sun.enterprise.security.ee.auth.realm.jdbc.JDBCRealm --property jaas-context=jdbcRealm:datasource-jndi=jdbc/_UserAuthPool:user-table=V_USER_ROLE:user-name-column=USERNAME:passwordcolumn=PASSWORD:group-table=V_USER_ROLE:group-name-column=GROUP_NAME newJdbcRealm
```

Определение пользовательских областей

Предопределенные типы областей с успехом могут использовать-ся в подавляющем большинстве случаев, тем не менее, GlassFish предоставляет возможность создавать новые типы областей, если имеющиеся не удовлетворяют нашим потребностям. Решение этой задачи подразумевает реализацию пользовательских классов `Realm` и `LoginModule`. Обсудим сначала пользовательский класс области:

```
package net.ensode.glassfishbook;

import java.util.Enumeration;
import java.util.Vector;

import com.sun.enterprise.security.auth.realm.IASRealm;
import com.sun.enterprise.security.auth.realm.InvalidOperationException;
import com.sun.enterprise.security.auth.realm.NoSuchUserException;

public class SimpleRealm extends IASRealm {

    @Override
    public Enumeration getGroupNames(String userName)
        throws InvalidOperationException, NoSuchUserException {
        Vector vector = new Vector();

        vector.add("appuser");
        vector.add("appadmin");

        return vector.elements();
    }

    @Override
```

```
public String getAuthType() {
    return "simple";
}

@Override
public String getJAASContext() {
    return "simpleRealm";
}

public boolean loginUser(String userName, String password) {
    boolean loginSuccessful = false;
    if ("glassfish".equals(userName) && "secret".equals(password))
    {
        loginSuccessful = true;
    }
    return loginSuccessful;
}
}
```

Пользовательский класс области должен наследовать `com.sun.enterprise.security.auth.realm.IASRealm`. Этот класс можно найти в файле `security.jar`; следовательно, этот JAR-файл должен быть добавлен в CLASSPATH, чтобы компиляция нашего класса `Realm` прошла успешно.



Файл security.jar можно найти в [каталог установки GlassFish]/glassfish/modules.

При использовании инструментов управления зависимостями Maven или Ivy этот JAR-файл можно найти в следующем репозитории: <http://download.java.net/maven/glassfish>.

Идентификатор группы: `org.glassfish.security`; **идентификатором артефакта:** `security`.

Класс должен переопределить метод `getGroupNames()`. Этот метод принимает единственный строковый параметр и возвращает `Enumeration`. Строковый параметр содержит имя пользователя, пытающегося зарегистрироваться в области. Возвращаемое значение типа `Enumeration` (перечисление) должно содержать коллекцию строк со списком групп, которым принадлежит пользователь. В нашем простом примере мы просто жестко «запилили» список групп в код, однако в реальных приложениях этот список должен извлекаться из некоторого хранилища (база данных, файл и т. д.).

Следующий метод, который должен быть переопределен в классе области, – `getAuthType()`. Он должен вернуть строку с описанием типа аутентификации, используемого этой областью.

Оба метода — `getGroupNames()` и `getAuthType()` — объявлены в родительском классе `IASRealm` как абстрактные. Хотя метод `getJAASContext()` не является абстрактным, его тоже нужно переопределить, поскольку значение, которое он возвращает, используется сервером приложений для определения типа аутентификации по файлу `login.conf`. По возвращаемому значению этого метода определяется модуль регистрации.

Наконец, класс области должен содержать метод аутентификации пользователя. Мы вольны назвать его как угодно. Кроме того, мы можем использовать столько параметров любого типа, сколько нам нужно. В нашем простом примере значения единственного имени пользователя и пароля жестко «зашиты» в код. И снова, в реальном приложении учетные данные извлекались бы из некоторого хранилища. Этот метод предназначается для вызова из соответствующего класса модуля регистрации:

```
package net.ensode.glassfishbook;

import java.util.Enumeration;
import javax.security.auth.login.LoginException;
import com.sun.appserv.security.AppservPasswordLoginModule;
import com.sun.enterprise.security.auth.realm
    .InvalidOperationException;
import com.sun.enterprise.security.auth.realm.NoSuchUserException;

public class SimpleLoginModule extends AppservPasswordLoginModule {

    @Override
    protected void authenticateUser() throws LoginException {
        Enumeration userGroupsEnum = null;
        String[] userGroupsArray = null;
        SimpleRealm simpleRealm;

        if (!(_currentRealm instanceof SimpleRealm)) {
            throw new LoginException();
        } else {
            simpleRealm = (SimpleRealm) _currentRealm;
        }

        if (simpleRealm.loginUser(_username, _password)) {
            try {
                userGroupsEnum = simpleRealm.getGroupNames(_username);
            } catch (InvalidOperationException e) {
                throw new LoginException(e.getMessage());
            } catch (NoSuchUserException e) {
```

```
        throw new LoginException(e.getMessage());
    }

    userGroupsArray = new String[2];
    int i = 0;

    while (userGroupsEnum.hasMoreElements()) {
        userGroupsArray[i++] = (
            (String)userGroupsEnum.nextElement());
    }
} else {
    throw new LoginException();
}
commitUserAuthentication(userGroupsArray);
}
}
```

Класс модуля регистрации должен наследовать класс `com.sun.appserv.security.AppservPasswordLoginModule`. Этот класс также находится внутри файла `security.jar`. Класс модуля регистрации нужен, только чтобы переопределить единственный метод `authenticateUser()`. Этот метод не принимает параметров и должен вызывать исключение `LoginException`, если аутентификация завершилась неудачей. В родительском классе определена переменная `_currentRealm`; она имеет тип `com.sun.enterprise.security.auth.realm`. Класс `realm` является предком для всех классов областей. Эта переменная инициализируется перед вызовом метода `authenticateUser()`. Класс модуля регистрации должен проверить, что данный класс имеет ожидаемый тип (в нашем примере – `SimpleRealm`), в противном случае – вызвать исключение `LoginException`.

Перед вызовом метода `authenticateUser()` инициализируются две другие переменные, которые определяются в родительском классе, – это `_username` и `_password`. Они содержат учетные данные пользователя, введенные в форме регистрации (при аутентификации на основе формы) или в диалоге (при стандартной аутентификации). В нашем примере эти значения просто передаются классу области, чтобы он мог проверить учетные данные пользователя.

После успешной аутентификации метод `authenticateUser()` должен вызвать метод `commitUserAuthentication()` родительского класса. Этот метод принимает массив строковых объектов с именами групп, которым принадлежит пользователь. В нашем примере он просто вызывает метод `getGroupNames()`, определенный в классе области, и добавляет элементы из коллекции, возвращаемой этим методом, в массив, а затем передает массив методу `commitUserAuthentication()`.

Очевидно, что GlassFish «не знает» о существовании наших классов пользовательской области и модуля регистрации. Мы должны добавить эти классы в CLASSPATH сервера GlassFish. Проще всего сделать это – скопировать JAR-файл с реализацией области и модуля регистрации в каталог [каталог установки GlassFish]/glassfish/domains/domain1/lib.

Последний шаг, которой нужно сделать прежде, чем можно будет аутентифицировать пользователей приложения через нашу область, заключается в добавлении новой области в файл домена login.conf:

```
fileRealm {  
    com.sun.enterprise.security.auth.login.FileLoginModule required;  
};  
  
ldapRealm {  
    com.sun.enterprise.security.auth.login.LDAPLoginModule required;  
};  
  
solarisRealm {  
    com.sun.enterprise.security.auth.login.SolarisLoginModule  
    required;  
};  
  
jdbcRealm {  
    com.sun.enterprise.security.auth.login.JDBCLoginModule required;  
};  
  
jdbcDigestRealm {  
    com.sun.enterprise.security.auth.login.JDBCDigestLoginModule  
    required;  
};  
  
pamRealm {  
    com.sun.enterprise.security.ee.auth.login.PamLoginModule  
    required;  
};  
  
simpleRealm {  
    net.ensode.glassfishbook.SimpleLoginModule required;  
};
```

Значение перед открывающей фигурной скобкой должно соответствовать значению, возвращаемому методом `getJAASContext()` класса области. Именно в этом файле классы области и модули регистрации встречаются друг с другом. После изменения файла домен GlassFish должен быть перезапущен, чтобы изменения вступили в силу.

Теперь все готово к использованию нашей области для аутентификации. Добавим новый тип области, созданной нами, через консоль администрирования GlassFish, как показано на рис. 9.31.

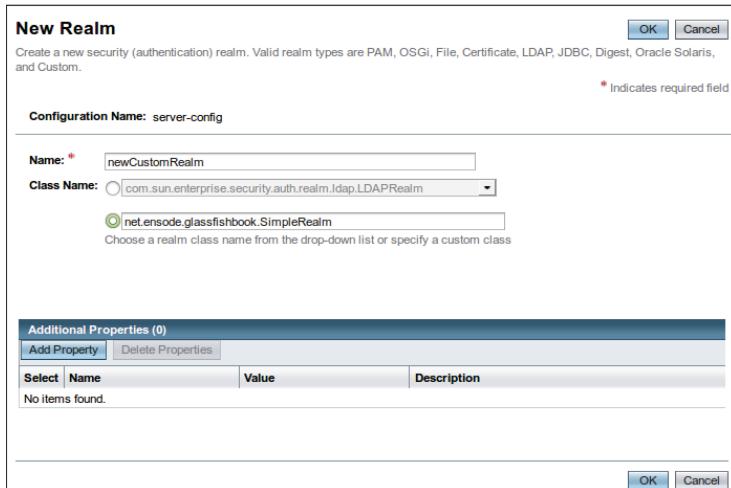


Рис. 9.31. Добавление области нового типа

Чтобы создать область, ей, как обычно, требуется дать имя. Имя класса в поле **Class Name** (Имя класса) нужно не выбирать в раскрывающемся списке, а ввести вручную в текстовое поле под списком. У нашей области не было никаких свойств, поэтому никаких свойств добавлять не нужно. Но если бы это было необходимо, можно было бы щелкнуть на кнопке **Add Property** (Добавить свойство) и ввести имя свойства и значение. Наша область могла бы затем получать свойства путем переопределения метода `init()` родительского класса. Данный метод имеет следующую сигнатуру:

```
protected void init(Properties arg0) throws BadRealmException,  
NoSuchRealmException
```

Экземпляры `java.util.Properties`, которые он принимает в параметре `arg0`, предварительно заполнены значениями, введенными в веб-консоли (напомним: у нашей области нет никаких свойств).

После того ввода соответствующей информации о нашей новой области, ее можно использовать точно так же, как любую из предопределенных областей. Приложения должны указывать ее имя в элементе `<realm-name>` дескриптора развертывания приложения `web.xml`. Ничего необычного на уровне приложения делать не нужно.

Точно так же, как и стандартные области, пользовательские области можно добавлять с помощью утилиты командной строки `asadmin`:

```
asadmin create-auth-realm --classname net.ensode.glassfishbook.  
SimpleRealm newCustomRealm
```

Резюме

В этой главе мы обсудили использование областей GlassFish по умолчанию для аутентификации веб-приложений. Рассмотрели область файла, которая хранит пользовательскую информацию в простом файле, и область сертификата, которая требует клиентских сертификатов для аутентификации пользователей.

Мы выяснили, как создавать дополнительные области, которые ведут себя точно так же, как области по умолчанию, используя классы областей, включенные в GlassFish.

Также речь шла о том, как использовать дополнительные классы областей, включенные в GlassFish для создания областей, осуществляющих аутентификацию через базу данных LDAP и реляционную базу данных, и о том, как создавать области, реализующие дополнительные механизмы аутентификации операционной системы Solaris.

Наконец, мы показали, как создавать собственные классы областей на тот случай, если вам потребуется что-то необычное.



ГЛАВА 10.

Веб-службы JAX-WS

Спецификация Java EE включает в себя JAX-WS API, как одну из своих технологий. JAX-WS – стандартный способ разработки на платформе Java веб-служб, действующих по **простому протоколу доступа к объектам (Simple Object Access Protocol, SOAP)**. Аббревиатура JAX-WS расшифровывается как «Java API for XML-Based Web Services» (Java API для веб-служб XML). JAX-WS – высокоуровневый API; обращение к веб-службам JAX-WS выполняется посредством механизма вызова удаленных процедур. JAX-WS – очень естественный API для разработчиков Java.

Веб-службы (web services) – это прикладные программные интерфейсы (Application Programming Interfaces, API), которые могут быть вызываться удаленно. Веб-службы можно вызывать из клиентов, написанных на любом языке программирования.

В этой главе мы рассмотрим следующие темы:

- ❖ разработка веб-служб с помощью JAX-WS API;
- ❖ разработка клиентов веб- служб с помощью JAX-WS API;
- ❖ добавление вложений в вызовы веб-служб;
- ❖ экспортование EJB как веб-служб;
- ❖ обеспечение безопасности веб-служб.

Разработка веб-служб с использованием JAX-WS API

JAX-WS – это высокоуровневый API, упрощающий разработку веб-служб, действующих по протоколу SOAP. Разработка веб-служб JAX-WS заключается в создании классов с открытыми методами, которые будут экспортirоваться как веб-службы. Класс должен быть декорирован аннотацией `@WebService`. Все открытые методы в классе автоматически экспортirуются как веб-службы; они могут дополнитель-

декорироваться аннотацией `@WebMethod`. Следующий пример иллюстрирует этот процесс:

```
package net.ensode.glassfishbook;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
public class Calculator {

    @WebMethod
    public int add(int first, int second) {
        return first + second;
    }

    @WebMethod
    public int subtract(int first, int second) {
        return first - second;
    }
}
```

Этот класс экспортирует два своих метода как веб-службы. Метод `add()` просто складывает два числа типа `int`, которые он получает в параметрах, и возвращает результат, а метод `subtract()` возвращает разность своих параметров.

Выше говорилось, что класс, реализующий веб-службу, должен декорироваться аннотацией `@WebService`. Любые методы, которые должны экспортироваться представить как веб-службы, могут дополнительно декорироваться аннотацией `@WebMethod`, но это не обязательно; все открытые методы автоматически экспортятся как веб-службы.

Чтобы развернуть веб-службу, нужно упаковать ее в WAR-файл. До версии Java EE 6, все WAR-файлы обязаны были содержать дескриптор развертывания `web.xml` в каталоге `WEB-INF`. Как мы уже говорилось в предыдущих главах, этот дескриптор развертывания стал необязательным в Java EE 6 (и выше) и не требуется для развертывания веб-служб в этой среде.

Если вы решите добавить дескриптор развертывания `web.xml`, вам не придется ничего добавлять в него. Чтобы успешно развернуть веб-службу, достаточно иметь пустой элемент `<web-app>` в дескрипторе развертывания:

```
<?xml version='1.0' encoding='UTF-8'?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="2.5"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema"
```

```

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://
java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
</web-app>

```

После компиляции, упаковки и развертывания этого кода, можно убедиться, что он был развернут успешно. Для этого нужно войти в веб-консоль администрирования сервера GlassFish и распахнуть узел **Applications** (Приложения) в панели навигации слева. В списке этого узла должна находиться только что установленная веб-служба, как показано на рис. 10.1.

Modules and Components (4)					
Module Name	Engines	Component Name	Type	Action	
Simple_Web_Service	[web, webservices]	-----	-----	Launch	
Simple_Web_Service		default	Servlet		
Simple_Web_Service		jsp	Servlet		
Simple_Web_Service		Calculator	Servlet	View Endpoint	

Рис. 10.1. Новая веб-служба была успешно развернута

Обратите внимание на ссылку **View Endpoint** (Посмотреть конечную точку) внизу справа. Щелчок на этой ссылке откроет страницу **Web Service Endpoint Information** (Информация о Конечной точке веб-службы) с некоторой информацией о веб-службе, как показано на рис. 10.2.

Web Service Endpoint Information

View details about a web service endpoint.

Application Name:	Simple_Web_Service
Tester:	/calculatorservice/CalculatorService?Tester
WSDL:	/calculatorservice/CalculatorService?wsdl
Endpoint Name:	Calculator
Service Name:	CalculatorService
Port Name:	CalculatorPort
Deployment Type:	109
Implementation Type:	SERVLET
Implementation Class Name:	net.ensode.glassfishbook.Calculator
Endpoint Address URI:	/calculatorservice/CalculatorService
Namespace:	http://glassfishbook.ensode.net/

Рис. 10.2. Страница с информацией о веб-службе

На рис. 10.2 можно видеть ссылку **Tester** (Тестер); щелчок на ней откроет автоматически сгенерированную страницу, позволяющую протестировать веб-службу, как показано на рис. 10.2.

CalculatorService Web Service Tester

This form will allow you to test your web service implementation ([WSDL File](#))

To invoke an operation, fill the method parameter(s) input boxes and click on the button labeled with the method name.

Methods :

public abstract int net.ensode.glassfishbook.Calculator.add(int,int)
 (,)

public abstract int net.ensode.glassfishbook.Calculator.subtract(int,int)
 (,)

Рис. 10.3. Страница тестирования веб-службы

Чтобы протестировать методы, можно просто ввести ряд параметров в текстовые поля и щелкнуть на соответствующей кнопке. Например, если ввести значения 2 и 3 в текстовых полях рядом с называнием метода add и щелкнуть на кнопке **add**, появится результат, как показано на рис. 10.4.

add Method invocation	
Method parameter(s)	
Type	Value
int	2
int	3
Method returned	
int : "5"	

Рис. 10.4. Результат тестирования метода add()

За кулисами JAX-WS использует протокол SOAP для обмена информацией между клиентами и веб-службами. Прокрутив страницу на рис. 10.4 вниз, можно увидеть запрос и отклик SOAP, сгенерированные тестом, как показано на рис. 10.5.

SOAP Request	
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xml:lang="en-US"><S:Header></S:Header><S:Body><ns2:add xmlns:ns2="http://glassfishbook.ensode.net/"><arg0>2</arg0><arg1>3</arg1></ns2:add></S:Body></S:Envelope>	
SOAP Response	
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xml:lang="en-US"><S:Header></S:Header><S:Body><ns2:addResponse xmlns:ns2="http://glassfishbook.ensode.net/"><return>5</return></ns2:addResponse></S:Body></S:Envelope>	

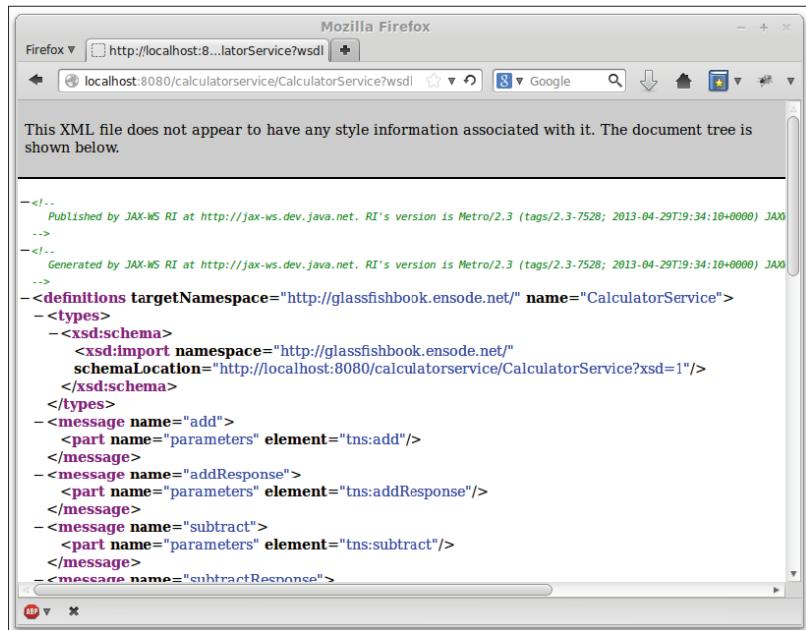
Рис. 10.5. Запрос и отклик SOAP, сгенерированные тестом

Эти запросы SOAP не представляют большого интереса для нас, как разработчиков приложений, поскольку о них автоматически заботится JAX-WS API.

Клиенты веб-службы нуждаются в файле на **языке определения веб-служб (Web Services Definition Language, WSDL)**, чтобы сге-

нериовать выполняемый код, который они будут использовать для вызова веб-службы. WSDL – это стандартный XML-подобный язык определения интерфейсов, на котором описывается функциональность веб-службы.

Файлы WSDL обычно помещаются на веб-сервер и клиенты получают к ним доступ через их URL. Когда развертывается веб-служба, разработанная с использованием JAX-WS, ее описание на языке WSDL генерируется автоматически. Мы можем просмотреть это описание вместе с его URL, щелкнув на ссылке **View WSDL** (Посмотреть WSDL) на странице **Web Service Endpoint Information** (Информация о Конечной точке веб-службы), как показано на рис. 10.6.



The screenshot shows a Mozilla Firefox window with the address bar containing `http://localhost:8080/calculatorservice/CalculatorService?wsdl`. The page content displays the WSDL XML document for the `CalculatorService`. The XML starts with a header indicating it was published by JAX-WS RI at `http://jax-ws.dev.java.net` with version `Metro/2.3 (tags/2.3-7528; 2013-04-29T19:34:10+0000)`. It defines a target namespace `http://glassfishbook.ensode.net/` and includes imports for `parameters` and `subtract` messages. The XML is displayed in a monospaced font within the browser's scrollable area.

```
<!--
Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is Metro/2.3 (tags/2.3-7528; 2013-04-29T19:34:10+0000) JAX
-->
<!--
Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is Metro/2.3 (tags/2.3-7528; 2013-04-29T19:34:10+0000) JAX
-->
<definitions targetNamespace="http://glassfishbook.ensode.net/" name="CalculatorService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://glassfishbook.ensode.net/" schemaLocation="http://localhost:8080/calculatorservice/CalculatorService?xsd=1"/>
    </xsd:schema>
  </types>
  <message name="add">
    <part name="parameters" element="tns:add"/>
  </message>
  <message name="addResponse">
    <part name="parameters" element="tns:addResponse"/>
  </message>
  <message name="subtract">
    <part name="parameters" element="tns:subtract"/>
  </message>
  <message name="subtractResponse">
```

Рис. 10.6. Описание веб-службы на языке WSDL

Обратите внимание на URL в адресной строке браузера; он понадобится нам при разработке клиента веб-службы.

Создание клиента веб-службы

Как уже отмечалось выше, выполняемый код для клиента генерируется на основе WSDL-описания веб-службы. Клиент сможет вызывать этот выполняемый код, чтобы получить доступ к веб-службе.

GlassFish включает утилиту `wsimport`, генерирующую Java-код на основе WSDL. Этую утилиту можно найти в каталоге [каталог установки Glassfish]/glassfish/bin/. Единственным обязательным аргументом `wsimport` является URL описания на языке WSDL, соответствующего веб-службе, например: `wsimport http://localhost:8080/calculatorservice/CalculatorService?wsdl`.

Эта команда сгенерирует много скомпилированных классов Java, с помощью которых клиенты смогут получить доступ к веб-службе. В данном случае это следующие классы:

- `Add.class;`
- `AddResponse.class;`
- `Calculator.class;`
- `CalculatorService.class;`
- `ObjectFactory.class;`
- `package-info.class;`
- `Subtract.class;`
- `SubtractResponse.class.`



Сохраните сгенерированный исходный код. По умолчанию файлы с исходным кодом сгенерированных классов автоматически удаляются; но их можно сохранить, передав параметр `-keep` утилите `wsimport`.

Эти классы следует добавить в CLASSPATH клиента, чтобы сделать их доступными для клиента. В дополнение к инструменту командной строки, GlassFish включает пользовательскую задачу для инструмента Ant, генерирующую код из WSDL. Следующий сценарий сборки Ant поясняет его использование:

```
<project name="calculatorserviceclient" default="wsimport"
         basedir=".">
  <target name="wsimport">
    <taskdef name="wsimport"
             classname="com.sun.tools.ws.ant.WsImport">
      <classpath path=
                  "/opt/glassfish-4.0/glassfish/modules/webservices-osgi.jar"/>
      <classpath path=
                  "/opt/glassfish-4.0/glassfish/modules/jaxb-osgi.jar"/>
      <classpath path=
                  "/opt/glassfish-4.0/glassfish/lib/javaee.jar"/>
    </taskdef>
    <wsimport wsdl="http://localhost:8080/
               calculatorservice/CalculatorService?wsdl"
              endorsed="true"/>
```

```
</target>
</project>
```

Этот пример демонстрирует самый минимальный сценарий сборки Ant, который всего лишь иллюстрирует, как настроить пользовательскую цель `<wsimport>` Ant. В действительности сценарий сборки проекта Ant может иметь множество других целей для компиляции, сборки WAR-файла и т. д.

Поскольку `<wsimport>` является пользовательской, а не стандартной целью Ant, в наш сценарий сборки нужно добавить элемент `<taskdef>`. Нам следует также определить атрибуты `name` и `classname`, как показано в приведенном примере. Кроме того, нижеперечисленные JAR-файлы необходимо добавить в CLASSPATH задачи с помощью вложенного элемента `<classpath>`:

- `webservices-osgi.jar`;
- `jaxb-osgi.jar`;
- `javaee.jar`.

Файлы `webservices-osgi.jar` и `jaxb-osgi.jar` находятся в каталоге [каталог установки Glassfish]/glassfish/modules. Файл `javaee.jar` содержит весь Java EE API и находится в каталоге в [каталог установки Glassfish]/glassfish/lib.

После определения задачи `<wsimport>` в элементе `<taskdef>` мы готовы ее использовать. Не забудьте указать местоположение WSDL в атрибуте `wsdl`. Сразу по выполнении задачи будет сгенерирован код Java, необходимый для доступа к веб-службе.

JDK 1.6 поставляется совместно с JAX-WS 2.1. Если вы используете эту версию JDK, сообщите Ant, что следует использовать JAX-WS 2.2 API, поставляемый с сервером GlassFish. Это легко сделать, установив атрибут `xendorsed` задачи `wsimport` в значение `true` (истина).

Читатели, использующие для сборки своих проектов инструмент Maven, могут воспользоваться преимуществами расширения `AntRun`, чтобы выполнить цель `wsimport` Ant и сгенерировать программный код. Этот подход показан в следующем файле `pom.xml`:

