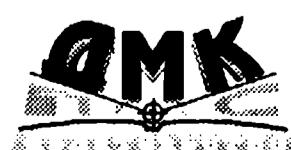


Дэвид Хеффельфингер

**Java EE 6
и сервер приложений
GlassFish 3**



Москва, 2013

Java EE 6 with GlassFish 3 Application Server

A practical guide to install and configure the GlassFish 3 Application Server and develop Java EE 6 applications to be deployed to this server

David R. Heffelfinger



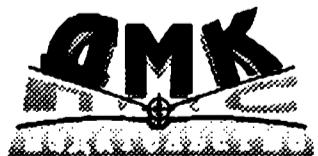
open source 
community experience distilled

BIRMINGHAM - MUMBAI

Java EE 6 и сервер приложений GlassFish 3

Практическое руководство по установке и конфигурированию сервера приложений GlassFish 3, а также по разработке приложений Java EE 6 и их развертыванию на этом сервере

Дэвид Хеффельфингер



УДК 004.438Java ЕЕ

ББК 32.973.26-018.2

X41

Дэвид Хеффельфингер

X41 Java EE 6 и сервер приложений GlassFish 3. Пер. с англ.: Карышев Е.Н. – М.: ДМК Пресс, 2013. – 416 с.: ил.

ISBN 978-5-94074-902-8

Книга представляет собой практическое руководство с очень удобным подходом, позволяющим читателю быстрее освоить технологии Java EE 6. Все рассмотренные основные интерфейсы Java EE 6 и подробная информация о сервере GlassFish 3 подкреплены практическими примерами их использования.

Платформа Java Enterprise Edition (Java EE) 6 является отраслевым стандартом для корпоративных вычислений Java, а сервер приложений GlassFish представляет собой эталонную среду реализации спецификации Java EE. В книге рассматриваются различные соглашения и аннотации Java EE 6, 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 3 для развертывания корпоративных приложений.

Настоящее издание предназначено для разработчиков Java, желающих стать специалистами в разработке корпоративных приложений с использованием платформы Java EE 6. 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.

Об авторе

Дэвид Хеффельфингер (David R. Heffelfinger) является техническим директором Enscape 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. Он использует Java в качестве основного языка программирования с 1996 года. Ему довелось работать во многих крупномасштабных проектах для ряда клиентов, в числе которых были 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 Дэвид является главным редактором Enscape.net (<http://www.enscape.net>), веб-сайта, посвященного Java, Linux и другим технологиям.

Я хотел бы поблагодарить всех, кто помогал мне в создании этой книги. Выражаю благодарность редакторам Мехулу Шетти (Mehul Shetty) и Дхираджу Чандирамани (Dhiraj Chandiramani), а также координаторам проекта Шубханджану Чаттерджи (Shubhanjan Chatterjee) и Паллаби Чаттерджи (Pallabi Chatterjee).

Отдельное спасибо техническим рецензентам Аллану Бонду (Allan Bond) и Аруну Гупте (Arun Gupta) за их важные замечания и предложения.

Кроме того, хочу поблагодарить группу GlassFish в Oracle (ранее Sun Microsystems) за разработку такого замечательного сервера приложений с открытым исходным кодом.

И, наконец, я признателен моей жене и дочери за их терпеливое отношение к моей дополнительной работе, в силу которой я был вынужден уделять меньше времени семье.

О рецензентах

Аллан Бонд (Allan Bond) – разработчик программного обеспечения, более 10 лет работающий в сфере ИТ. Его основное внимание сосредоточено на разработке систем с использованием Java и связанных с ней технологий. В процессе работы Аллану довелось консультировать множество организаций, от малых предприятий до компаний, входящих в список Fortune 500, а также государственных учреждений. Он имеет степень магистра в области управления информационными системами Университета Бригама Янга.

Я хотел бы поблагодарить свою жену и детей за их терпение вочные часы (а иногда и выходные дни), когда мне нужно было завершить рецензирование этой книги.

Арун Гупта (Arun Gupta) – апологет Java EE и GlassFish, работающий в компании Oracle. Арун имеет более чем 14-летний опыт в индустрии программного обеспечения и работает с платформой Java^(TM) и несколькими связанными с ней интернет-технологиями. В настоящий момент его деятельность направлена на создание и укрепление сообщества пользователей Java EE 6 и GlassFish. Арун сотрудничал с несколькими организациями по стандартизации и осуществлял взаимодействие с представителями других компаний. Он входил в команду Java EE начиная с момента ее создания и в той или иной степени внес свой вклад во все релизы Java EE. Имеет обширный опыт обсуждения множества тем в международном сообществе и обожает широкомасштабные проекты.

Арун является активным блогером (<https://blogs.oracle.com/arungupta/>). В этом блоге более 1000 записей, и его часто читают посетители со всего мира; посещаемость блога достигает 25 000 посещений в день.

Содержание

Об авторе	5
О рецензентах	6
Содержание	7
Предисловие	16
Темы, освещаемые в книге	16
Что нужно для чтения этой книги	19
Для кого эта книга	19
Соглашения	19
Поддержка клиентов	20
Сообщения об ошибках	20
Незаконное воспроизведение содержимого книги	21
Вопросы	21
1. Знакомство с сервером GlassFish	22
Общий обзор Java EE и GlassFish	22
Новые возможности Java EE	23
JavaServer Faces (JSF) 2.0	23
Enterprise JavaBeans (EJB) 3.1	23
API Персистентности Java (JPA) 2.0	24
Контексты и инжекция зависимости для Java (Web Beans 1.0)	24
API Java Сервлета 3.0	24
API Java для веб-сервисов RESTful (JAX-RS) 1.1	25
API Java для веб-сервисов XML (JAX-WS) 2.2	25
Архитектура Java для связывания с XML (JAXB) 2.2	25
Новые возможности GlassFish v.3	25
Преимущества GlassFish	26

Получение GlassFish	27
Установка GlassFish	28
Зависимости GlassFish	28
Выполнение установки	29
Проверка установки	34
Развертывание нашего первого приложения Java EE	35
Развертывание приложения через веб-консоль	35
Отмена развертывания приложения через веб-консоль	37
Развертывание приложения с помощью командной строки	38
Каталог autodeploy	39
Утилита командной строки asadmin	40
Домены GlassFish	41
Создание доменов	41
Удаление доменов	43
Остановка домена	43
Настройка подключения к базе данных	43
Создание пулов соединений	44
Создание источников данных	48
Заключительные замечания	49
Резюме	49
2. Разработка и развертывание сервлета	50
Что такое сервлет?	50
Написание нашего первого сервлета	51
Компиляция сервлета	52
Конфигурирование сервлета	53
Упаковка веб-приложения	55
Развертывание веб-приложения	56
Тестирование веб-приложения	56
Обработка HTML-форм	58
Переадресация запросов и перенаправление откликов	64
Переадресация запроса	64
Перенаправление отклика	67

Сохранение данных приложения между запросами	69
Новые возможности, появившиеся в Сервлете 3.0	71
Необязательный дескриптор развертывания web.xml	71
Аннотация @WebServlet	71
Передача сервлету параметров инициализации через аннотации .	73
Аннотация @WebFilter	74
Аннотация @WebListener	76
Подключаемость	78
Программное конфигурирование веб-приложений	80
Асинхронная обработка	82
Резюме	84
3. JavaServer Pages	85
Введение в JavaServer Pages	85
Разработка нашей первой JSP-страницы	86
Неявные объекты JSP	90
JSP и JavaBeans	97
Повторное использование JSP-контента	101
Пользовательские теги JSP	103
Расширение класса SimpleTagSupport	104
Использование файлов тегов для создания	
пользовательских тегов JSP	109
Унифицированный язык выражений	113
XML-синтаксис JSP	116
Резюме	118
4. Библиотека стандартных тегов JSTL	119
JSTL-библиотека базовых тегов	119
JSTL-библиотека тегов форматирования	127
JSTL-библиотека SQL-тегов	131
JSTL-библиотека XML-тегов	136
Функции JSTL	140
Резюме	143

5. Подключение к базе данных	144
База данных CUSTOMERDB	144
JDBC	145
Извлечение данных из базы данных	146
Изменение информации в базе данных	152
API Персистентности Java	154
Отношения сущности	159
Отношения «один к одному»	159
Отношения «один ко многим»	164
Отношения «многие ко многим»	168
Составные первичные ключи	173
Язык запросов персистентности Java	177
Новые функции, введенные в JPA 2.0	181
API Критериев	181
Поддержка проверки допустимости со стороны бинов	184
Резюме	186
6. Java Server Faces	187
Введение в JSF 2.0	187
Фэйслеты	187
Необязательный файл faces-config.xml	188
Стандартное расположение ресурсов	188
Разработка нашего первого JSF 2.0-приложения	189
Фэйслеты	189
Этапы проекта	193
Проверка допустимости	195
Группировка компонентов	197
Отправка формы	197
Управляемые бины	198
Контексты управляемых бинов	199
Навигация	200
Пользовательская проверка допустимости данных	201
Создание нестандартных элементов верификации	202
Методы блока проверки допустимости	204

Настройка сообщений JSF по умолчанию	207
Настройка стилей сообщения	207
Настройка текста сообщения	209
Интеграция JSF и JPA	211
Включение Ajax в приложения JSF 2.0	218
Стандартные компоненты JSF	222
Базовые компоненты JSF	222
Тег <f:actionListener>	222
Тег <f:ajax>	222
Тег <f:attribute>	223
Тег <f:convertDateTime>	223
Тег <f:convertNumber>	223
Тег <f:converter>	224
Тег <f:event>	224
Тег <f:facet>	225
Тег <f:loadBundle>	225
Тег <f:metadata>	225
Тег <f:param>	226
Тег <f:phaseListener>	226
Тег <f:selectItem>	226
Тег <f:selectItems>	226
Тег <f:setPropertyActionListener>	227
Тег <f:subview>	227
Тег <f:validateBean>	227
Тег <f:validateDoubleRange>	228
Тег <f:validateLength>	228
Тег <f:validateLongRange>	228
Тег <f:validateRegex>	229
Тег <f:validateRequired>	229
Тег <f:validator>	229
Тег <f:valueChangeListener>	229
Тег <f:verbatim>	230
Тег <f:view>	230
Тег <f:viewParam>	230

HTML-компоненты JSF	231
Тег <h:body>	231
Тег <h:button>	231
Тег <h:column>	231
Тег <h:commandButton>	231
Тег <h:commandLink>	232
Тег <h:dataTable>	232
Тег <h:form>	232
Тег <h:graphicImage>	233
Тег <h:head>	233
Тег <h:inputHidden>	233
Тег <h:inputSecret>	233
Тег <h:inputText>	234
Тег <h:inputTextarea>	234
Тег <h:link>	234
Тег <h:message>	234
Тег <h:messages>	235
Тег <h:outputFormat>	235
Тег <h:outputLabel>	235
Тег <h:outputLink>	236
Тег <h:outputScript>	236
Тег <h:outputStylesheet>	236
Тег <h:outputText>	236
Тег <h:panelGrid>	236
Тег <h:panelGroup>	237
Тег <h:selectBooleanCheckbox>	238
Тег <h:selectManyCheckbox>	238
Тег <h:selectManyListbox>	238
Тег <h:selectManyMenu>	239
Тег <h:selectOneListbox>	239
Тег <h:selectOneMenu>	239
Тег <h:selectOneRadio>	239
Дополнительные библиотеки компонентов JSF	240
Резюме	240

7. Служба обмена сообщениями Java	241
Настройка GlassFish для использования JMS	241
Создание фабрики JMS-соединений	241
Создание очереди JMS-сообщений	243
Создание темы JMS-сообщений	244
Очереди сообщений	245
Отправка сообщений в очередь сообщений	245
Извлечение сообщений из очереди сообщений	249
Асинхронный прием сообщений из очереди сообщений	250
Просмотр очередей сообщений	253
Темы сообщений	254
Отправка сообщений теме сообщений	254
Получение сообщений от темы сообщений	255
Создание долговременных подписчиков	257
Резюме	260
8. Безопасность	261
Области безопасности	261
Предопределенные области безопасности	262
Область администратора	262
Область файла	264
Стандартная аутентификация через область файла	265
Область сертификата	276
Создание самоподписанных сертификатов	276
Конфигурирование приложений для использования области сертификата	280
Определение дополнительных областей	283
Определение дополнительных областей файла	283
Определение дополнительных областей сертификата	285
Определение области LDAP	286
Определение области Solaris	287
Определение области JDBC	288
Определение пользовательских областей	293
Резюме	298

9. Enterprise JavaBeans	299
Сеансовые бины	300
Простой сеансовый бин	300
Более реалистический пример	303
Вызов сеансовых бинов из веб-приложений	305
Одноэлементный сеансовый бин (Singleton)	306
Асинхронные вызовы метода	307
Управляемые сообщением бины	309
Транзакции в Enterprise JavaBeans	310
Транзакции, управляемые контейнером	311
Транзакции, управляемые бином	313
Жизненный цикл Enterprise JavaBeans	315
Жизненный цикл сеансового бина с сохранением состояния	316
Жизненный цикл сеансового бина, не сохраняющего состояние ..	319
Жизненный цикл управляемых сообщением бинов	321
Служба таймера EJB	322
Выражения таймера EJB на основе календаря	324
Безопасность EJB	326
Аутентификация клиента	329
Резюме	330
10. Контексты и инжекция зависимости	332
Именованные бины	332
Инжекция зависимости	334
Квалификаторы	335
Контексты именованных бинов	338
Резюме	345
11. Веб-сервисы JAX-WS	346
Разработка веб-сервисов JAX-WS	346
Разработка клиента веб-сервиса	351
Отправка вложений веб-сервисам	357

Представление EJB как веб-сервисов	359
Клиенты веб-сервиса EJB	360
Безопасность веб-сервисов	360
Безопасность веб-сервисов EJB	362
Резюме	364
12. RESTful веб-сервисы в Jersey и JAX-RS	365
Введение в веб-сервисы RESTful и JAX-RS	365
Разработка простого веб-сервиса RESTful	366
Конфигурирование пути к ресурсам REST для нашего приложения	368
Конфигурирование через web.xml	368
Конфигурирование через аннотацию @ApplicationPath	369
Тестирование нашего веб-сервиса	370
Преобразование данных между Java и XML с помощью JAXB	372
Разработка клиента веб-сервиса RESTful	375
Параметры запроса и пути	377
Параметры запроса	377
Отправка параметров запроса через клиентский API Jersey	378
Параметры пути	380
Отправка параметров пути через клиентский API Jersey	382
Приложение А: Отправка электронной почты из приложений Java EE	384
Конфигурирование сервера GlassFish	384
Реализация функциональности доставки электронной почты	387
Приложение Б: Интеграция с IDE	390
NetBeans	390
Eclipse	392
Алфавитный указатель	397

Предисловие

Изложение материала в данной книге начинается с установки сервера GlassFish v.3 и развертывания Java-приложений. Затем объясняется, как разработать, сконфигурировать, упаковать и развернуть сервлеты; помимо прочего, уделяется внимание изучению обработки HTML-форм. По мере продвижения в изучении материала мы будем разрабатывать Серверные страницы Java (Java Server Pages (JSP)) и узнаем о неявных объектах JSP. Также мы познакомимся со всеми Библиотеками стандартных тегов JSP (JSP Standard Tag Library (JSTL)). Эта книга позволит нам лучше понять, как управлять данными, хранящимися в базе данных, через API Подключения к базе данных Java (Java Database Connectivity (JDBC)) и через API Персистентности Java (Java Persistence API (JPA)). Кроме того, мы узнаем о новых функциях, введенных в JPA 2.0, и разработаем приложения JSF 2.0 для их изучения и настройки. Затем настроим сервер GlassFish для работы с API Системы обмена сообщениями Java (Java Messaging System (JMS)) и изучим работу тем и очередей сообщений. Позже мы будем использовать API Контекстов и инжекции зависимости (Context and Dependency Injection (CDI)) для интегрирования различных уровней приложения, а также изучим веб-сервис на основе SOAP, используя для его разработки спецификацию JAX-WS. Наконец, мы узнаем больше о разработке веб-сервиса RESTful с использованием спецификации JAX-RS.

Наконец, в книге обсуждаются различные соглашения и аннотации Java EE 6, которые помогут упростить разработку корпоративных приложений Java. Также рассматриваются самые последние версии спецификаций Сервлета (Servlet), JSF, JPA, EJB и JAX-WS, а также новые дополнения к спецификации, такие как JAX-RS и CDI.

Темы, освещаемые в книге

В главе 1, «*Знакомство с сервером GlassFish*», объясняется, как загрузить и установить сервер GlassFish. Здесь мы рассмотрим несколько методов развертывания приложений Java EE: через веб-консоль GlassFish, с помощью утилиты командной строки `asadmin` и путем копирования файла приложения в каталог авторазвертывания. Мы рассмотрим основные задачи администрирования GlassFish, такие как настройка доменов и настройка соединений с базой данных, добавление пулов соединений и источников данных.

В главе 2, «*Разработка и развертывание сервлета*», показано, как разработать, сконфигурировать, упаковать и развернуть сервлеты. Также мы рассмотрим, как обра-

ботать информацию HTML-формы, получая доступ к объекту HTTP-запроса. В дополнение будет объяснена переадресация HTTP-запросов от одного сервлета к другому, наряду с перенаправлением HTTP-отклика на другой сервер. Мы обсудим, как сохранить объекты в памяти с помощью запросов, присоединяя их к контексту сервлета и HTTP-сессии. Наконец, мы рассмотрим все важные новые функции Сервлета 3.0, включая конфигурирование веб-приложений с помощью аннотаций, подключаемость с помощью `web-fragment.xml`, программное конфигурирование сервлета и асинхронную обработку.

Глава 3, «*JavaServer Pages*», содержит сведения о том, как разработать и развернуть простую JSP-страницу. Мы рассмотрим, как получить доступ к неявным объектам, таким как запрос, сеанс и т. д., из JSP-страницы. Кроме того, мы выясним, как установить и получить значения свойств JavaBean с помощью тега `<jsp:useBean>`. Поговорим и о том, как включить одну JSP-страницу в другую во время выполнения с помощью тега `<jsp:include>` и во время компиляции с помощью директивы JSP `include`. Мы обсудим, как написать пользовательские JSP-теги, расширяющие `javax.servlet.jsp.tagext.SimpleTagSupport` или TAG-файлы. Также будет показано, как получить доступ к JavaBeans и их свойствам с помощью Унифицированного языка выражений (Unified Expression Language). Наконец, мы рассмотрим XML-синтаксис JSP, который позволяет нам разрабатывать XML-совместимые JSP-страницы.

Глава 4, «*Библиотека стандартных тегов JSP*», познакомит нас со всеми Библиотеками стандартных тегов JSP (JSP Standard Tag Library (JSTL)), включая библиотеки базовых тегов, тегов форматирования, SQL- и XML-тегов. Кроме этого, будут объяснены функции JSTL. В главе приведены примеры, поясняющие использование наиболее распространенных тегов JSTL; также упоминаются и описываются дополнительные JSTL-теги.

В главе 5, «*Подключение к базе данных*», рассказывается о том, как получить доступ к данным в базе данных с помощью API Подключения к базе данных Java (Java Database Connectivity (JDBC)) и через API Персистентности Java (Java Persistence API (JPA)). Будут рассмотрены определения односторонних и двунаправленных отношений «один к одному», «один ко многим» и «многие ко многим» между JPA-сущностями. Помимо прочего, мы обсудим, как использовать первичные ключи составной JPA-сущности путем разработки пользовательских классов первичного ключа. Также будет показано, как получить объекты из базы данных путем использования Языка запросов персистентности Java (Java Persistence Query Language (JPQL)). Вы узнаете, как создавать программные запросы с помощью API Критерий JPA 2.0 (JPA 2.0 Criteria API) и автоматизировать проверку допустимости данных с помощью поддерживаемой JPA 2.0 Проверки допустимости со стороны бинов (Bean Validation Support).

Глава 6, «*JavaServer Faces*», посвящена разработке веб-приложений с использованием JavaServer Faces – стандартного каркаса компонентов для платформы Java EE 5. Мы поговорим о том, как написать простое приложение, создав JSP-страницу, содержащую JSF-теги и управляемые бины (managed beans). Обсудим, как проверить

данные, вводимые пользователем, путем использования стандартных блоков проверки допустимости JSF и создаваемых нами собственных блоков проверки допустимости или путем написания методов блока проверки допустимости. Рассмотрим, как настроить стандартные сообщения об ошибках JSF – и текст сообщения, и стиль сообщения (шрифт, цвет и т. д.). Наконец, узнаем, как написать приложения, интегрирующие технологии JSF и API Персистентности Java (JPA).

Глава 7, «*Служба обмена сообщениями Java*», повествует о том, как настроить в GlassFish фабрики соединений JMS, очереди и темы сообщений JMS, используя веб-консоль GlassFish. Мы рассмотрим, как отправлять и получать сообщения в очередь и из очереди сообщений JMS. Обсудим, как отправлять и получать сообщения в тему и из темы сообщений JMS. Узнаем, как просмотреть сообщения в очереди сообщений не удаляя их из очереди. В завершение будет показано, как создать и настроить темы JMS и взаимодействовать с долговременными подписчиками (*durable subscriptions*) на них.

Глава 8, «*Безопасность*», рассказывает о том, как использовать для аутентификации наших веб-приложений установленные по умолчанию области (realms) GlassFish. Мы рассмотрим область файла (file realm), которая хранит пользовательскую информацию в плоском файле, и область сертификата (certificate realm), которая требует клиентских сертификатов для пользовательской аутентификации. Кроме того, обсудим, как создать дополнительные области аутентификации, которые ведут себя точно так же, как области по умолчанию, при использовании классов области, включенных в GlassFish.

Глава 9, «*Enterprise JavaBeans*», объясняет, как реализовать бизнес-логику с помощью сохраняющих и не сохраняющих состояние сеансовых бинов (session beans). Кроме того мы разъясним понятия транзакций, управляемых контейнером, и транзакций, управляемых бином. Будут рассмотрены жизненные циклы для различных типов Enterprise JavaBeans. Мы поговорим о том, как периодически вызывать методы EJB – контейнером EJB, используя возможности службы таймера EJB. Наконец, мы расскажем, как сделать так, чтобы EJB методы могли быть вызваны только авторизованными пользователями.

Глава 10, «*Контексты и инжекция зависимости*», расскажет нам о том, как JSF страница, может получить доступ к именованным бинам (named beans) CDI, как будто они являются управляемыми бинами (managed beans) JSF. Мы объясним, как CDI облегчает внедрение (инжекцию) зависимостей в наш код. Мы обсудим, как можно использовать квалифиликаторы (qualifiers), чтобы определить, какие конкретно реализации зависимостей инжектировать в наш код. Наконец, мы рассмотрим все контексты, в которые может быть помещен бин CDI.

Глава 11, «*Веб-сервисы JAX-WS*», рассматривает, как разработать веб-сервисы и клиентов веб-сервисов с помощью API JAX-WS. Мы обсудим, как отправлять вложения веб-сервису, как представить методы EJB в качестве веб-сервисов и, наконец, как обеспечить безопасность веб-сервисов, чтобы они не были доступны неавторизованным клиентам.

В главе 12, «*RESTful веб-сервисы в Jersey и JAX-RS*», обсуждается, как легко и быстро разработать RESTful веб-сервисы, используя API JAX-RS – новое дополнение к спецификации Java EE. Мы объясним, как автоматически преобразовать данные между Java и XML, используя возможности API Java для связывания с XML (Java API for XML Binding (JAXB)). Наконец, мы рассмотрим, как передать параметры нашим RESTful веб-сервисам с помощью аннотаций @PathParam и @QueryParam.

Что нужно для чтения этой книги

Для чтения этой книги потребуется установить Комплект разработчика Java (Java Development Kit (JDK)) 1.5 или более новую версию, а также GlassFish v.3 или v.3.1. Настоятельно рекомендуется установить Maven 2, поскольку он используется во всех примерах кода, приведенного в книге. Наличие IDE Java, таких как NetBeans, Eclipse или IntelliJ IDEA, необязательно.

Для кого эта книга

Если Вы являетесь разработчиком Java и хотите стать специалистом по Java EE 6, эта книга для Вас. Чтобы чтение пошло Вам на пользу, необходимо иметь некоторый опыт работы с Java, а также опыт разработки и развертывания собственных приложений, однако никаких предварительных знаний о Java EE или J2EE не требуется. Вы также изучите, как использовать сервер приложений GlassFish v.3 для разработки и развертывания приложений.

Соглашения

В этой книге используется несколько стилей и пометок для выделения особо важной информации. Ниже мы поясним все эти типы выделений.

Элементы кода в тексте обозначаются моноширинным шрифтом, например: «XML-теги < servlet > и < servlet-mapping > используются для фактического конфигурирования нашего сервлета».

Блоки кода представлены следующим образом:

```
<servlet-mapping>
    <servlet>SimpleServlet</servlet>
    <url-pattern>*.foo</url-pattern>
</servlet-mapping>
```

Если мы хотим привлечь ваше внимание к определенной части блока кода, соответствующие фрагменты выделяются еще и жирным шрифтом:

```
<b>Application Menu</b>
<ul>
    <li/> <a href="main.jsp">Main</a>
    <li/> <a href="secondary.jsp">Secondary</a>
</ul>
Current page: <%= pageName %>
```

Ввод или вывод командной строки записывается так:

```
javac -cp /opt/s ges-v3/glassfish/lib/javaee.jar  
net/ensode/glassfishbook/simpleapp/ SimpleServlet.java
```

Важные (ключевые) слова в тексте выделяются курсивом.

Элементы интерфейса программы, например пункты меню или поля в диалоговых окнах, отмечены жирным шрифтом: «Щелкните по кнопке Далее (Next)».

Для обозначения последовательно выполняемых действий используется символ | (вертикальная черта): «Щелкните по узлам Ресурсы (Resources) | JDBC | Пулы соединений (Connection Pools) в панели навигации».

Для удобства восприятия перечисляемые элементы оформлены в виде маркированных списков, например:

Элемент <tag> содержит несколько подэлементов:

- подэлемент <name>, который присваивает логическое имя пользовательскому тегу;
- подэлемент <tag-class>, который идентифицирует полностью определенное (квалифицированное) имя пользовательского тега;
- один или более подэлементов <attribute>, которые определяют атрибуты пользовательского тега.

Веб-адреса выделяются подчеркиванием и специальным шрифтом, например: «Мы сможем увидеть отображение файла `dataentry.html`, вводя в адресной строке обозревателя `http://localhost:8080/formhandling`».



Предупреждения или важные примечания отмечены в тексте таким образом.



Советы и рекомендации обозначены так.



Символ в правом нижнем углу нечетной страницы указывает что листинг кода имеет продолжение на следующем развороте страниц.

Поддержка клиентов

Теперь, когда Вы являетесь счастливым обладателем книги, у нас имеется возможность помочь Вам извлечь максимум пользы из Вашего приобретения.



Загрузите примеры кода для этой книги

Файлы с примерами кода для данной книги можно загрузить с сайта издательства :

<http://www.dmk-press.ru/>

Сообщения об ошибках

Хотя мы делаем все возможное для того, чтобы не допустить ошибок в наших изданиях, Вы можете встретить в тексте какие-либо неточности. В таком случае мы будем рады, если Вы сообщите нам об этом.

Сообщения об ошибках в русскоязычном издании этой книги можно оставить на сайте издательства «ДМК Пресс»: <http://www.dmk-press.ru/contacts1/contacts/>.

Незаконное воспроизведение содержимого книги

Пиратские копии – повсеместная проблема. Если Вам встретились незаконным образом растиражированные экземпляры данной книги (в любом формате), пожалуйста, сообщите источник публикации, написав письмо по адресу dm@dmk-press.ru или copyright@packtpub.com.

Вопросы

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

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 сервера приложений GlassFish;
- проверка установки сервера GlassFish;
- развертывание приложения Java EE;
- установка соединения с базой данных.

Общий обзор Java EE и GlassFish

Спецификация Java EE (ранее называемая J2EE) включает в себя стандартный набор технологий для разработки серверных приложений Java. Технологии Java EE включают, среди прочего, Сервлеты (Servlets), Серверные страницы Java (Java Server Pages (JSP)), Каркас стандартных компонентов Java Server Faces (JSF), Компоненты корпоративных приложений (Enterprise Java Beans (EJB)), Службу обмена сообщениями Java (Java Messaging Service (JMS)), API Персистентности Java (Java Persistence API (JPA)), используемый для сохранения сущностей в базе данных, API Java для веб-сервисов XML (Java API for XML Web Services (JAX-WS)) и API Java для веб-сервисов RESTful (Java API for RESTful Web Services (JAX-RS)). Существует несколько коммерческих вариантов серверов приложений и несколько вариантов серверов приложений с открытым исходным кодом. Серверы приложений Java EE позволяют разработчикам разрабатывать и развертывать Java EE-совместимые приложения; одним из таких серверов приложений является сервер GlassFish. В числе других серверов приложений Java EE с открытым исходным кодом – JBoss Red Hat, Apache Software Foundation's Geronimo и ObjectWeb JOnAS. Коммерческие варианты сервера – Oracle (ранее BEA) Weblogic, IBM Websphere и Oracle Application Server.

GlassFish является сервером приложений Java EE с открытым исходным кодом, находящимся в свободном доступе. GlassFish лицензируется в соответствии с Общей лицензией разработки и распространения (Common Development and Distribution License (CDDL)).



Чтобы узнать больше о лицензиях GlassFish, обратитесь к сайту http://glassfish.java.net/public/CDDL+GPL_1_1.html.

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

Новые возможности Java EE

Java EE 6 на сегодняшний день 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) 2.0

Java EE 6 включает новую версию JSF. В каркасе стандартных компонентов JSF 2.0 появились следующие важные возможности:

- JSF 2.0 принял фэйслеты (Facelets) в качестве части официальной спецификации. Фэйслеты – технология представления, специально разработанная для JSF. В число преимуществ фэйслетов входят следующие возможности: возможность определения представления в XHTML, возможность упрощения создания шаблонов и возможность разработки компонентов JSF только с использованием разметки – без использования кода Java;
- JSF 2.0 также включает возможность конфигурирования приложения JSF с использованием аннотаций, значительно снижая, таким образом, а во многих случаях и полностью устранивая необходимость использования XML для конфигурирования.

Enterprise JavaBeans (EJB) 3.1

Ранние версии спецификации EJB приобрели репутацию сложных в практическом применении.

EJB 3.0 сделал большой шаг по пути к упрощению разработки EJB. EJB 3.1, в свою очередь, добавил новые возможности, которые еще больше упростили разработку EJB, а именно:

- локальные интерфейсы теперь являются не обязательными, поскольку фактический экземпляр бина может быть инжектирован (*injected*) в локального клиента;
- одиночный (*Singleton*) сеансовый бин может использоваться для управления состояниями приложения;

- сеансовые бины теперь могут быть вызваны асинхронно, позволяя нам использовать сеансовые бины для задач, которые ранее были зарезервированы для JMS и управляемых сообщением бинов;
- улучшенная служба таймера EJB теперь позволяет нам планировать задания декларативно через аннотации;
- Enterprise JavaBeans могут быть упакованы в файл веб-архива (Web ARchive (WAR)). Эта возможность значительно упрощает упаковку EJB, поскольку ранее требовался файл корпоративного архива (Enterprise ARchive (EAR)) для совместной упаковки в один модуль веб-функциональности и EJB-функциональности.

API Персистентности Java (JPA) 2.0

JPA был введен в качестве части стандартной спецификации Java EE в версии 5. JPA был призван заменить Сущностные бины (Entity Beans), использовавшиеся на тот момент в качестве стандартного каркаса для объектно-реляционного отображения в Java EE. JPA перенял идеи сторонних объектно-реляционных каркасов, таких как Hibernate, JDO и т. д., и сделал их частью стандарта.

JPA 2.0 в сравнении с JPA 1.0 улучшен по нескольким направлениям:

- несущностные коллекции теперь могут быть сохранены с помощью использования аннотаций: `@ElementCollection` и `@CollectionTable`;
- запросы JPA могут быть созданы с помощью нового API Критериев, уменьшая тем самым зависимость от JPQL;
- улучшен язык запросов JPA (JPQL) путем добавления поддержки для SQL-подобных выражений CASE, а также операторов NULLIF и COALESCE.

Контексты и инжекция зависимости для Java (Web Beans 1.0)

Контексты (Context) и инжекция (Injection) зависимости представляют собой API, который помогает упростить разработку корпоративных приложений. Этот API помогает унифицировать веб- и транзакционные уровни приложения Java EE. Например, контексты и инжекция зависимости позволяют использовать Enterprise JavaBeans (EJB) в качестве управляемых бинов JSF.

API Java Сервлета 3.0

Сервлеты являются строительными блоками всех веб-приложений Java. Ранние веб-приложения Java обращались к API сервлета напрямую. За прошедшие годы было создано несколько API-надстроек над API сервлета, некоторые из них – в рамках стандарта, а некоторые – сторонними разработчиками. Все каркасы веб-приложений Java, такие как JSF, Struts, Wicket, Tapestry и т. д., опираются на API сервлета, вы-

полняя его работу «за кулисами». API самого сервлета не сильно изменился за прошедшее время. Java EE 6 включает несколько усовершенствований API сервлета, таких как аннотации, веб-фрагменты и асинхронные запросы.

API Java для веб-сервисов RESTful (JAX-RS) 1.1

JAX-RS является API Java для разработки веб-сервисов JAX-RS. Веб-сервисы JAX-RS используют архитектуру передачи состояния представления (Representational State Transfer (REST)).

Java EE 6 принял JAX-RS в качестве части официальной спецификации Java EE.

API Java для веб-сервисов XML (JAX-WS) 2.2

JAX-WS является API Java для веб-сервисов XML. JAX-WS используется для разработки традиционных веб-сервисов на основе SOAP. Java EE 6 включает обновленную спецификацию JAX-WS. JAX-WS 2.2 является корректировочной версией с незначительными улучшениями и усовершенствованиями по сравнению с JAX-WS 2.0.

Архитектура Java для связывания с XML (JAXB) 2.2

JAXB используется для отображения классов Java на XML и обратно. Java EE 6 включает обновленную корректировочную версию JAXB.

Новые возможности GlassFish v.3

GlassFish v.3 – первый сервер приложений, поддерживающий спецификацию Java EE 6 в полном объеме. Данное обстоятельство не должно удивлять, поскольку GlassFish является эталонной реализацией спецификации Java EE. GlassFish v.3 предлагает следующие заслуживающие внимания особенности:

- *имеет модульную архитектуру, основанную на OSGi.* Архитектура на основе OSGi позволяет GlassFish иметь подключаемые модули, позволяя нам, таким образом, запускать его только с теми функциональными возможностями, которые нам действительно необходимы. В результате не придется тратить впустую ресурсы, такие как память и вычислительные ресурсы ЦП, на функциональность, которая нами не используется;
- *является встраиваемым сервером; может быть встроен в существующую JVM.* Это позволяет нам писать приложения Java со встроенным

в них сервером GlassFish. Чтобы использовать эту возможность, следует просто добавить библиотеки GlassFish к нашему проекту;

- является расширяемым сервером; может быть адаптирован для поддержки дополнительных технологий, которые не являются частью спецификации Java EE. Несколько расширений для него доступны из формы центра обновления GlassFish, например поддержка Grails (платформа веб-приложений на основе Groovy) и JRuby on Rails. Функция расширяемости GlassFish v.3 позволяет разработчикам и поставщикам приложений реализовывать свои собственные расширения GlassFish.

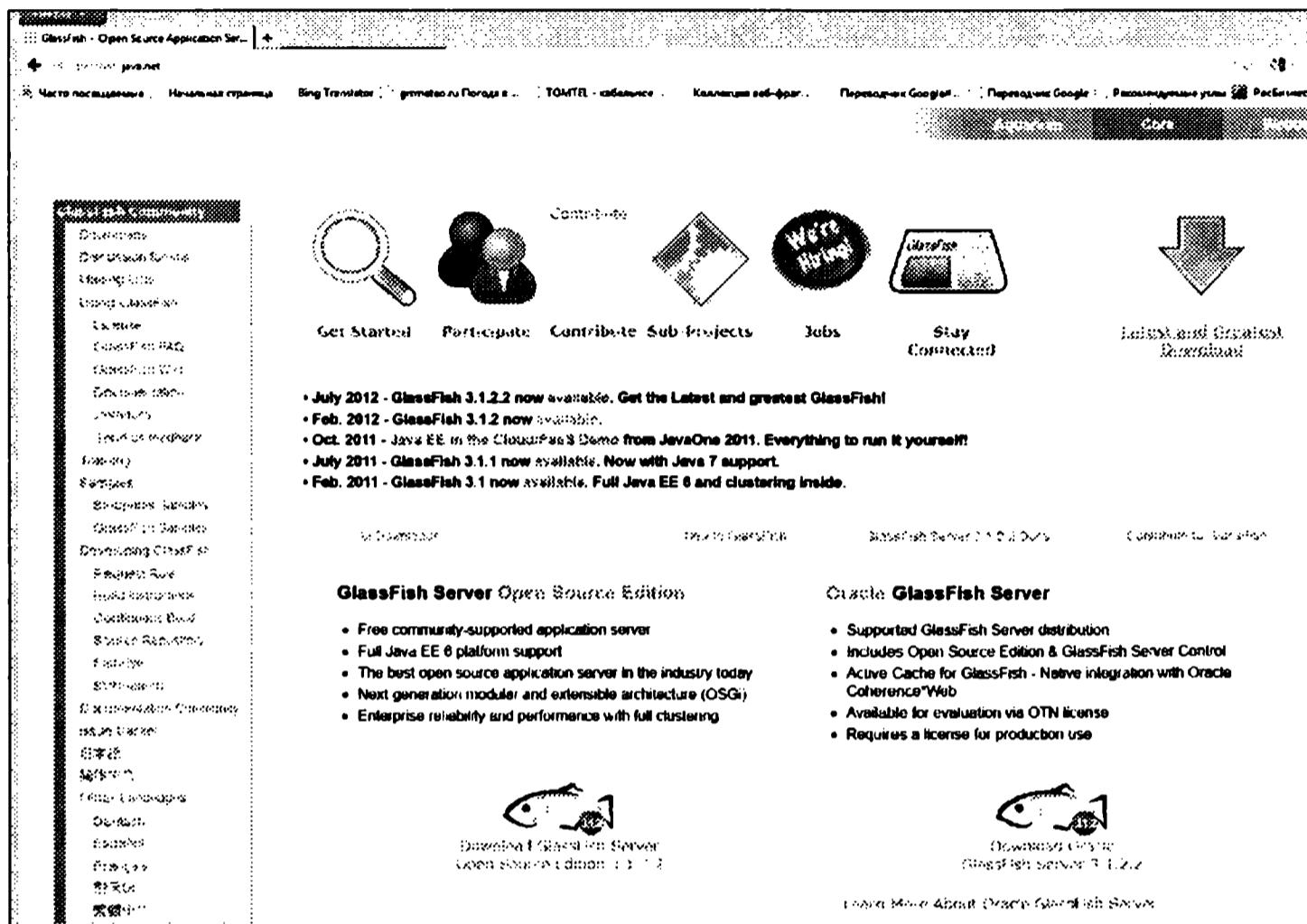
Преимущества GlassFish

Имеется много вариантов серверов приложений Java EE, но почему чаще всего выбирают именно GlassFish? Помимо очевидных преимуществ бесплатного сервера, GlassFish предлагает и многие другие:

- имеет доступную коммерческую поддержку. Коммерческая поддержка относительно недорога. Многие корпоративные покупатели программного обеспечения не будут использовать ПО, для которого не доступна коммерческая поддержка, поэтому невысокая стоимость коммерческой поддержки позволяет использовать GlassFish и в тех рыночных «нишах», где иного не предусмотрено;
- является эталонной реализацией спецификации Java EE. Это означает, что другие производители серверов приложений могут использовать GlassFish для того, чтобы убедиться, что их продукт удовлетворяет спецификации. Теоретически GlassFish может использоваться для устранения неисправностей других серверов приложений. Если приложение, развернутое на другом сервере приложений, не функционирует надлежащим образом, при том что оно работает правильно, будучи развернутым на сервере GlassFish, ошибка с наибольшей степенью вероятности заключена в другом сервере приложений;
- поддерживает самые последние версии спецификации Java EE. Поскольку GlassFish является эталонной реализацией спецификации Java EE, он реализует самые последние нововведения в спецификации раньше, чем любые другие серверы приложений на рынке. Действительно, на момент написания этой книги GlassFish является единственным Java EE-совместимым сервером приложений на рынке, который поддерживает спецификацию Java EE 6 в полном объеме.

Получение GlassFish

Сервер GlassFish может быть загружен с веб-узла <http://glassfish.java.net/>. При вводе этого URL в адресной строке веб-обозревателя открывается страница, показанная на следующем снимке экрана:



Щелкнув по ссылке для скачивания, мы перейдем к странице, содержащей таблицу такого вида:

GlassFish Server	The GlassFish Server is available in different releases to address different requirements. For more details, check out this detailed comparison between v2 and v3 .	
GlassFish v2.1	A final release based on the Java EE 5 standard, with clustering, load balancing and high availability and with Update Center. Commercially supported by Sun and recommended for production environments. Community Release Sun's Supported Releases Quick Start Guide Documentation Support Training	 Download
GlassFish v3 Preview	An unsupported, early access release implementing the latest version of the Java EE 6 standard. Includes an extensible core based on OSGi, Admin Console and Update Center. It does not have HA, clustering and other features. Community Release Java EE 6 SDK Preview Quick Start Guide Installation Guide Documentation	 Download
GlassFish v3 Prelude	A final release implementing the web layer of the Java EE 6 standard. It is based on the same core as GF v3 Preview; it does not have production features but it is more stable. Community Release Oracle Supported Release Quick Start Guide Installation Guide Documentation Support Patches	 Download

Во время написания этой книги GlassFish 3 еще не был выпущен официально, но, как видно из предыдущего снимка экрана, имелась Java EE 6-совместимая версия,

доступная для предварительного просмотра. Щелкнув по ссылке для загрузки (**Download**) для этой версии, мы перейдем к следующей странице:

How do I get GlassFish v3 Preview?

GlassFish v3 Preview Community Distributions

GlassFish v3 Preview (en)	Size (MB)	GlassFish v3 Web Profile Preview (en)	Size (MB)	Description
Windows Installer file	50	Windows Installer File	30	GUI-based installer for Windows
Self-Extracting Installer File	50	Self-Extracting Installer File	30	GUI-based Installer for Solaris, Linux and MacOS X
Zip File	71	Zip File	40	Platform-independent download file

Required JDK Version

Installations require JDK 6. The minimum (and certified) version of the JDK software that is required depends on the operating system:

- For supported operating systems except MacOS, the minimum required version is 1.6.0_13
- For the MacOS operating system, the minimum required version is 1.6.0_7

Как видно из рисунка, страница имеет ссылки загрузки для всех официально поддерживаемых платформ (Windows, Solaris, Linux и Mac OSx) и, дополнительно, – платформонезависимый ZIP-файл.

Чтобы загрузить GlassFish, нужно просто щелкнуть по ссылке для используемой нами платформы. Файл должен начать загружаться сразу. После того как он будет загружен, мы увидим название вроде такого: GlassFish-v3-preview-unix.sh, GlassFish-v3-preview-windows.exe или GlassFish-v3-preview.zip. Точное имя файла зависит от конкретной версии GlassFish и используемой нами платформы.

Установка GlassFish

Мы будем использовать установщик Unix для пояснения процесса установки. Этот установщик работает под Linux, Solaris и Mac OSx. Установка для Windows практически не отличается от демонстрационного примера.

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



NetBeans 6.8 поставляется в комплекте с GlassFish v.3. При установке дистрибутива NetBeans Java, сервер GlassFish также будет установлен автоматически.

Зависимости GlassFish

Для установки GlassFish v.3 на нашей рабочей станции должна иметься свежая версия Комплекта разработчика Java (Java Development Kit (JDK)) (требуется JDK 1.6 или более поздняя версия), а в нашем системном пути должны быть исполняемые программы Java. Самый последний JDK можно загрузить с веб-узла: <http://www.oracle.com/technetwork/java/javase/downloads/index.html>.

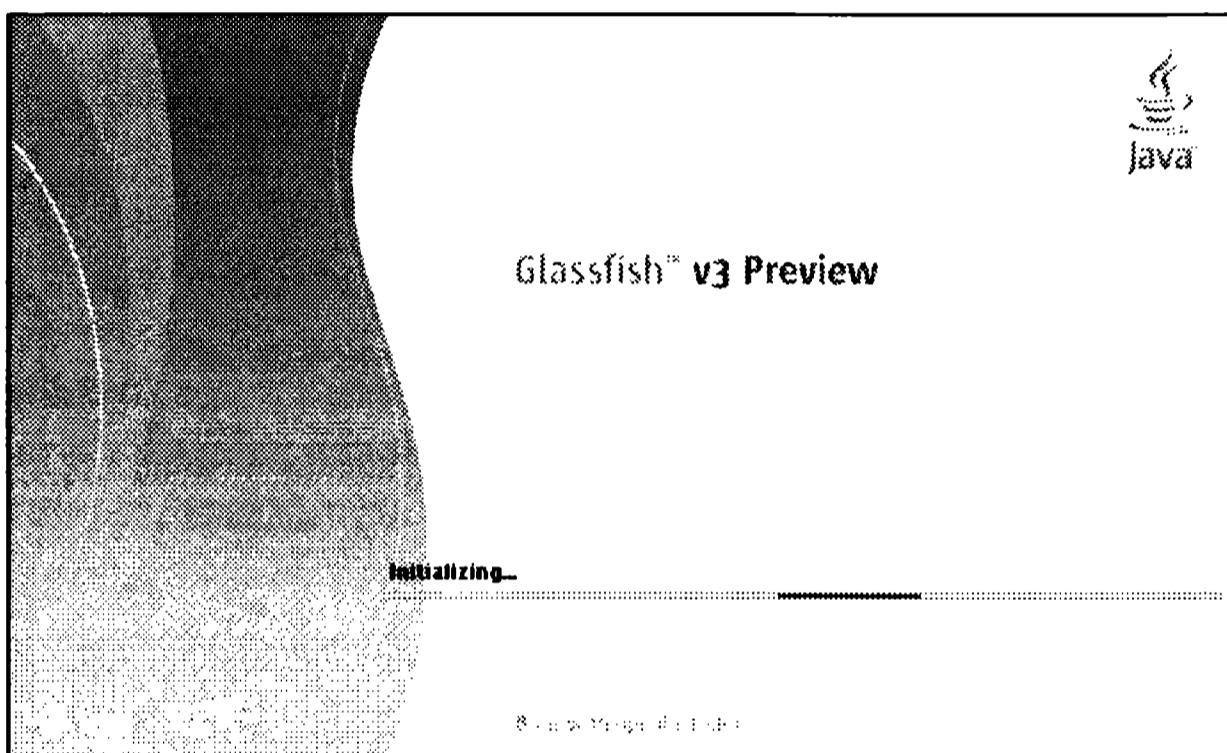
Выполнение установки

После того как будет установлен JDK, можно приступить к установке GlassFish v.3, просто запустив на выполнение загруженный файл (возможно, нам придется изменить полномочия для запуска программы-установщика, например запуск от имени администратора, для Windows):

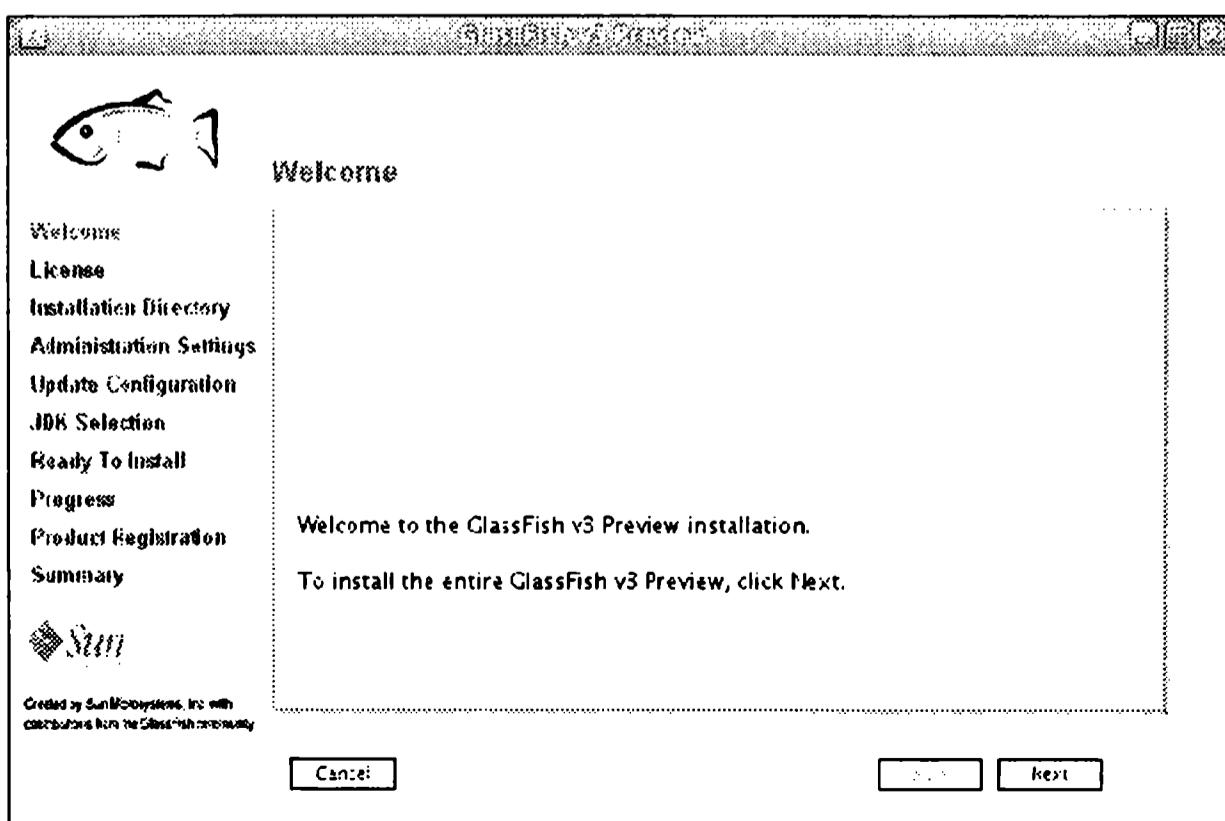
```
./GlassFish-v3-preview-Unix.sh
```

Фактическое имя файла будет зависеть от версии загруженного GlassFish. Для успешной установки GlassFish нужно выполнить следующие действия:

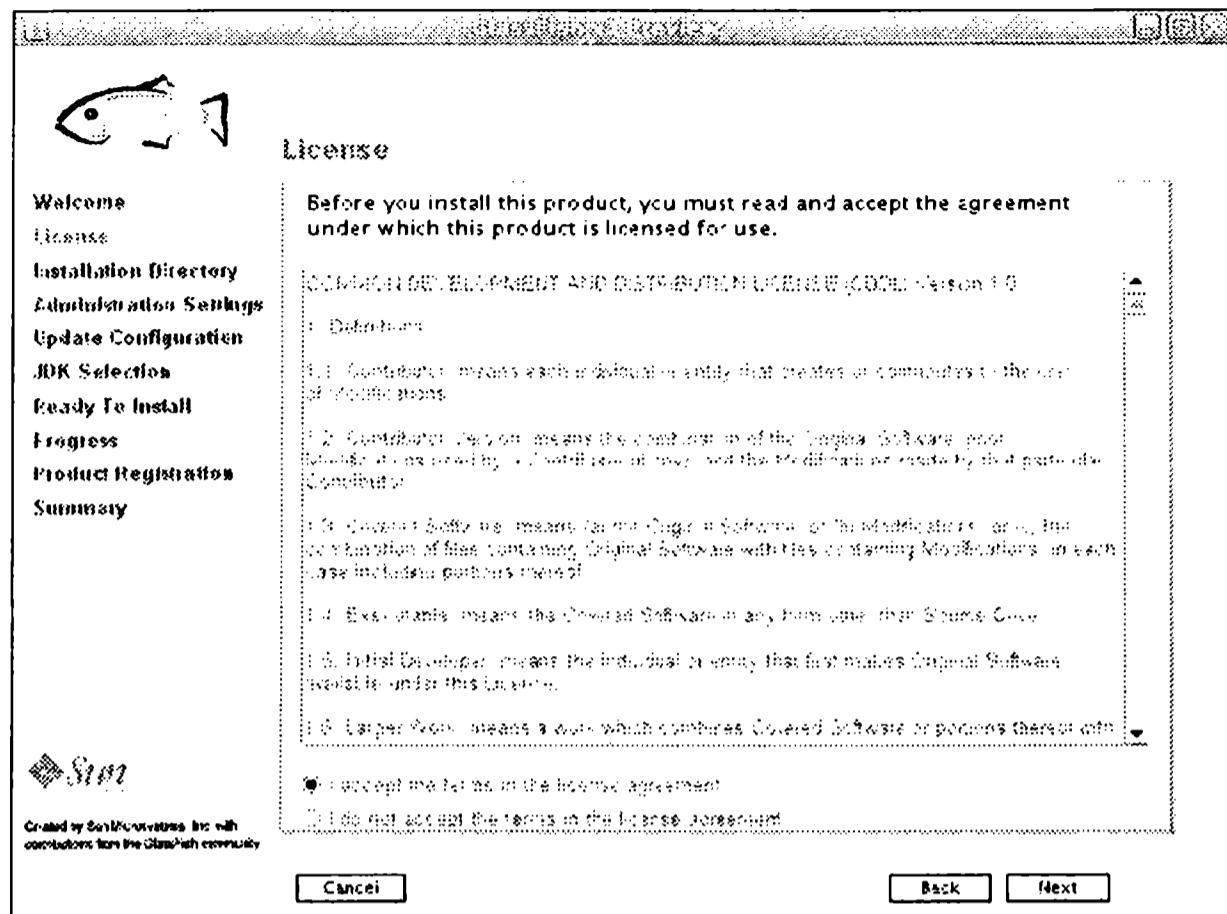
1. После выполнения предыдущей команды установщик GlassFish начнет инициализацию:



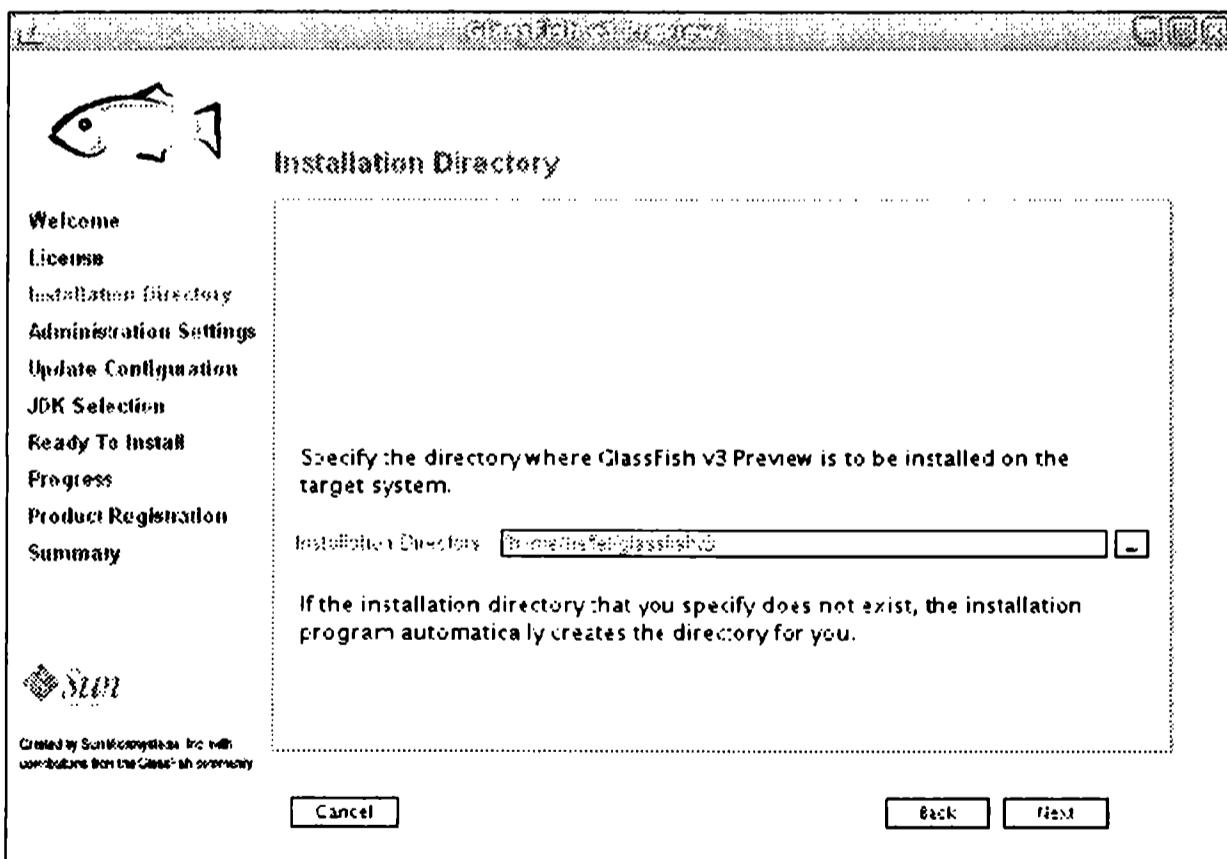
- По прошествии нескольких секунд мы увидим экран приветствия установщика:



2. После щелчка по кнопке **Далее (Next)** появится следующий экран установщика, предлагающий нам принять условия лицензионного соглашения:

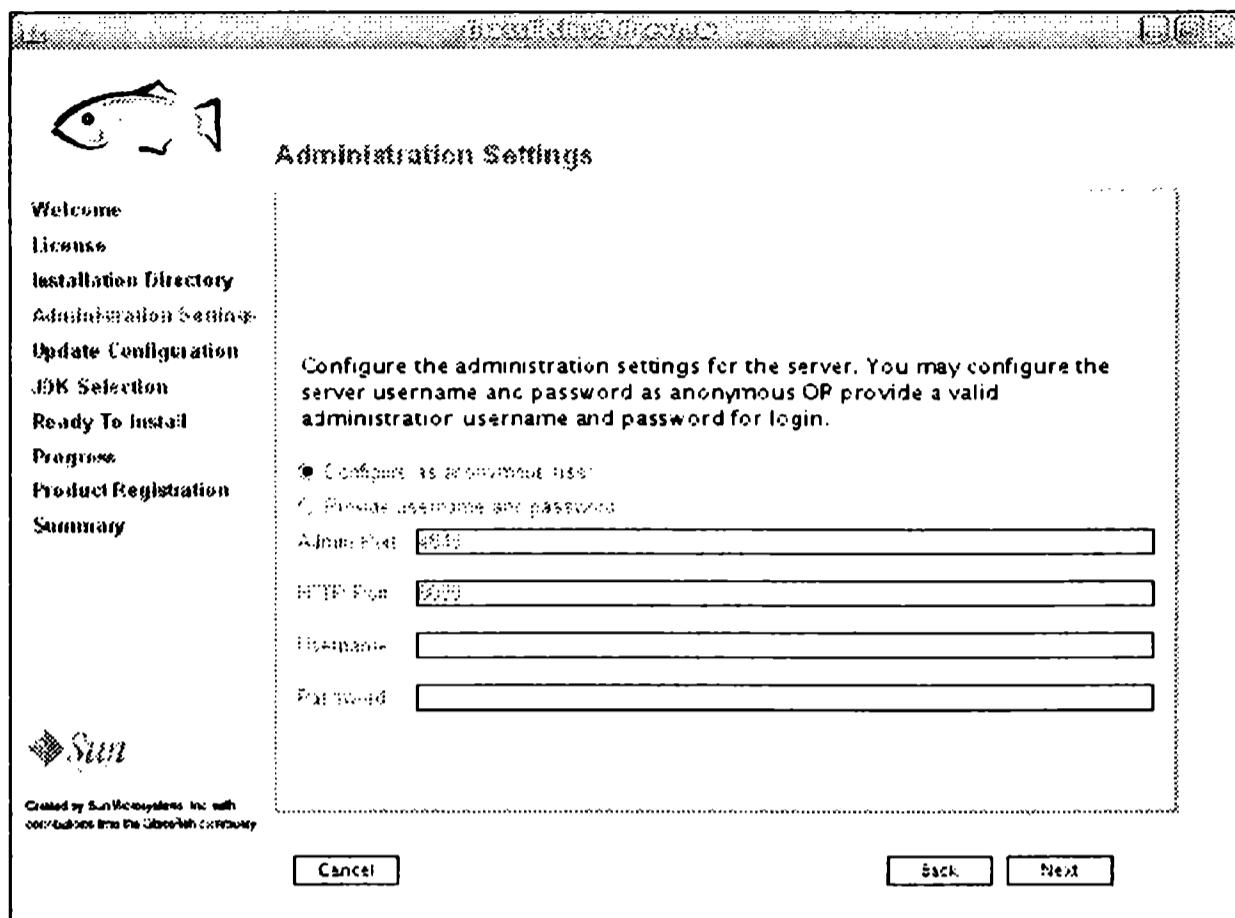


3. Следующая страница установщика запросит у нас каталог установки. Значением по умолчанию для каталога установки является каталог, называемый GlassFishv3 в нашем домашнем каталоге. Будет разумным оставить это значение, хотя мы можем его изменить.