```
<?xml version='1.0' encoding='UTF-8'?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
          http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>net.enseode.glassfishbook</groupId>
    <artifactId>calculatorserviceclient</artifactId>
```

```
<packaging>jar</packaging>
<name>Simple Web Service Client</name>
<version>1.0</version>
<url>http://maven.apache.org</url>
<repositories>
    <repository>
        <id>maven2-repository.dev.java.net</id>
        <name>Java.net Repository for Maven 2</name>
        <url>http://download.java.net/maven/2/</url>
    </repository>
</repositories>
<dependencies>
    <dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-api</artifactId>
        <version>6.0</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
<build>
    <finalName>calculatorserviceclient</finalName>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-antrun-plugin</artifactId>
            <executions>
                <execution>
                    <phase>generate-sources</phase>
                    <configuration>
                        <tasks>
                            <property name="target.dir" value="target" />
                            <delete dir="${target.dir}/classes/com/
                                testapp/ws/client" />
                            <delete dir="${target.dir}/generated-sources/
                                main/java/com/testapp/ws/client" />
                            <mkdir dir="${target.dir}/classes" />
                            <mkdir dir="${target.dir}/generated-sources/
                                main/java" />
                        </tasks>
                    </configuration>
                    <taskdef name="wsimport"
                        classname="com.sun.tools.ws.ant.WsImport">
                        <classpath path="/home/heffel/sgesv3/
                            glassfish/modules/webservices-osgi.jar" />
                        <classpath path="/home/heffel/sges-v3/
                            glassfish/modules/jaxb-osgi.jar" />
                        <classpath path="/home/heffel/sges-v3/
                            glassfish/lib/javaee.jar" />
                    </taskdef>
                    <wsimport wsdl="http://localhost:8080/
                        calculatorservice/CalculatorService?wsdl"
                        destdir="${target.dir}/classes" verbose="true"
                        keep="true" sourceDestDir="${target.dir}/
                        generated-sources/main/java" xendorsed="true" />
                </execution>
            </executions>
        </plugin>
    </plugins>

```

```

        </tasks>
        <sourceRoot>${project.build.directory} /
            generated-sources/main/java</sourceRoot>
    </configuration>
    <goals>
        <goal>run</goal>
    </goals>
    </execution>
</executions>
</plugin>
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <configuration>
        <archive>
            <manifest>
                <mainClass>net.ensode.glassfishbook
                    .CalculatorServiceClient</mainClass>
                <addClasspath>true</addClasspath>
            </manifest>
        </archive>
    </configuration>
</plugin>
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
        <source>1.6</source>
        <target>1.6</target>
    </configuration>
</plugin>
</plugins>
</build>
</project>

```

Внутрь тега `<configuration>`, в файле `pom.xml`, соответствующего расширению `AntRun`, мы помещаем любые задачи Ant, которые нужно выполнить. Неудивительно, что тело этого тега в нашем примере почти идентично содержимому файла сборки Ant, представленного выше.

Теперь, когда мы знаем, как сгенерировать выполняемый код с помощью Ant или Maven, можно приступать к созданию простого клиента для работы с веб-службой:

```

package net.ensode.glassfishbook;

import javax.xml.ws.WebServiceRef;

public class CalculatorServiceClient {
    @WebServiceRef(wsdlLocation =

```

```
"http://localhost:8080/calculatorservice/CalculatorService?wsdl")
private static CalculatorService calculatorService;

public void calculate() {
    Calculator calculator =
        calculatorService.getCalculatorPort();

    System.out.println("1 + 2 = "
        + calculator.add(1, 2));
    System.out.println("1 - 2 = "
        + calculator.subtract(1, 2));
}

public static void main(String[] args) {
    new CalculatorServiceClient().calculate();
}
```

Аннотация `@WebServiceRef` внедряет экземпляр веб-службы в клиентское приложение. Ее атрибут `wsdlLocation` определяет URL описания WSDL используемой веб-службы.

Обратите внимание, что веб-служба является экземпляром класса `CalculatorService`; этот класс был создан вызовом утилиты `wsimport`. Утилита `wsimport` всегда генерирует класс с именем, идентичным имени реализованного класса, плюс окончание `Service`. Этот класс службы мы используем для получения экземпляра класса веб-службы, созданного нами. В данном примере это осуществляется вызовом метода `getCalculatorPort()` экземпляра `CalculatorService`. Обычно методам, предназначеннм для получения экземпляра класса веб-службы, дают имена, в соответствии с шаблоном именования `getNamePort()`, где `Name` – имя класса, реализующего веб-службу. После получения экземпляра класса веб-службы, можно просто вызывать его методы, как если бы он был простым объектом Java.



Строго говоря, метод `getNamePort()` класса службы возвращает экземпляр класса, реализующего интерфейс, сгенерированный утилитой `wsimport`. Этому интерфейсу присваивается имя класса веб-службы и он объявляет все методы, которые были объявлены как веб-службы. Для всех практических целей использования, возвращаемый объект эквивалентен классу веб-службы.

Как уже говорилось ранее, чтобы задействовать механизм внедрения ресурсов в автономных клиентах (не развернутых на сервере GlassFish), необходимо запускать их с помощью утилиты `appclient`. Предположим, мы упаковали нашего клиента в JAR-файл

calculatorserviceclient.jar; тогда команда для его запуска будет выглядеть так:

```
appclient -client calculatorserviceclient.jar
```

После ввода этой команды, в консоли должен появиться вывод нашего клиента:

```
1 + 2 = 3  
1 - 2 = -1
```

В этом примере в параметрах и возвращаемых значениях передаются простые типы. Однако, точно так же можно передавать объекты – и в качестве параметров, и в качестве возвращаемых значений. К сожалению, не все стандартные классы Java и простые типы могут использоваться при работе с веб-службами. Причина в том, что за кулисами параметры и возвращаемые значения отображаются в XML-определения, и не каждый тип может быть отображен таким способом.

Ниже перечислены типы, которые могут использоваться в вызовах веб-служб JAX-WS:

- java.awt.Image;
- java.lang.Object;
- java.lang.String;
- java.math.BigDecimal;
- java.math.BigInteger;
- java.net.URI;
- java.util.Calendar;
- java.util.Date;
- java.util.UUID;
- javax.activation.DataHandler;
- javax.xml.datatype.Duration;
- javax.xml.datatype.XMLGregorianCalendar;
- javax.xml.namespace.QName;
- javax.xml.transform.Source.

Дополнительно могут использоваться следующие простые типы:

- boolean;
- byte;
- byte[];
- double;
- float;
- int;

- long;
- short.

Можно также использовать экземпляры собственных классов, при условии, что переменные-члены в этих классах будут иметь типы из числа перечисленных выше.

Кроме того, допустимо использовать массивы – как в качестве параметров методов, так и в качестве возвращаемых значений. Однако при выполнении `wsimport` эти массивы преобразуются в списки (`List`), из-за чего появляется несоответствие между сигнатурой метода веб-службы и сигнатурой метода ее вызова на стороне клиента. По этой причине в качестве параметров метода и/или возвращаемых значений вместо массивов предпочтительнее использовать списки, так как в этом случае не возникает несоответствий между клиентом и сервером.



Для создания сообщения SOAP из вызовов методов, JAX-WS внутренне использует архитектуру Java для связывания с XML (Java Architecture for XML Binding, JAXB). Типы, которые разрешается использовать для параметров методов и возвращаемых значений, должны поддерживаться JAXB. Более подробную информацию можно получить по адресу: <http://jaxb.dev.java.net/>.

Отправка вложений веб-службам

Помимо отправки и приема данных в виде значений типов, перечисленных в предыдущем разделе, методы веб-служб могут отправлять и принимать вложенные файлы. Следующий пример демонстрирует, как это сделать:

```
package net.ensode.glassfishbook;

import java.io.FileOutputStream;
import java.io.IOException;

import javax.activation.DataHandler;
import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
public class FileAttachment {

    @WebMethod
    public void attachFile(DataHandler dataHandler) {
        FileOutputStream fileOutputStream;
```

```
try {

    // замените "/tmp/attachment.gif" на фактический
    // путь, если необходимо.
    fileOutputStream = new FileOutputStream(
        "/tmp/attachment.gif");

    dataHandler.writeTo(fileOutputStream);

    fileOutputStream.flush();
    fileOutputStream.close();
} catch (IOException e) {
    e.printStackTrace();
}
}
```

Чтобы написать метод веб-службы, который получает одно или более вложений, нужно всего лишь добавить параметр типа `javax.activation.DataHandler` для каждого вложения, которое получит метод. В вышеприведенном примере метод `attachFile()` принимает единственный параметр этого типа и просто записывает его в файловую систему.

Так же, как в случае с любой стандартной веб-службой, предыдущий код должен быть упакован в WAR-файл и развернут. После развертывания автоматически будет сгенерировано описание WSDL. Далее следует с помощью утилиты `wsimport` сгенерировать код, который будет использоваться клиентом веб-службы для получения доступа к ней. Как упоминалось выше, утилиту `wsimport` можно вызвать из командной строки непосредственно или через пользовательскую цель Ant.

После получения программного кода для доступа к веб-службе, можно написать и скомпилировать клиентский код:

```
package net.ensode.glassfishbook;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

import javax.xml.ws.WebServiceRef;

public class FileAttachmentServiceClient {

    @WebServiceRef(wsdlLocation =
```

```
"http://localhost:8080/fileattachmentservice/"
+ "FileAttachmentService?wsdl")
private static FileAttachmentService fileAttachmentService;

public static void main(String[] args) {
    FileAttachment fileAttachment = fileAttachmentService.
        getFileAttachmentPort();
    File fileToAttach = new File("src/main/resources/logo.gif");

    byte[] fileBytes = fileToByteArray(fileToAttach);

    fileAttachment.attachFile(fileBytes);
    System.out.println("Successfully sent attachment.");
}

static byte[] fileToByteArray(File file) {
    byte[] fileBytes = null;

    try {
        FileInputStream fileInputStream;
        fileInputStream = new FileInputStream(file);

        FileChannel fileChannel = fileInputStream.getChannel();
        fileBytes = new byte[(int) fileChannel.size()];
        ByteBuffer byteBuffer = ByteBuffer.wrap(fileBytes);
        fileChannel.read(byteBuffer);
    } catch (IOException e) {
        e.printStackTrace();
    }
    return fileBytes;
}
```

Клиент веб-службы, отправляющий ей одно или более вложений, сначала должен получить экземпляр веб-службы, как обычно. Затем создать экземпляр `java.io.File`, передав в параметре конструктора местоположение файла для вложения. После того как будет создан экземпляр `java.io.File` с файлом для вложения, нужно преобразовать этот файл в массив байтов и передать его методу веб-службы.

Обратите внимание, что, в отличие от передачи обычных параметров, тип параметра в вызове метода, принимающего вложение, отличается от типа параметра метода в коде веб-службы. Метод в веб-службе ожидает экземпляр `javax.activation.DataHandler` для каждого вложения. Однако код, сгенерированный утилитой `wsimport`, ожидает массив байтов для каждого вложения. Эти массивы преобразуются в правильный тип (`javax.activation.DataHandler`) кодом, сгенерированным утилитой `wsimport`. Как разработчиков приложе-

ний нас не должны интересовать тонкости происходящего. Следует только иметь в виду, что при отправке вложений методам веб-служб типы параметров в коде веб-службы и в клиентском вызове будут отличаться.

Экспортирование компонентов EJB в виде веб-служб

Веб-службы можно создавать не только, как описано в предыдущем разделе, открытые методы сеансовых компонентов без сохранения состояния, легко могут быть экспортированы как веб-службы. Как это сделать, демонстрирует следующий пример:

```
package net.ensode.glassfishbook;

import javax.ejb.Stateless;
import javax.jws.WebService;

@Stateless
@WebService
public class DectoHexBean {

    public String convertDecToHex(int i) {
        return Integer.toHexString(i);
    }
}
```

Как видите, чтобы экспортировать открытые методы сеансового компонента без сохранения состояния в виде веб-служб, достаточно декорировать объявление класса аннотацией `@WebService`. Разумеется, класс сеансового компонента также должен быть декорирован аннотацией `@Stateless`.

Точно так же, как обычные сеансовые компоненты без сохранения состояния, компоненты с методами, экспортруемыми в виде веб-служб, должны быть упакованы в JAR-файл. После развертывания этого файла можно будет увидеть новую веб-службу в узле **Applications** (Приложения) в веб-консоли сервера GlassFish. Щелкнув на узле приложения, можно увидеть некоторые подробности, как показано на рис. 10.7.

Обратите внимание, что столбец **Type** (Тип) в строке с нашим компонентом имеет значение `StatelessSessionBean`. Это позволяет сразу увидеть, что веб-служба реализована как компонент Enterprise JavaBean.

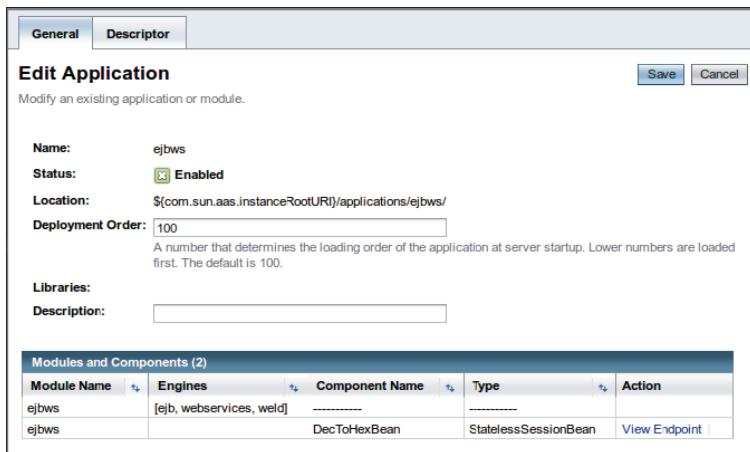


Рис. 10.7. Дополнительная информация о компоненте EJB

Так же как для стандартных веб-служб, для веб-служб EJB автоматически генерируется описание WSDL. После развертывания компонента EJB, доступ к нему можно получить тем же путем – щелкнув на ссылке **View EndPoint** (Посмотреть конечную точку).

Клиенты веб-служб EJB

Следующий класс демонстрирует процедуру получения доступа к методу веб-службы EJB из клиентского приложения:

```
package net.ensode.glassfishbook;

import javax.xml.ws.WebServiceRef;

public class DecToHexClient {

    @WebServiceRef(wsdlLocation =
        "http://localhost:8080/DecToHexBeanService/DecToHexBean?wsdl")
    private static DecToHexBeanService decToHexBeanService;

    public void convert() {
        DecToHexBean decToHexBean =
            decToHexBeanService.getDecToHexBeanPort();

        System.out.println("decimal 4013 in hex is: "
            + decToHexBean.convertDecToHex(4013));
    }

    public static void main(String[] args) {
```

```
    new DecToHexClient().convert();  
}  
}
```

Как видите, здесь нет ничего особенного, что нужно было бы сделать для получения доступа к веб-службе EJB со стороны клиента. Процедура та же, что и для обычных веб-служб.

Поскольку предыдущий код представляет собой автономное приложение, он должен запускаться с помощью утилиты `appclient`:

```
appclient -client ejbwsclient.jar
```

Эта команда выведет следующий результат:

```
decimal 4013 in hex is: fad
```

Безопасность веб-служб

Так же как обычные веб-приложения, веб-службы можно защитить, чтобы только авторизованные пользователи могли получить к ним доступ. Этого можно достичь, изменив дескриптор развертывания веб-службы `web.xml`:

```
<?xml version='1.0' encoding='UTF-8'?>  
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="2.5"  
         xmlns:xsi="http://www.w3.org/2001/XMLSchema"  
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/  
                           http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">  
  
<security-constraint>  
    <web-resource-collection>  
        <web-resource-name>Calculator Web Service</web-resource-name>  
        <url-pattern>/CalculatorService/*</url-pattern>  
        <http-method>POST</http-method>  
    </web-resource-collection>  
    <auth-constraint>  
        <role-name>user</role-name>  
    </auth-constraint>  
</security-constraint>  
    <login-config>  
        <auth-method>BASIC</auth-method>  
        <realm-name>file</realm-name>  
    </login-config>  
</web-app>
```

В этом примере мы изменим службу калькулятора так, чтобы только авторизованные пользователи могли получить к ней доступ. Обратите внимание, что для обеспечения безопасности веб-службы нужно внести точно такие же изменения, как для обеспечения безопасности

любого другого веб-приложения. Шаблон URL для элемента `<url-pattern>` можно получить щелчком на ссылке **View WSDL** (Посмотреть WSDL), соответствующей нашей службе. В данном примере URL выглядит так:

```
http://localhost:8080/calculatorservice/CalculatorService?wsdl
```

В элементе `<url-pattern>` должна быть указана часть URL, следующая сразу за корнем контекста (в нашем примере это `/CalculatorService`) и перед вопросительным знаком, сопровождающаяся наклонной чертой и звездочкой.



Особо отметим, что предыдущий дескриптор развертывания web.xml защищает только POST-запросы HTTP. Причина в том, что wsimport использует GET-запрос, чтобы получить WSDL и сгенерировать соответствующий код. Если GET-запросы будут защищены, утилита wsimport перестанет работать, поскольку будет лишен доступа к WSDL.

Следующий код демонстрирует, как автономный клиент может получить доступ к защищенной веб-службе:

```
package net.ensode.glassfishbook;

import javax.xml.ws.BindingProvider;
import javax.xml.ws.WebServiceRef;

public class CalculatorServiceClient {

    @WebServiceRef(
        wsdlLocation =
            "http://localhost:8080/securecalculatorservice/CalculatorService?
wsdl")
    private static CalculatorService calculatorService;

    public void calculate() {
        // чтобы воспользоваться веб-службой, добавьте в область файла
        // пользователя "joe" с паролем "password".
        // пользователь "joe" должен входить в группу "appuser".
        Calculator calculator = calculatorService.getCalculatorPort();
        ((BindingProvider) calculator).getRequestContext().put(
            BindingProvider.USERNAME_PROPERTY, "joe");
        ((BindingProvider) calculator).getRequestContext().put(
            BindingProvider.PASSWORD_PROPERTY, "password");

        System.out.println("1 + 2 = " + calculator.add(1, 2));
        System.out.println("1 - 2 = " + calculator.subtract(1, 2));
    }
}
```

```
}

public static void main(String[] args) {
    new CalculatorServiceClient().calculate();
}
}
```

Это – измененная версия автономного клиента веб-службы калькулятора, представленного выше. Данная версия была изменена для получения доступа к защищенной веб-службе. Как видите, для выполнения данной задачи необходимо включить имя пользователя и пароль в контекст запроса. Имя пользователя и пароль должны быть допустимыми для области, используемой при аутентификации.

Добавить имя пользователя и пароль в контекст запроса можно путем приведения класса конечной точки веб-службы к типу `javax.xml.ws.BindingProvider` и вызова его метода `get RequestContext()`. Этот метод возвращает экземпляр `java.util.Map`, куда можно добавить имя пользователя и пароль, вызывая метод `put()` и используя в качестве ключей константы `USERNAME_PROPERTY` и `PASSWORD_PROPERTY`, определенные в `BindingProvider`, а в качестве значений – соответствующие строковые объекты.

Безопасность веб-служб EJB

Подобно обычным веб-службам, веб-службы EJB также можно защищать и разрешить доступ к ним только для авторизованных клиентов. Эту задачу можно решить настройкой компонента EJB в файле `glassfish-ejb-jar.xml`:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE glassfish-ejb-jar PUBLIC "-//GlassFish.org//DTD
  GlassFish Application Server 3.1 EJB 3.1//EN"
  "http://glassfish.org/dtds/glassfish-ejb-jar_3_1-1.dtd">
<glassfish-ejb-jar>
  <ejb>
    <ejb-name>SecureDecToHexBean</ejb-name>
    <webservice-endpoint>
      <port-component-name>
        SecureDecToHexBean
      </port-component-name>
      <login-config>
        <auth-method>BASIC</auth-method>
        <realm>file</realm>
      </login-config>
    </webservice-endpoint>
```

```
</ejb>  
</glassfish-ejb-jar>
```

Как видите, безопасность веб-служб EJB настраивается иначе, чем безопасность обычных компонентов EJB. Для веб-служб EJB настройки безопасности помещаются в элемент `<webservice-endpoint>` дескриптора развертывания `glassfish-ejb-jar.xml`.

Элемент `<port-component-name>` должен содержать имя EJB, экспортируемого в виде веб-службы. Это имя определяется в элементе `<ejb-name>`.

Элемент `<login-config>` очень похож на соответствующий элемент в дескрипторе развертывания веб-приложения `web.xml`. Он должен содержать метод авторизации, в подэлементе `<auth-method>`, и область безопасности, в подэлементе `<realm>`.



Не используйте аннотацию @RolesAllowed для компонентов EJB, экспортируемых в виде веб-служб. Эта аннотация предназначена для методов EJB, доступ к которым осуществляется через удаленный или локальный интерфейс. Если компонент EJB или один или более его методов декорировать этой аннотацией, при вызове таких методов будет генерироваться исключение безопасности.

После настройки аутентификации для веб-службы EJB, ее можно упаковать в JAR-файл и развернуть как обычно. Теперь веб-служба EJB готова для обработки запросов клиентов.

Следующий пример демонстрирует, как клиент может получить доступ к защищенной веб-службе EJB:

```
package net.ensode.glassfishbook;  
  
import javax.xml.ws.BindingProvider;  
import javax.xml.ws.WebServiceRef;  
  
public class DecToHexClient {  
  
    @WebServiceRef(  
        wsdlLocation =  
            "http://localhost:8080/SecureDecToHexBeanService/  
SecureDecToHexBean?wsdl")  
    private static SecureDecToHexBeanService  
        secureDecToHexBeanService;  
  
    public void convert() {  
        SecureDecToHexBean secureDecToHexBean =  
            secureDecToHexBeanService.
```

```
    getSecureDecToHexBeanPort();

    ((BindingProvider) secureDecToHexBean).getRequestContext().put(
        BindingProvider.USERNAME_PROPERTY, "joe");
    ((BindingProvider) secureDecToHexBean).getRequestContext().put(
        BindingProvider.PASSWORD_PROPERTY, "password");

    System.out.println("decimal 4013 in hex is: "
        + secureDecToHexBean.convertDecToHex(4013));
}

public static void main(String[] args) {
    new DecToHexClient().convert();
}
}
```

Как видите, процедура доступа к веб-службе EJB идентична процедуре доступа к обычной веб-службе. Способ реализации веб-службы клиенту не важен.

Резюме

В этой главе мы рассмотрели особенности разработки веб-служб и их клиентов с использованием JAX-WS API. Мы объяснили, как сгенерировать код доступа к веб-службе для клиентов путем использования инструментов сборки Ant и Maven. Также мы обсудили, какие типы данных могут передаваться в вызовы методов через JAX-WS. Дополнительно была рассмотрена возможность отправки вложений веб-службам. Мы также выяснили, как экспорттировать методы компонентов EJB в виде веб-служб. Наконец, мы посмотрели, как обезопасить веб-службы от доступа неавторизованных клиентов.

В следующей главе мы исследуем создание веб-служб RESTful с применением JAX-RS API.



ГЛАВА 11.

Веб-службы RESTful JAX-RS

Передача представительного состояния (Representational State Transfer, REST) – это архитектурный стиль, в котором веб-службы рассматриваются как ресурсы и могут идентифицироваться унифицированными идентификаторами ресурсов (Uniform Resource Identifiers, URI).

Веб-службы, разработанные в стиле REST, называют веб-службами RESTful.

В Java EE 6 добавлена поддержка веб-служб RESTful в виде Java API для веб-служб RESTful (Java API for RESTful Web Services, JAX-RS). В течение некоторого времени JAX-RS API был доступен как автономный прикладной интерфейс; но в Java EE 6 он стал частью спецификации. В этой главе мы покажем, как создавать веб-службы RESTful с использованием JAX-RS API.

В главе будут затронуты следующие темы:

- ◊ введение в веб-службы RESTful и JAX-RS;
- ◊ создание простой веб-службы RESTful;
- ◊ создание клиента веб-службы RESTful;
- ◊ параметры пути;
- ◊ параметры запроса.

Введение в веб-службы RESTful и JAX-RS

Веб-службы RESTful отличаются большой гибкостью. Они могут принимать и возвращать данные нескольких разных MIME-типов, однако чаще в них предусматривается прием/передача данных в формате XML или JSON.

Веб-службы должны поддерживать один или несколько методов HTTP из следующих четырех:

- `GET`: в соответствии с соглашениями запросы этого типа используются для получения существующих ресурсов;
- `POST`: в соответствии с соглашениями запросы этого типа используются для изменения существующего ресурса;
- `PUT`: – в соответствии с соглашениями запросы этого типа используются для создания нового ресурса;
- `DELETE`: – в соответствии с соглашениями запросы этого типа используются для удаления существующего ресурса.

Мы создадим веб-службу RESTful, определив класс с аннотированными методами, которые будут вызываться при получении веб-службой одного из четырех перечисленных HTTP-запросов. После создания и развертывания веб-службы RESTful мы реализуем клиента, который будет отправлять запросы нашей службе. В JAX-RS 2.0 был стандартизован клиентский API, который можно использовать для упрощения разработки клиентов веб-служб RESTful.

Создание простой веб-службы RESTful

В этом разделе мы создадим простую веб-службу, чтобы показать, как заставить методы веб-службы отвечать на разные HTTP-запросы.