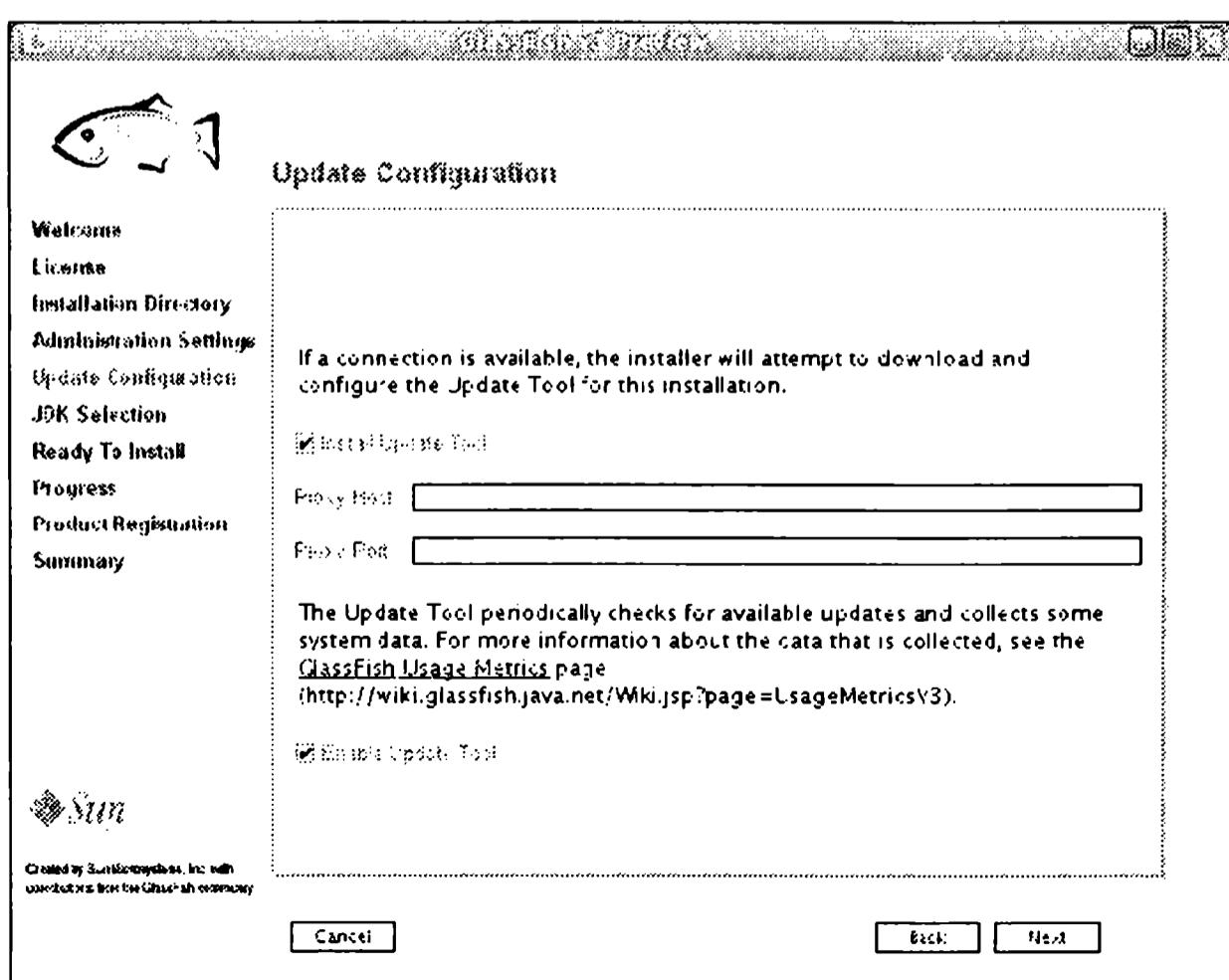


4. Следующая страница установщика позволяет нам настроить порты администратора и HTTP для GlassFish. Кроме того, здесь мы можем указать имя и пароль пользователя, выполняющего роль администратора. По умолчанию не требуется никакой комбинации имени пользователя и пароля для входа в консоль администрирования. Это поведение по умолчанию

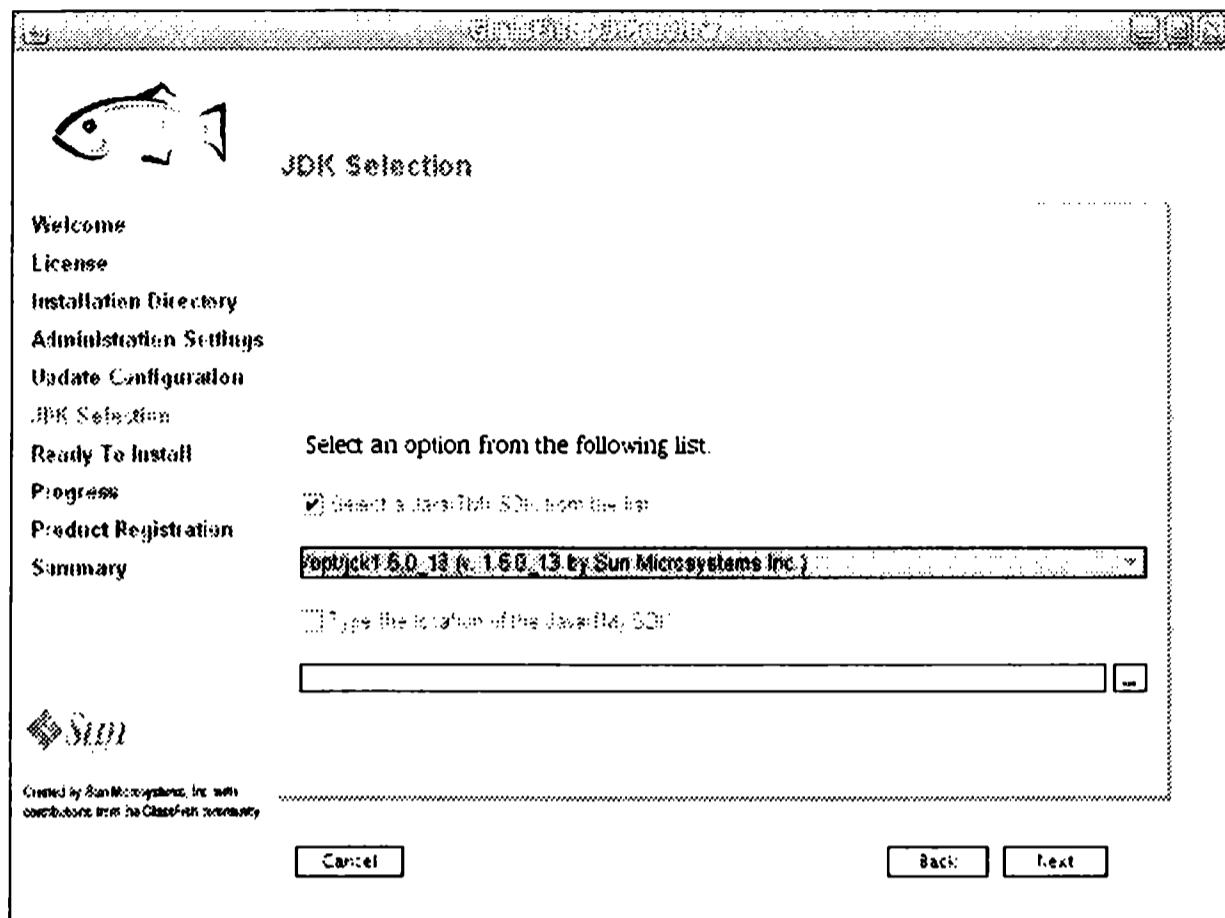
предусмотрено для режима разработки. Мы можем переопределить это поведение и предоставить имя пользователя и пароль на данном шаге мастера установки.



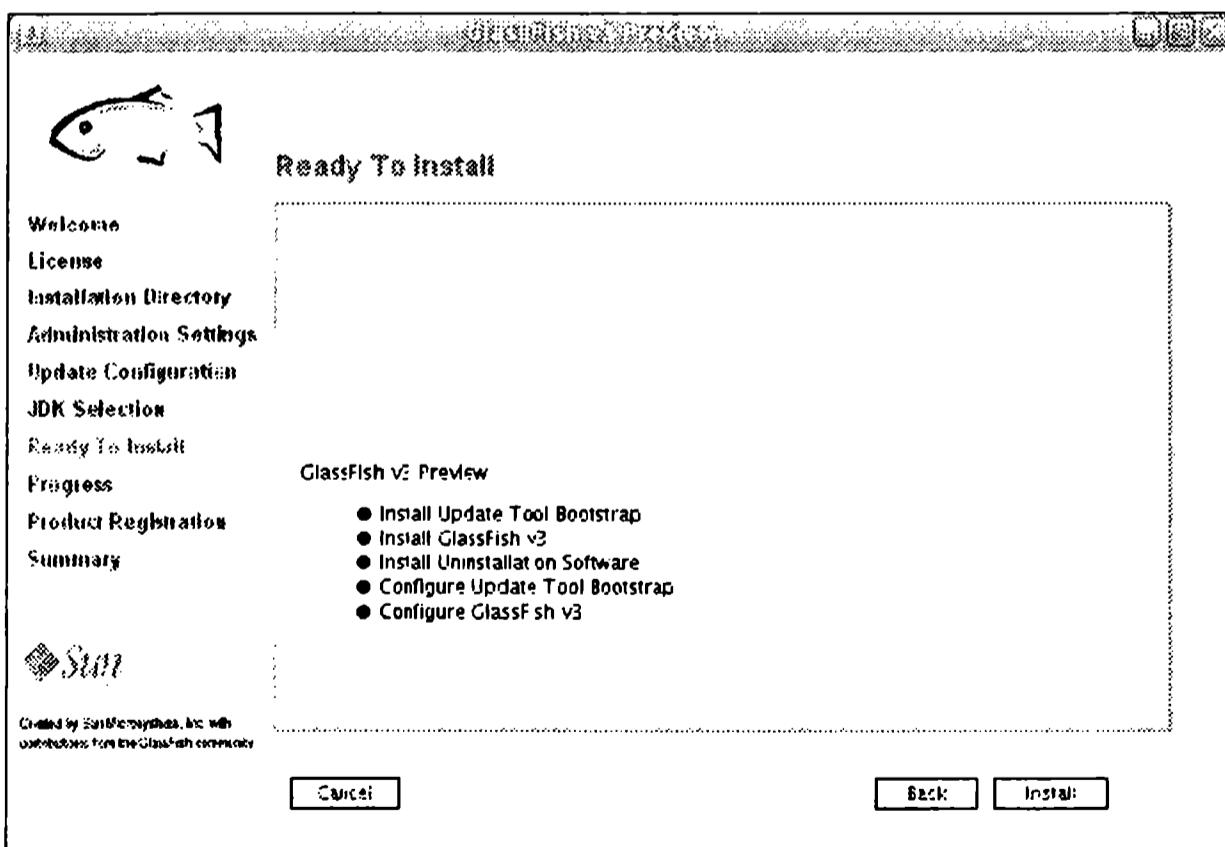
5. На данном этапе установки следует указать, хотим ли мы установить инструмент обновления GlassFish. Инструмент обновления дает возможность легко устанавливать дополнительные модули GlassFish. Поэтому, если позволяет дисковое пространство, рекомендуется его установить. Если мы получаем доступ в Интернет через прокси-сервер, можно ввести имя его хоста или IP-адрес и порт на этой странице мастера установки.

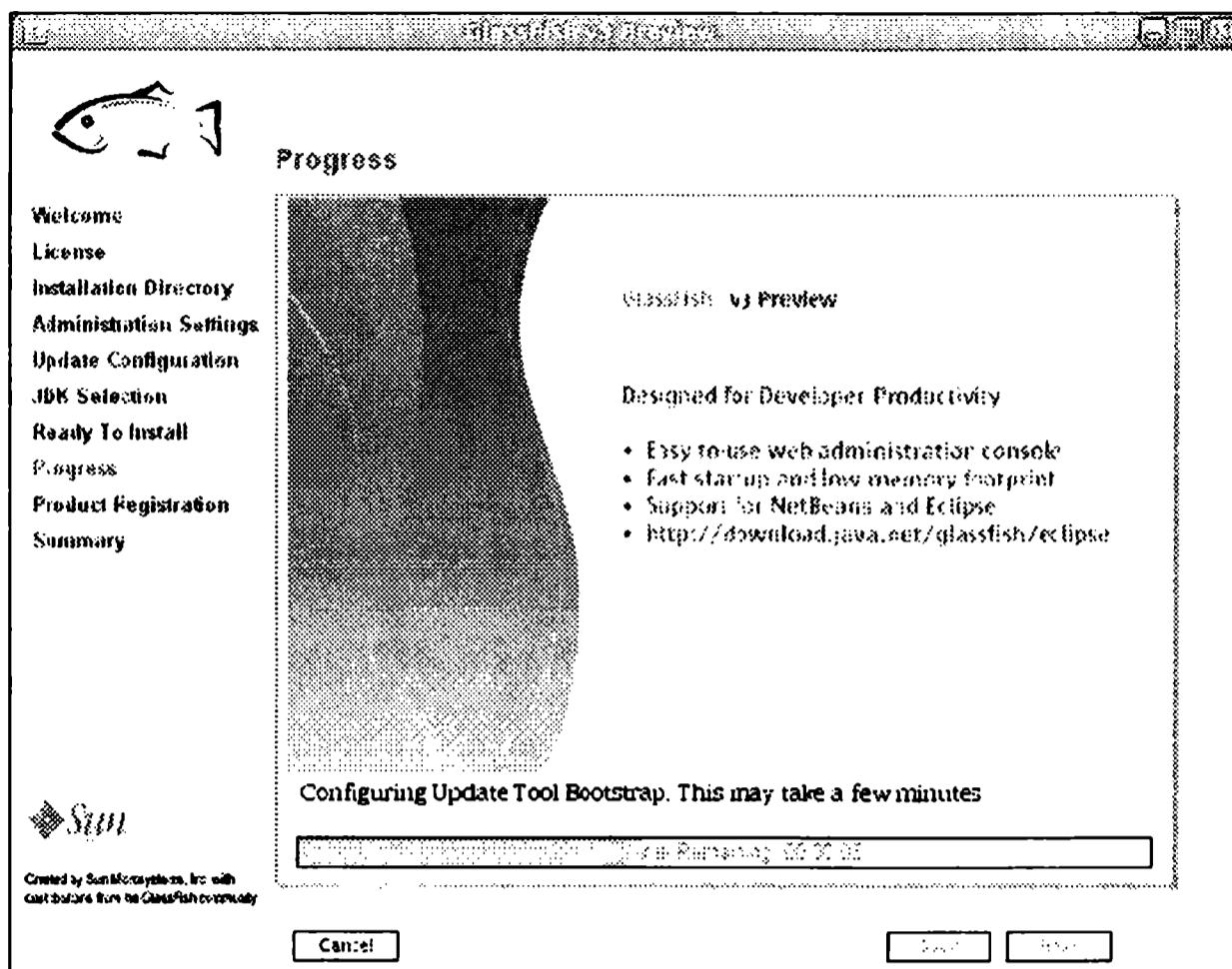


6. Теперь нам предлагаются на выбор два варианта: либо автоматическое обнаружение Java SDK, либо указание местонахождения SDK. По умолчанию выбирается тот Java SDK, который соответствует значению переменной среды JAVA_HOME.

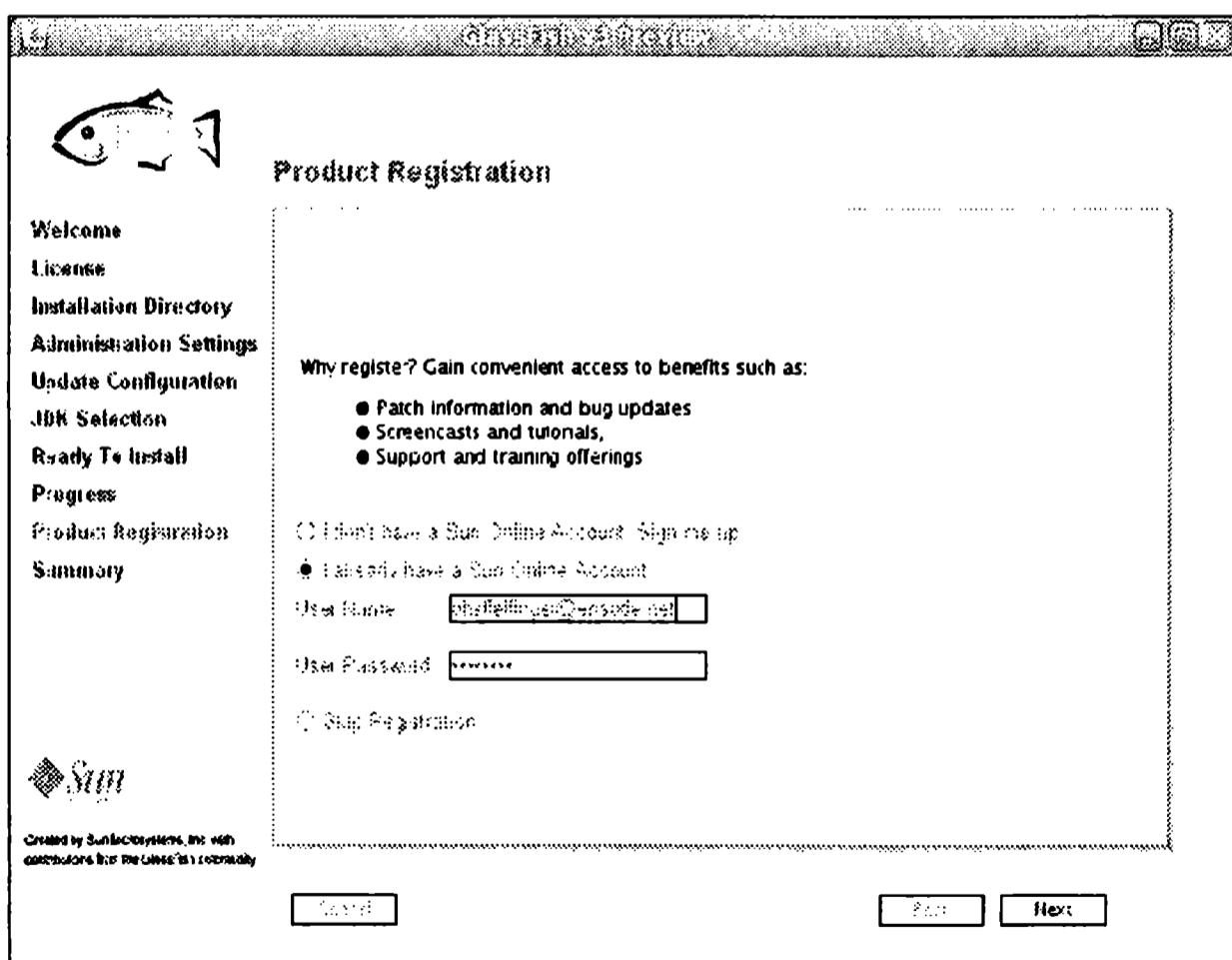


7. На данном этапе установщик резюмирует действия, которые он собирается предпринять в процессе установки. Щелкните по кнопке Установить (Install), чтобы начать установку:



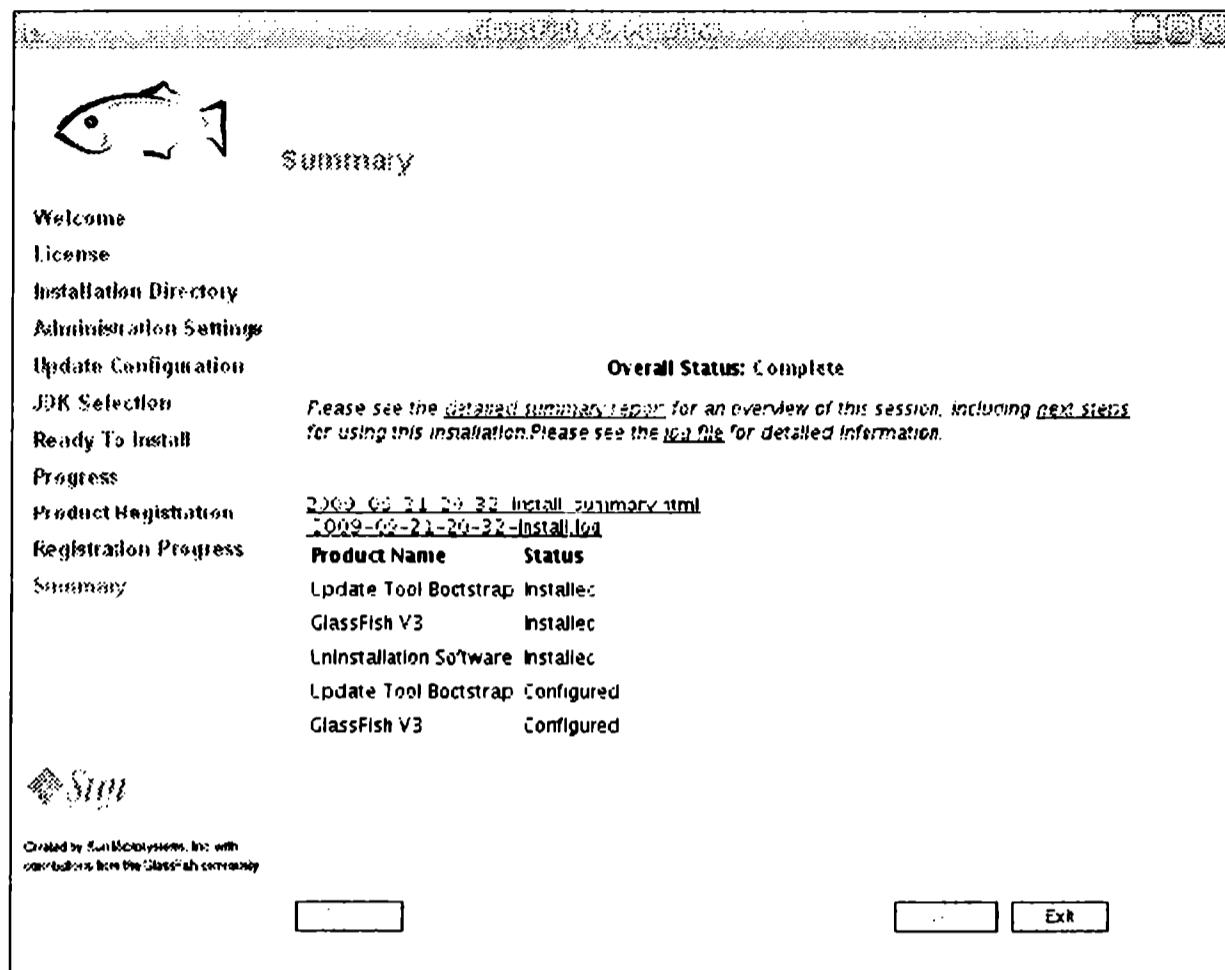
8. Процесс установки показан на следующем снимке экрана:

9. По завершении установки нам будет предложено зарегистрировать нашу копию GlassFish. На данном этапе мы можем связать установку GlassFish с существующей у нас онлайновой учетной записью Sun¹, либо создать новую онлайновую учетную запись Sun, либо пропустить регистрацию:



¹ Здесь речь идет об учетной записи Oracle (начиная с 2011 года, после поглощения Sun Microsystems), к которой можно получить доступ по адресу: <https://login.oracle.com/mysso/signon.jsp>. – Прим. перев.

10. Следующая страница установщика показывает сводную информацию об установке. Здесь нам нужно просто щелкнуть по кнопке **Выход** (Exit), чтобы выйти из установщика:



Проверка установки

Чтобы запустить GlassFish, нужно изменить текущий каталог на [*Каталог установки GlassFish*]/GlassFishv3/bin и выполнить следующую команду:

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

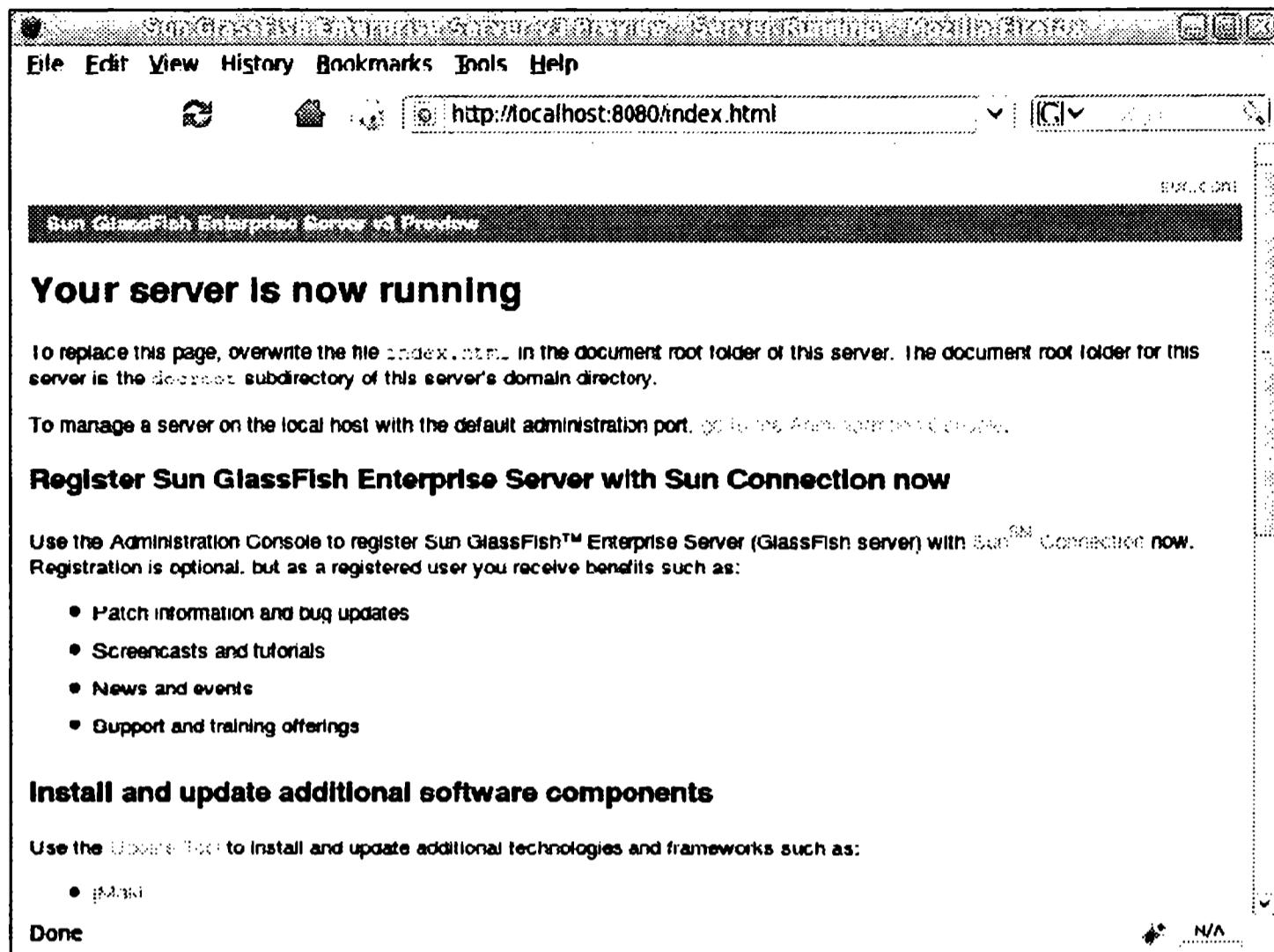
 Эта команда, как и большинство команд, приведенных в этой главе, предполагает использование Unix или Unix-подобной операционной системы. Для систем Windows начальных символов ./ не требуется.

Через несколько секунд после выполнения предыдущей команды мы должны увидеть внизу окна терминала сообщение, подобное следующему:

Имя запущенного домена: [domain1] и его расположение: [/home/heffel/GlassFishv3/GlassFish/domains/domain1]. Порт администратора для домена: [4848].
 (Name of the domain started: [domain1] and its location: [/home/heffel/glassfishv3/glassfish/domains/domain1]. Admin port for the domain: [4848].)

Затем мы можем открыть окно обозревателя и ввести в его адресной строке следующий URL: <http://localhost:8080>.

Если бы все было нормально, то мы не должны были бы видеть страницу, подобную следующему снимку экрана:



Получение справки

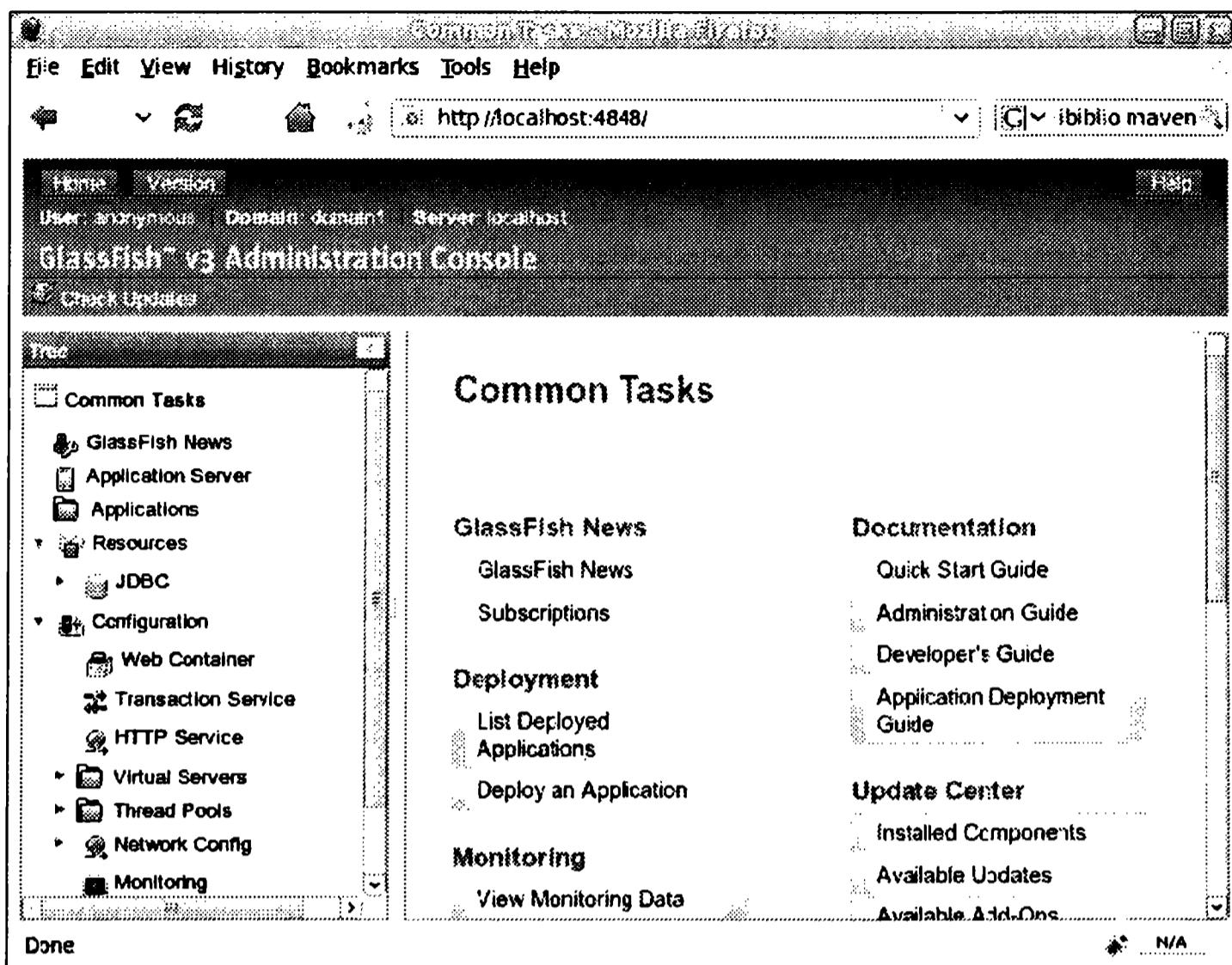
Если какой-либо из наших предыдущих шагов завершился неудачно либо нам требуется справка по общим вопросам, касающимся GlassFish, то мы можем обратиться к большому информационному ресурсу – форуму GlassFish, который можно найти по адресу: <http://www.java.net/forums/glassfish/glassfish>.

Развертывание нашего первого приложения Java EE

Чтобы проверить, правильно ли работает установленный нами сервер GlassFish, развернем WAR-файл (веб-архив) и убедимся, что он развертывается и выполняется надлежащим образом. Прежде чем двигаться дальше, пожалуйста, загрузите файл `simpleapp.war` с веб-сайта www.dimk-press.ru.

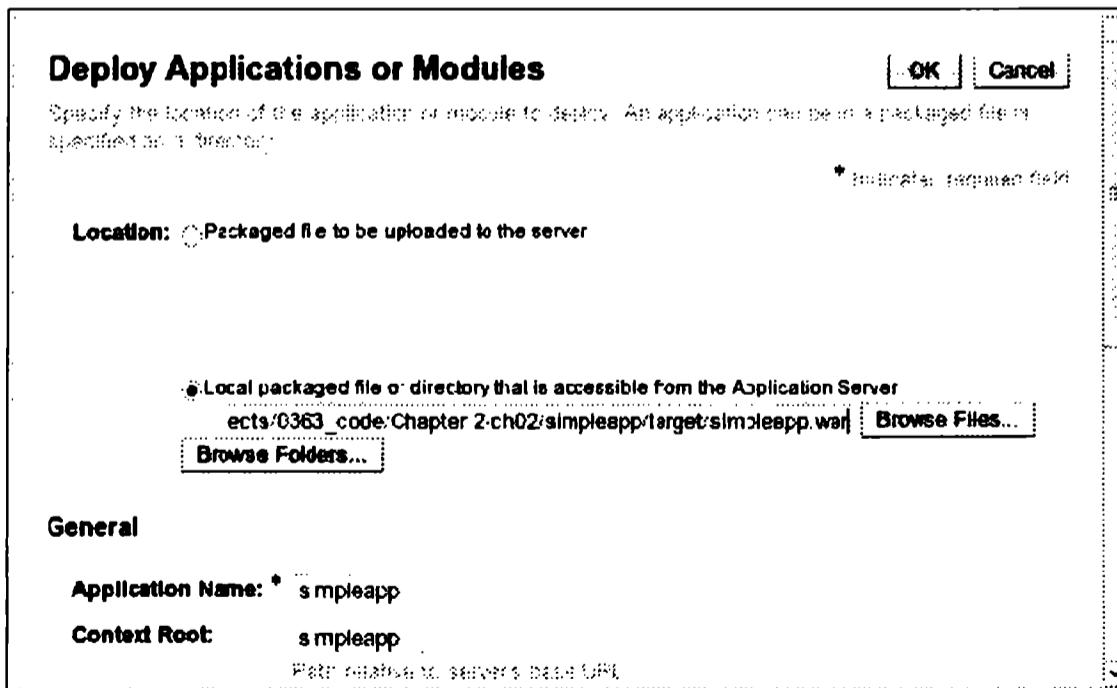
Развертывание приложения через веб-консоль

Чтобы развернуть `simpleapp.war`, откройте обозреватель и перейдите к следующему URL: <http://localhost:4848>. Вы должны увидеть экран общих задач консоли администрирования, который выглядит примерно так:



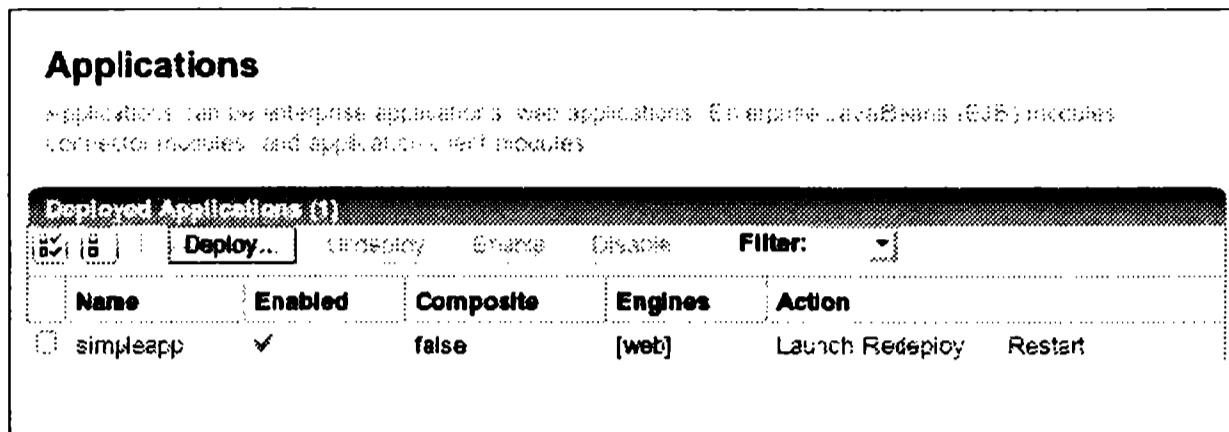
Если вход в GlassFish был сконфигурирован для анонимного пользователя, мы сразу увидим предыдущую страницу. В противном случае эта страница появится только после ввода учетных данных администратора, заданных нами во время установки.

Теперь мы должны щелкнуть по элементу **Развертывание приложения** (Deploy an Application) в разделе **Развертывание** (Deployment) в основной панели окна.



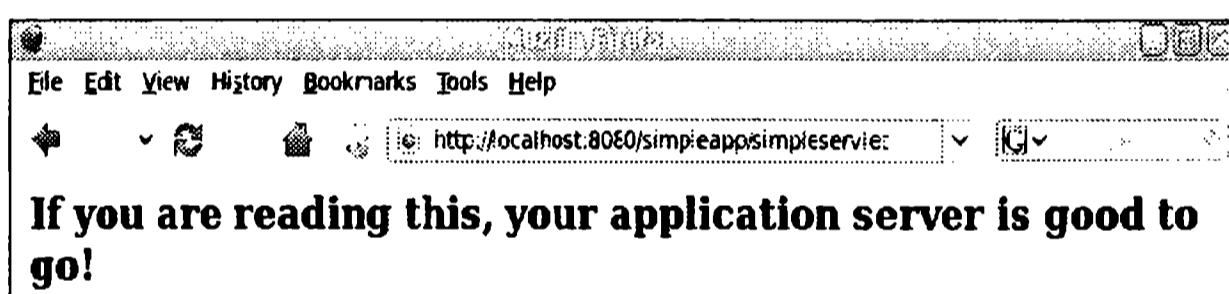
Далее на открывшейся странице мы должны установить переключатель в положение **Локальный упакованный файл или каталог, который доступен из сервера приложений** (Local packaged file or directory that is accessible from the Application Server) и ввести путь к нашему WAR-файлу или выбрать его, щелкнув по кнопке **Просмотр файлов...** (Browse Files...).

После того как мы выберем наш WAR-файл, остается лишь щелкнуть по кнопке **OK**, чтобы развернуть его.



Как видно на предыдущем снимке экрана, наше приложение simpleapp теперь развернуто на сервере.

Для выполнения приложения simpleapp введите в адресной строке обозревателя: <http://localhost:8080/simpleapp/simple servlet>. Открывшаяся в результате этого страница должна выглядеть следующим образом:



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

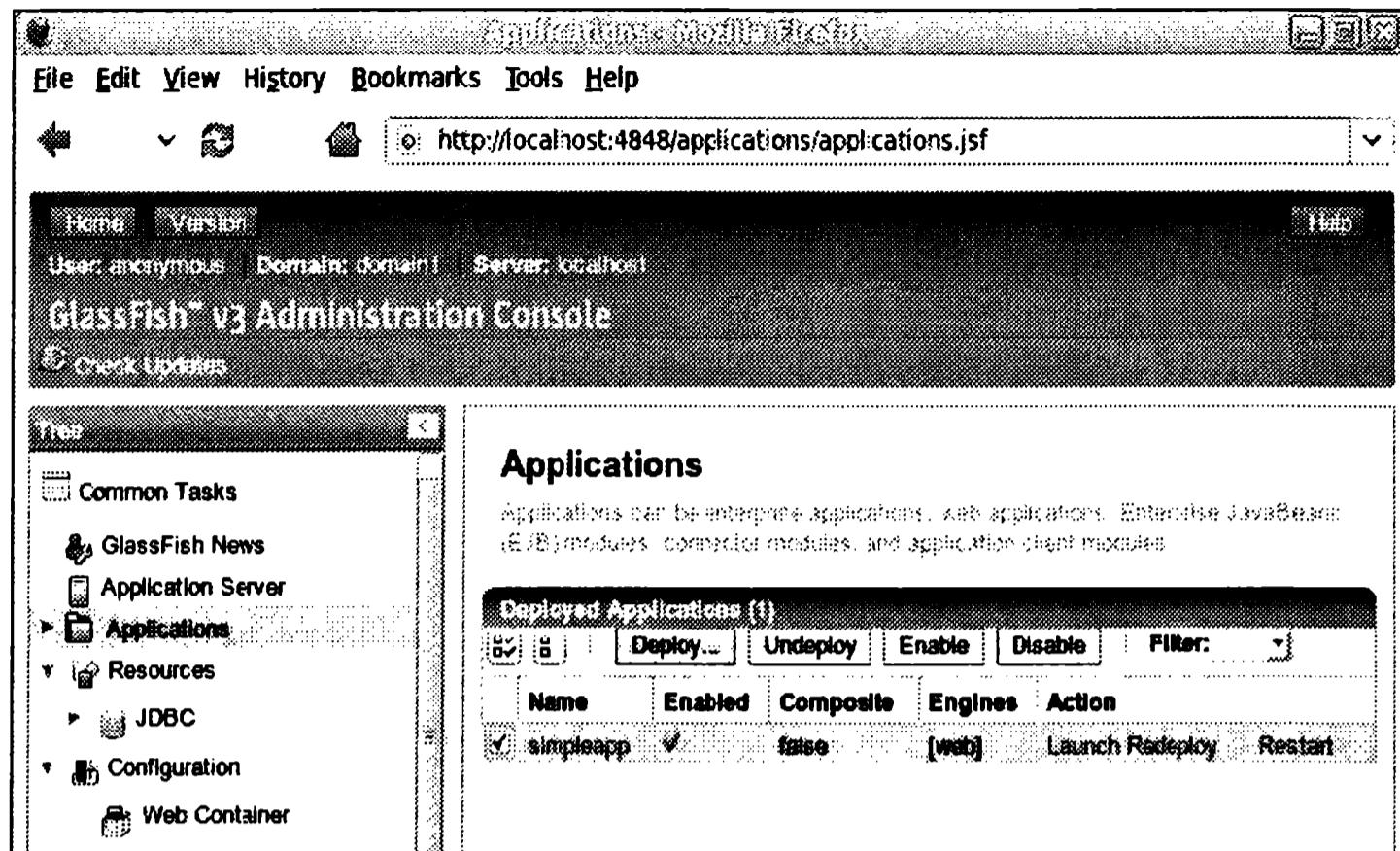
Отмена развертывания приложения через веб-консоль

В следующем разделе мы объясним, как развернуть веб-приложение с помощью утилиты командной строки. Для того чтобы выполнить инструкции следующего раздела, вначале потребуется отменить развертывание приложения simpleapp.war.

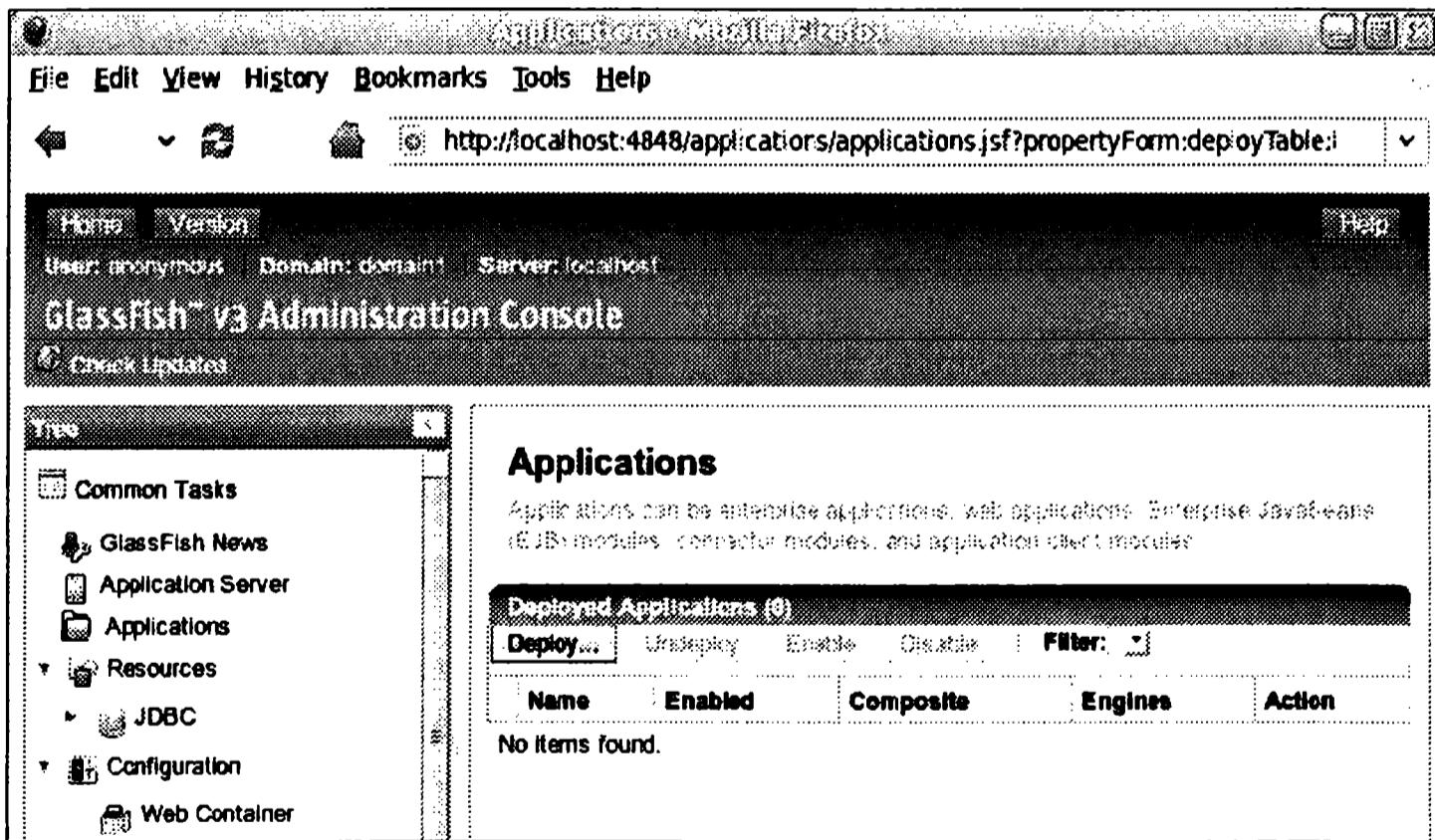
Чтобы отменить развертывание приложения, которое мы развернули в предыдущем разделе, войдите в Консоль администрирования GlassFish, введя следующий URL в адресной строке обозревателя: <http://localhost:4848>.

Затем, или щелкните по пункту меню (узлу) **Приложения** (Applications) в верхней левой части панели навигации страницы, или щелкните по элементу **Список развернутых приложений** (List Deployed Applications) в основной панели страницы консоли администрирования.

В любом случае мы должны попасть на страницу управления приложениями:



Отмена развертывания приложения может быть выполнена путем простого выбора его из списка развернутых приложений и щелчка по кнопке **Отменить развертывание** (Undeploy). После этого страница управления приложениями будет выглядеть следующим образом:



Развертывание приложения с помощью командной строки

Имеется два способа, которыми приложение может быть развернуто при использовании командной строки. Это может быть сделано или путем копирования артефакта, который мы хотим развернуть, в каталог `autodeploy`, или путем использования утилиты командной строки GlassFish – `asadmin`.

Каталог autodeploy

Теперь, когда мы отменили развертывание файла simpleapp.war, мы готовы развернуть его с использованием командной строки. Чтобы развернуть приложение таким способом, просто скопируйте файл simpleapp.war в [Каталог установки GlassFish]/GlassFishv3/GlassFish/domains/domain1/autodeploy. Приложение будет автоматически развернуто при его копировании только в этот каталог.

Для того чтобы убедиться, что приложение было успешно развернуто, необходимо просмотреть записи в журнале сервера. Журнал сервера можно найти в [Каталог установки GlassFish]/GlassFishv3/GlassFish/domains/domain1/logs/server.log. Последние несколько строк из этого файла должны выглядеть примерно так:

```
[#|2009-09-23T19:26:39.463-0400|INFO|glassfish|javax.enterprise.system.tools.deployment.org.glassfish.deployment.common|_ThreadID=20;_ThreadName=Thread-1;|[AutoDeploy] Selecting file /home/heffel/glassfishv3/glassfish/domains/domain1/autodeploy/simpleapp.war for autodeployment.|#]

[#|2009-09-23T19:26:39.635-0400|INFO|glassfish|null|_ThreadID=20;_ThreadName=Thread-1;|Deployment expansion took 64|#]

[#|2009-09-23T19:26:41.132-0400|INFO|glassfish|null|_ThreadID=20;_ThreadName=Thread-1;|DOL Loading time938|#]

[#|2009-09-23T19:26:41.190-0400|INFO|glassfish|org.apache.catalina.loader.WebappLoader|_ThreadID=20;_ThreadName=Thread-1;|Unknown loader org.glassfish.internal.api.DelegatingClassLoader@55d866c5 class org.glassfish.internal.api.DelegatingClassLoader|#]

[#|2009-09-23T19:26:41.610-0400|INFO|glassfish|javax.enterprise.system.container.web.com.sun.enterprise.web|_ThreadID=20;_ThreadName=Thread-1;|Loading application simpleapp at /simpleapp|#]

[#|2009-09-23T19:26:41.808-0400|INFO|glassfish|javax.enterprise.system.tools.admin.org.glassfish.server|_ThreadID=20;_ThreadName=Thread-1;|Deployment of simpleapp done is 2,239 ms|#]

[#|2009-09-23T19:26:41.810-0400|INFO|glassfish|javax.enterprise.system.tools.deployment.org.glassfish.deployment.common|_ThreadID=20;_ThreadName=Thread-1;|[AutoDeploy] Successfully autodeployed : /home/heffel/glassfishv3/glassfish/domains/domain1/autodeploy/simpleapp.war.|#]
```

Конечно, можно дополнительно проверить развертывание, перейдя к URL приложения, который будет таким же, как и используемый нами ранее, при развертывании приложения через веб-консоль: <http://localhost:8080/simpleapp/simple-servlet>. Приложение должно выполнится надлежащим образом.

Отмена развертывания приложения, развернутого таким образом, может быть выполнена простым удалением артефакта (в нашем случае WAR-файла) из каталога

autodeploy. После удаления файла мы увидим сообщение в журнале сервера, подобное следующему:

```
[#|2009-09-23T19:09.835-0400|INFO|glassfish|javax.enterprise.system.tools.deployment.org.glassfish.deployment.common|_ThreadID=20;_ThreadName=Thread-1;|Autoundeploying application :simpleapp|#]

[#|2009-09-23T19:09.909-0400|INFO|glassfish|javax.enterprise.system.tools.deployment.org.glassfish.deployment.common|_ThreadID=20;_ThreadName=Thread-1;|[AutoDeploy] Successfully autoundeployed : /home/heffel/glassfishv3/glassfish/domains/domain1/autodeploy/simpleapp.war.|#]
```

Утилита командной строки asadmin

Альтернативным способом развертывания приложения с помощью командной строки является использование команды:

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

Файл журнала сервера должен показать сообщение, подобное следующему:

```
[#|2009-09-23T19:35:04.012-0400|INFO|glassfish|null|_ThreadID=16;_ThreadName=Thread-1;|Deployment expansion took 76|#]

[#|2009-09-23T19:35:04.986-0400|INFO|glassfish|null|_ThreadID=16;_ThreadName=Thread-1;|DOL Loading time707|#]

[#|2009-09-23T19:35:05.025-0400|INFO|glassfish|org.apache.catalina.loader.WebappLoader|_ThreadID=16;_ThreadName=Thread-1;|Unknown loader org.glassfish.internal.api.DelegatingClassLoader@55d866c5 class org.glassfish.internal.api.DelegatingClassLoader|#]

[#|2009-09-23T19:35:05.238-0400|INFO|glassfish|javax.enterprise.system.container.web.com.sun.enterprise.web|_ThreadID=16;_ThreadName=Thread-1;|Loading application simpleapp at /simpleapp|#]

[#|2009-09-23T19:35:05.321-0400|INFO|glassfish|javax.enterprise.system.tools.admin.org.glassfish.server|_ThreadID=16;_ThreadName=Thread-1;|Deployment of simpleapp done is 1,576 ms|#]
```

Утилита asadmin также может использоваться для отмены развертывания приложения путем выполнения следующей команды:

```
asadmin undeploy simpleapp
```

В этом случае будет показано следующее сообщение в нижней части окна терминала:

Команда отмены развертывания успешно выполнена (Command undeploy executed successfully).

Пожалуйста, обратите внимание, что при развертывании приложения расширение файла не используется. Аргументом для asadmin undeploy должен быть корневой

контекст приложения (он вводится сразу после `http://localhost:4848` для получения доступа к приложению через адресную строку обозревателя), который по умолчанию является именем WAR-файла.

В следующей главе мы увидим, как изменить корневой контекст по умолчанию для приложения.

Домены GlassFish

Внимательный читатель, возможно, заметил, что каталог `autodeploy` является подкаталогом каталога `domains/domain1`. GlassFish использует концепцию *доменов* (`domains`). Домены позволяют совместно развертывать наборы связанных между собой приложений. Несколько доменов могут быть запущены одновременно; при этом они ведут себя как индивидуальные экземпляры сервера GlassFish. Домен по умолчанию, называемый `domain1`, создается при установке сервера GlassFish.

Создание доменов

Дополнительные домены могут быть созданы из командной строки путем выполнения следующей команды:

```
asadmin create-domain имя_домена
```

Эта команда принимает несколько параметров, указывающих номера портов, которые домен будет прослушивать для нескольких служб (HTTP, Администратор, JMS, IIOP, защищенный HTTP и т. д.). Чтобы увидеть все эти параметры, введите в командной строке:

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

Если мы хотим, чтобы несколько доменов работали одновременно на одном и том же сервере, выбор портов для них должен осуществляться с известной степенью осторожности, поскольку в случае указания одних и тех же портов для различных служб (либо для одной и той же службы разных доменов) возникнут проблемы в связи с тем, что один из доменов не будет работать должным образом.

Номера портов, по умолчанию присваиваемые домену `domain1`, являющемуся доменом по умолчанию, перечислены в таблице:

Служба	Порт
Администратор	4848
HTTP	8080
Система обмена сообщениями Java (JMS)	7676
Интернет-протокол Inter-ORB (IIOP)	3700
Защищенный HTTP (HTTPS)	8181

Служба	Порт
Защищенный ПОР	3820
Взаимная авторизация ПОР	3920
Администрирование Расширения управления Java (JMX)	8686

Пожалуйста, обратите внимание, что при создании домена единственным портом, который должен быть указан явно, является порт администратора. Если значения других портов не указаны, то для них будут использоваться значения портов по умолчанию, перечисленные в предыдущей таблице. При создании домена необходимо соблюдать осторожность, поскольку, как уже было объяснено ранее, два домена не смогут работать одновременно на одном и том же сервере, если какая-либо из их служб прослушивает один и тот же порт.

Альтернативным методом создания домена, без необходимости указывать порты для каждой службы, является следующая команда утилиты командной строки:

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

Значение параметра `--portbase` определяет номер базового порта для домена. Номера портов для других служб домена будут смещениями от данного номера порта. В следующей таблице перечислены номера портов, назначаемые в таком случае всем прочим службам:

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 ПОР	portbase+ 38
Взаимная авторизация ПОР	portbase+ 39
Администрирование Расширения управления Java (JMX)	portbase+ 86

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



Как показывает опыт, при использовании числа `portbase`, превышающего 8000 и делящегося на 1000, создаются домены, которые не конфликтуют друг с другом. Например, будет безопасным создание одного домена с использованием `portbase 9000`, а другого — с использованием `portbase 10000` и т. д.

Удаление доменов

Удалить домен очень просто – достаточно выполнить следующую команду утилиты командной строки:

```
asadmin delete-domain имя_домена
```

После этого в окне терминала появится сообщение наподобие следующего:

Команда delete-domain успешно выполнена (Command delete-domain executed successfully).

Пожалуйста, используйте эту команду с необходимыми мерами предосторожности, поскольку после того, как домен будет удален, его не удастся легко и без потерь восстановить (все развернутые приложения будут уничтожены, равно как и пулы соединений, источники данных и другие инфраструктурные элементы).

Остановка домена

Домен, находящийся в рабочем состоянии, может быть остановлен выполнением следующей команды утилиты командной строки:

```
asadmin stop-domain имя_домена
```

Эта команда остановит домен **имя_домена**.

Если на сервере работает только один домен, аргумент **имя_домена** является необязательным.



В этой книге предполагается, что читатель работает с доменом по умолчанию **domain1**, для которого используются номера портов по умолчанию. Если дело обстоит иначе, то приведенные выше команды необходимо модифицировать таким образом, чтобы они соответствовали нужному 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 JavaDB, Oracle, Derby, Sybase, DB2, Pointbase, MySQL, PostgreSQL, Informix, Cloudscape и SQL Server. В этом разделе мы покажем, как настроить GlassFish для установки соединения с базой данных MySQL. Для других СУРБД процедура будет аналогичной.



GlassFish поставляется в комплекте с СУРБД, именуемой JavaDB. Эта СУРБД основана на проекте Derby Apache. Чтобы минимизировать необходимость различных загрузок и конфигурирования при выполнении примеров кода, написанных для этой книги, все примеры, которым требуется СУРБД, будут использовать встроенную СУРБД – JavaDB. Инструкции, приведенные в этом разделе, также поясняют, как GlassFish может установить соединение с СУРБД сторонних производителей.

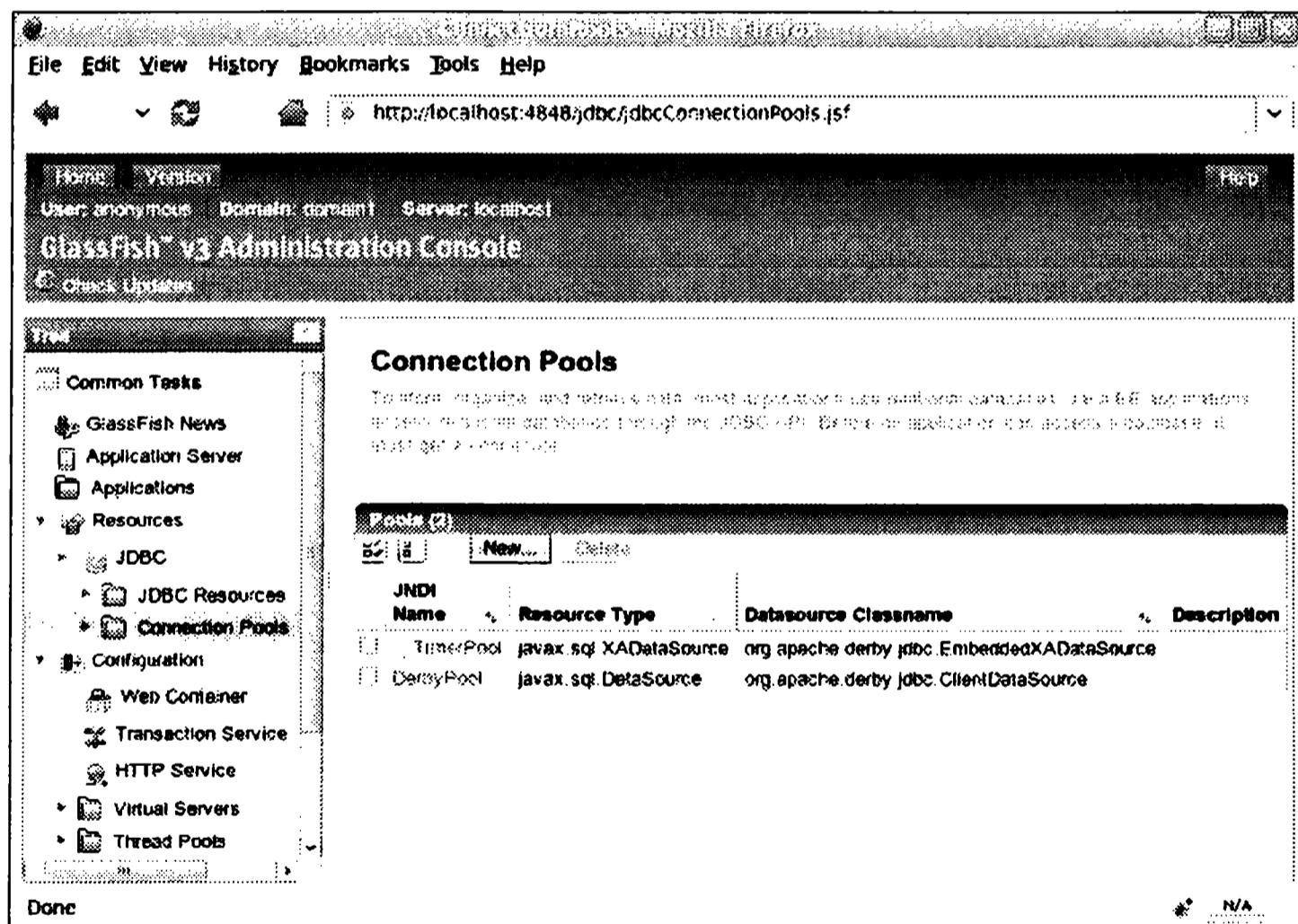
Создание пулов соединений

Первый шаг, который предстоит выполнить при создании пула соединений, – копирование JAR-файла, содержащего драйвер JDBC для нашей СУРБД, в каталог `lib` домена (информацию о том, где можно взять этот JAR-файл, следует получить из документации по СУРБД). Если домен GlassFish, в который мы будем добавлять пул соединений, находится в работающем состоянии, то после копирования драйвера JDBC его нужно перезапустить для того, чтобы изменения вступили в силу. Домен можно перезапустить, выполнив команду утилиты командной строки:

```
asadmin restart-domain
```

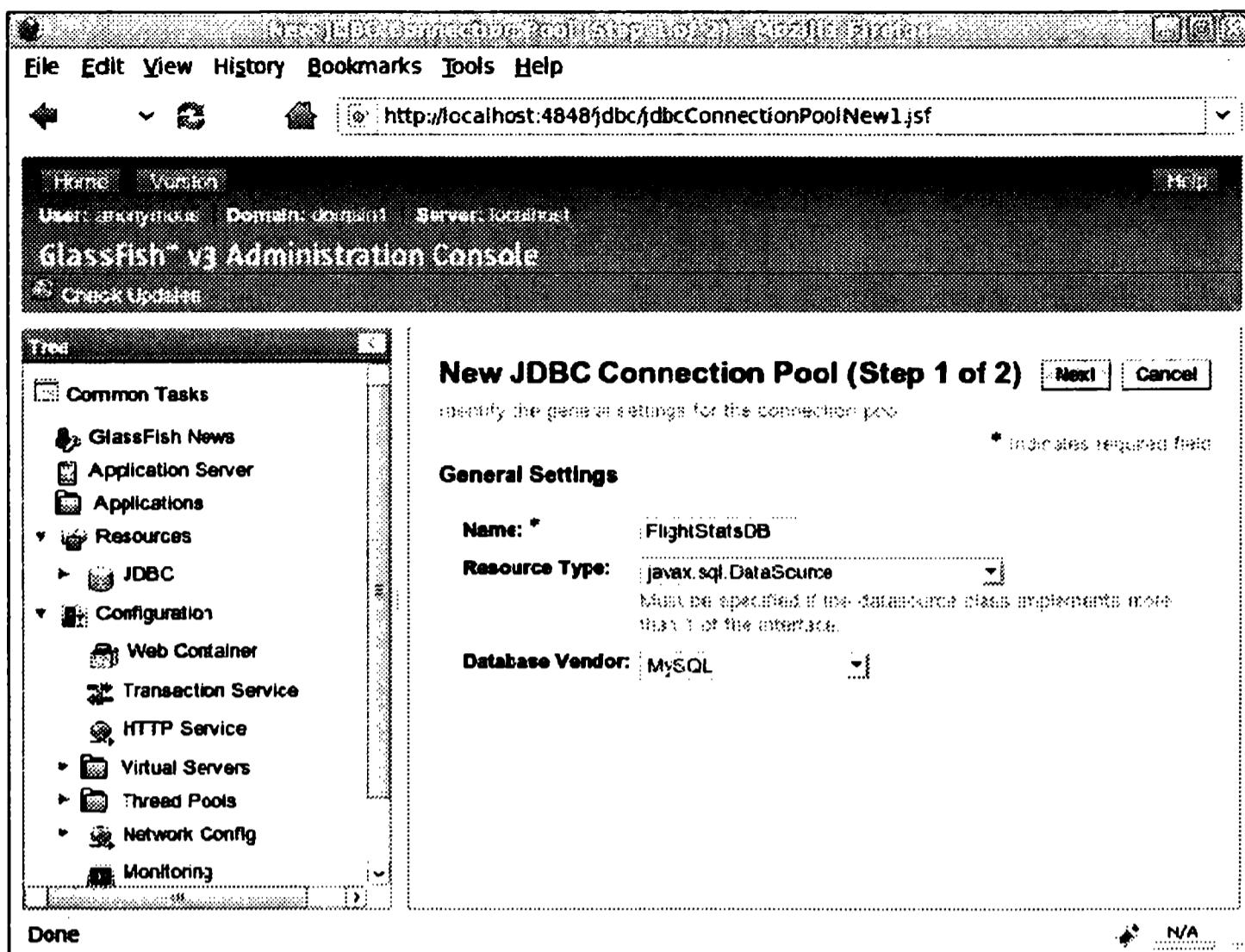
После того как драйвер JDBC будет скопирован в требуемое место и сервер приложений будет перезапущен, нужно войти в консоль администрирования, указав в адресной строке обозревателя URL: `http://localhost:4848`.

Затем нужно щелкнуть по узлам **Ресурсы (Resources) | JDBC | Пулы соединений (Connection Pools)** в панели навигации, в левой части консоли администрирования. Содержимое обозревателя теперь должно выглядеть примерно так:

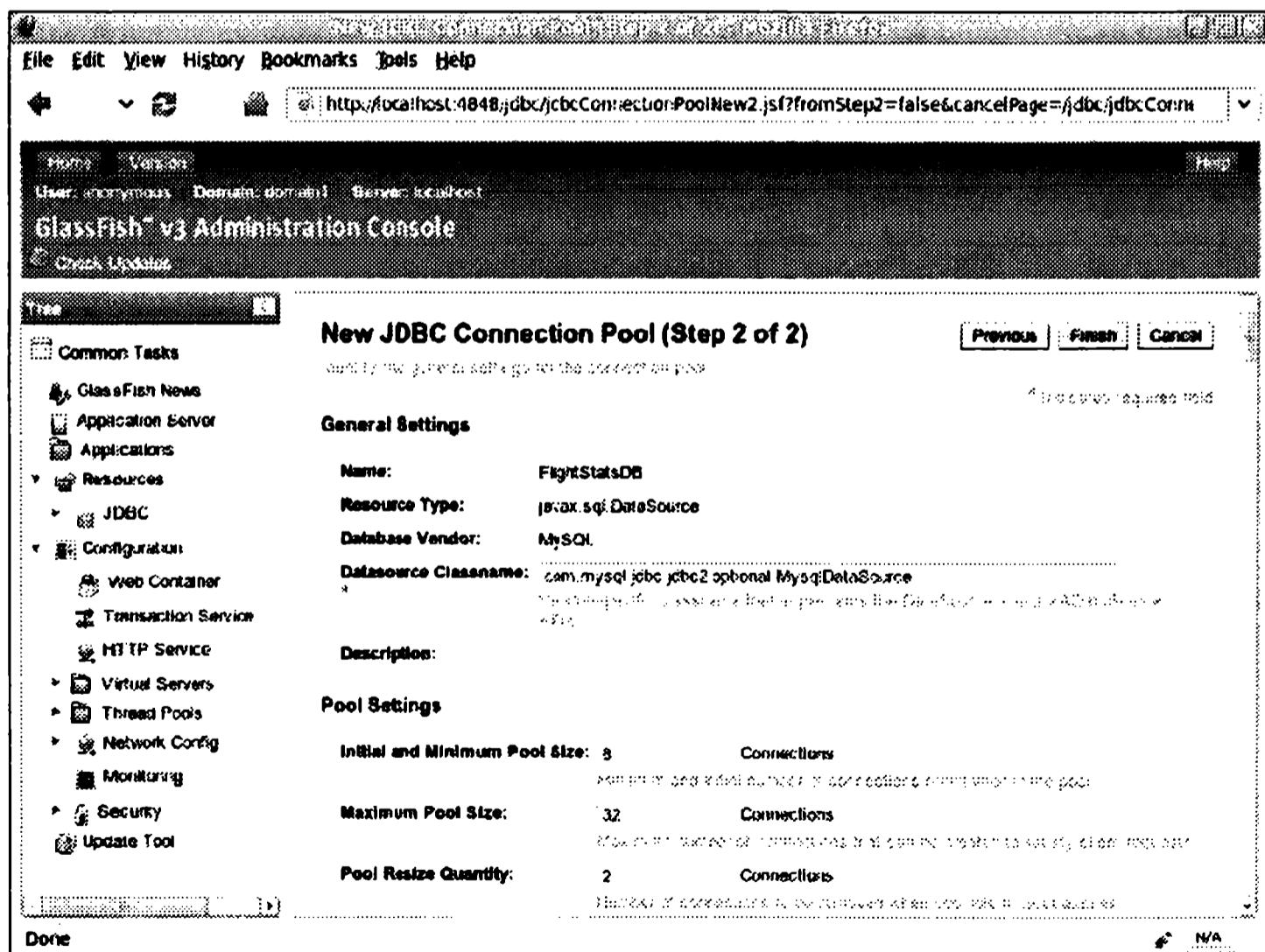


Далее нужно щелкнуть по кнопке **Новый... (New...)**. После ввода соответствующих данных для нашей СУРБД в поля, имеющиеся на странице, страница консоли администрирования должна будет выглядеть примерно так, как показано на следующем снимке экрана.