Создание веб-служб RESTful с использованием JAX-RS не вызывает никаких сложностей. Все веб-службы RESTful должны вызываться через унифицированные идентификаторы ресурса (URI). Определяются эти URI с помощью аннотации `@Path`, которой следует отметить класс RESTful-ресурса веб-службы.

В процессе разработки веб-службы RESTful необходимо определить методы, которые будут вызываться для обработки HTTP-запросов от одного до четырех типов, а именно: `GET`, `POST`, `PUT` и/или `DELETE`.

JAX-RS API предоставляет четыре аннотации для декорирования методов веб-служб. Аннотации имеют соответствующие имена: `@GET`, `@POST`, `@PUT` и `@DELETE`. Декорирование метода веб-службы одной из этих аннотаций заставит его отвечать на соответствующий тип HTTP-запросов.

Кроме того, каждый метод в службе должен принимать и/или возвращать определенный MIME-тип.



Многоцелевые расширения электронной почты Интернета (Multipurpose Internet Mail Extensions, MIME) – это стандарт, определяющий порядок передачи через Интернет данных, которые не являются ASCII-текстом. Первоначально стандарт MIME созда-

вался с целью определить порядок передачи нетекстовой информации по электронной почте, но позднее он был расширен включением в него других форм передачи данных, таких как веб-службы RESTful.

MIME-тип, который будет возвращаться, должен быть определен аннотацией @Produces. В свою очередь MIME-тип, который будет приниматься, должен быть определен аннотацией @Consumes.



Обратите внимание, что этот пример ничего «существенного» не делает – его цель в том, чтобы показать, как заставить разные методы класса RESTful-ресурса веб-службы отвечать на разные HTTP-запросы.

Следующий пример демонстрирует практическое применение понятий, которые только что были описаны.

```
package com.ensode.jaxrsintro.service;

import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("customer")
public class CustomerResource {

    @GET
    @Produces("text/xml")
    public String getCustomer() {
        // в "настоящей" веб-службе данные могли бы извлекаться
        // из базы данных и возвращаться в формате XML.

        System.out.println(" --- " + this.getClass().getCanonicalName()
            + ".getCustomer() invoked");

        return "<customer>\n"
            + "<id>123</id>\n"
            + "<firstName>Joseph</firstName>\n"
            + "<middleName>William</middleName>\n"
            + "<lastName>Graystone</lastName>\n"
            + "</customer>\n";
    }

    /**

```

```

    * Создает нового клиента
    * @param customerXML - информация в формате XML о новом заказчике
    */
@PUT
@Consumes("text/xml")
public void createCustomer(String customerXML) {
    // в "настоящей" веб-службе RESTful можно было бы разобрать XML
    // в параметре customerXML и затем добавить запись в базу данных.
    System.out.println("--- " + this.getClass().getCanonicalName()
        + ".createCustomer() invoked");
    System.out.println("customerXML = " + customerXML);
}

@POST
@Consumes("text/xml")
public void updateCustomer(String customerXML) {
    // в "настоящей" веб-службе RESTful можно было бы разобрать XML
    // в параметре customerXML и затем изменить запись в базе данных.
    System.out.println("--- " + this.getClass().getCanonicalName()
        + ".updateCustomer() invoked");

    System.out.println("customerXML = " + customerXML);
}

@DELETE
@Consumes("text/xml")
public void deleteCustomer(String customerXML) {
    // в "настоящей" веб-службе RESTful можно было бы разобрать XML
    // в параметре customerXML и затем удалить запись из базы данных.
    System.out.println("--- " + this.getClass().getCanonicalName()
        + ".deleteCustomer() invoked");

    System.out.println("customerXML = " + customerXML);
}
}

```

Обратите внимание, что класс декорирован аннотацией `@Path`. Она определяет унифицированный идентификатор ресурса (URI) для веб-службы RESTful. Полный URI для нашей службы будет включать протокол, имя сервера, порт, корневой контекст, путь к ресурсам REST (см. следующий подраздел) и значение, переданное в эту аннотацию.

Предположим, что веб-служба была развернута на сервере `example.com` с использованием протокола HTTP по порту 8080, имеет корневой контекст `jaxrsintro` и путь `resources` к ресурсам REST. В таком случае полный URI для нашей веб-службы будет выглядеть так: `http://example.com:8080/jaxrsintro/resources/customer`.



Веб-браузеры генерируют `GET`-запросы, когда в адресной строке водится `URL`, поэтому протестировать метод `GET` службы можно, просто задав в адресной строке браузера `URI` нашей службы.

Обратите внимание, что каждый из методов в классе аннотирован одной из аннотаций `@GET`, `@POST`, `@PUT` или `@DELETE`. Эти аннотации заставляют методы отвечать на HTTP-запросы соответствующих типов.

В дополнение ко всему, если метод возвращает данные клиенту, с помощью аннотации `@Produces` объявляется, какой МИМЕ-тип возвращаемых данных. В нашем примере только метод `getCustomer()` возвращает данные клиенту. Данные будут возвращаться в формате XML, поэтому в аннотации `@Produces` указано значение `text/xml`. Аналогично, если метод должен принимать данные от клиента, нужно указать МИМЕ-тип входных данных. Это делается с помощью аннотации `@Consumes`. Все методы в нашей службе, кроме `getCustomer()`, принимают данные. Во всех случаях ожидается, что данные будут в формате XML, поэтому снова указываем `text/xml` в качестве МИМЕ-типа.

Настройка пути к ресурсам REST в приложении

В предыдущем разделе упоминалось, что, прежде чем развернуть веб-службу RESTful, необходимо настроить путь к ресурсам REST для нашего приложения. Сделать это можно, определив класс, наследующий `javax.ws.rs.core.Application` и отмеченный аннотацией `@ApplicationPath`.

Настройка с использованием аннотации `@ApplicationPath`

Как говорилось в предыдущих главах, в Java EE 6 был добавлен ряд новых возможностей с целью свести к минимуму необходимость создания дескриптора развертывания `web.xml`. JAX-RS не является исключением. Мы можем настроить путь к REST-ресурсам в коде Java через аннотацию.

Чтобы настроить путь к REST-ресурсам без использования дескриптора развертывания `web.xml`, нужно определить класс, наследующий `javax.ws.ApplicationPath`, и декорировать его аннотацией `@ApplicationPath`. Значение, которое будет передано этой аннотации, определяет путь к REST-ресурсам служб.

Этот процесс иллюстрирует следующий пример:

```
package com.enseode.jaxrsintro.service.config;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("resources")
public class JaxRsConfig extends Application { }
```

Обратите внимание, что класс не должен реализовывать какие-либо методы. Он просто должен наследовать javax.ws.rs.Application и декорироваться аннотацией @ApplicationPath. Класс должен быть общедоступным (public), может иметь любое имя и помещен в любой пакет.

Тестирование веб-службы

Как отмечалось выше, веб-браузеры отправляют GET-запросы по любым URL, указанным в адресной строке. Поэтому самый простой способ протестировать GET-запросы к службе – просто указать URI веб-службы в адресной строке браузера, как показано на рис. 11.1.

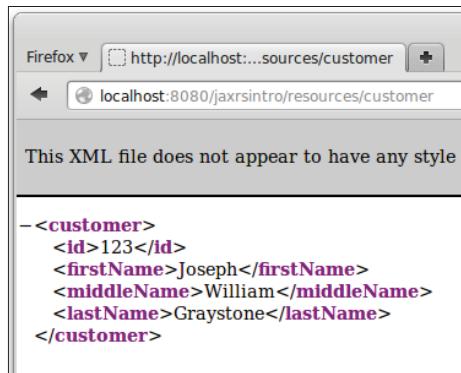


Рис. 11.1. Чтобы протестировать GET-запрос к службе, достаточно просто указать URI в адресной строке браузера

Веб-браузеры поддерживают только GET- и POST-запросы. Чтобы протестировать POST-запрос с помощью браузера, нужно написать приложение с HTML-формой, имеющей атрибут action со значением URI веб-службы. Для одной веб-службы это приложение будет тривиально простым, но может стать громоздким и неудобным, если

распространить его действие на все веб-службы RESTful, которые мы разрабатываем.

К счастью, есть утилита `curl` с открытым исходным кодом, которую можно использовать для тестирования наших веб-служб. Утилита `curl` включена в большинство дистрибутивов Linux и нет никаких препятствий получить версию для Windows, Mac OS X и некоторых других платформ (адрес для загрузки: <http://curl.haxx.se/>).

Утилита `curl` может передать все четыре типа запросов (`GET`, `POST`, `PUT` и `DELETE`) нашей веб-службе. Ответ сервера будет выведен в окне консоли. Утилита `curl` принимает параметр командной строки `-X`, который позволяет указать метод запроса. Чтобы отправить `GET`-запрос, достаточно просто запустить команду:

```
curl -XGET http://localhost:8080/jaxrsintro/resources/customer
```

Результатом должен стать следующий вывод:

```
<customer>
  <id>123</id>
  <firstName>Joseph</firstName>
  <middleName>William</middleName>
  <lastName>Graystone</lastName>
</customer>
```

Неудивительно, что это тот же результат, который мы видели, когда указывали URI в адресной строке браузера (см. рис.11.1).

По умолчанию `curl` генерирует `GET`-запросы, поэтому параметр `-X` в предыдущем примере избыточен. Тот же результат можно было бы получить следующей командой:

```
curl http://localhost:8080/jaxrsintro/resources/customer
```

После выполнения любой из предыдущих команд, в журнале GlassFish должны появиться строки, которые выводятся инструкцией `System.out.println()` в методе `getCustomer()`:

```
INFO: --- com.ensode.jaxrsintro.service.CustomerResource.getCustomer()
invoked
```

Для всех других типов запросов необходимо отправить некоторые данные нашей службе. Это можно сделать с помощью аргумента командной строки `--data`:

```
curl -XPUT -HContent-type:text/xml --data "<customer><id>321
</id><firstName>Amanda</firstName><middleName>Zoe
</middleName><lastName>Adams</lastName></customer>" http://localhost:
8080/jaxrsintro/resources/customer
```

Как видите, мы должны передать утилите curl параметр -H с MIME-типовом, используя формат, который мы видели в примере выше.

Убедиться, что предыдущая команда сработала как ожидалось, можно, если заглянуть в журнал GlassFish:

```
INFO: --- com.ensode.jaxrsintro.service.CustomerResource.createCustomer()
invoked

INFO: customerXML = <customer><id>321</id><firstName>Amanda
</firstName><middleName>Zoe</middleName><lastName>Adams
</lastName></customer>
```

Так же просто можно протестировать другие типы запросов:

```
curl -XPOST -HContent-type:text/xml --data "<customer><id>321
</id><firstName>Amanda</firstName><middleName>Tamara
</middleName><lastName>Adams</lastName></customer>" http://localhost:
8080/jaxrsintro/resources/customer
```

В журнале GlassFish должны появиться следующие строки:

```
INFO: --- com.ensode.jaxrsintro.service.CustomerResource.updateCustomer()
invoked

INFO: customerXML = <customer><id>321</id><firstName>Amanda
</firstName><middleName>Tamara</middleName><lastName>Adams
</lastName></customer>
```

Протестировать метод DELETE можно командой:

```
curl -XDELETE -HContent-type:text/xml --data "<customer><id>321
</id><firstName>Amanda</firstName><middleName>Tamara
</middleName><lastName>Adams</lastName></customer>" http://localhost:
8080/jaxrsintro/resources/customer
```

И снова, в журнале GlassFish должны появиться следующие строчки:

```
INFO: --- com.ensode.jaxrsintro.service.CustomerResource.deleteCustomer()
invoked

INFO: customerXML = <customer><id>321</id><firstName>Amanda
</firstName><middleName>Tamara</middleName><lastName>Adams
</lastName></customer>
```

Преобразование данных в/из XML с помощью JAXB

В предыдущем примере мы никак не обрабатывали данные в формате XML, полученные в параметре, а так же возвращали «жестко заши-

тый» код XML клиенту. В реальном приложении мы более чем вероятно проанализировали бы XML, полученный от клиента, и использовали его для заполнения объекта Java. Кроме того, любые данные XML, которые мы должны вернуть клиенту, должны были бы создаваться из объектов Java.

Преобразование данных в формат XML и обратно настолько частый случай в практике, что спецификация Java EE предоставляет для этих целей API, который называется **Java API для связывания с XML (Java API for XML Binding, JAXB)**.

JAXB делает преобразование данных из/в XML прозрачным и тривиальным. Для этого нужно лишь декорировать класс, который потребуется преобразовывать в XML, аннотацией `@XmlRootElement`. Следующий пример показывает, как это сделать:

```
package com.ensode.jaxrtest.entity;

import java.io.Serializable;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Customer implements Serializable {

    private Long id;
    private String firstName;
    private String middleName;
    private String lastName;

    public Customer() {
    }

    public Customer(Long id, String firstName,
                    String middleInitial, String lastName) {
        this.id = id;
        this.firstName = firstName;
        this.middleName = middleInitial;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public Long getId() {
```

```

        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getMiddleName() {
        return middleName;
    }

    public void setMiddleName(String middleName) {
        this.middleName = middleName;
    }

    @Override
    public String toString() {
        return "id = " + getId() + "\nfirstName = " + getFirstName()
            + "\nmiddleName = " + getMiddleName() + "\nlastName = "
            + getLastname();
    }
}

```

Как видите, кроме аннотации `@XmlRootElement` на уровне класса здесь нет ничего необычного.