Настройка подключения к базе данных



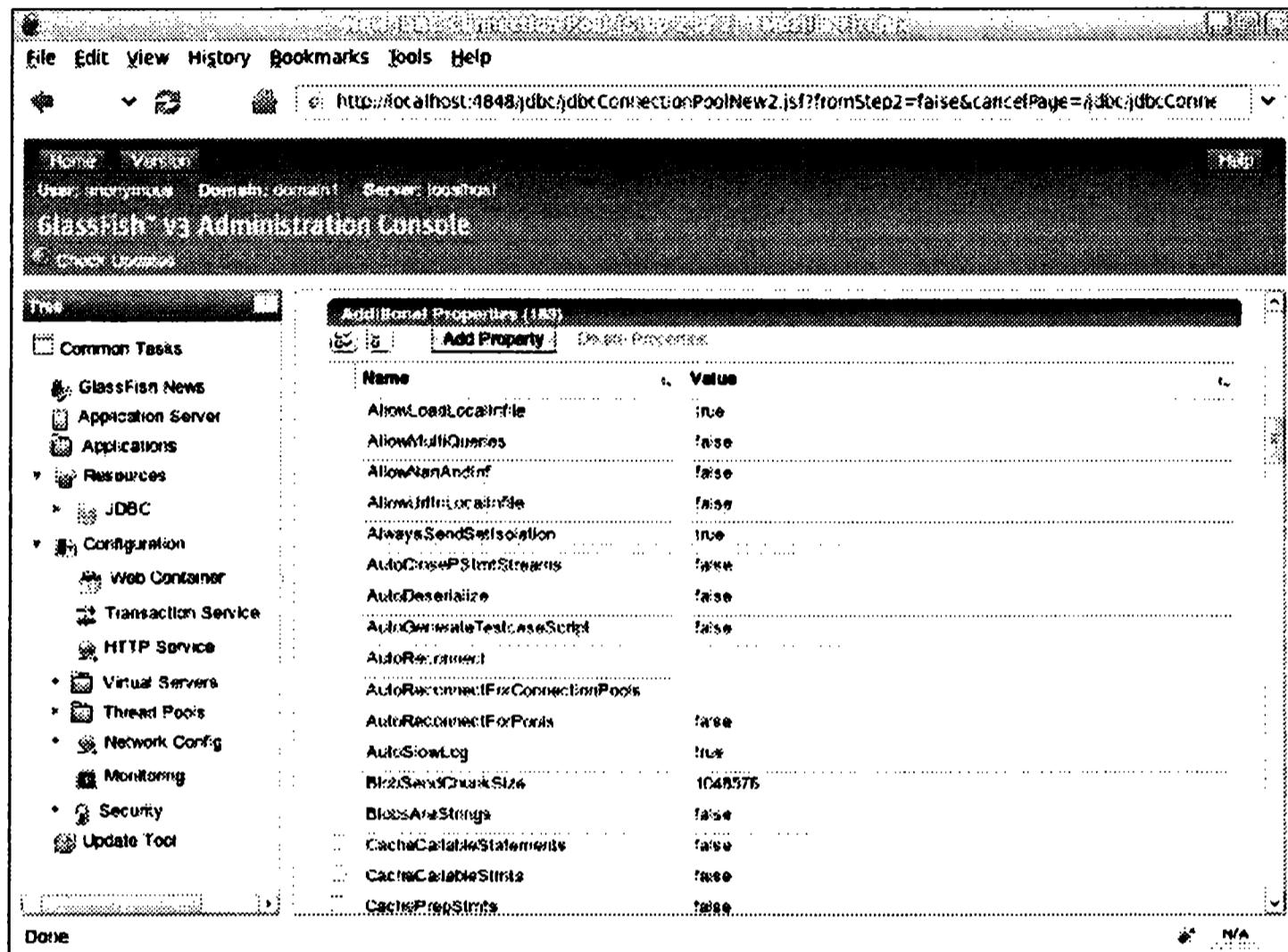
После ввода соответствующих данных для СУРБД и щелчка по кнопке **Далее (Next)** мы увидим страницу наподобие следующей:



Большинство значений по умолчанию в верхней части этой страницы вполне приемлемы и не требуют изменений. Прокрутив страницу до конца и введя соответствую-

щие данные для нашей СУРБД, мы должны щелкнуть по кнопке **Завершить** (Finish) в верхнем правом углу основной панели страницы.

Состав и наименования свойств изменяются в зависимости от конкретной СУРБД, которую мы используем, но обычно у всех них есть поле URL, куда мы должны ввести URL JDBC для нашей базы данных, а также поля ввода имени пользователя и пароля, где мы должны указать учетные данные для аутентификации в нашей базе данных. Список свойств для нашего примера показан на следующем снимке экрана:



Созданный нами новый пул соединений теперь должен быть видимым в списке пулов соединений:

JNDI Name	Resource Type	Datasource Classname	Description
TimerPool	javax.sql.XADataSource	org.apache.derby.jdbc.EmbeddedXADataSource	
FlightStatsDB	javax.sql.DataSource	com.mysql.jdbc.optional.MysqlDataSource	
DerbyPool	javax.sql.DataSource	org.apache.derby.jdbc.ClientDataSource	

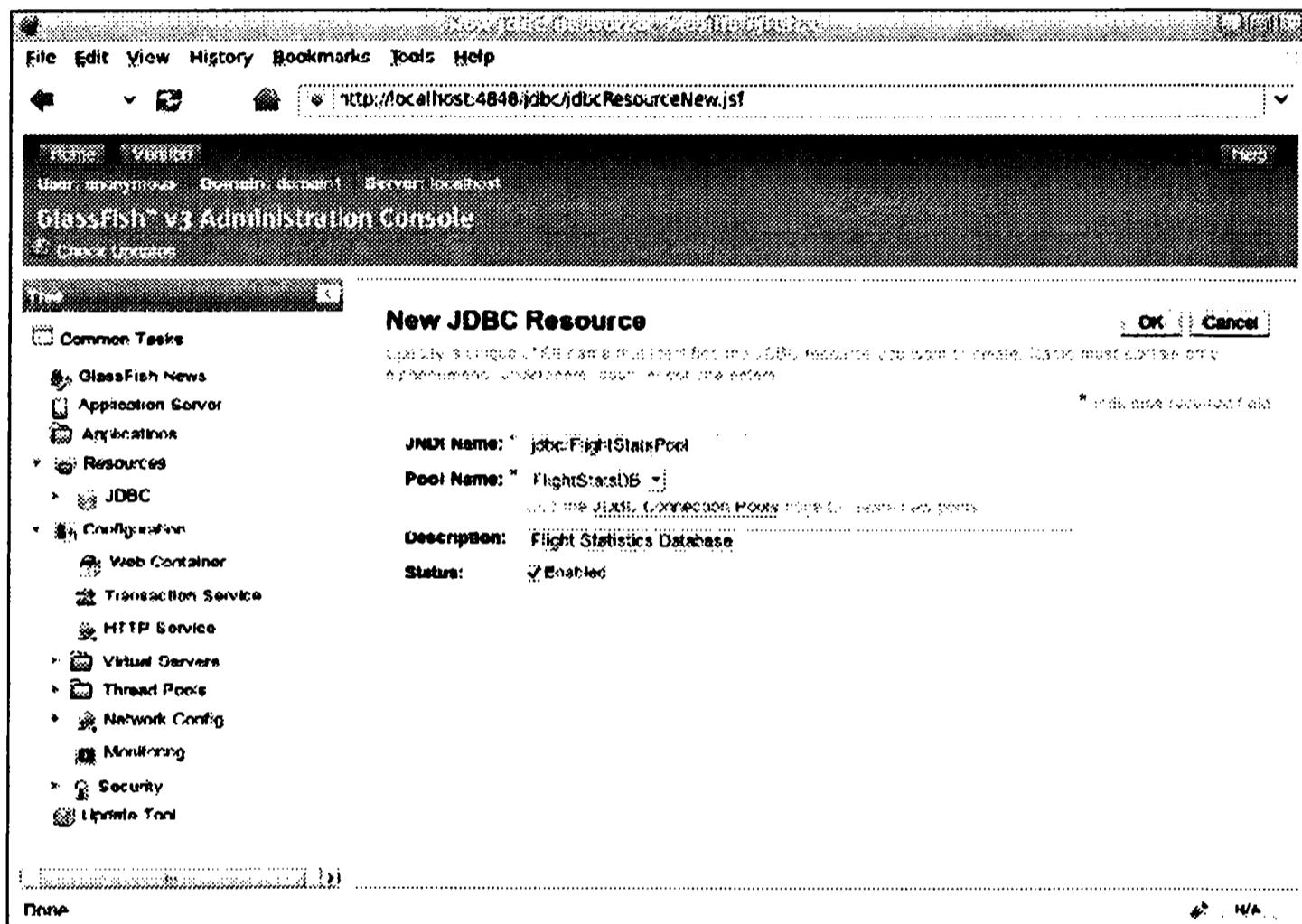
В большинстве случаев GlassFish должен быть перезапущен после создания нового пула соединений.

После перезапуска сервера и навигации к странице пулов соединений мы сможем убедиться в том, что наш пул соединений был создан успешно, щелкнув по его **JNDI-имени** (JNDI Name) и затем – на открывшейся странице с данными созданного пула, в панели основного содержания, – по кнопке **Пинг (Ping)**:

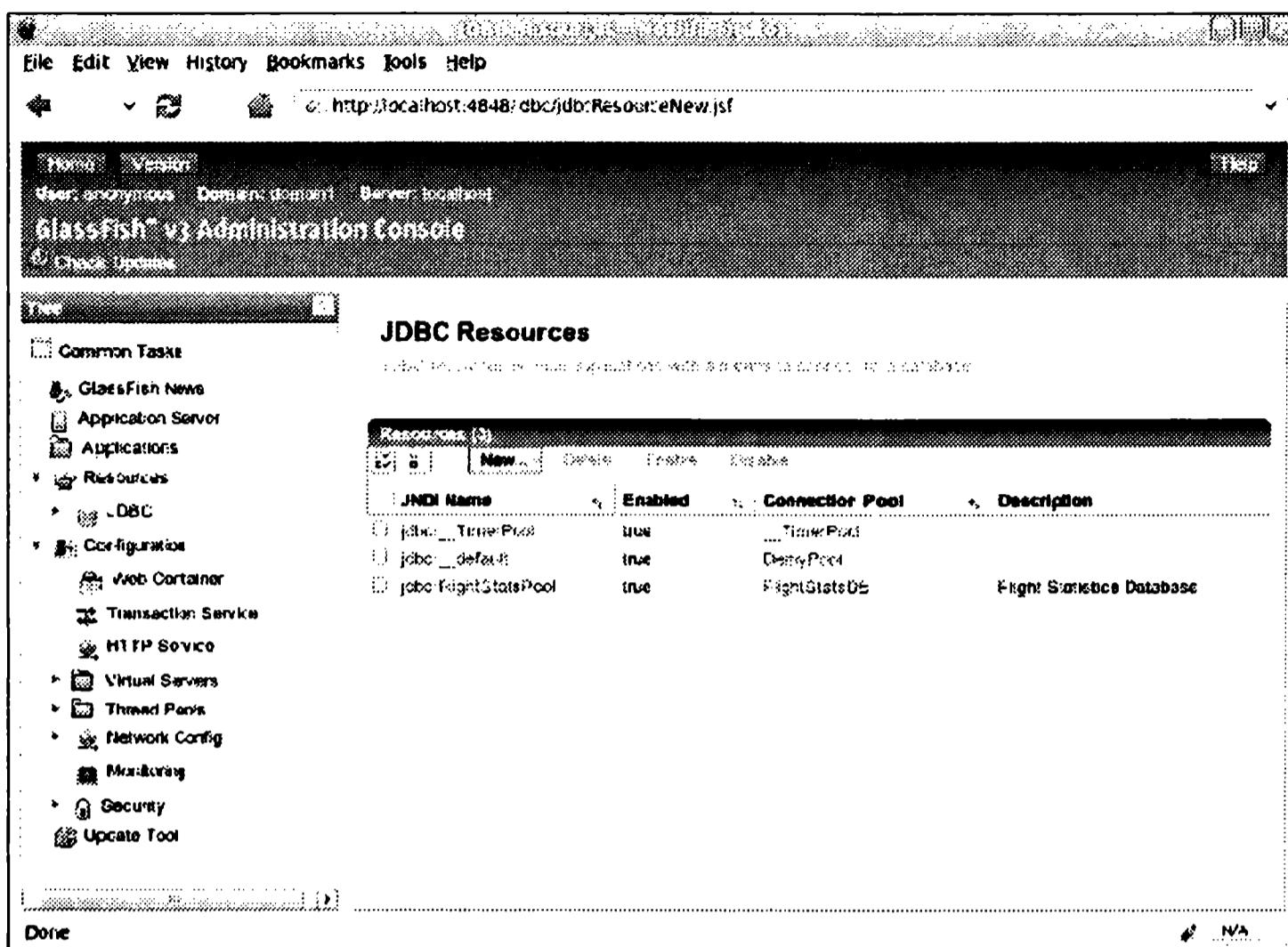
Теперь наш пул соединений готов к использованию нашими приложениями.

Создание источников данных

Приложения Java EE не получают доступ к пулам соединения напрямую. Вместо этого они получают доступ к источнику данных, который указывает на пул соединений. Для создания нового источника данных нам нужно щелкнуть по узлу **Ресурсы JDBC** (JDBC Resources) в панели навигации в левой части веб-консоли, а затем по кнопке **Новый...** (New...). После ввода соответствующей информации для нового источника данных в имеющиеся на странице поля мы увидим примерно такую страницу:



После щелчка по кнопке **OK** мы сможем увидеть созданный нами новый источник данных в таблице доступных ресурсов:



Заключительные замечания

Большинство примеров в этой книге предназначены для использования в среде IDE. Для упрощения разработки и развертывания приложений такие IDE, как NetBeans и Eclipse, можно интегрировать с сервером GlassFish. Читатели, желающие использовать одну из этих интегрированных сред разработки, могут обратиться к разделу «Приложение Б: Интеграция с IDE» на странице 390 для получения инструкций по интеграции IDE с сервером GlassFish.

Резюме

В этой главе мы обсудили, как загрузить и установить GlassFish. Мы также обсудили несколько методов развертывания приложения Java EE: через веб-консоль GlassFish, посредством утилиты командной строки `asadmin` и путем копирования файла в каталог `autodeploy`. Кроме того, мы обсудили основные задачи администрирования GlassFish, такие как создание доменов, создание соединений с базами данных, а также добавления пулов соединений и источников данных.

2

Разработка и развертывание сервлета

В этой главе мы обсудим, как разработать и развернуть *Сервлеты Java* (Java Servlets). Вот некоторые из затрагиваемых ниже тем:

- что такое сервlet;
- разработка, конфигурирование, упаковка и развертывание нашего первого сервлета;
- обработка HTML-формы;
- переадресация HTTP-запросов;
- перенаправление HTTP-откликов;
- сохранение данных между HTTP-запросами;
- новые возможности, появившиеся в Сервлете 3.0.

Что такое сервлет?

Сервлет является классом Java, который используется для расширения возможностей серверов, предназначенных для размещения приложений. Сервлеты могут отвечать на запросы и генерировать отклики. Базовым классом для всех сервлетов является `javax.servlet.GenericServlet`. Этот класс определяет обобщенный, независимый от протокола сервлет.

Безусловно, наиболее распространенный тип сервлета – HTTP-сервлет. Этот тип сервлета используется в обработке HTTP-запросов и генерировании HTTP-откликов. HTTP-сервлет представляет собой класс, который расширяет класс `javax.servlet.http.HttpServlet`, являющийся подклассом базового класса `javax.servlet.GenericServlet`.

Сервлет должен реализовывать один или более методов для ответов на определенные HTTP-запросы. Эти переопределяемые методы определены в родительском классе `HttpServlet`. Как видно из нижеприведенной таблицы, эти методы названы таким образом, чтобы можно было интуитивно понять, какой метод использовать в том или ином случае:

HTTP-запрос	Метод HTTP-сервлета
GET	doGet(HttpServletRequest request, HttpServletResponse response)
POST	doPost(HttpServletRequest request, HttpServletResponse response)
PUT	doPut(HttpServletRequest request, HttpServletResponse response)
DELETE	doDelete(HttpServletRequest request, HttpServletResponse response)

Каждый из этих методов принимает одни и те же два параметра, а именно: экземпляр класса, реализующего интерфейс `javax.servlet.http.HttpServletRequest`, и экземпляр класса, реализующего интерфейс `javax.servlet.http.HttpServletResponse`. Эти интерфейсы будут подробно описаны далее в этой главе.



Разработчики приложений никогда не вызывают эти методы напрямую. Их автоматически вызывает сервер приложений всякий раз, когда получает соответствующий HTTP-запрос.

Из четырех методов, перечисленных выше, методы `doGet()` и `doPost()` на сегодняшний день используются наиболее часто.

HTTP-запрос GET генерируется всякий раз, когда пользователь вводит URL сервлета в адресной строке обозревателя, либо щелкает по ссылке, указывающей на URL сервлета, либо отправляет HTML-форму с использованием метода GET, в которой атрибут `action` указывает на URL сервлета. В любом из этих случаев код сервлета внутри метода `doGet()` запускается на выполнение.

HTTP-запрос POST обычно генерируется, когда пользователь отправляет HTML-форму с использованием метода POST и ее атрибут `action` указывает на URL сервлета. В этом случае код сервлета внутри метода `doPost()` запускается на выполнение.

Написание нашего первого сервлета

В разделе «*Развертывание нашего первого приложения Java EE*» главы 1 на стр. 35, мы развернули простое приложение, которое печатало сообщение в окне обозревателя. Это приложение в целом состояло из одного сервлета. В этом разделе мы увидим, как этот сервлеt был разработан, сконфигурирован и упакован.

Код сервлета следующий:

```
package net.ensode.glassfishbook.simpleapp;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class SimpleServlet extends HttpServlet
{
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response)
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("Hello, World!");
    }
}
```



```
try
{
    response.setContentType("text/html");
    PrintWriter printWriter = response.getWriter();
    printWriter.println("<h2>");
    printWriter.println("Если вы читаете это, ваш сервер приложений");
    printWriter.println("работает нормально!");
    printWriter.println("</h2>");
}
catch (IOException ioException)
{
    ioException.printStackTrace();
}
}
```

Поскольку этот сервлет предназначается для выполнения, когда пользователь вводит его URL в адресной строке окна обозревателя, мы должны переопределить в нем метод `doGet()`, определенный в родительском классе `HttpServlet`. Как мы объяснили ранее, этот метод принимает два параметра: экземпляр класса, реализующего интерфейс `javax.servlet.http.HttpServletRequest`, и экземпляр класса, реализующего интерфейс `javax.servlet.http.HttpServletResponse`.



Даже при том, что `HttpServletRequest` и `HttpServletResponse` являются интерфейсами, разработчики приложений обычно не пишут классы, реализующие их. Когда управление из HTTP-запроса передается сервлету, сервер приложений (в нашем случае GlassFish) сам предоставляет объекты, реализующие эти интерфейсы.

Первым делом наш метод `doGet()` устанавливает тип контента для объекта `HttpServletResponse` в "text/html". Если мы забудем это сделать, используемым типом контента по умолчанию будет "text/plain"; это означает, что HTML-теги, использующие несколько строк, будут выведены на экран в окне обозревателя, т. е. не будут интерпретироваться как HTML-теги.

Затем мы получаем экземпляр `java.io.PrintWriter`, вызывая метод `HttpServletResponse.getWriter()`. Затем мы можем отправить текстовый вывод обозревателю, вызывая методы `PrintWriter.print()` и `PrintWriter.println()` (предыдущий пример использует исключительно `println()`). Поскольку мы устанавливаем тип контента в "text/html", любые HTML-теги должным образом интерпретируются обозревателем.

Компиляция сервлета

Чтобы скомпилировать сервлет, библиотека Java, включенная в GlassFish, должна находиться в CLASSPATH. Эту библиотеку называют `javaee.jar`; ее можно найти в каталоге `[Каталог установки glassfish]/glassfish/lib`.

Чтобы выполнить компиляцию из командной строки, используя компилятор `javac`, потребуется выполнить команду наподобие следующей¹:

¹ Все должно быть записано в одной строке. Прим. перев.

```
javac -cp /opt/sges-v3/glassfish/lib/javaee.jar net/ensode/
glassfishbook/simpleapp/SimpleServlet.java
```

Конечно, теперь очень немногие из разработчиков компилируют «сырой код» `javac` компилятором. Вместо него используются или графический IDE, или инструмент сборки командной строки – такой, например, как Apache ANT или Apache Maven. Обратитесь к документации по IDE или инструменту построения, чтобы узнать, как добавить библиотеку `javaee.jar` в его CLASSPATH.



Maven

Apache Maven – инструмент сборки, подобный ANT. Вместе с тем Maven предлагает много преимуществ по сравнению с ANT, включая автоматическую загрузку зависимостей и служебных команд для компиляции и упаковки приложений. Именно Maven использовался для компиляции и упаковки всех примеров в этой книге, поэтому рекомендуется установить его для упрощения создания примеров. При использовании этого инструмента сборки код может быть скомпилирован и упакован путем запуска следующей команды в корневом каталоге проекта (в нашем случае – `simpleapp`): `mvn package`. Сам Maven можно загрузить с веб-сайта <http://maven.apache.org/>.

Конфигурирование сервлета

Прежде чем мы сможем развернуть наш сервлет, мы должны его сконфигурировать. Все веб-приложения Java EE могут быть сконфигурированы через XML-файл дескриптора развертывания, называемый `web.xml`, или через аннотации. В этом разделе мы обсудим, как сконфигурировать веб-приложение Java EE через `web.xml`; далее рассмотрим конфигурирование с помощью аннотаций. Дескриптор развертывания `web.xml` для нашего сервлета следующий:

```
<?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">
    <servlet>
        <servlet-name>SimpleServlet</servlet-name>
        <servlet-class>
            net.ensode.glassfishbook.simpleapp.SimpleServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>SimpleServlet</servlet-name>
        <url-pattern>/simpleservlet</url-pattern>
    </servlet-mapping>
</web-app>
```

Первые несколько строк являются шаблоном XML, объявляющим версию XML и тип кодировки, а также схему, используемую для XML-файла, и другую информацию. Можно смело копировать и вставлять эти строки для их повторного использования в другие приложения. XML-теги `<servlet>` и `<servlet-mapping>` используются для фактического конфигурирования нашего сервлета.

Тег `<servlet>` содержит два вложенных тега: `<servlet-name>` определяет логическое имя сервлета, а `<servlet-class>` указывает класс Java, определяющий сервлет.

Тег `<servlet-mapping>` также содержит два вложенных тега: `<servlet-name>` соответствует значению аналогичного тега внутри тега `<servlet>`, а `<url-pattern>` устанавливает шаблон URL, для которого будет выполняться сервлет.

Тег `<url-pattern>` может быть указан одним из двух способов: с помощью префикса пути (что и наблюдается в предыдущем примере) либо путем указания суффикса расширения.

Значения префикса пути для `<url-pattern>` указывают для того, чтобы любые пути URL, начинающиеся с данного шаблона, могли обслуживаться соответствующим сервлетом. Значения префикса пути должны начинаться с символа / (наклонная черта).



Веб-приложения Java EE запускаются внутри корневого контекста. Корневой контекст – это первая часть вхождения строки в URL, которая не является ни именем сервера, ни IP-адресом, ни портом. Например, в URL `http://localhost:8080/simpleapp/simpleServlet` вхождение строки `simpleapp` является корневым контекстом. Значение для `<url-pattern>` устанавливается относительно этого корневого контекста приложения.

Значения суффикса расширения для `<url-pattern>` указывают, что любые URL, заканчивающиеся на вхождение строки, содержащей данный суффикс, будут обслуживаться соответствующим сервлетом. В предыдущем примере мы использовали префикс пути. Если бы мы использовали суффикс расширения, то тег `<servlet-mapping>` выглядел бы примерно так:

```
<servlet-mapping>
    <servlet>SimpleServlet</servlet>
    <url-pattern>*.foo</url-pattern>
</servlet-mapping>
```

Это позволит направлять любые URL, заканчивающиеся вхождением строки .foo, к нашему сервлету.

Причина, по которой тег `<servlet-name>` указывается дважды (первый раз – в теге `<servlet>`, второй раз – в теге `<servlet-mapping>`), состоит в том, что у веб-приложения Java EE может быть более одного сервлета. У каждого из сервлетов должен быть тег `<servlet>` в файле дескриптора развертывания приложения `web.xml`. У каждого тега `<servlet>` должен быть соответствующий тег `<servlet-mapping>`. Вложенный в него тег `<servlet-name>` используется для того, чтобы указать, какому из тегов `<servlet>` соответствует тег, выполняющий отображение `<servlet-mapping>`.



Java EE-файл `web.xml` может содержать много дополнительных XML-тегов. Однако дополнительные теги для данного простого примера не требуются. В следующих примерах дополнительные теги будут обсуждаться по мере необходимости.

Прежде чем мы сможем выполнить наш сервлет, его нужно упаковать как часть веб-приложения в WAR-файл (веб-архив).

Упаковка веб-приложения

Все веб-приложения Java EE должны быть упакованы в WAR-файл (веб-архив), прежде чем они смогут быть развернуты. WAR-файл является всего лишь сжатым файлом, содержащим наш код и конфигурацию. WAR-файлы могут быть созданы любой утилитой, которая может создавать файлы в формате ZIP (например, WinZip, 7-Zip и др.). Кроме того, многие IDE Java и инструменты сборки, такие как ANT и Maven, автоматизируют создание WAR-файла.

WAR-файл должен содержать следующие каталоги (в дополнение к его корневому каталогу):

- **WEB-INF**
- **WEB-INF/classes**
- **WEB-INF/lib**

Корневой каталог содержит страницы JSP (рассматриваются в следующей главе), файлы HTML, файлы JavaScript и файлы CSS.

- **WEB-INF** содержит descriptors развертывания, такие как `web.xml`;
- **WEB-INF/classes** содержит скомпилированный код (`.class`-файлы) и может дополнительно содержать файлы свойств. Точно так же, как и для любых Java-классов, структура каталогов должна соответствовать структуре пакета. По этой причине данный каталог обычно содержит несколько подкаталогов, соответствующих содержащимся в них классам;
- **WEB-INF/lib** содержит JAR-файлы с любыми библиотеками, от которых может зависеть наш код.

У корневых каталогов **WEB-INF** и **WEB-INF/classes** могут быть подкаталоги. К любым ресурсам в подкаталоге корневого каталога (кроме **WEB-INF**) можно получить доступ, предварительно добавив имя подкаталога к имени файла. Например, если имеется подкаталог, называемый `CSS` и содержащий CSS-файл, называемый `style.css`, к этому CSS-файлу можно получить доступ из JSP-страниц и HTML-файлов в корневом каталоге с помощью строки:

```
<link rel="stylesheet" type="text/css" media="screen"
      href="css/style.css"/>
```

Обратите внимание на префикс `css` в имени файла, соответствующий каталогу, где находится CSS-файл.

Для создания WAR-файла с нуля следует создать описанную выше структуру каталогов в любом каталоге нашей системы, а затем выполнить следующие действия:

1. Скопировать файл `web.xml` в каталог **WEB-INF**.
2. Создать в каталоге **WEB-INF/classes** следующую структуру каталогов: `net/ensode/glassfishbook/simpleapp`.

3. Скопировать SimpleServlet.class в каталог simpleapp, созданный на предыдущем шаге.
4. Из командной строки выполнить следующую команду, находясь при этом в каталоге, расположенном выше каталога WEB-INF:
jar cvf simpleapp.war *.

После этого у нас должен появиться WAR-файл, готовый к развертыванию.



В случае использования Maven для построения кода, WAR-файл генерируется автоматически при выполнении команды mvn package. После этого WAR-файл можно найти в соответствующем target-каталоге. Он будет называться simpleapp.war.

Прежде чем мы сможем выполнить наше приложение, его нужно развернуть.

Развертывание веб-приложения

Как мы обсуждали в главе 1, существует несколько способов развертывания приложения. Самый легкий и самый простой способ развертывания любых приложений Java EE состоит в копировании файла развертывания (в нашем случае WAR-файла) в [Каталог установки glassfish]/glassfish/domains/domain1/autodeploy.

После копирования WAR-файла в каталог autodeploy системный журнал должен показать сообщение, подобное следующему:

```
[#|2010-04-08T19:39:48.313-0400|INFO|glassfishv3.0|javax.enterprise.system.tools.deployment.org.glassfish.deployment.common|_ThreadID=28;_ThreadName=Thread-1|[AutoDeploy] Successfully autodeployed : /home/heffel/sges-v3/glassfish/domains/domain1/autodeploy/simpleapp.war.|#]
```



Системный журнал можно найти по адресу: [Каталог установки glassfish]/glassfish/domains/domain1/logs/server.log.

Последняя строка сообщения должна содержать строку «Successfully autodeployed», указывающую, что наш WAR-файл успешно развернут.

Тестирование веб-приложения

Чтобы убедиться в том, что сервлет был развернут должным образом, мы должны указать в адресной строке нашего обозревателя адрес:

<http://localhost:8080/simpleapp/simple servlet>

После его обработки обозревателем мы должны будем увидеть примерно такую страницу:



Неудивительно, что здесь показано то же самое сообщение, которое мы видели при развертывании приложения в главе 1 (см. стр. 37), поскольку здесь мы развертываем то же самое приложение.

Ранее в этой главе мы упоминали, что пути URL для приложения Java EE указываются относительно их корневого контекста. Корневым контекстом по умолчанию для WAR-файла является непосредственно имя WAR-файла (минус расширение .war). Как видно из предыдущего снимка экрана, корневым контекстом нашего приложения является simpleapp, что, как оказывается, соответствует имени WAR-файла. Это значение по умолчанию может быть изменено путем добавления дополнительных файлов конфигурации в каталог WEB-INF WAR-файла. Имя этого конфигурационного файла должно быть sun-web.xml. Приведем пример файла sun-web.xml, который изменяет корневой контекст нашего приложения со значения по умолчанию simpleapp на simple:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD
Application Server 8.1 Servlet 2.4//EN" "http://www.sun.com/
software/appserver/dtds/sun-web-app_2_4-1.dtd">
<sun-web-app>
    <context-root>/simple</context-root>
</sun-web-app>
```

Как видно из этого примера, корневой контекст приложения должен быть указан в теге <context-root> конфигурационного файла sun-web.xml. После повторного развертывания файла simpleapp.war направление нашего обозревателя по адресу `http://localhost:8080/simple/simple servlet` приведет к выполнению нашего сервлета.



Файл sun-web.xml может содержать много дополнительных тегов для конфигурирования различных аспектов приложения. Дополнительные теги будут обсуждаться в соответствующих разделах этой книги.

Обработка HTML-форм

К сервлетам редко получают доступ путем ввода их URL непосредственно в адресной строке обозревателя. Наиболее популярный способ использования сервлетов заключается в обработке данных, вводимых пользователями в HTML-форму. Ниже мы поясним этот процесс.

Прежде чем углубиться в код сервлета и HTML-разметку, давайте рассмотрим файл web.xml для нашего нового приложения:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
           http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <servlet>
        <servlet-name>FormHandlerServlet</servlet-name>
        <servlet-class>
            net.enseode.glassfishbook.formhandling.FormHandlerServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>FormHandlerServlet</servlet-name>
        <url-pattern>/formhandlerservlet</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>dataentry.html</welcome-file>
    </welcome-file-list>
</web-app>
```

Данный файл web.xml очень похож на тот, который мы видели в предыдущем разделе. Однако он содержит XML-тег, который ранее нам не встречался, а именно <welcome-file>. Тег <welcome-file> определяет, к какому файлу непосредственно будет обращаться пользователь, когда он вводит URL, оканчивающийся в корневом контексте приложения (для данного примера URL был бы таким: <http://localhost:8080/formhandling>, по названию WAR-файла formhandling.war), и не указывает пользовательский корневой контекст. HTML-файл, содержащий форму, мы назовем dataentry.html. Таким образом, GlassFish будет отображать его в обозревателе, в случае когда пользователь введет URL нашего приложения и не укажет имя файла.

 **Примечание** Если ни один файл <welcome-file> не указан в дескрипторе развертывания приложения web.xml, то GlassFish будет искать файл под названием index.html и использовать его в качестве файла приглашения. Если и этот файл не обнаружится, GlassFish будет искать файл, называемый index.jsp и использовать его в качестве файла приглашения. Если ни один из перечисленных файлов не найден, на экран будет выведен список имеющихся каталогов.

HTML-файл, содержащий форму для нашего приложения, выглядит следующим образом:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
          "http://www.w3.org/TR/html4/loose.dtd">
```

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Страница ввода данных</title>
  </head>
  <body>
    <form method="post" action="formhandlerservlet">
      <table cellpadding="0" cellspacing="0" border="0">
        <tr>
          <td>Пожалуйста введите какой-нибудь текст:</td>
          <td><input type="text" name="enteredValue" /></td>
        </tr>
        <tr>
          <td></td>
          <td><input type="submit" value="Отправить"></td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

Обратите внимание на то, что значение атрибута `action` тега `<form>` соответствует значению в теге `<url-pattern>` сервлета в файле дескриптора развертывания приложения `web.xml` (минус начальная наклонная черта). Поскольку значением атрибута `method` тега `<form>` является "post", будет выполняться метод `doPost()` нашего сервлета при отправке формы.

Теперь давайте рассмотрим код нашего сервлета:

```
package net.ensode.glassfishbook.formhandling;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class FormHandlerServlet extends HttpServlet
{
    protected void doPost(HttpServletRequest request, HttpServletResponse
                          response)
    {
        String enteredValue;
        enteredValue = request.getParameter("enteredValue");
        response.setContentType("text/html");
        PrintWriter printWriter;
        try
        {
            printWriter = response.getWriter();
            printWriter.println("<p>");
            printWriter.print("Вы ввели: ");
            printWriter.print(enteredValue);
            printWriter.print("</p>");
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

Как видно из примера, мы получаем ссылку на значение, введенное пользователем, вызывая метод `request.getParameter()`. Этот метод получает объект типа `String` в качестве единственного параметра. Значение этой строки должно соответствовать имени поля ввода в HTML-файле. В нашем случае в HTML-файле текстовое поле называется `enteredValue`:

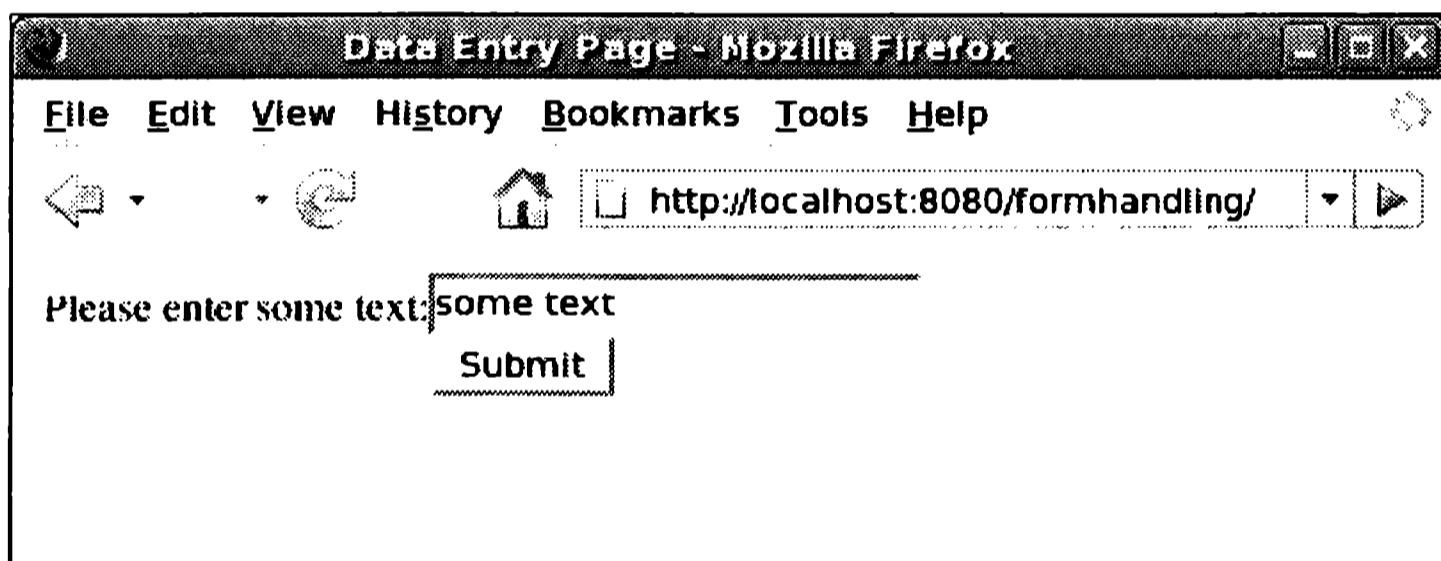
```
<input type="text" name="enteredValue"/>
```

Поэтому у сервлета имеется соответствующая строка:

```
enteredValue = request.getParameter("enteredValue");
```

Эта строка программы используется для получения текста, введимого пользователем, и сохранения его в строковой переменной, называемой `enteredValue` (имя переменной не должно соответствовать имени поля ввода, но, назвав ее таким образом, мы лучше запомним, какое значение эта переменная хранит).

После упаковки предыдущих трех файлов в WAR-файл `formhandling.war` и развертывания WAR-файла мы сможем увидеть отображение файла `dataentry.html`, вводя в адресной строке обозревателя `http://localhost:8080/formhandling`:



После того как пользователь введет *некоторый текст* в текстовое поле и отправит форму (либо нажав на клавиатуре клавишу **Enter**, либо щелкнув по кнопке **Отправить** (`Submit`)), мы должны будем увидеть вывод сервлета:



Метод `HttpServletRequest.getParameter()` может использоваться для получения значения любого поля ввода HTML, которое может возвратить только одно значение (текстовые поля, текстовые области, одиночный выбор, переключатели,

скрытые поля и т. д.). Процедура получения любого из значений этих полей идентична. Другими словами, сервлет не заботится о том, ввел ли пользователь значение в текстовое поле, выбрал ли элемент из набора переключателей и т. д. До тех пор пока имя поля ввода соответствует запрашиваемому имени, значение будет передаваться методом `getParameter()` и предыдущий код будет работать.



При работе с переключателями у всех связанных переключателей должно быть одно и то же имя. Вызов метода `HttpServletRequest.getParameter()` и передача ему имени переключателя возвратит значение установленного переключателя.

Некоторые поля ввода HTML, такие как флагки и поля множественного выбора, позволяют пользователю выбирать более одного значения. Для этих полей вместо метода `HttpServletRequest.getParameter()` используется метод `HttpServletRequest.getParameterValues()`. Он также получает строку, содержащую имя поля ввода, в качестве его единственного параметра и возвращает массив строк, содержащих все значения, которые были выбраны пользователем.

Давайте добавим второй HTML-файл и второй сервлет к нашему приложению, чтобы пояснить данный случай. Соответствующие разделы этого HTML-тега показаны в следующем коде:

```
<form method="post" action="multiplevaluefieldhandlerservlet">
<p>Пожалуйста выберите одну или более опций.</p>
<table cellpadding="0" cellspacing="0" border="0">
<tr>
    <td><input name="options" type="checkbox" value="опция1"/>
        Option 1
    </td>
</tr>
<tr>
    <td><input name="options" type="checkbox" value="опция2"/>
        Option 2
    </td>
</tr>
<tr>
    <td><input name="options" type="checkbox" value="опция3"/>
        Option 3
    </td>
</tr>
<tr>
    <td><input type="submit" value="Отправить"/></td>
    <td></td>
</tr>
</table>
</form>
```

Новый HTML-файл содержит простую форму с тремя флагками и кнопкой **Отправить** (Submit). Обратите внимание, что каждый флагок имеет одно и то же значение атрибута `name`. Как мы упоминали ранее, любые флагки, по которым щелкает пользователь, будут отправлены сервлету.

Теперь рассмотрим сервлет, который обрабатывает эту HTML-форму:

```
package net.ensode.glassfishbook.formhandling;
import java.io.IOException;
```



```
import java.io.PrintWriter;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class MultipleValueFieldHandlerServlet extends HttpServlet
{
    protected void doPost(HttpServletRequest request, HttpServletResponse
                          response)
    {
        String[] selectedOptions = request.getParameterValues("options");
        response.setContentType("text/html");
        try
        {
            PrintWriter printWriter = response.getWriter();
            printWriter.println("<p>");
            printWriter.print("Были выбраны следующие опции:");
            printWriter.println("<br/>");
            if (selectedOptions != null)
            {
                for (String option : selectedOptions)
                {
                    printWriter.print(option);
                    printWriter.println("<br/>");
                }
            }
            else
            {
                printWriter.println("Ни каких опций выбрано не было.");
            }
            printWriter.println("</p>");
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

Код вызывает метод `request.getParameterValues()` и присваивает возвращаемое им значение переменной `selectedOptions`. Ниже, в методе `doPost()`, код обходит массив `selectedOptions` и печатает выбранные значения в окне обозревателя.



Предыдущий код использует улучшенный цикла `for`, введенный в язык начиная с JDK 1.5.

Если ни один флажок не будет выбран, метод `request.getParameterValues()` возвратит `null`. Поэтому правильным решением будет выполнение проверки на `null` перед попыткой выполнения обхода возвращаемых этим методом значений.

Прежде чем этот новый сервлет может быть развернут, нужно добавить следующие строки в файл дескриптора развертывания приложения `web.xml`:

```
<servlet>
    <servlet-name>MultipleValueFieldHandlerServlet</servlet-name>
    <servlet-class>
        net.ensode.glassfishbook.formhandling.
        MultipleValueFieldHandlerServlet
```

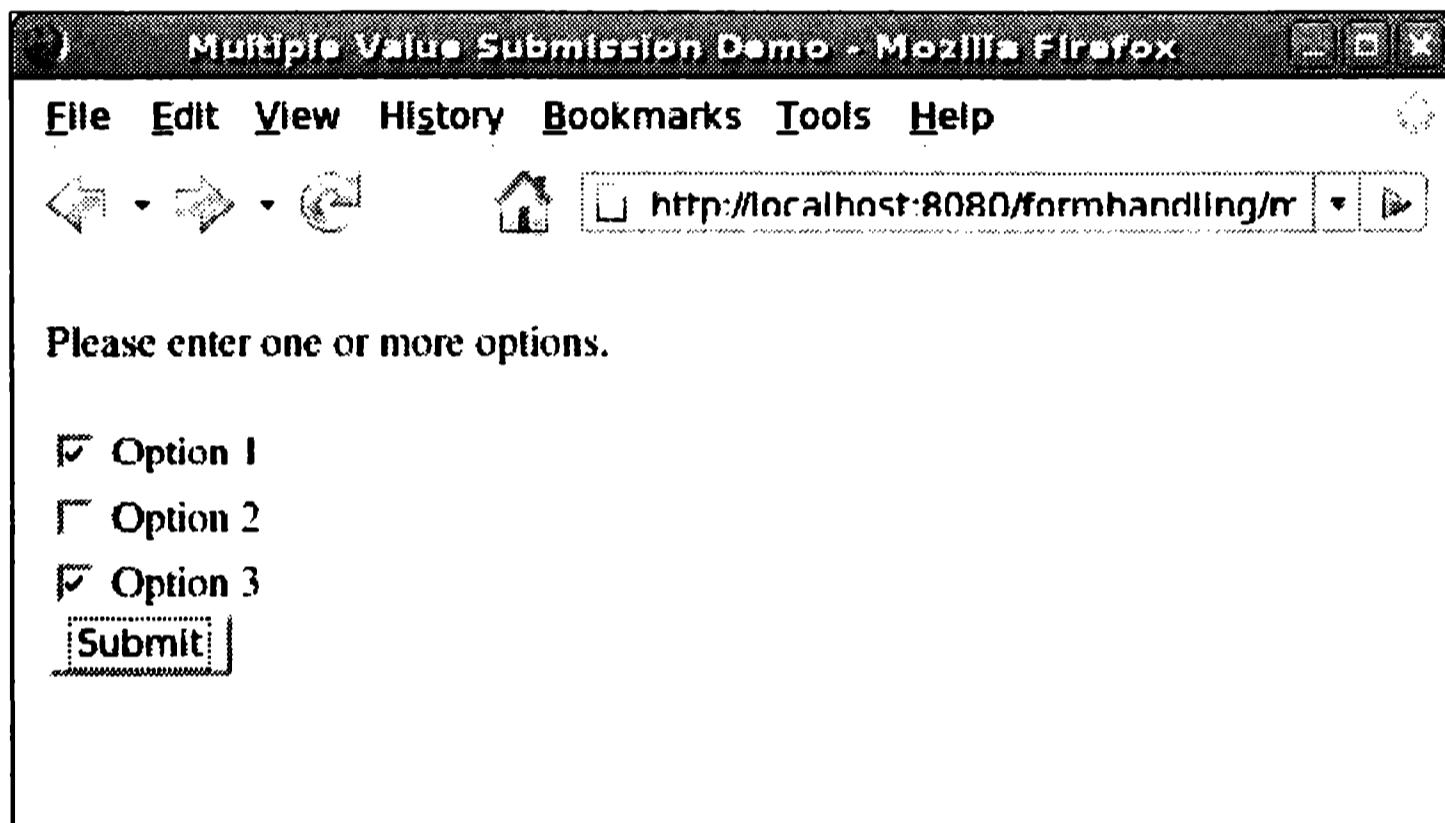
```
</servlet-class>
</servlet>
```

Мы должны были бы также добавить следующие строки кода:

```
<servlet-mapping>
    <servlet-name>MultipleValueFieldHandlerServlet</servlet-name>
    <url-pattern>/multiplevaluefieldhandlerServlet</url-pattern>
</servlet-mapping>
```

Эти строки присваивают логическое имя и URL новому сервлету.

После повторного создания файла `formhandling.war` с добавлением к нему скомпилированного сервлета и HTML-файла, а также его повторного развертывания мы сможем увидеть результат произведенных изменений, введя в адресной строке обозревателя следующий URL: `http://localhost:8080/formhandling/multiplevaluedataentry.html`.



После отправки формы управление будет передано нашему сервлету, и окно обозревателя приобретет примерно такой вид:



Конечно, фактическое сообщение, которое мы увидим в окне обозревателя, будет зависеть от того, по каким флагкам пользователь щелкнул.

Переадресация запросов и перенаправление откликов

Во многих случаях сервер обрабатывает данные формы, а затем передает управление другому серверу или странице JSP, чтобы произвести дополнительную обработку или вывести на экран подтверждающее сообщение. Сделать это можно двумя способами: или *запрос переадресуется* (forwarded), или *отклик перенаправляется* (redirected) к другому серверу или странице.

Переадресация запроса

Обратите внимание, что текст, выведенный на экран в примере из предыдущего раздела, соответствует значению атрибута value флагков, по которым был выполнен щелчок, а не меткам, выведенным на экран на предыдущей странице. Это может смутить пользователя. Давайте модифицируем сервер, чтобы изменить эти значения таким образом, чтобы они соответствовали меткам, а затем переадресуем запрос к другому серверу, который выведет на экран подтверждающее сообщение в окне обозревателя.

Новая версия `MultipleValueFieldHandlerServlet` содержит следующий код:

```
package net.ensode.glassfishbook.formhandling;

import java.io.IOException;
import java.util.ArrayList;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class MultipleValueFieldHandlerServlet extends HttpServlet
{
    protected void doPost(HttpServletRequest request, HttpServletResponse
        response)
    {
        String[] selectedOptions = request.getParameterValues("options");
        ArrayList<String> selectedOptionLabels = null;
        if (selectedOptions != null)
        {
            selectedOptionLabels = new ArrayList<String>(selectedOptions.
                length);
            for (String selectedOption : selectedOptions)
            {
                if (selectedOption.equals("опция1"))
                {
                    selectedOptionLabels.add("Опция 1");
                }
                else if (selectedOption.equals("опция2"))
                {
                    selectedOptionLabels.add("Опция 2");
                }
                else if (selectedOption.equals("опция3"))
                {
                    selectedOptionLabels.add("Опция 3");
                }
            }
        }
        request.setAttribute("checkedLabels", selectedOptionLabels);
```

```

try
{
    request.getRequestDispatcher("ConfirmationServlet").
        forward(request, response);
}
catch (ServletException e)
{
    e.printStackTrace();
}
catch (IOException e)
{
    e.printStackTrace();
}
}
}
}

```

Эта версия сервлета выполняет итерации по выбранным опциям и добавляет соответствующую метку строки к ArrayList. Затем эта строка кода присоединяет объект к запросу, вызывая метод `request.setAttribute()`. Данный метод используется для присоединения любого объекта к запросу для того, чтобы у любого другого кода, которому мы передадим запрос, позже был доступ к нему.



Предыдущий код использует функцию generics, введенную в язык Java начиная с JDK 1.5 (для получения подробной информации см. <http://docs.oracle.com/javase/1.5.0/docs/guide/language/generics.html>).

После присоединения ArrayList к запросу мы переадресуем запрос к новому сервлету, используя следующую строку программы:

```

request.getRequestDispatcher("ConfirmationServlet").
    forward(request, response);

```

Аргумент String этого метода должен соответствовать значению тега `<url-pattern>` сервлета в файле дескриптора развертывания приложения `web.xml`.

На этом этапе исполнения управление переходит к нашему новому сервлету. Код этого нового сервлета следующий:

```

package net.ensode.glassfishbook.requestforward;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.List;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class ConfirmationServlet extends HttpServlet
{
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
        response)
    {
        try
        {
            PrintWriter printWriter;
            List<String> checkedLabels =
                (List<String>) request.getAttribute("checkedLabels");

```



```
response.setContentType("text/html");
printWriter = response.getWriter();
printWriter.println("<p>");
printWriter.print("Были выбраны следующие опции:");
printWriter.println("<br/>");
if (checkedLabels != null)
{
    for (String optionLabel : checkedLabels)
    {
        printWriter.print(optionLabel);
        printWriter.println("<br/>");
    }
}
else
{
    printWriter.println("Никаких опций выбрано не было.");
}
printWriter.println("</p>");

catch (IOException ioException)
{
    ioException.printStackTrace();
}
}
}
```

Этот код получает `ArrayList`, который был присоединен к запросу предыдущим сервлетом. Он выполняется при вызове метода `request.getAttribute()`. Параметр для этого метода должен соответствовать значению, использованному при присоединении объекта к запросу.

Как только предыдущий сервлет получит список меток опций, он выполнит их обход и затем выведет их на экран в окне обозревателя:



Переадресация запроса, описанная выше, работает только для других ресурсов (сервлетов и JSP-страниц) в том же самом контексте что и код, выполняющий переадресацию. Говоря простым языком, сервлет или JSP-страница, которым мы хотим передать управление, должны быть упакованы в том же самом WAR-файле, что и код,зывающий метод `request.getRequestDispatcher().forward()`. Если мы должны переадресовать пользователя к странице в другом контексте (или даже на другом сервере), мы можем сделать это путем перенаправления отклика (`redirecting response`) объекта.

Перенаправление отклика

Недостаток переадресации запроса, описанной в предыдущем разделе, заключается в том, что запросы могут быть переадресованы только к другим сервлетам или JSP-страницам в том же самом контексте. Если же мы должны переадресовать пользователя к странице в другом контексте (развернутой в другом WAR-файле на том же самом сервере или развернутой на другом сервере), мы должны использовать метод `HttpServletResponse.sendRedirect()`.

Чтобы пояснить перенаправление отклика, давайте разработаем простое веб-приложение, которое предлагает пользователю выбрать его любимую поисковую систему, затем направляет пользователя к предпочтаемой им/ей поисковой системе. HTML-страница для этого приложения могла бы выглядеть следующим образом:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Демонстрация перенаправления отклика</title>
  </head>
  <body>
    <form method="post" action="responseredirectionservlet">
      Пожалуйста выберите предпочтаемую вами поисковую систему:
      <table>
        <tr>
          <td><input type="radio" name="searchEngine"
            value="http://www.google.com">Google</td>
        </tr>
        <tr>
          <td><input type="radio" name="searchEngine"
            value="http://www.msn.com">MSN</td>
        </tr>
        <tr>
          <td><input type="radio" name="searchEngine"
            value="http://www.yahoo.com">Yahoo!</td>
        </tr>
        <tr>
          <td colspan="2"><input type="submit"
            value="Отправить"/></td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

HTML-форма в этом коде разметки содержит три переключателя. Значением для каждого из них является URL поисковой системы, соответствующей выбору пользователя. Обратите внимание, что для атрибута `name` каждого переключателя установлено одно и то же значение, а именно – `searchEngine`. Сервlet получит значение установленного переключателя, вызывая метод `request.getParameter()` и передавая строку `searchEngine` в качестве параметра. Это продемонстрировано в следующем коде:

```
package net.ensode.glassfishbook.responseredirection;
import java.io.IOException;
import java.io.PrintWriter;
```



```
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class ResponseRedirectionServlet extends HttpServlet
{
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
                          response) throws IOException
    {
        String url = request.getParameter("searchEngine");
        if (url != null)
        {
            response.sendRedirect(url);
        }
        else
        {
            PrintWriter printWriter = response.getWriter();
            printWriter.println("Никакая поисковая система выбрана не была.");
        }
    }
}
```

Вызывая `request.getParameter("searchEngine")`, предыдущий код присваивает URL выбранной поисковой системы – переменной `URL`. Затем (после того как будет выполнена проверка на `null` в случае, если пользователь щелкнул по кнопке **Отправить** (`Submit`), не выбрав никакую поисковую систему) код перенаправляет пользователя к выбранной поисковой системе, вызывая метод `response.sendRedirect()` и передавая переменную URL в качестве параметра.

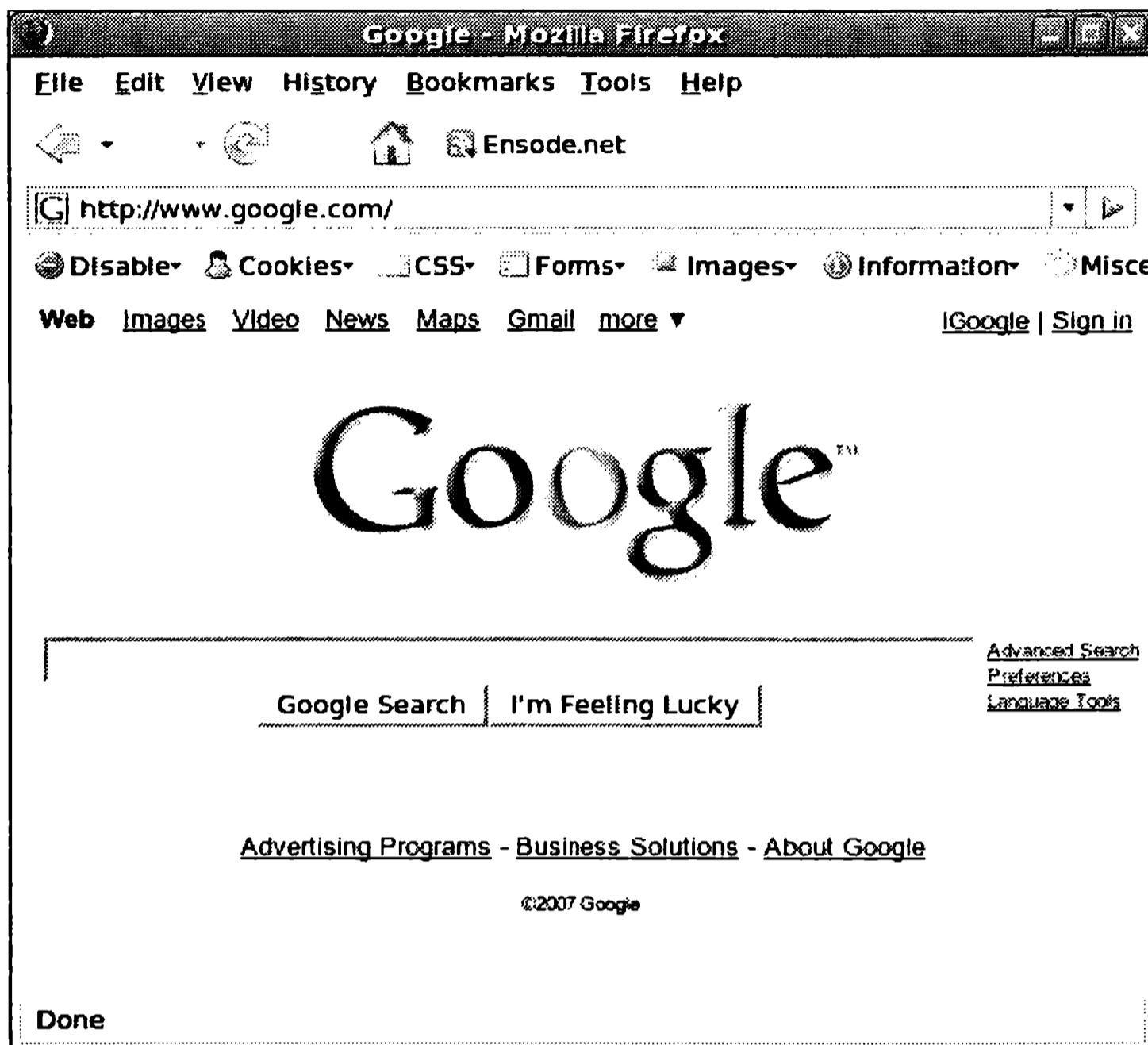
Файл `web.xml` для этого приложения будет довольно простым и незатейливым; здесь он не показан (его можно найти в загрузке кода для этой книги).

После упаковки кода и его развертывания мы сможем увидеть его в действии, введя следующий URL в адресную строку обозревателя: `http://localhost:8080/responseredirection/`.



После щелчка по кнопке **Отправить** (`Submit`) пользователь будет перенаправлен к предпочтаемой им поисковой системе.

Следует отметить, что перенаправление отклика, как было показано выше, создает новый HTTP-запрос для страницы, к которой мы перенаправляемся. Поэтому любые параметры и атрибуты исходного запроса будут потеряны.



Сохранение данных приложения между запросами

В предыдущем разделе мы видели, как можно сохранить объект в запросе, вызывая метод `HttpRequest.setAttribute()`, и как этот объект позже может быть получен путем вызова метода `HttpRequest.getAttribute()`. Этот подход работает только в том случае, если запрос был переадресован к сервлету вызовом метода `getAttribute()`. Если дело обстоит не так, метод `getAttribute()` возвратит нуль.

Имеется возможность сохранения объекта между запросами. Помимо присоединения объекта к объекту запроса допускается и присоединение объекта к объекту сеанса или к контексту сервлета. Различие между этими двумя подходами состоит в том, что объекты, присоединенные к сеансу, не будут видимы для других пользователей, тогда как объекты, присоединенные к контексту сервлета, будут.

Присоединение объектов к сеансу и контексту сервлета очень походит на присоединение объектов к запросу. Чтобы присоединить объект к сеансу, нужно вызвать

метод `HttpServletRequest.getSession()`. Этот метод возвращает экземпляр `javax.servlet.http.HttpSession`. Затем мы вызываем метод `HttpSession.setAttribute()`, чтобы присоединить объект к сеансу. Следующий фрагмент кода поясняет этот процесс:

```
protected void doPost(HttpServletRequest request, HttpServletResponse
    response)
{
    .
    .
    .
    Foo foo = new Foo(); // гипотетический объект
    HttpSession session = request.getSession();
    session.setAttribute("foo", foo);
    .
    .
    .
}
```

Далее мы можем получить объект из сеанса, вызывая метод `HttpSession.getAttribute()`:

```
protected void doPost(HttpServletRequest request, HttpServletResponse
    response)
{
    HttpSession session = request.getSession();
    Foo foo = (Foo) session.getAttribute("foo");
}
```

Обратите внимание, что возвращаемое методом `session.getAttribute()` значение должно быть преобразовано к требуемому типу. Это необходимо, поскольку данный метод возвращает значение `java.lang.Object`.

Процедура присоединения объектов к контексту сервлета и получения их из него очень похожи. Сервлет должен вызвать метод `getServletContext()` (метод определен в классе `GenericServlet`, являющимся родительским классом для `HttpServlet`, который, в свою очередь, является родительским классом для наших сервлетов). Этот метод возвращает экземпляр `javax.servlet.ServletContext`, который определяет методы `setAttribute()` и `getAttribute()`. Данные методы работают так же, как и их аналоги: `HttpServletRequest` и `HttpServletResponse`.

Процедура присоединения объекта к контексту сервлета поясняется в следующем фрагменте кода:

```
protected void doPost(HttpServletRequest request, HttpServletResponse
    response)
{
    // Метод getServletContext() определен выше в иерархии наследования.
    ServletContext servletContext = getServletContext();
    Foo foo = new Foo();
    servletContext.setAttribute("foo", foo);
    .
    .
    .
}
```

Этот код присоединяет объект `foo` к контексту сервлета. Данный объект будет доступен любому сервлету в нашем приложении и будет сохраняться между сессиями. Он может быть получен путем вызова метода `ServletContext.getAttribute()`, как это иллюстрируется ниже:

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
{
    ServletContext ServletContext = getServletContext();
    Foo foo = (Foo) ServletContext.getAttribute("foo");
    .
    .
    .
}
```

Этот код получает объект `foo` из контекста запроса. Вновь приведенный здесь метод `ServletContext.getAttribute()`, как и его аналоги, возвращает экземпляр `java.lang.Object`.

 Объекты, присоединенные к контексту сервлета, также называют имеющими контекст приложения. Аналогичным образом объекты, присоединенные к сессии, также называют имеющими контекст сессии, а объекты, присоединенные к запросу, – имеющими контекст запроса.

Новые возможности, появившиеся в Сервлете 3.0

Java EE 6 включает новую версию API Сервлета – Сервлет 3.0. Эта версия API Сервлета включает несколько новых функций, которые упрощают разработку сервлета. Сервлет 3.0 также облегчает использование преимуществ современных технологий веб-приложений, таких как Ajax.

В следующих нескольких разделах мы обсудим некоторые из наиболее важных дополнений к API Сервлета.

Необязательный дескриптор развертывания web.xml

Сервлет 3.0 делает дескриптор развертывания приложения, файл `web.xml`, совершенно необязательным. Сервлеты могут быть сконфигурированы с помощью аннотаций вместо использования XML.

 Если веб-приложение конфигурируется и с помощью аннотаций и с помощью дескриптора развертывания `web.xml`, то настройки, указанные в файле `web.xml`, имеют приоритет.

Аннотация @WebServlet

Сервлеты могут быть декорированы аннотацией `@WebServlet` для указания их имени, шаблона URL, параметров инициализации и других элементов конфигурации, которые обычно указываются в файле дескриптора развертывания `web.xml`.

Как минимум у сервлета, который будет сконфигурирован через аннотации, должна быть аннотация `@WebServlet`, указывающая шаблон URL сервлета.