После определения класса с аннотацией `@XmlRootElement` необходимо изменить тип параметра веб-службы со `String` на наш класс:

```

package com.ensode.jaxbxmlconversion.service;

import com.ensode.jaxbxmlconversion.entity.Customer;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("customer")
public class CustomerResource {

    private Customer customer;

```

```
public CustomerResource() {
    // "имитация" данных, в действующем приложении данные
    // будут извлекаться из базы данных.
    customer = new Customer(1L, "David",
                           "Raymond", "Heffelfinger");
}

@GET
@Produces("text/xml")
public Customer getCustomer() {
    // в "настоящей" веб-службе данные скорее всего будут
    // извлекаться из базы данных и возвращаться
    // в формате XML.
    System.out.println("--- " + this.getClass().getCanonicalName()
        + ".getCustomer() invoked");
    return customer;
}

@POST
@Consumes("text/xml")
public void updateCustomer(Customer customer) {
    // в "настоящей" веб-службе XML-данные скорее всего будут
    // анализироваться с помощью JAXB и сохраняться
    // в базе данных.

    System.out.println("--- " + this.getClass().getCanonicalName()
        + ".updateCustomer() invoked");
    System.out.println("---- got the following customer: "
        + customer);
}

@PUT
@Consumes("text/xml")
public void createCustomer(Customer customer) {
    // в "настоящей" веб-службе XML-данные скорее всего будут
    // анализироваться с помощью JAXB и сохраняться
    // в базе данных.

    System.out.println("--- " + this.getClass().getCanonicalName()
        + ".createCustomer() invoked");
    System.out.println("customer = " + customer);
}

@DELETE
@Consumes("text/xml")
public void deleteCustomer(Customer customer) {
    // в "настоящей" веб-службе скорее всего потребовалось бы
    // удалить из базы данных запись, соответствующую информации
    // в параметре customer.
```

```
        System.out.println("--- " + this.getClass().getCanonicalName()
            + ".deleteCustomer() invoked");
        System.out.println("customer = " + customer);
    }
}
```

Как видите, разница между этой и предыдущей версией веб-службы RESTful заключается в том, что все типы параметров и возвращаемых значений изменены со `String` на `Customer`. JAXB заботится о преобразовании параметров и возвращаемых значений в XML и обратно, по мере необходимости. При использовании JAXB объекты пользовательских классов автоматически заполняются данными из документа XML, полученного от клиента. Точно так же возвращаемые значения прозрачно преобразуются в XML.

Создание клиента веб-службы RESTful

Хотя утилита `curl` позволяет быстро протестировать веб-службы RESTful и является дружественным для разработчика инструментом, она все же недостаточно удобна для пользователя. Едва ли кто-то из пользователей пожелает вводить команды `curl` в командной строке, чтобы воспользоваться нашей веб-службой. Поэтому, помимо веб-служб мы должны создавать клиентов для них. Спецификация JAX-RS 2.0 определяет стандартный клиентский API, который можно использовать для упрощения разработки клиентских приложений.

Следующий пример демонстрирует, как использовать клиентский API в JAX-RS:

```
package com.ensode.jaxrsintroclient;

import com.ensode.jaxbxmlconversion.entity.Customer;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;

public class App {

    public static void main(String[] args) {
        App app = new App();
        app.insertCustomer();
    }

    public void insertCustomer() {
```

```
Customer customer = new Customer(234L, "Tamara", "A",
                                  "Graystone");
Client client = ClientBuilder.newClient();
client.target(
    "http://localhost:8080/jaxbxmlconversion/resources/customer").
    request().put(
        Entity.entity(customer, "text/xml"),
        Customer.class);
}
```

Первое, что нужно сделать, – создать экземпляр javax.ws.rs.client.Client вызовом статического метода newClient() класса javax.ws.rs.client.ClientBuilder.

Затем вызывается метод target() экземпляра Client и ему передается URI веб-службы RESTful. Метод target() вернет экземпляр класса, реализующего интерфейс javax.ws.rs.client.WebTarget.

В этой точке тут же вызывается метод request() экземпляра WebTarget, который вернет реализацию интерфейса javax.ws.rs.client.Invocation.Builder.

В данном конкретном примере веб-службе отправляется HTTP-запрос PUT, поэтому здесь вызывается метод put() реализации Invocation.Builder. В первом параметре метод put() принимает экземпляр javax.ws.rs.client.Entity. Этот экземпляр можно создать «на лету», вызовом статического метода entity() класса Entity. В первом параметре этот метод принимает объект для передачи веб-службе RESTful, а во втором – строку с названием MIME-типа данных, подлежащих передаче веб-службе RESTful. Во втором параметре метод put() принимает тип ответа, ожидаемый клиентом. После вызова метода put() отправляется HTTP-запрос PUT веб-службе RESTful и вызывается ее метод, декорированный аннотацией @Put (createCustomer() в данном примере). В клиенте определены аналогичные методы get(), post() и delete(), которые можно вызывать для отправки соответствующих HTTP-запросов веб-службе RESTful.

Параметры запроса и пути

В нашем предыдущем примере мы работали с веб-службой RESTful, управляющей единственным объектом customer. Очевидно, что на практике это не имеет особого смысла. Обычно веб-службы RESTful разрабатываются для обработки коллекций объектов (в нашем при-

мере – заказчиков). Чтобы определить, с каким конкретно объектом в наборе мы работаем, можно передавать веб-службе RESTful дополнительные параметры. Существуют два типа параметров, которые мы можем использовать: параметры запроса и параметры пути.

Параметры запроса

Мы можем добавить параметры в методы веб-службы, которые обрабатывают HTTP-запросы. Параметры, декорированные аннотацией `@QueryParam`, будут извлекаться из URL запроса.

Следующий пример поясняет, как можно использовать параметры запроса в нашей веб-службе JAX-RS RESTful:

```
package com.ensode.queryparams.service;

import com.ensode.queryparams.entity.Customer;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;

@Path("customer")
public class CustomerResource {

    private Customer customer;

    public CustomerResource() {
        customer = new Customer(1L, "Samuel",
                               "Joseph", "Willow");
    }

    @GET
    @Produces("text/xml")
    public Customer getCustomer(@QueryParam("id") Long id) {
        // в "настоящей" веб-службе данные скорее всего будут
        // извлекаться из базы данных с использованием
        // переданного значения id.
        System.out.println("--- " + this.getClass().getCanonicalName()
                           + ".getCustomer() invoked, id = " + id);
        return customer;
    }

    /**
     * Создает запись с информацией о новом заказчике
    }
```

```
* @param customer - XML-документ с информацией о новом заказчике
*/
@PUT
@Consumes("text/xml")
public void createCustomer(Customer customer) {
    // в "настоящей" веб-службе скорее всего был бы
    // выполнен парсинг XML, полученного в параметре
    // customer, а затем добавлена новая запись в базу данных
    System.out.println("--- " + this.getClass().getCanonicalName()
        + ".createCustomer() invoked");
    System.out.println("customer = " + customer);
}

@POST
@Consumes("text/xml")
public void updateCustomer(Customer customer) {
    // в "настоящей" веб-службе скорее всего был бы
    // выполнен парсинг XML, полученного в параметре
    // customer, а затем произведено изменение
    // записи в базе данных.
    System.out.println("--- " + this.getClass().getCanonicalName()
        + ".updateCustomer() invoked");
    System.out.println("customer = " + customer);
    System.out.println("customer= " + customer);
}

@DELETE
@Consumes("text/xml")
public void deleteCustomer(@QueryParam("id") Long id) {
    // в "настоящей" веб-службе скорее всего был бы
    // вызван объект DAO и с его помощью удалена запись
    // из базы данных с первичным ключом, переданным в
    // параметре "id".
    System.out.println("--- " + this.getClass().getCanonicalName()
        + ".deleteCustomer() invoked, id = " + id);
    System.out.println("customer = " + customer);
}
}
```

Обратите внимание, что от нас потребовалось лишь декорировать параметры аннотацией `@QueryParam`. Она позволяет JAX-RS получать любые параметры запроса, соответствующие значению аннотации, и присваивать их переменным параметров.

Добавить параметр в URL веб-службы можно точно так же, как при передаче любых других параметров в URL:

```
curl -XGET -HContent-type:text/xml http://localhost:8080/queryparams/
resources/customer?id=1
```

Отправка параметров запроса через клиентский JAX-RS API

Клиентский JAX-RS API предоставляет простой и прямой способ отправки параметров запроса веб-службам RESTful, как можно видеть в следующем примере:

```
package com.ensode.queryparamsclient;

import com.ensode.queryparamsclient.entity.Customer;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;

public class App {

    public static void main(String[] args) {
        App app = new App();
        app.getCustomer();
    }

    public void getCustomer() {
        Client client = ClientBuilder.newClient();
        Customer customer = client.target(
            "http://localhost:8080/queryparams/resources/customer").
            queryParam("id", 1L).
            request().get(Customer.class);
        System.out.println(
            "Received the following customer information:");
        System.out.println("Id: " + customer.getId());
        System.out.println("First Name: " + customer.getFirstName());
        System.out.println("Middle Name: " +
            customer.getMiddleName());
        System.out.println("Last Name: " + customer.getLastName());
    }
}
```

Чтобы передать параметр, нужно вызвать метод `queryParam()` экземпляра `javax.ws.rs.client.WebTarget`, возвращаемого вызовом метода `target()` экземпляра `client`. В первом аргументе этому методу передается имя параметра, которое должно соответствовать значению аннотации `@QueryParam` веб-службы. Во втором аргументе передается значение параметра. Чтобы передать веб-службе несколько параметров, можно составить цепочку из вызовов метода `queryParam()`, по одному для каждого параметра.

Параметры пути

Другой способ передачи параметров веб-службам RESTful – использовать параметры пути. Следующий пример демонстрирует, как организовать прием параметров пути в веб-службе RESTful:

```
package com.ensode.pathparams.service;

import com.ensode.pathparams.entity.Customer;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rsPathParam;
import javax.ws.rs.Produces;

@Path("/customer/")
public class CustomerResource {

    private Customer customer;

    public CustomerResource() {
        customer = new Customer(1L, "William",
                               "Daniel", "Graystone");
    }

    @GET
    @Produces("text/xml")
    @Path("{id}/")
    public Customer getCustomer(@PathParam("id") Long id) {
        // в "настоящей" веб-службе RESTful данные могли бы извлекаться
        // из базы данных с учетом значения параметра id.
        System.out.println("--- " + this.getClass().getCanonicalName()
                           + ".getCustomer() invoked, id = " + id);
        return customer;
    }

    @PUT
    @Consumes("text/xml")
    public void createCustomer(Customer customer) {
        // в "настоящей" веб-службе скорее всего был бы
        // выполнен парсинг XML, полученного в параметре
        // customer, а затем добавлена новая запись в базу данных
        System.out.println("--- " + this.getClass().getCanonicalName()
                           + ".createCustomer() invoked");
        System.out.println("customer = " + customer);
    }

    @POST
    @Consumes("text/xml")
    public void updateCustomer(Customer customer) {
        // в "настоящей" веб-службе скорее всего был бы
        // выполнен парсинг XML, полученного в параметре
        // customer, а затем произведено изменение
        // записи в базе данных.
```

```

        System.out.println(" --- " + this.getClass().getCanonicalName()
            + ".updateCustomer() invoked");
        System.out.println("customer = " + customer);
        System.out.println("customer= " + customer);
    }

    @DELETE
    @Consumes("text/xml")
    @Path("/{id}/")
    public void deleteCustomer(@PathParam("id") Long id) {
        // в "настоящей" веб-службе скорее всего был бы
        // вызван объект DAO и с его помощью удалена запись
        // из базы данных с первичным ключом, переданным в
        // параметре "id".
        System.out.println(" --- " + this.getClass().getCanonicalName()
            + ".deleteCustomer() invoked, id = " + id);
        System.out.println("customer = " + customer);
    }
}

```

Любой метод, принимающий параметр пути, должен декорироваться аннотацией `@Path`. Атрибут значения этой аннотации должен записываться в формате "`{paramName}/`", где `paramName` – параметр для передачи методу. Кроме того, параметры метода должны декорироваться аннотацией `@PathParam`. Значение этой аннотации должно соответствовать имени параметра, объявленному в аннотации `@Path` для метода.