Используя аннотации нового Сервлета 3.0, можно переписать наш первый пример из этой главы следующим образом:

```
package net.ensode.glassfishbook.simpleapp;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(urlPatterns = {"/simpleservlet"})
public class SimpleServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response)
    {
        try
        {
            response.setContentType("text/html");
            PrintWriter printWriter = response.getWriter();
            printWriter.println("<h2>");
            printWriter.println("Если вы читаете это, ваш сервер приложений
                работает normally!");
            printWriter.println("</h2>");
        }
        catch (IOException ioException)
        {
            ioException.printStackTrace();
        }
    }
}
```

Обратите внимание, что нам нужно было всего лишь декорировать наш сервлет аннотацией `@WebServlet` и указать его шаблон URL в качестве значения атрибута `urlPatterns`.

Точно так же как в случае с `web.xml`, мы можем указать более одного шаблона URL для нашего сервлета. В этом случае достаточно отделить шаблоны URL друг от друга запятыми. Например, если мы хотим, чтобы наш сервлет обрабатывал все URL, заканчивающиеся суффиксом `.foo`, в дополнение ко всем URL, начинающимся с префикса `/simpleServlet`, понадобится аннотировать его следующим образом:

```
@WebServlet(urlPatterns = {"/simpleservlet", "*.foo"})
```

Благодаря этому простому дополнению к нашему коду мы будем избавлены от необходимости создавать и заполнять файл `web.xml` для нашего приложения.

После упаковки и развертывания этой новой версии приложения оно будет работать точно так же, как предыдущая версия.

Передача сервлету параметров инициализации через аннотации

Иногда полезно передать сервлету некоторые параметры инициализации. Таким образом мы сможем сообщить сервлету, что ему нужно вести себя по-разному в зависимости от параметров, которые ему отправлены. Например, нам может понадобиться сконфигурировать сервлет так, чтобы он действовал неодинаково в средах разработки и производственных средах.

Традиционно параметры инициализации сервлета отправлялись ему с помощью тега `<init-param>` в файле дескриптора развертывания `web.xml`. Начиная с Сервлета 3.0 параметры инициализации можно передавать сервлету в качестве значения атрибута `initParams` аннотации `@WebServlet`. Следующий пример поясняет, как это сделать:

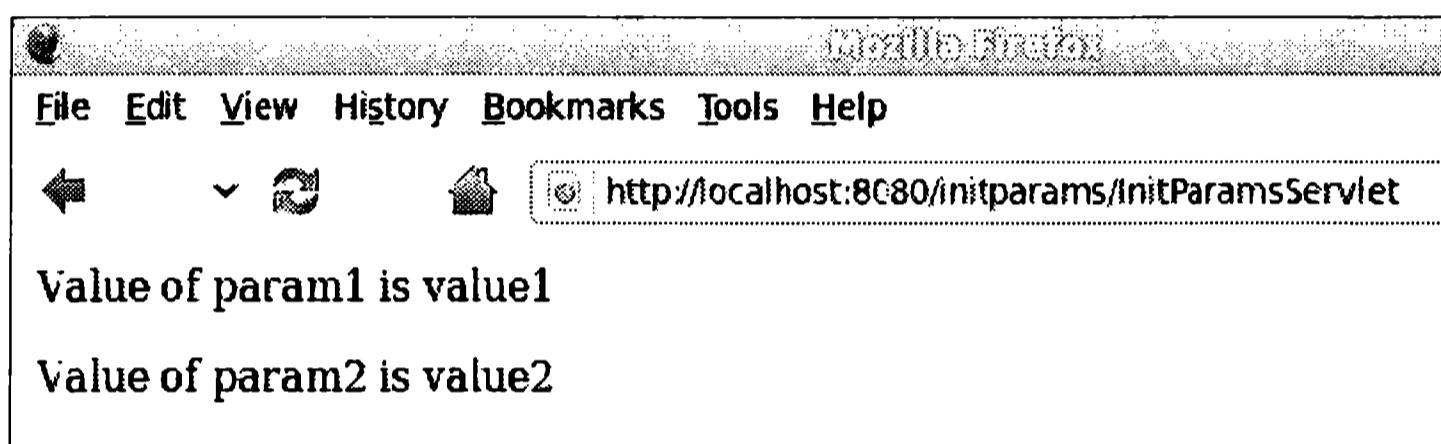
```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebInitParam;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet(name = "InitParamsServlet",
            urlPatterns = {"/InitParamsServlet"},
            initParams = {
                @WebInitParam(name = "param1", value = "value1"),
                @WebInitParam(name = "param2", value = "value2")})
public class InitParamsServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
                         response) throws ServletException, IOException
    {
        ServletConfig servletConfig = getServletConfig();
        String param1Val = servletConfig.getInitParameter("param1");
        String param2Val = servletConfig.getInitParameter("param2");
        response.setContentType("text/html");
        PrintWriter printWriter = response.getWriter();
        printWriter.println("<p>");
        printWriter.println("Значением param1 является " + param1Val);
        printWriter.println("</p>");
        printWriter.println("<p>");
        printWriter.println("Значением param2 является " + param2Val);
        printWriter.println("</p>");
    }
}
```

Как видно из приведенного кода, значение атрибута `initParams` аннотации `@WebServlet` является массивом аннотаций `@WebInitParam`. У каждой аннотации `@WebInitParam` имеются два атрибута: `name`, который соответствует названию параметра, и `value`, который соответствует значению параметра.

Мы можем получить значения наших параметров, вызывая метод `getInitParameter()` на классе `javax.servlet.ServletConfig`. Этот метод в качестве единственного параметра получает аргумент типа `String`, соответствующий названию параметра, и возвращает объект типа `String`, соответствующий значению параметра.

У каждого сервлета имеется присвоенный ему экземпляр `ServletConfig`. Как видно из приведенного примера, мы можем получить этот экземпляр, вызывая метод `getServletConfig()`, унаследованный от `javax.servlet.GenericServlet` – родительского класса для `HttpServlet`, расширением которого являются наши сервлеты.

После упаковки нашего сервлета в WAR-файл и развертывания его на сервере GlassFish с помощью инструмента командной строки `asadmin`, веб-консоли GlassFish либо путем копирования его в каталог `autodeploy` нашего домена мы увидим следующую страницу в окне обозревателя:



Как видно из снимка экрана, отображаемые в обозревателе значения соответствуют значениям, которые мы установили в каждой из аннотаций `@WebInitParam`.

Аннотация `@WebFilter`

Фильтры были введены в спецификацию сервлета начиная с версии 2.3. Фильтр является объектом, который может динамически перехватывать запрос и манипулировать его данными до того, как запрос будет обработан сервлетом. Фильтры также могут манипулировать откликом после того, как свою работу завершит метод сервлета `doGet()` или `doPost()`, но прежде, чем вывод будет отправлен обозревателю.

Единственный способ конфигурирования фильтра в более ранних спецификациях сервлета заключался в том, чтобы использовать тег `<filter-mapping>` в файле дескриптора развертывания `web.xml`. Сервлет 3.0 предоставил возможность конфигурировать сервлеты через аннотацию `@WebFilter`.

Следующий фрагмент кода поясняет, как это сделать:

```
package net.ensode.glassfishbook.simpleapp;

import java.io.IOException;
import java.util.Enumeration;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
```

```
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.annotation.WebInitParam;
@WebFilter(filterName = "SimpleFilter", initParams = {
@WebInitParam(name = "filterparam1", value = "filtervalue1")},
urlPatterns = {"/InitParamsServlet"})
public class SimpleFilter implements Filter
{
    private FilterConfig filterConfig;
    @Override
    public void init(FilterConfig filterConfig) throws ServletException
    {
        this.filterConfig = filterConfig;
    }
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
        servletResponse, FilterChain filterChain) throws IOException,
        ServletException
    {
        ServletContext servletContext = filterConfig.getServletContext();
        servletContext.log("Входим в doFilter()");
        servletContext.log("инициализируем параметры: ");
        Enumeration<String> initParameterNames = filterConfig.
            getInitParameterNames();
        String parameterName;
        String parameterValue;
        while (initParameterNames.hasMoreElements())
        {
            parameterName = initParameterNames.nextElement();
            parameterValue = filterConfig.getInitParameter(parameterName);
            servletContext.log(parameterName + " = " + parameterValue);
        }
        servletContext.log("Вызываем сервлет ...");
        filterChain.doFilter(servletRequest, servletResponse);
        servletContext.log("Возвращаемся из вызова сервлета");
    }
    @Override
    public void destroy()
    {
        filterConfig = null;
    }
}
```

Как видно из этого кода, у аннотации `@WebFilter` есть несколько атрибутов, которые мы можем использовать для конфигурирования фильтра. Атрибут `urlPatterns` имеет особое значение. Этот атрибут получает массив строковых (`String`) объектов в качестве его значения. Каждый элемент в массиве соответствует тому URL, который будет перехватывать наш фильтр. В нашем примере мы перехватываем единственный шаблон URL, который соответствует сервлету, написанному нами в предыдущем разделе.

Другие атрибуты в аннотации `@WebFilter` включают дополнительный атрибут `filterName`, который мы можем использовать для присвоения нашему фильтру имени. Если мы не указываем имя нашего фильтра, то значением имени фильтра по умолчанию будет имя класса фильтра.

Как видно из предыдущего примера кода, мы можем отправить параметры инициализации фильтру. Это делается точно так же, как и в случае, когда мы отправ-

ляем параметры инициализации сервлету. У аннотации @WebFilter есть атрибут initParams, который получает массив аннотаций @WebInitParam в качестве его значения. Мы можем получить значения указанных параметров, вызывая метод getInitParameter() на javax.servlet.FilterConfig, как продемонстрировано в примере выше.

Наш фильтр довольно прост – он всего лишь отправляет некоторый вывод журналу сервера до и после вызова сервлета. Проверка журнала сервера после развертывания нашего приложения и указания в адресной строке обозревателя URL сервлета должна показать примерно такой вывод из нашего фильтра:

```
[#|2009-09-30T19:38:15.454-0400|INFO|glassfish|javax.enterprise.system.container.web.com.sun.enterprise.web|_ThreadID=17;_ThreadName=Thread-1;|PWC1412: WebModule[/servlet30filter] ServletContext.log(): Входим вdoFilter()|#]

[#|2009-09-30T19:38:15.456-0400|INFO|glassfish|javax.enterprise.system.container.web.com.sun.enterprise.web|_ThreadID=17;_ThreadName=Thread-1;|PWC1412: WebModule[/servlet30filter] ServletContext.log(): инициализируем параметры: |#]

[#|2009-09-30T19:38:15.459-0400|INFO|glassfish|javax.enterprise.system.container.web.com.sun.enterprise.web|_ThreadID=17;_ThreadName=Thread-1;|PWC1412: WebModule[/servlet30filter] ServletContext.log(): filterparam1 = filtervalue1|#]

[#|2009-09-30T19:38:15.461-0400|INFO|glassfish|javax.enterprise.system.container.web.com.sun.enterprise.web|_ThreadID=17;_ThreadName=Thread-1;|PWC1412: WebModule[/servlet30filter] ServletContext.log(): Вызываем сервлет ...|#]

[#|2009-09-30T19:38:15.471-0400|INFO|glassfish|javax.enterprise.system.container.web.com.sun.enterprise.web|_ThreadID=17;_ThreadName=Thread-1;|PWC1412: WebModule[/servlet30filter] ServletContext.log(): Возвращаемся из вызова сервлета|#]
```

Конечно, существует масса других способов применения фильтров сервлета. Они могут использоваться для профилирования веб-приложения, для применения политики безопасности и для сжатия данных, а также во многих других случаях.

Аннотация @WebListener

За время жизни типичного веб-приложения происходит множество событий, таких как создание или уничтожение HTTP-запросов, добавление, удаление или модификация запросов или атрибутов сеанса и т. д.

API Сервлета предоставляет ряд интерфейсов слушателя (listener), которые мы можем реализовать для того, чтобы реагировать на подобные события. Все эти интерфейсы находятся в пакете javax.servlet. Они перечислены в следующей таблице:

Интерфейс слушателя	Описание
ServletContextListener	Содержит методы для обработки событий инициализации контекста и уничтожения
ServletContextAttributeListener	Содержит методы для реагирования на добавление, удаление или замену любых атрибутов в контексте сервлета (контекст приложения)
ServletRequestListener	Содержит методы для обработки событий инициализации и уничтожения запросов
ServletRequestAttributeListener	Содержит методы для реагирования на добавление, удаление или замену атрибутов в запросе
HttpSessionListener	Содержит методы для обработки событий инициализации HTTP-сеансов и уничтожения
HttpSessionAttributeListener	Содержит методы для реагирования на добавление, удаление или замену любых атрибутов в HTTP-сеансах

Для обработки любого из событий, которые обрабатываются интерфейсами, описанными в вышеприведенной таблице, мы должны реализовать один из этих интерфейсов и декорировать его аннотацией `@WebListener` или объявить его в файле дескриптора развертывания `web.xml` с помощью тега `<listener>`. Неудивительно, что в версии 3.0 спецификации сервлета была введена возможность использования аннотаций для регистрации слушателя.

API для всех этих интерфейсов довольно прост и интуитивно понятен. Мы покажем пример для одного из этих интерфейсов; для других они будут аналогичными.

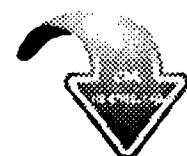


JavaDoc для всех предыдущих интерфейсов можно найти по адресу <http://docs.oracle.com/javaee/6/api/javax/servlet/http/package-summary.html>.

Следующий пример кода поясняет, как реализовать интерфейс `ServletRequestListener`, который может использоваться для выполнения некоторых действий всякий раз, когда создается или уничтожается HTTP-запрос:

```
package net.ensode.glassfishbook.listener;

import javax.servlet.ServletContext;
import javax.servlet.ServletRequestEvent;
import javax.servlet.ServletRequestListener;
import javax.servlet.annotation.WebListener;
@WebListener()
public class HttpRequestListener implements ServletRequestListener
{
    @Override
    public void requestInitialized(ServletRequestEvent servletRequestEvent)
    {
        ServletContext servletContext = servletRequestEvent.
            getServletContext();
```



```
    servletContext.log("Инициализирован новый запрос");
}
@Override
public void requestDestroyed(ServletRequestEvent servletRequestEvent)
{
    ServletContext servletContext = servletRequestEvent.
        getServletContext();
    servletContext.log("Запрос уничтожен");
}
}
```

Из приведенного примера кода мы видим, что для активации нашего класса слушателя мы должны декорировать его аннотацией `@WebListener`. Наш слушатель должен также реализовывать один из интерфейсов слушателя, которые мы перечисляли ранее. В нашем примере мы выбрали для реализации интерфейс `javax.servlet.ServletRequestListener`. У этого интерфейса есть методы, которые автоматически вызываются всякий раз при инициализации либо уничтожении HTTP-запроса.

У интерфейса `ServletRequestListener` есть два метода: `requestInitialized()` и `requestDestroyed()`. В нашей предыдущей простой реализации мы просто отправили некоторый вывод журналу, но, разумеется, можем сделать и нечто другое, что может нам понадобиться в наших реализациях.

Процесс развертывания нашего предыдущего слушателя наряду с простым сервлетом, который мы разработали ранее в этой главе, можно проследить по следующему выводу в журнале сервера GlassFish:

```
[#|2009-10-03T10:37:53.465-0400|INFO|glassfish|javax.enterprise.system.
container.web.com.sun.enterprise.web|_ThreadID=39;_ThreadName=Thread-
2;|PWC1412: WebModule[/nbServlet30listener] ServletContext.log():
Инициализирован новый запрос|#]

[#|2009-10-03T10:37:53.517-0400|INFO|glassfish|javax.enterprise.system.
container.web.com.sun.enterprise.web|_ThreadID=39;_ThreadName=Thread-
2;|PWC1412: WebModule[/nbServlet30listener] ServletContext.log():
Запрос уничтожен|#]
```

Реализация иных интерфейсов слушателя столь же проста и понятна.

Подключаемость

Еще в конце 1990-х годов – в те времена, когда появился изначальный API Сервлета, – написание сервлета было единственным способом написания серверных веб-приложений на Java. С тех пор поверх API Сервлета было создано несколько стандартных каркасов Java EE и каркасов сторонних производителей. Примерами таких стандартных каркасов являются JSP- и JSF-страницы; каркасов сторонних производителей – Struts, Wicket, Spring Web MVC и ряд других.

В настоящее время очень немногие (если таковые вообще имеются) веб-приложения Java создаются с использованием API Сервлета напрямую. Вместо этого в подавляющем большинстве проектов используется один из нескольких доступных каркасов веб-приложений Java. Все эти каркасы «под капотом» используют API Сервлета. Поэтому установка приложения для использования одного из этих каркасов всегда

включала создание некоторой конфигурации в файле дескриптора развертывания приложения web.xml. В некоторых случаях отдельные приложения используют более одного каркаса одновременно. Это приводит к тому, что файл дескриптора развертывания web.xml становится довольно большим и трудно поддерживаемым.

Сервлет 3.0 вводит концепцию подключаемости (pluggability). Разработчики каркасов веб-приложений теперь имеют возможность предоставить разработчикам приложений, использующим их каркас, не один, а два способа, с помощью которых удается избежать модификации файла дескриптора развертывания web.xml при использовании каркаса. Разработчики каркасов теперь могут выбрать использование аннотаций вместо файла web.xml для конфигурирования их сервлетов. В этом случае для использования каркаса необходимо включить JAR-файл (файлы) библиотеки (библиотек), предоставленный разработчиками каркаса, в WAR-файл приложения. Либо же разработчики каркаса могут выбрать включение файла web-fragment.xml в качестве части JAR-файла, который должен быть включен в веб-приложение для использования их каркаса.

Файл web-fragment.xml практически идентичен web.xml. Основное различие состоит в том, что корневым элементом файла web-fragment.xml является <web-fragment>, в противоположность элементу <web-app>, используемому для web.xml. Следующий код показывает пример файла web-fragment.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-fragment 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-fragment_3_0.xsd">
    <servlet>
        <servlet-name>WebFragment</servlet-name>
        <servlet-class>
            net.ensode.glassfishbook.webfragment.WebFragmentServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>WebFragment</servlet-name>
        <url-pattern>/WebFragment</url-pattern>
    </servlet-mapping>
</web-fragment>
```

Как видно из примера кода, файл web-fragment.xml практически идентичен типичному файлу web.xml. В этом простом примере мы используем только элементы <servlet> и <servlet-mapping>. Однако все другие типичные элементы файла web.xml, такие как <filter>, <filter-mapping> и <listener>, тоже доступны.

Как указано в нашем файле web-fragment.xml, наш сервлет может быть вызван через его шаблон URL – /WebFragment. Таким образом, URL для выполнения нашего сервлета после его развертывания в качестве части веб-приложения будет следующим: http://localhost:8080/webfragmentapp/WebFragment. Конечно, имя хоста, порт и корневой контекст должны быть скорректированы по мере необходимости.

Чтобы принять настройки `web-fragment.xml` для сервера GlassFish или любого другого Java EE 6-совместимого сервера приложений, нам нужно просто поместить файл в папку `META-INF` библиотеки, в которую мы упаковываем наш сервлет, фильтр и/или слушатель, а затем поместить JAR-файл созданной нами библиотеки в каталог `lib` WAR-файла, содержащего наше приложение.

Программное конфигурирование веб-приложений

В дополнение к возможности конфигурировать веб-приложения с помощью аннотаций и с помощью файла дескриптора развертывания `web-fragment.xml` API Сервлета 3.0 также позволяет нам программно конфигурировать наши веб-приложения во время их выполнения.

У класса `ServletContext` имеются новые методы для программного конфигурирования сервлетов, фильтров и слушателей. Следующий пример поясняет, как программно сконфигурировать сервлет во время выполнения, не обращаясь к аннотации `@WebServlet` или к XML:

```
package net.ensode.glassfishbook.servlet;

import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;
import javax.servlet.annotation.WebListener;
@WebListener()
public class ServletContextListenerImpl implements ServletContextListener
{
    @Override
    public void contextInitialized(ServletContextEvent servletContextEvent)
    {
        ServletContext servletContext = servletContextEvent.
            getServletContext();
        try
        {
            ProgrammaticallyConfiguredServlet servlet =
                servletContext.createServlet(
                    ProgrammaticallyConfiguredServlet.class);
            servletContext.addServlet("ProgrammaticallyConfiguredServlet",
                servlet);
            ServletRegistration servletRegistration =
                servletContext.getServletRegistration(
                    "ProgrammaticallyConfiguredServlet");
            servletRegistration.addMapping(
                "/ProgrammaticallyConfiguredServlet");
        }
        catch (ServletException servletException)
        {
            servletContext.log(servletException.getMessage());
        }
    }
    @Override
    public void contextDestroyed(ServletContextEvent servletContextEvent)
    {
    }
}
```

В этом примере мы вызываем метод `createServlet()` контекста `ServletContext` для создания сервлета, который мы собираемся сконфигурировать. Этот метод получает экземпляр класса `java.lang.Class`, соответствующий классу нашего сервлета. Метод возвращает класс, реализующий интерфейс `javax.servlet.Servlet` или любой из его дочерних интерфейсов (благодаря возможности обобщения Generics – языка Java, появившейся начиная с Java 5, нам больше не нужно явно приводить тип возвращаемого значения к фактическому типу нашего сервлета).

После того как мы создадим наш сервлет, мы должны вызвать метод `addServlet()` на нашем экземпляре `ServletContext`, чтобы зарегистрировать наш сервлет в контейнере сервлета. Этот метод получает два параметра: первый является строкой, соответствующей имени сервлета, второй – экземпляром сервлета, возвращаемого вызовом метода `createServlet()`.

После того как мы зарегистрируем наш сервлет, мы должны добавить шаблон URL, отображающийся на него. Чтобы это сделать, нам нужно вызвать метод `getServletRegistration()` на нашем экземпляре `ServletContext`, передавая имя сервлета в качестве параметра. Данный метод возвращает реализацию контейнера сервлета `javax.servlet.ServletRegistration`. Из этого объекта мы должны вызвать его метод `addMapping()`, передавая отображение URL, которое мы хотим установить, для обработки нашим сервлетом.

Пример нашего сервлета очень прост. Он просто выводит на экран текстовое сообщение в окне обозревателя.

```
package net.ensode.glassfishbook.servlet;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class ProgrammaticallyConfiguredServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException
    {
        ServletOutputStream outputStream = response.getOutputStream();
        outputStream.println("Это сообщение было генерировано из сервлета,
            который был сконфигурирован программным путем.");
    }
}
```

После упаковки нашего кода в WAR-файл, развертывания его на сервере GlassFish и указания в адресной строке обозревателя соответствующего URL (`http://localhost:8080/programmaticServletwebapp/ProgrammaticallyConfiguredServlet` – предполагая, что мы упаковали приложение в WAR-файле `programmaticServletwebapp.war` и не переопределяли корневой контекст по умолчанию), мы должны увидеть следующее сообщение в окне обозревателя:

Это сообщение было сгенерировано из сервлета, который был сконфигурирован программным путем.

У интерфейса `ServletContext` имеются методы создания и добавления фильтров сервлета и слушателей. Работа этих методов очень напоминает работу методов `addServlet()` и `createServlet()`, поэтому мы не будем обсуждать их подробно. Обратитесь к документации по API Java EE 6 – см. <http://docs.oracle.com/javaee/6/api/> – для получения подробной информации.

Асинхронная обработка

Традиционно сервлеты создавали один поток на запрос в веб-приложениях Java. После того как запрос обработан, поток становится доступным для использования другими запросами. Эта модель довольно хорошо работает на традиционных веб-приложениях, в которых HTTP-запросы относительно немногочисленны и по времени выполнения далеко отстоят друг от друга. Однако самые современные веб-приложения используют преимущества технологии Ajax (Асинхронный JavaScript и XML – Asynchronous JavaScript and XML), благодаря чему они являются более отзывчивыми по сравнению с традиционными веб-приложениями. У Ajax имеется побочный эффект, заключающийся в генерировании намного большего количества HTTP-запросов, чем в традиционных веб-приложениях. Если некоторые из этих потоков в течение долгого времени блокируются ожиданием готовности ресурса либо делают нечто требующее много времени на обработку, наше приложение может начать страдать от недостатка потоковых ресурсов.

Чтобы облегчить положение дел, спецификация Сервлета 3.0 вводит асинхронную обработку. Используя эту новую возможность, мы больше не ограничены условием «один поток на один запрос». Теперь мы можем порождать отдельный поток и возвращать исходный поток пулу, который будет повторно использоваться другими клиентами.

Следующий пример поясняет, как реализовать асинхронную обработку с использованием новых возможностей, введенных в спецификацию Сервлета 3.0:

```
package net.ensode.glassfishbook.asynchronousservlet;

import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.servlet.AsyncContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet(name = "AsynchronousServlet", urlPatterns = {
    "/AsynchronousServlet"}, asyncSupported = true)
public class AsynchronousServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException
```

```
{  
    final Logger logger = Logger.getLogger(AsynchronousServlet.class.  
        getName());  
    logger.log(Level.INFO, "--- Данные от doGet()");  
    final AsyncContext ac = request.startAsync();  
    logger.log(Level.INFO, "---- Вызов ac.start()");  
    ac.start(new Runnable()  
    {  
        @Override  
        public void run()  
        {  
            logger.log(Level.INFO, "внутри потока");  
            try  
            {  
                // Имитация длительного процесса.  
                Thread.sleep(10000);  
            }  
            catch (InterruptedException ex)  
            {  
                Logger.getLogger(AsynchronousServlet.class.getName()).  
                    log(Level.SEVERE, null, ex);  
            }  
            try  
            {  
                ac.getResponse().getWriter().println("Вы должны увидеть  
                    это после короткого ожидания");  
                ac.complete();  
            }  
            catch (IOException ex)  
            {  
                Logger.getLogger(AsynchronousServlet.class.getName()).  
                    log(Level.SEVERE, null, ex);  
            }  
        }  
    });  
    logger.log(Level.INFO, "Покидаем doGet()");  
}  
}
```

Чтобы убедиться в том, что код асинхронной обработки работает как ожидалось, нам нужно установить атрибут `asyncSupported` аннотации `@WebServlet` в значение `true`.

Для того чтобы фактически породить асинхронный процесс, мы должны вызвать метод `startAsync()` на экземпляре `HttpServletRequest`, который мы получаем в качестве параметра метода `doGet()` или `doPost()` в нашем сервлете. Данный метод возвращает экземпляр класса `javax.servlet.AsyncContext`. У этого класса есть метод `start()`, который получает экземпляр класса, реализующего интерфейс `java.lang.Runnable` в качестве единственного параметра. В нашем примере мы использовали анонимный внутренний класс для соответствующей реализации интерфейса `Runnable`. Конечно, стандартный класс Java, реализующий интерфейс `Runnable`, также может использоваться.

Когда мы вызываем метод `start()` класса `AsyncContext`, порождается новый поток и выполняется метод `run()`, `Runnable` экземпляра. Этот поток выполняется в фоновом режиме, метод `doGet()` возвращается сразу и поток запроса тут же становится доступен для обслуживания других клиентов. Важно отметить, что даже при том, что метод `doGet()` возвращается сразу, отклика (ответа) не происходит, пока порожденный поток не будет завершен. Порожденный поток сможет сообщить о завершении обработки, вызвав метод `complete()` на `AsyncContext`.

В предыдущем примере мы отправили некоторые записи в файл журнала сервера GlassFish, чтобы лучше пояснить, что происходит. Просматривая журнал сервера GlassFish сразу по завершении выполнения нашего сервлета, мы должны заметить, что все записи в журнале разделяет временной интервал в пределах долей секунды. Сообщение «Вы должны увидеть это после короткого ожидания» не показывается в окне обозревателя до тех пор, пока запись в журнале не засвидетельствует, что мы покинули метод `doGet()`.

Резюме

В этой главе было показано, как разрабатывать, конфигурировать, упаковывать и развертывать сервлеты.

Мы узнали, как обработать информацию из HTML-формы, получая доступ к объекту HTTP-запроса.

Кроме того, было рассказано о переадресации HTTP-запросов от одного сервлета к другому и перенаправлении HTTP-отклика на другой сервер.

Также мы обсудили, как сохранить объекты в памяти между запросами, присоединяя их к контексту (окружению) сервлета и HTTP-сессии.

Наконец, были рассмотрены все основные новые возможности спецификации Сервлета 3.0, включая конфигурирование веб-приложений с помощью аннотаций, подключаемость с помощью файла `web-fragment.xml`, программное конфигурирование сервлета и асинхронную обработку.

3

JavaServer Pages

В предыдущей главе мы рассмотрели, как разработать сервлеты Java. Сервлеты хорошо подходят для обработки форм ввода, но код сервлета, который выводит HTML-разметку в окно обозревателя, обычно весьма громоздок для написания, чтения и отладки. Лучшим способом является отправка вывода для обозревателя через *Серверные страницы Java (JavaServer Pages (JSP))*.

В этой главе мы затронем следующие темы:

- разработка нашей первой JSP-страницы;
- неявные объекты JSP;
- JSP и JavaBeans;
- повторное использование JSP-контента;
- создание пользовательских тегов.

Введение в JavaServer Pages

Первоначально технология сервлетов была единственным API, доступным в Java для разработки серверных веб-приложений. У сервлетов было много преимуществ по сравнению со сценариями CGI, которые в те времена были (и до некоторой степени все еще остаются) доминирующей технологией. К таким преимуществам относились, например, увеличенная производительность и улучшенная безопасность.

Тем не менее у сервлетов имелся один существенный недостаток. Поскольку код HTML, который будет представлен в окне обозревателя, должен был встраиваться в код Java, большую часть кода сервлета оказывалось очень трудно поддерживать. Для устранения этого недостатка была создана технология Серверных страниц Java (JSP), использующая сочетание статического HTML-контента и динамического контента для генерации веб-страниц. Поскольку статический контент отделяется от динамического, JSP намного легче поддерживать, чем сервлеты, генерирующие HTML-вывод.

В самых современных приложениях, использующих JSP, все еще используются сервлеты, однако они обычно играют роль контроллеров в рамках шаблона проектирования *Модель-Представление-Контроллер* (Model-View-Controller (MVC)), а JSP-страницы в рамках этого шаблона играют роль представлений. Поскольку у сервлетов-контроллеров нет никакого пользовательского интерфейса, мы не сталкиваемся с проблемой наличия HTML-разметки в коде Java.

В этой главе мы расскажем, как разработать серверное веб-приложение, используя технологию JavaServer Pages.

Разработка нашей первой JSP-страницы

JSP – это страница, содержащая и статическую HTML-разметку, и динамический контент. Динамический контент может быть сгенерирован путем использования фрагментов кода Java, называемых *скриптлетами* (scriptlets), или путем использования стандартных или пользовательских JSP-тегов. Давайте рассмотрим очень простой JSP-код, который выводит на экран текущее время сервера в окне обозревателя:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
           pageEncoding="UTF-8" %>
<%@ page import="java.util.Date" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
           "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Дата и время сервера</title>
    </head>
    <body>
        <p>Дата и время сервера: <% out.print(new Date()); %></p>
    </body>
</html>
```

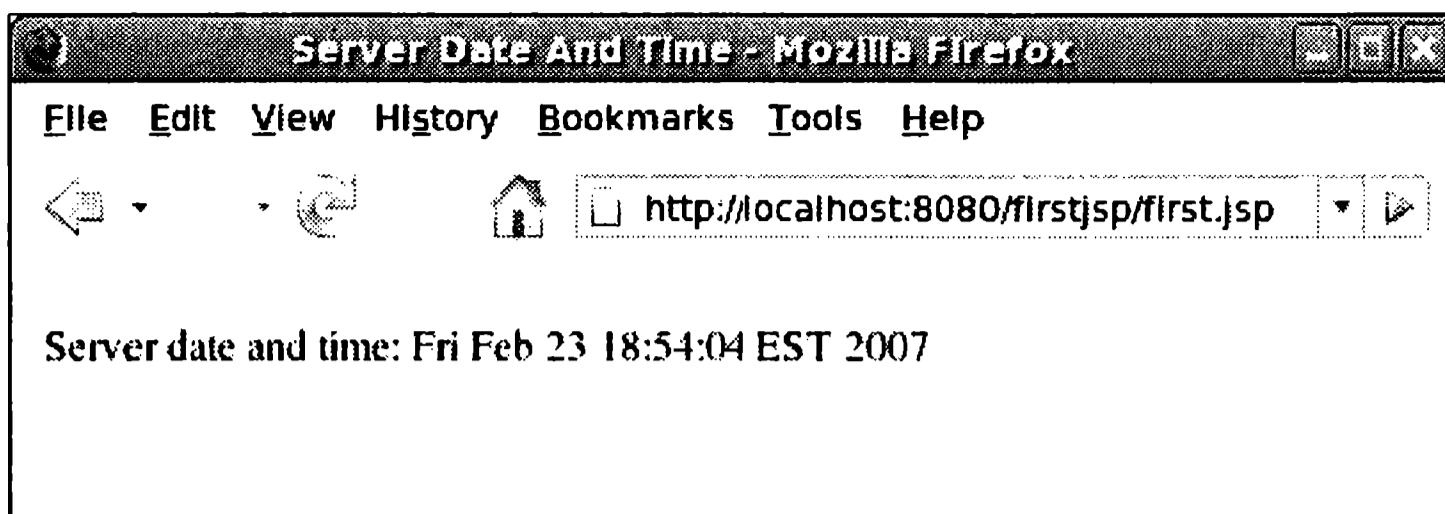
Чтобы развернуть это JSP-приложение, нам достаточно поместить его в WAR-файл, а как мы уже упоминали ранее, самый простой способ развертывания WAR-файла на сервере заключается в его копировании в каталог: [Каталог установки Glassfish]/glassfish/domains/domain1/autodeploy.