Параметры пути можно передать из командной строки путем изменения URI веб-службы. Например, параметр `id` со значением `1` можно передать методу `getCustomer()` (который обрабатывает HTTP-запросы `GET`), как показано ниже:

```
curl -XGET -HContent-type:text/xml http://localhost:8080/pathparams/
resources/customer/1
```

Эта команда вернет XML-представление объекта `Customer`, возвращаемого методом `getCustomer()`:

```
<?xml version='1.0' encoding='UTF-8' standalone="yes"?><customer>
<firstName>William</firstName><id>1</id><lastName>Graystone
</lastName><middleName>Daniel</middleName></customer>
```

Отправка параметров пути через клиентский JAX-RS API

Отправка параметров пути веб-службе через клиентский JAX-RS API производится просто: достаточно добавить пару вызовов методов,

определяющих параметр пути и его значение. Следующий пример демонстрирует, как это сделать:

```
package com.ensode.pathparamsclient;

import com.ensode.pathparamsclient.entity.Customer;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;

public class App {

    public static void main(String[] args) {
        App app = new App();
        app.getCustomer();
    }

    public void getCustomer() {
        Client client = ClientBuilder.newClient();
        Customer customer = client.target(
            "http://localhost:8080/pathparams/resources/customer") .
            path("{id}") .
            resolveTemplate("id", 1L) .
            request().get(Customer.class);

        System.out.println(
            "Received the following customer information:");
        System.out.println("Id: " + customer.getId());
        System.out.println("First Name: " +
            customer.getFirstName());
        System.out.println("Middle Name: " +
            customer.getMiddleName());
        System.out.println("Last Name: " +
            customer.getLastName());
    }
}
```

В этом примере мы вызвали метод `path()` экземпляра `WebTarget`, который вернул вызов `client.target()`. Этот метод добавляет указанный путь в экземпляр `WebTarget`; значение этого метода должно соответствовать со значением аннотации `@Path` в веб-службе RESTful.

После вызова метода `path()` экземпляра `WebTarget` нужно вызвать метод `resolveTemplate()` и передать ему в первом аргументе имя параметра (без фигурных скобок) и во втором аргументе – значение параметра.

Если потребуется передать более одного параметра, достаточно просто использовать следующий формат определения параметров в аннотации `@Path` на уровне метода:

```
@Path("/{paramName1}/{paramName2}/")
```

А затем декорировать соответствующие параметры метода аннотацией `@PathParam`:

```
public String someMethod(@PathParam("paramName1") String param1,  
    @PathParam("paramName2") String param2)
```

После этого можно вызвать веб-службу, изменив ее URI для передачи параметров в порядке, указанном в аннотации `@Path`. Например, следующий URI передает значения 1 и 2 для `paramName1` и `paramName2`:

```
http://localhost:8080/contextroot/resources/customer/1/2
```

Этот URI будет работать как из командной строки, так и в клиенте веб-службы, созданном с помощью клиентского JAX-RS API.

Резюме

В этой главе мы обсудили, как быстро создавать веб-службы RESTful, используя JAX-RS – новое дополнение к спецификации Java EE.

Мы показали процесс разработки веб-службы путем добавления нескольких простых аннотаций в программный код. Также мы объяснили, как автоматически преобразовать данные Java в XML и обратно, используя возможности Java API для связывания с XML (JAXB).

Также мы показали, как создать клиента веб-службы RESTful с использованием клиентского JAX-RS API.

И наконец, мы рассмотрели передачу параметров веб-службам RESTful через аннотации `@PathParam` и `@QueryParam`.



ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Символы

< оператор 118
<= оператор 118
= оператор 118
> оператор 118
>= оператор 118
:name, параметр 117
_currentRealm, переменная 275
<f:ajax>, тег 71
 event, атрибут 73
 render, атрибут 73
<f:validateBean>, тег 54
<f:validateDoubleRange>, тег 54
<f:validateLength>, тег 54, 70
<f:validateLongRange>, тег 54
<f:validateRegex>, тег 54
<f:validateRequired>, тег 54
<f:validator>, тег 62
<h:commandButton>, тег 55
<h:form>, тег 47, 250
<h:inputText>, тег 49
<h:messages>, тег 47, 68
<h:outputLabel>, тег 49
<h:outputStylesheet>, тег 46
<h:outputText>, тег 59, 73
<h:panelGrid>, тег 47
<h:panelGroup>, тег 55
<login-config>, элемент 299
<message-bundle>, элемент 70
<port-component-name>, элемент 299
<realm-name>, элемент 277
@ApplicationPath, аннотация 305
@ApplicationScoped, аннотация 57, 177
@Asynchronous, аннотация 140
@ClientEndPoint, аннотация 207

@Column, аннотация 88
@Consumes, аннотация 305
@ConversationScoped, аннотация 58, 177
@DELETE, аннотация 305
@Dependent, аннотация 58, 178
@EJB, аннотация 166
@Entity, аннотация 88
@FacesValidator, аннотация 61
@FlowScoped, аннотация 81
@GET, аннотация 305
@IdClass, аннотация 113
@Id, аннотация 88
@JoinColumn, аннотация 106
@JoinTable, аннотация 106, 108
@ManyToMany, аннотация 106
@MessageDriven, аннотация 143
@NamedQuery, аннотация 118
@Named, аннотация 56, 57
@NotNull, аннотация 128
@OnClose, аннотация 200
@OnError, аннотация 200, 207
@OneToMany, аннотация 100
@OneToOne, аннотация 95
@OnMessage, аннотация 199, 208
@OnOpen, аннотация 199
@PathParam, аннотация 318
@Path, аннотация 304, 319
@PostActivate, аннотация 153, 154
@PostConstruct, аннотация 154
@POST, аннотация 305
@PreDestroy, аннотация 154
@PrePassivate, аннотация 153, 154
@Produces, аннотация 305
@PUT, аннотация 305
@QueryParam, аннотация 315

@Remove, аннотация 154
 @RequestScoped, аннотация 57, 58, 177
 @Resource?, аннотация 218
 @RolesAllowed, аннотация 164, 299
 @Schedule, аннотация 161
 dayOfMonth, атрибут 161
 dayOfWeek, атрибут 161
 hour, атрибут 161
 minute, атрибут 161
 month, атрибут 161
 second, атрибут 161
 timezone, атрибут 161
 year, атрибут 161
 @ServerEndpoint, аннотация 199
 @SessionScoped, аннотация 58, 177
 @Size, аннотация 128
 @Stateless, аннотация 294
 @Table, аннотация 88
 @TransactionAttribute, аннотация 145
 TransactionAttributeType.MANDATORY 145
 TransactionAttributeType.NEVER 145
 TransactionAttributeType.NOT_SUPPORTED 145
 TransactionAttributeType.REQUIRED 146
 TransactionAttributeType.REQUIRES_NEW 146
 TransactionAttributeType.SUPPORTS 146
 @TransactionManagement, аннотация 150
 @WebMethod, аннотация 280
 @WebServiceRef, аннотация 289
 @WebService, аннотация 279
 @XmlRootElement, аннотация 310

A

архитектура Java для связывания с XML
 (Java Architecture for XML Binding, JAXB) 291
 асинхронные методы

cancel(boolean mayInterruptIfRunning) 142
 get() 142
 get(long timeout, TimeUnit unit) 142
 isCancelled() 142
 isDone() 143
 введение 140

Б

безопасность
 веб-служб 296
 EJB 298
 безопасность EJB 162
 аутентификация клиентов 165
 библиотеки компонентов JSF
 ICEfaces 84
 Primefaces 84
 RichFaces 84

В

валидаторы
 нестандартные 60
 методы 63
 создание 60
 веб-приложения
 вызов сеансовых компонентов 137
 веб-службы
 безопасность 296
 клиенты веб-служб EJB 295
 компоненты EJB 294
 отправка вложений 291
 создание клиента 284
 веб-службы, JAX-WS 279
 веб-сокеты
 Java API 197
 введение 197
 дополнительная информация 208
 клиентский код на JavaScript 201
 клиенты на Java 204
 серверные конечные точки
 разработка 197
 внедрение зависимостей 171

выражения связывания значений 49
выражения связывания методов 55

Г

группировка компонентов 55

Д

долговременная подписка
создание 228
домены GlassFish
номера портов 34
остановка 36
создание 34
удаление 35
дополнительные области
JDBC 267
LDAP 265
Solaris 266
определение 262
сертификата 264

И

именованные компоненты 169
именованные компоненты CDI 56
источники данных
настройка 40

К

каскадные таблицы стилей (Cascading Style Sheets, CSS) 67
квалифициаторы (CDI) 173
клиентский код на JavaScript 201
клиенты
веб-служб EJB 295
компоненты
именованные 169
компоненты-одиночки (singleton) 139
конвертеры 20
константы JsonParser.Event
END_ARRAY 195
END_OBJECT 195
KEY_NAME 195

START_ARRAY 195
START_OBJECT 195
VALUE_FALSE 195
VALUE_NULL 195
VALUE_NUMBER 195
VALUE_STRING 195
VALUE_TRUE 195
контексты и внедрение зависимостей
(Contexts and Dependency Injection, CDI) 43
контексты именованных компонентов 57
@ApplicationScoped, аннотация 57, 177
@ConversationScoped, аннотация 58, 177
@Dependent, аннотация 58, 178
@RequestScoped, аннотация 58, 177
@SessionScoped, аннотация 58, 177

М

местоположение ресурсов, стандартное 44
многие ко многим, отношения 104
многоцелевые расширения электронной почты Интернета (Multipurpose Internet Mail Extensions, MIME) 302

модель обмена сообщениями публикация/подписка (publish/subscribe, pub/sub) 210
модель обмена сообщениями точка-точка (point-to-point, PTP) 210

Н

навигация, JSF 58

О

области безопасности
JDBC 267
LDAP 265
Solaris 266
введение 232

дополнительные области 262
сертификата 264
файла 262
область сертификата 253
настройка приложений 258
создание самоподписанных сертификатов 253
область файла 236
пользовательские 272
предопределенные 233
область сертификата 253
настройка приложений 258
создание самоподписанных сертификатов 253
область файла 236
стандартная аутентификация 238
облегченный протокол доступа к каталогам (Lightweight Directory Access Protocol, LDAP) 265
объектно реляционное отображение (Object Relational Mapping, ORM) 85
объекты доступа к данным (Data Access Objects, DAO) 128, 135
один к одному, отношения 93
один ко многим, отношения 99
отношения между сущностями
введение 92
многие ко многим 104
один к одному 93
один ко многим 99
очереди сообщений 216
асинхронный прием сообщений из очереди 221
асинхронный прием сообщений из темы 228
отправка сообщений 216
прием сообщений из очереди 219
прием сообщений из темы 226
просмотр сообщений 224
очереди сообщений JMS
настройка 213

П

параметры запроса 314
отправка 316
параметры пути 316
отправка 318
первичные ключи
составные 110, 113
требования 112
передача репрезентативного состояния (Representational State Transfer, REST) 301
пользовательская проверка допустимости 60
пользовательские области безопасности 272
предопределенные области безопасности 233
область сертификата 253
настройка приложений 258
создание самоподписанных сертификатов 253
область файла 236
проверка допустимости 53
проверка допустимости на стороне компонентов 126
проекты, этапы 50
модульное тестирование 51
промышленная эксплуатация 51
разработка 51
системное тестирование 51
простая веб-служба
создание 302
простой протокол доступа к объектам (Simple Object Access Protocol, SOAP) 279
простые старые объекты Java (Plain Old Java Objects, POJO) 90
пульсы соединений
настройка 37

С

сеансовые компоненты

асинхронные вызовы методов 140
введение 131
вызов из веб-приложений 137
компоненты-одиночки 139
пример 135
простой сеансовый компонент 131
сеансовые компоненты с сохранением состояния
жизненный цикл 151
методы интерфейса SessionBean 151
сертификат, самоподписанные
создание 253
сквозные элементы HTML 5 78
служба аутентификации и авторизации Java (Java Authentication and Authorization Service, JAAS) 232
служба обмена сообщениями Java (Java Messaging Service, JMS) 210
служба таймеров EJB (EJB Timer Service) 157
сообщения
асинхронный прием сообщений из очереди 221
асинхронный прием сообщений из темы 228
отправка в очередь 216
отправка в тему 225
прием сообщений из очереди 219
прием сообщений из темы 226
просмотр списка сообщений в очереди 224
сообщения по умолчанию, JSF 66
изменение текста 69
настройка стилей 67
составные первичные ключи 110, 113
требования 112
стандартное местоположение ресурсов 44
структура приложения 242
сущности, отношения
введение 92

многие ко многим 104
один к одному 93
один ко многим 99

Т
таймеры EJB на основе календаря 160
темы сообщений 225
отправка сообщений 225
темы сообщений JMS
настройка 214
транзакции
Enterprise JavaBeans 144
транзакции, управляемые компонентом (Bean-Managed Transactions) 148
транзакции, управляемые контейнером (Container-Managed Transactions) 144

Ф
фабрика соединений JMS
настройка 210
фейслеты
первое приложение JSF 45
фейслеты, JSF 42

Ц
центры сертификации 247

Э
этапы проекта 50
модульное тестирование 51
промышленная эксплуатация 51
разработка 51
системное тестирование 51

Я
язык запросов JPA (Java Persistence Query Language, JPQL) 115
язык определения веб-служб (Web Services Definition Language, WSDL) 283

A

actionListener, атрибут 74
 addMessage(), метод 47
 ADDRESS_TYPES, таблица 86
 add(), метод 186
 admin, группа 234
 Ajax, поддержка в JSF 71
 поддерживаемые события JavaScript 74
 Apache Commons Validator, библиотека 61
 appclient, утилита 166, 220
 asadmin, группа 234
 attachFile(), метод 292
 authenticateUser(), метод 275

B

BASIC, метод аутентификации 240
 Bean-Managed Transactions (транзакции, управляемые компонентом) 148
 begin(), метод 178
 blur, событие 74
 build(), метод 187
 BytesMessage, тип сообщений 218

C

cancel(), метод 158
 CDI
 введение 169
 внедрение зависимостей 171
 именованные компоненты 169
 квалификаторы 173
 CDI (Contexts and Dependency Injection
 контексты и внедрение зависимостей) 43
 change, событие 74
 click, событие 74
 CLIENT-CERT, метод аутентификации 240
 closeConnection(), функция JavaScript 204

commitUserAuthentication(), метод 275
 connectToServer(), метод 207
 Container-Managed Transactions (транзакции, управляемые контейнером) 144
 createBrowser(), метод 225
 createConnection(), метод 217
 createConsumer(), метод 220, 231
 createContext(), метод 220
 createCriteriaDelete(), метод 126
 createDurableSubscriber(), метод 231
 createJsonParser(), метод 194
 createNamedQuery(), метод 118
 createProducer(), метод 218
 createQuery(), метод 121
 createReader(), метод 190
 createTimer(), метод 158
 createWriter(), метод 187
 Criteria API
 введение 119
 изменение данных 122
 использование 119
 удаление данных 124
 CRUD, операции 85
 CSS (Cascading Style Sheets каскадные таблицы стилей) 67
 curl? команда 307
 CustomerDB, база данных
 ADDRESS_TYPES, таблица 86
 TELEPHONE_TYPES, таблица 86
 US_STATES, таблица 86
 введение 85
 customerinfo-return.xhtml, файл 81
 customerinfo.xhtml, файл 81
 служба сообщений Java (Java Message Service, JMS) 143
 сокеты 197

D

DAO (Data Access Objects объекты доступа к данным) 128, 135
 dblclick, событие 74

deleteCustomer(), метод 136
delete(), метод 313
DELETE, метод 302
Digest Algorithm, свойство 271
digest(), метод 269
DIGEST, метод аутентификации 240

E

EJB
веб-службы
безопасность 298
экспортирование в виде веб-служб 294
ejbActivate(), метод 153
ejbPassivate(), метод 153
ejbRemove(), метод 154
EJB Timer Service (служба таймеров EJB) 157
encryptPassword(), метод 269
end(), метод 178
Enterprise JavaBeans
безопасность 162
аутентификация клиентов 165
введение 130
жизненные циклы 150
компоненты без сохранения состояния 154
компоненты с сохранением состояния 151
компоненты, управляемые сообщениями 156
транзакции 144
EntityManager.createQuery(), метод 117
EntityManager.find(), метод 115
entity(), метод 313

F

faces-config.xml, файл 43
стандартное местоположение ресурсов 44
Faces Flows, JSF 2.2 80
страница подтверждения 83

findByPrimaryKey(), метод 91
focus, событие 74
FORM, метод аутентификации 240

G

getAuthType(), метод 273
getBasicRemote(), метод 199, 208
getBoolean(String name), метод 190
getCalculatorPort(), метод 289
getCriteriaBuilder(), метод 120, 124
getCustomer(), метод 305, 318
GetCustomer(), метод 136
getDeclaredSingularAttribute(), метод 121
getEnumeration(), метод 225
getGroupNames(), метод 273
getInfo(), метод 158
getInt(String name), метод 190
getJAASContext(), метод 274
getJSONArray(String name), метод 190
getJsonNumber(String name), метод 190
getJsonObject(String name), метод 190
getJsonString(String name), метод 190
getLabel(), метод 61
getNamePort(), метод 289
get(Object key), метод 190
getOpenSessions(), метод 199
getOrders(), метод 104
getReasonPhrase(), метод 207
getRequestContext(), метод 298
getResultList(), метод 117
getSingleResult(), метод 117
getString(String Name), метод 190
getString(), метод 195
getText(), метод 222
getTimers(), метод 158
getWebSocketContainer(), метод 207
get(), метод 313
GET, метод 302
GlassFish 18
URL 22
домены 33
номера портов 34

остановка 36
создание 34
удаление 35
зависимости 24
запуск, из командной строки 25
получение 22
предопределенные области безопасности 233
преимущества 21
развертывание приложений Java EE 26
установка 24

GlassFish, настройка
очереди сообщений 213
темы сообщений 214
фабрики соединений 210

Group Name Column, свойство 271

Group Table, свойство 271

H

hasNext(), метод 195

HTML5-совместимая разметка 76

I

ICEfaces
 URL 84

invalidate(), метод 251

isIntegralNumber()?, метод 196

isOpen(), метод 200

J

JAAS Context, поле 271

JAAS (Java Authentication and Authorization Service служба аутентификации и авторизации Java) 232

Java
 клиенты веб-сокетов 204

Java API для веб-служб RESTful (JAX-RS) 2.0 20

Java API для веб-сокетов 1.0 21

Java API для обработки JSON (JSON-P) 1.0 21

Java API для связывания с XML (Java API for XML Binding, JAXB) 309

Java EE 5 43

Java EE 7 197

Java EE 7, новые возможности 19

Java API для веб-служб RESTful (JAX-RS) 2.0 20

Java API для веб-сокетов 1.0 21

Java API для обработки JSON (JSON-P) 1.0 21

Java Message Service (JMS) 2.0 20

Java Persistence API (JPA) 2.1 20

JavaServer Faces (JSF) 19

Java EE, приложения
 развертывание 26

Java Message Service (JMS) 2.0 20

Java Persistence API 87

Java Persistence API (JPA) 2.1 20

java.security.MessageDigest, класс 268

JavaServer Faces (JSF) 19

javax.JMS.ConnectionFactory, тип ресурса 213

javax.jms.QueueBrowser, интерфейс 225

javax.JMS.QueueConnectionFactory, тип ресурса 213

javax.JMS.TopicConnectionFactory, тип ресурса 213

JAXB (Java API for XML Binding Java API для связывания с XML) 309

JAXB (Java Architecture for XML Binding архитектура Java для связывания с XML) 291

JAX-WS API 279

JDK 24
 URL 24

JMS (Java Message Service служба сообщений Java) 143

JMS (Java Messaging Service служба обмена сообщениями Java) 210

JPA 87

- Criteria API 119
JPQL 115
Metamodel API 121
введение 85
отношения между сущностями 92
проверка допустимости на стороне компонентов 126
составные первичные ключи 110
JPQL 115
JPQL (Java Persistence Query Language)
язык запросов JPA) 115
- JSF
faces-config.xml, файл 43
Faces Flows, JSF 2.2 80
страница подтверждения 83
введение 42
группировка компонентов 55
именованные компоненты CDI 56
навигация 58
нестандартные валидаторы 60
методы 63
создание 60
отправка формы 55
поддержка Ajax 71
поддерживаемые события JavaScript 74
пользовательская проверка допустимости 60
проверка допустимости 53
сообщения по умолчанию 66
изменение текста 69
настройка стилей 67
фейслеты 42
первое приложение 45
- JSF 2.2 HTML5
разметка 76
сквозные элементы 78
- JSON
Model API 185
Streaming API 185
введение 185
парсинг с использованием Model API 185
- парсинг с использованием Streaming API 193
создание с использованием Model API 186
создание с использованием Streaming API 191
- JsonGenerator, класс 191
JsonGenerator, методы
write(String name, BigDecimal value) 192
write(String name, BigInteger value) 193
write(String name, boolean value) 193
write(String name, double value) 193
write(String name, int value) 193
write(String name, JsonValue value) 193
write(String name, long value) 193
write(String name, String value) 193
- JSON (JavaScript Object Notation) форма записи объектов JavaScript 21
- JsonObjectBuilder, методы
add(String name, BigDecimal value) 188
add(String name, BigInteger value) 188
add(String name, boolean value) 188
add(String name, double value) 188
add(String name, int value) 188
add(String name, JSONArrayBuilder value) 188
add(String name, JsonObjectBuilder value) 188
add(String name, JsonValue value) 188
add(String name, long value) 188
add(String name, String value) 188
- JSON-P 21
- JsonParser, класс 194
JsonParser, методы
getBigDecimal() 196
getInt() 196
getLong() 196
- JSON-P Model API 186
JSON-P Streaming API 191
- K**
- keep, параметр 285

keydown, событие 74
 -keypass, параметр 253
 keypress, событие 74
 -keystore, параметр 253
 keytool, утилита 253
 keyup, событие 74

L

LDAP (Lightweight Directory Access Protocol облегченный протокол доступа к каталогам) 265
 like(), метод 122
 LoginInfo, поле 95
 LoginModule, класс 272
 logout(), метод 252

M

MapMessage, тип сообщений 218
 mappedName, атрибут 143, 218
 MessageReceiver, класс 228
 MessageSender, класс 217, 226, 228
 Metamodel API 121
 MIME (Multipurpose Internet Mail Extensions многоцелевые расширения электронной почты Интернета) 302
 mousedown, событие 74
 mousemove, событие 74
 mouseout, событие 74
 mouseover, событие 74
 mouseup, событие 74

N

newClient(), метод 313
 next(), метод 195

O

ObjectMessage, тип сообщений 218
 onError(), метод 200
 onMessage(), метод 222
 OrderItemPK, класс 113

ORM (Object Relational Mapping объектно-реляционное отображение) 85

P

path(), метод 319
 POJO (Plain Old Java Objects простые старые объекты Java) 90
 --portbase, параметр 35
 post(), метод 313
 POST, метод 302
 Primefaces
 URL 84
 processMessage(), метод 199
 produceMessages(), метод 217
 property, параметр 266
 put(), метод 298
 put(), метод 313
 PUT, метод 302

Q

queryParam(), метод 316

R

Realm, класс 272
 reciveBody(), метод 220
 request(), метод 313
 RESTful, веб-службы
 DELETE, метод 302
 GET, метод 302
 POST, метод 302
 PUT, метод 302
 настройка пути к ресурсам 305
 с аннотацией @ApplicationPath 305
 преобразование данных XML с JAXB 308
 создание клиентов 312
 тестирование с curl 306

REST (Representational State Transfer
 передача репрезентативного состояния) 301

RichFaces

URL 84
rollback(), метод 150

S

saveCustomer(), метод 136, 148
-savemasterpassword=true, параметр 257
saveMultipleNewCustomers(), метод 150
saveNewCustomer(), метод 148
select, событие 74
sendMessage(), метод 208
sendMessage(), функция JavaScript 204
sendText(), метод 199, 208
send(), метод 218
SessionBeanClient, класс 134
setCustomerId(), метод 90
setCustomer(), метод 104
setItems(), метод 110
setParameter(), метод 117
setRollBackOnly(), метод 148
set(), метод 124
SOAP (Simple Object Access Protocol
простой протокол доступа к объ-
ектам) 279
-storepass, параметр 253
StreamMessage, тип сообщений 219
styleClass, атрибут 67

T

target(), метод 313
TELEPHONE_TYPES, таблица 86
TextMessage, тип сообщений 219
toString(), метод 180, 187
TransactionAttributeType.MANDATORY,
значение 145
TransactionAttributeType.NEVER, значе-
ние 145
TransactionAttributeType.NOT_
SUPPORTED, значение 145
TransactionAttributeType.REQUIRED,
значение 146
TransactionAttributeType.REQUIRES_
NEW, значение 146

TransactionAttributeType.SUPPORTS,
значение 146

U

updateCustomer(), метод 148
User Name Column, свойство 271
User Table, свойство 271
US_STATES, таблица 86

V

validate(), метод 61, 64
validator, атрибут 64
valueChange, событие 74

W

websocketError(), функция JavaScript
204
webSocketMessage(), функция JavaScript
204
websocketOpen(), функция JavaScript
204
writeEnd(), метод 193
writeObject(), метод 187
writeStartObject(), метод 192
write(), метод 192, 193
WSDL (Web Services Definition Language
язык определения веб-служб)
283
wsimport, утилита 285, 292

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.**

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: www.aliants-kniga.ru.

Оптовые закупки: тел. +7 (499) 782-38-89.

Электронный адрес: books@aliants-kniga.ru.

Дэвид Хеффельфингер

Java EE 7 и сервер приложений GlassFish 4

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Перевод с английского *Киселев А. Н.*

Корректор *Синяева Г. И.*

Верстка *Паранская Н. В.*

Дизайн обложки *Мовчан А. Г.*

Формат 60×90 $\frac{1}{16}$. Гарнитура «Петербург».

Печать офсетная. Усл. печ. л. 20,33.

Тираж 200 экз.

Веб-сайт издательства: www.dmk.ru