Быстрое развертывание простого приложения JSP

Простые JSP-приложения могут быть быстро развернуты без необходимости их упаковки в WAR-файлы путем простого копирования их в каталог: [Каталог установки glassfish]/glassfish/domains/domain1/docroot/ и последующего просмотра в обозревателе при указании на них в адресной строке обозревателя: <http://localhost:8080/jspname.jsp> (где jspname.jsp – имя развернутого JSP-приложения).

После успешного развертывания приложения, указав в адресной строке обозревателя адрес <http://localhost:8080/firstjsp/first.jsp>, мы должны будем увидеть примерно такой результат:



Строка **Дата и время сервера** (*Server date and time*) взята из статического текста сразу после тега `<p>` в JSP-странице. Фактические дата и время, выведенные на экран, являются датой и временем сервера. Значение взято из вывода кода между разделителями `<%` и `%>`. Мы можем поместить между ними любой допустимый код Java. Код в этих разделителях называется *скриптлетом* (*scriptlet*). Скриптлет в показанном выше приложении JSP использует неявный объект. Неявные объекты JSP – это такие объекты, которые могут с готовностью использоваться в любой JSP-странице без необходимости их объявления или инициализации. Неявный объект `out` является экземпляром класса `javax.servlet.jsp.JspWriter`. Он может считаться эквивалентом вызова метода `HttpServletResponse.getWriter()`.

Первые две строки в показанном выше коде являются *директивами JSP-страницы* (*JSP page directives*). Директивы JSP-страницы определяют атрибуты, которые применяются ко всей JSP-странице. Атрибутов может быть несколько. В вышеприведенном примере первая директива страницы устанавливает следующие атрибуты: язык (*language*), тип содержимого (*contentType*), набор символов (*charset*) и кодировка страницы (*PageEncoding*). Вторая директива добавляет оператор *import* для страницы.

Как видно из примера, атрибуты директивы JSP-страницы могут быть объединены в одну директиву либо для каждого атрибута может использоваться отдельная директива страницы.

В следующей таблице перечислены все атрибуты, которые могут использоваться в директиве страницы:

Атрибут	Описание	Допустимые значения	Значение по умолчанию
<code>autoFlush</code>	Определяет, нужно ли автоматически очищать буфер, когда он заполнен	true или false	true
<code>buffer</code>	Размер буфера вывода в килобайтах	Nkb, где N – целое число. "none" также является допустимым значением	8kb

Атрибут	Описание	Допустимые значения	Значение по умолчанию
contentType	Определяет MIME-тип HTTP-страницы отклика, а также ее кодировку	Любая комбинация из допустимых MIME-типа и кодировки	text/html; charset=ISO-8859-1
deferredSyntaxAllowsStringLiteral	В более ранних версиях спецификации JSP синтаксическая конструкция выражений языка #{} не была зарезервирована. Для целей обратной совместимости этот атрибут устанавливает выражения с использованием синтаксиса строковых литералов	true или false	false
errorCode	Указывает, на какую страницу нужно перейти, когда JSP генерирует исключение	Любые допустимые относительные URL другой JSP	Не задано
extends	Указывает на класс, который расширяет эта JSP	Полностью определенное (квалифицированное) имя родительского класса JSP	Не задано
import	Импортирует один и более классов для использования их в скриптлете	Полностью определенное (квалифицированное) имя классов или имя пакета + .* для импорта всех необходимых классов из пакета (например: <%@ page import Java.util.* %>)	Не задано
info	Значение этого атрибута включается в скомпилированную JSP. Позже оно может быть извлечено вызовом метода getServletInfo() страницы	Любая строка	Не задано

Атрибут	Описание	Допустимые значения	Значение по умолчанию
isELIgnored	Установка этого атрибута в значение <code>true</code> предотвращает интерпретацию выражения языка интерпретатором выражений	<code>true</code> или <code>false</code>	<code>false</code>
isErrorPage	Определяет, является ли страница страницей ошибки	<code>true</code> или <code>false</code>	<code>false</code>
isThreadSafe	Определяет, является ли страница потокобезопасной	<code>true</code> или <code>false</code>	<code>true</code>
language	Определяет язык сценариев, используемый в скриптлетах, декларациях и выражениях на странице JSP	Любой язык сценариев, который может выполняться в виртуальной машине Java (Groovy, JRuby и т.д.)	<code>java</code>
pageEncoding	Определяет кодировку страницы, например: «UTF-8»	Любая допустимая кодировка страницы.	Не задано
session	Определяет, имеет ли страница доступ к HTTP-сессии	<code>true</code> или <code>false</code>	<code>true</code>
trimDirectiveWhitespaces	Когда JSP-страница отображается в виде HTML в обозревателе, при генерации разметки в ней часто возникает много пустых строк. Установка этого атрибута в значение <code>true</code> предотвращает генерацию этих посторонних пустых строк в разметке	<code>true</code> или <code>false</code>	<code>false</code>

Из всех атрибутов, перечисленных в таблице, обычно используются `errorPage`, `import` и `isErrorPage`. Другие атрибуты имеют разумные значения по умолчанию, поэтому, как правило, явно не задаются.

При развертывании на сервере приложения, JSP транслируются (компилируются) в сервлеты. Атрибут `extends` директивы страницы указывает родительский класс сгенерированного сервлета. Значение этого атрибута должно быть подклассом `javax.servlet.GenericServlet`.

Хотя атрибут `language` может принять любой язык, который может быть выполнен виртуальной машиной Java, крайне редко используется какой-либо другой язык, отличный от Java.

Неявные объекты JSP

Неявные объекты JSP – это объекты, которые могут использоваться в JSP без необходимости их объявления или инициализации. Они фактически объявляются и инициализируются сервером приложений «за кулисами», во время развертывания JSP.

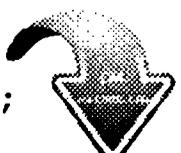
В примере предыдущего раздела мы использовали неявный объект JSP – `out`. Этот объект для всех практических целей эквивалентен вызову метода сервлета `HttpServletResponse.getWriter()`. В дополнение к объекту `out` имеются несколько других неявных объектов, которые могут использоваться в JSP-скриптлетах. Эти неявные объекты перечислены в следующей таблице:

Неявный объект	Класс неявного объекта	Описание
<code>application</code>	<code>javax.servlet.ServletContext</code>	Эквивалент вызова метода <code>getServletContext()</code> сервлета
<code>config</code>	<code>javax.servlet.ServletConfig</code>	Эквивалент вызова метода <code>getServletConfig()</code> сервлета
<code>exception</code>	<code>java.lang.Throwable</code>	Доступен только в случае, если атрибут <code>isErrorPage</code> директивы страницы установлен в значение <code>true</code> . Предоставляет доступ к тем исключениям, по причине возникновения которых был сделан вызов данной страницы
<code>out</code>	<code>javax.servlet.jsp.JspWriter</code>	Эквивалент значения, возвращаемого методом <code>HttpServletResponse.getWriter()</code>
<code>page</code>	<code>java.lang.Object</code>	Предоставляет доступ к странице, генерирующей сервлет
<code>pageContext</code>	<code>javax.servlet.jsp.PageContext</code>	Предоставляет несколько методов управления различными контекстами веб-приложения (запрос, сеанс, приложение). Обратитесь к JavaDoc для <code>PageContext</code> по адресу: http://docs.oracle.com/javaee/5/api/javax/servlet/jsp/PageContext.html .

Неявный объект	Класс неявного объекта	Описание
request	javax.servlet.ServletRequest	Эквивалент экземпляра HttpServletRequest, который мы получаем в качестве параметра для методов doGet() и doPost() в сервлете
response	javax.servlet.ServletResponse	Эквивалент экземпляра HttpServletResponse, который мы получаем в качестве параметра для методов doGet() и doPost() в сервлете
session	javax.servlet.http.HttpSession	Эквивалент значения, возвращаемого методом HttpServletRequest.getSession()

Следующий пример поясняет использование некоторых неявных объектов JSP:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
           pageEncoding="UTF-8" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
           "http://www.w3.org/TR/html4/loose.dtd">
<%@ page import="java.util.Enumeration" %>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Демонстрация неявных объектов</title>
    </head>
    <body>
        <p>Эта JSP-страница использует неявные объекты для присоединения
           объектов к контекстам запроса (request), сеанса
           (session) и приложения (application).<br/>
           Она также получает некоторые параметры инициализации,
           передаваемые через конфигурационный файл web.xml.<br/>
           И наконец, она получает размер буфера из неявного
           объекта.<br/>
        </p>
        <p>
            <%
                application.setAttribute("applicationAttribute", new String(
                    "Эта строка доступна для всех сеансов."));
                session.setAttribute("sessionAttribute", new String(
                    "Эта строка доступна для всех запросов."));
                request.setAttribute("requestAttribute", new String(
                    "Эта строка доступна в одном запросе."));
            <%
            Enumeration initParameterNames = config.getInitParameterNames();
            out.print("Инициализация параметров, полученных ");
            out.print("из неявных <br/>");
            out.println("объектов конфигурации:<br/><br/>");
            while (initParameterNames.hasMoreElements())
            {
                String parameterName = (String) initParameterNames.
                   .nextElement();
                out.print(parameterName + " = ");
                out.print(config.getInitParameter((String) parameterName));
                out.print("<br/>");
            }
        
```



```

        out.println("<br/>");
        out.println("Неявный объект <b>page</b> имеет тип "
            + page.getClass().getName() + "<br/><br/>");
        out.println("Размер буфера: " + response.getBufferSize()
            + " байт");
    %>
</p>
<p>
    <a href="implicitobjects2.jsp">Щелкните здесь для продолжения.</a>
</p>
</body>
</html>

```

Эта JSP-страница использует большинство неявных объектов, доступных для JSP-скриптов. Первое, что она делает, – присоединяет к объектам application, session и request неявные объекты. Затем она получает все параметры инициализации из неявного объекта config и выводит на экран в обозревателе их имена и значения, используя неявный объект out. Далее она выводит на экран полное (калифицированное) имя неявного объекта page и, наконец, размер буфера, получая доступ к неявному объекту response.

Параметры инициализации JSP (и, возможно, сервлета) объявляются в файле дескриптора развертывания приложения web.xml. Для данного приложения файл web.xml выглядит следующим образом:

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
           http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <servlet>
        <servlet-name>ImplicitObjectsJsp</servlet-name>
        <jsp-file>/implicitobjects.jsp</jsp-file>
        <init-param>
            <param-name>webxmlparam</param-name>
            <param-value>
                Это устанавливается в файле web.xml
            </param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>ImplicitObjectsJsp</servlet-name>
        <url-pattern>/implicitobjects.jsp</url-pattern>
    </servlet-mapping>
</web-app>

```

Помните, что JSP-страница компилируется в сервlet во время ее первого выполнения при обращении к ней после развертывания. Таким образом, с точки зрения файла web.xml она может рассматриваться как сервlet. Чтобы иметь возможность передать параметры инициализации JSP-странице, мы должны обработать ее как сервlet, поскольку параметры инициализации помещаются между XML-тегами <init-param> и </init-param>. Как показано в предыдущем файле web.xml, наименование параметра помещается между тегами <param-name> и </param-name>, а значение параметра – между тегами <param-value> и </param-value>.

У сервлета (и у JSP) может быть несколько параметров инициализации. Каждый параметр инициализации должен быть объявлен в отдельном теге `<init-param>`.

Обратите внимание, что в предыдущем файле `web.xml` мы объявили отображение сервлета для нашей JSP. Это было необходимо для того, чтобы позволить веб-контейнеру GlassFish передавать параметры инициализации JSP-странице. Поскольку мы не хотим, чтобы URL JSP изменился, мы использовали фактический URL JSP в качестве значения тега `<url-pattern>`. Если мы захотим получить доступ к JSP через другой URL (не обязательно оканчивающийся на `.jsp`), мы можем поместить требуемый URL в тег `<url-pattern>`.

В самом низу файла `implicitobjects.jsp` присутствует гиперссылка на вторую JSP, названную `implicitobjects2.jsp`. Разметка и код для `implicitobjects2.jsp` выглядят следующим образом:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
           pageEncoding="UTF-8" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
           "http://www.w3.org/TR/html4/loose.dtd">
<%@ page import="java.util.Enumeration" %>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Проверка на возможность извлечения</title>
    </head>
    <body>
        <p>Эта страница позволяет убедиться, что мы можем получить
           атрибуты приложения, сеанса и запроса, установленные
           на предыдущей странице.<br/>
        </p>
        <p>значение атрибутов приложения (applicationAttribute) :
            <%= application.getAttribute("applicationAttribute") %>
            <br/>
            значение атрибутов сеанса (sessionAttribute) :
            <%= session.getAttribute("sessionAttribute") %>
            <br/>
            значение атрибутов запроса (requestAttribute) :
            <%= request.getAttribute("requestAttribute") %>
            <br />
        </p>
        <p>В контексте приложения были обнаружены следующие атрибуты:
            <br/><br/>
            <%
                Enumeration applicationAttributeNames = pageContext
                    .getAttributeNamesInScope(pageContext.APPLICATION_SCOPE);
                while (applicationAttributeNames.hasMoreElements())
                {
                    out.println(applicationAttributeNames.nextElement() + "<br/>");
                }
            %
        </p>
        <p><a href="buggy.jsp">Эта гиперссылка указывает на JSP, к которой
           будет выполняться переход при возникновении исключения.
            </a>
        </p>
    </body>
</html>
```

На этой второй JSP-странице мы получаем объекты, которые были присоединены к следующим объектам: приложение (application), сеанс (session) и запрос (request). Присоединенные объекты могут быть получены с помощью вызова метода соответствующего неявного объекта `getAttribute()`. Обратите внимание, что все вызовы метода `getAttribute()` помещаются между разделителями `<%=` и `%>`. Отрывки кода между этими разделителями называются *выражениями JSP* (JSP expressions). Выражения JSP вычисляются, и возвращаемое ими значение выводится на экран в обозревателе без необходимости вызова метода `out.print()`.

Данная JSP-страница также получает имена всех объектов, присоединенных к контексту приложения, и выводит их на экран в окне обозревателя.

Внизу кода предыдущей JSP-страницы присутствует гиперссылка на третью JSP-страницу. Эта третья JSP-страница вызывает `buggy.jsp`. Ее цель – демонстрация атрибута `errorPage` директивы `page`, атрибута `error` директивы `page` и `exception` неявного объекта. Поэтому она не так уж сложна.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
       pageEncoding="UTF-8" errorPage="error.jsp" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
         "http://www.w3.org/TR/html4/loose.dtd">

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP-страница с ошибкой</title>
  </head>
  <body>
    <p>
      Этот текст никогда не будет видно в обозревателе, так как
      исключение возникает прежде, чем страница будет отображена.
    <%
      Object o = null;
      // здесь возникает исключение NullPointerException
      out.println(o.toString());
    %>
    </p>
  </body>
</html>
```

Эта JSP-страница всего лишь заставляет возникать исключение `NullPointerException`, что приведет в контейнере сервлета GlassFish к направлению пользователя на страницу, объявленную в качестве страницы ошибки в атрибуте `errorPage` директивы `page`. Последняя страница называется `error.jsp`; ее разметка и код показаны ниже:

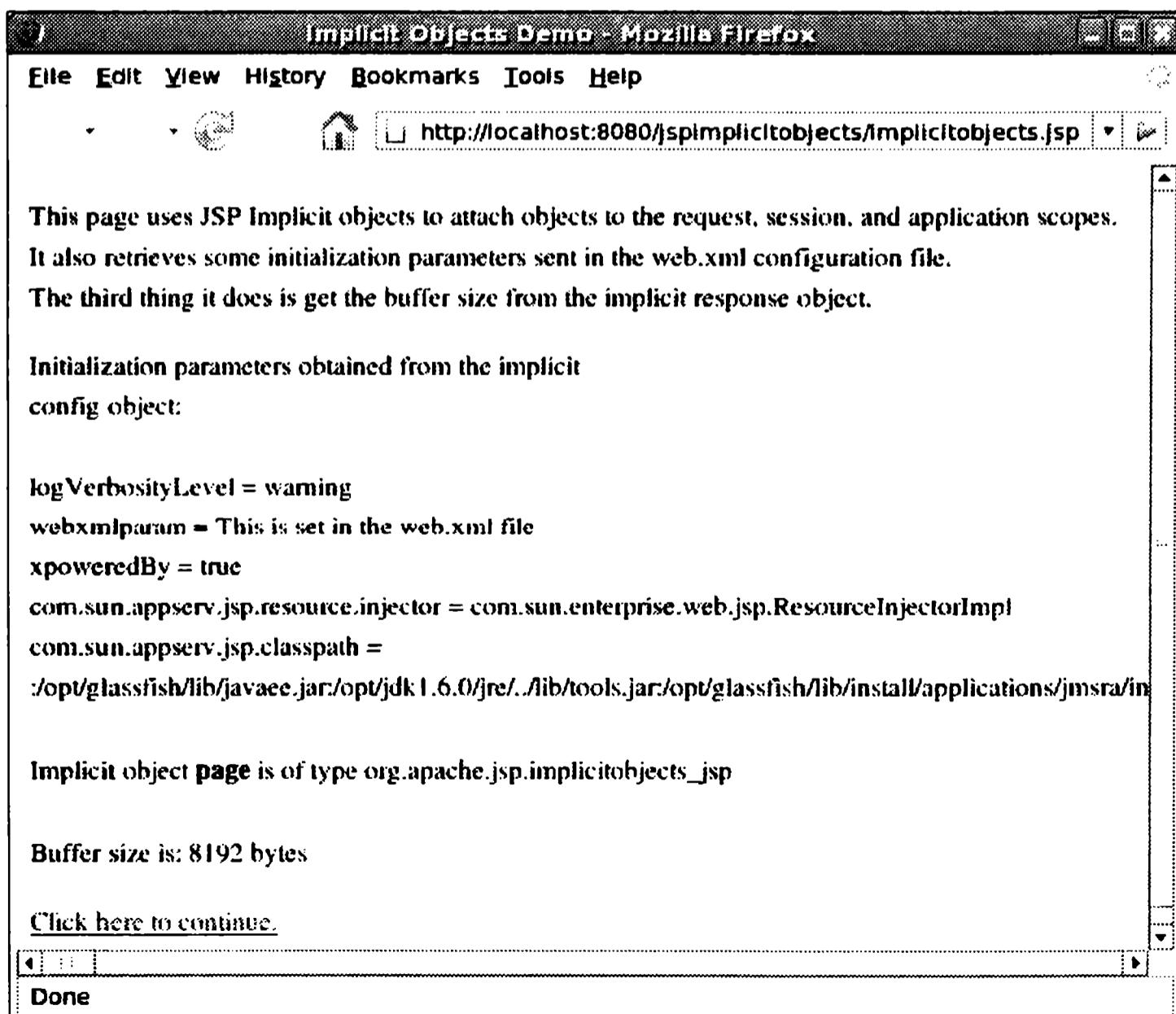
```
<%@ page language="java" contentType="text/html; charset=UTF-8"
       pageEncoding="UTF-8" isErrorPage="true" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
         "http://www.w3.org/TR/html4/loose.dtd">
<%@ page import="java.io.StringWriter" %>
<%@ page import="java.io.PrintWriter" %>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>В приложении возникла ошибка</title>
  </head>
  <body>
    <h2>Перехвачено исключение</h2>
```

```
<p>Трассировочная информация из стека вызовов:<br/>
<%
    StringWriter stringWriter = new StringWriter();
    PrintWriter printWriter = new PrintWriter(stringWriter);
    exception.printStackTrace(printWriter);
    out.write(stringWriter.toString());
%
</p>
</body>
</html>
```

Обратите внимание, что эта страница объявляет саму себя страницей ошибки, устанавливая атрибут `isErrorPage` директивы `page` в значение `true`. Как страница ошибки она имеет доступ к `exception` неявного объекта. Эта страница просто вызывает метод `printStackTrace()` директивы `exception` неявного объекта и отправляет его вывод обозревателю через неявный объект. В реальном приложении, вероятнее всего, выводилось бы на экран понятное для пользователя сообщение об ошибке.

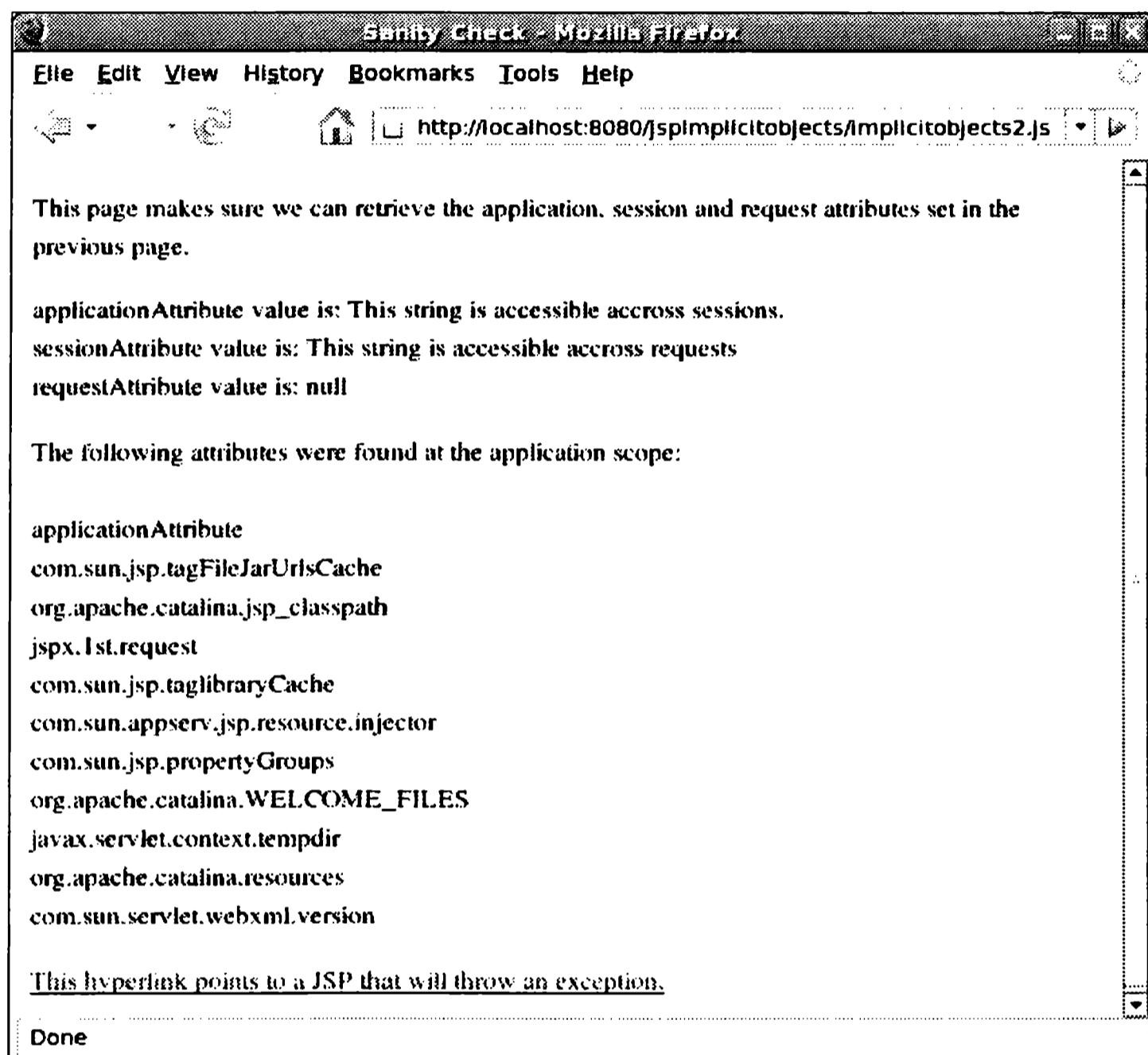
Поскольку предыдущее приложение состоит только из трех JSP-страниц, его упаковка для развертывания состоит из простого помещения всех JSP-страниц в корне WAR-файла, а также помещения файла `web.xml` в его обычное место (в каталоге `WEB-INF` WAR-файла).

После развертывания приложения и ввода в адресной строке обозревателя адреса `http://localhost:8080/jspimplicitobjects/implicitobjects.jsp` мы должны будем увидеть страницу `implicitobjects.jsp` отображенной в обозревателе:



Как видно из снимка экрана, у JSP имеется много «тайных» параметров инициализации в дополнение к тому, который мы устанавливаем в файле дескриптора развертывания приложения web.xml. Эти дополнительные параметры инициализации устанавливаются автоматически веб-контейнером сервера GlassFish.

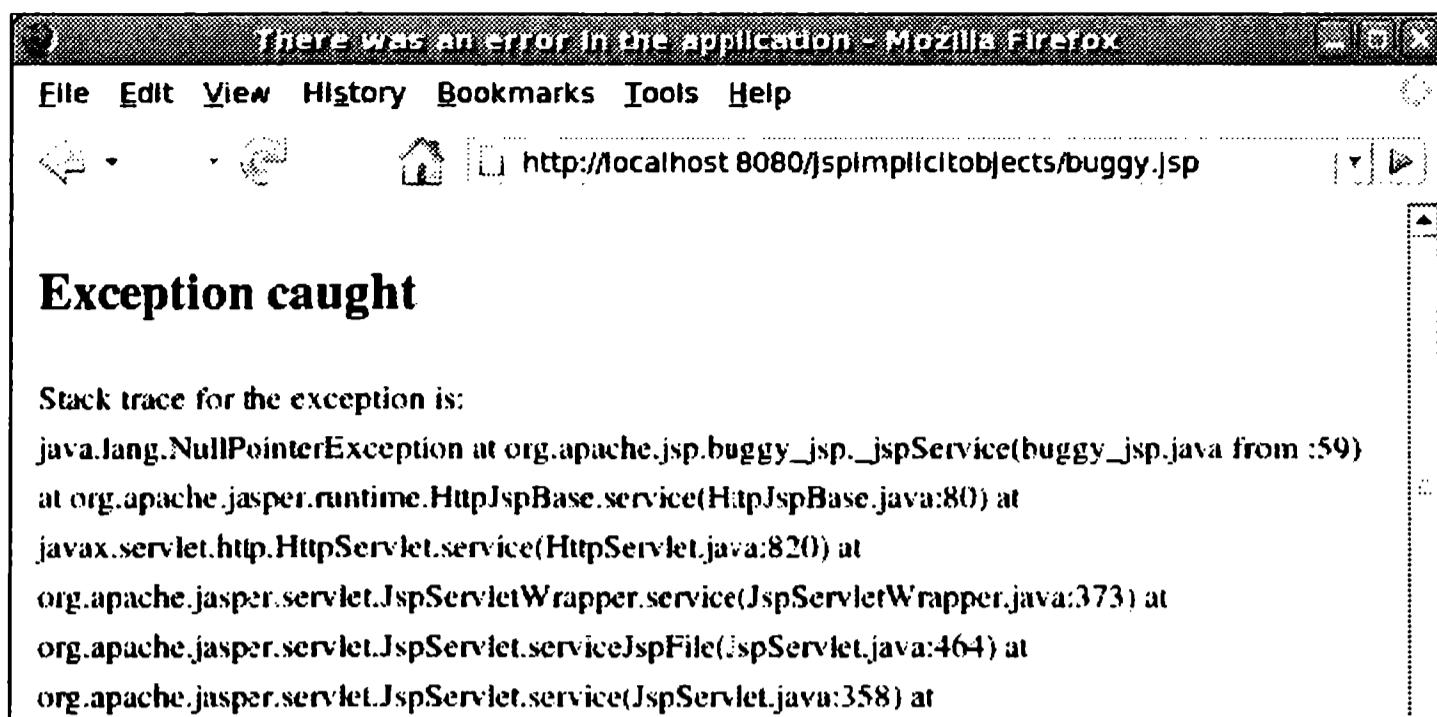
Щелкнув по гиперссылке внизу страницы, мы перейдем к странице implicitobjects2.jsp:



Здесь значение для атрибута запроса появляется как null. Причина в том, что когда мы щелкнули по гиперссылке на предыдущей странице, был создан новый HTTP-запрос, поэтому были потеряны любые атрибуты, присоединенные к предыдущему запросу. Если бы мы были переадресованы запросом к этой JSP-странице, то увидели бы предполагаемое значение в окне обозревателя.

Обратите внимание, что мы присоединили к приложению дополнительный атрибут. GlassFish также присоединяет много других атрибутов к этому неявному объекту.

Наконец, щелкнув по гиперссылке внизу страницы, мы перейдем к JSP-странице, содержащей ошибку, которая не отобразится. Вместо этого управление будет передано странице error.jsp:



Ничего особенного и заслуживающего внимания здесь не отображается; мы видим трассировку стека исключения, как и ожидалось.

JSP и JavaBeans

С помощью JSP-страниц очень легко устанавливать и получать свойства JavaBean. JavaBean является разновидностью класса Java. Для того чтобы класс можно было квалифицировать как JavaBean, он должен обладать следующими характеристиками:

- он должен иметь открытый (`public`) конструктор без аргументов;
- к его переменным можно получить доступ с помощью геттеров и сеттеров (методов `get()` и `set()`);
- он должен реализовывать интерфейс `java.io.Serializable`;
- хотя это и не строгое требование, существует своеобразное «правило хорошего тона» в кодировании, когда все переменные – члены JavaBean объявляются частными (`private`).

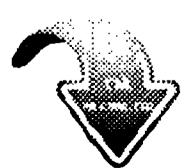


Не путайте JavaBeans с Enterprise JavaBeans. Это далеко не одно и то же. Enterprise JavaBeans подробно описывается в главе 9.

Все примеры в этом разделе будут использовать следующий JavaBean для пояснения интегрирования JavaBean и JSP:

```
package net.ensode.glassfishbook.javabeansproperties;

public class CustomerBean
{
    public CustomerBean()
    {
    }
    String firstName;
    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 видно из приведенного кода, этот класс квалифицируется как JavaBean, поскольку он удовлетворяет всем требованиям, перечисленным ранее. Обратите внимание, что имена методов геттеров и сеттеров отвечают соглашению о присвоении имен. Так, названия методов геттеров начинаются со слова `get`, за которым следует имя свойства, а названия методов сеттеров – со слова `set`, за которым следует имя свойства. Таким образом, имена методов состоят из имени свойства с добавленными к ним указанными префиксами. Важно придерживаться этого соглашения при интеграции работы JSP и JavaBean.

Страницы JSP объявляют, что будут использовать JavaBean с помощью тега `<jsp:useBean>`. Свойства JavaBean устанавливаются с помощью тега `<jsp:setProperty>` и получаются через тег `<jsp:getProperty>`.



В терминологии JavaBean свойство просто ссылается на одну из переменных класса JavaBean.

Следующий пример поясняет использование этих тегов:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
           pageEncoding="UTF-8" %>
<jsp:useBean id="customer"
              class="net.ensode.glassfishbook.javabeanproperties.
              CustomerBean"
              scope="page">
</jsp:useBean>
<jsp:setProperty name="customer" property="firstName" value="Albert"/>
<jsp:setProperty name="customer" property="lastName" value="Chan"/>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
          "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Свойства JavaBean</title>
    </head>
    <body>
        <form>
            <table cellpadding="0" cellspacing="0" border="0">
                <tr>
                    <td align="right">Имя: </td>
                    <td>
                        <input type="text" name="firstName"
                               value='<jsp:getProperty name="customer"
                               property="firstName"/>'>
                    </td>
                
```

```
</tr>
<tr>
    <td align="right">Фамилия:&nbsp;</td>
    <td>
        <input type="text" name="lastName"
               value='<jsp:getProperty name="customer"
               property="lastName"/>'>
    </td>
</tr>
<tr>
    <td></td>
    <td><input type="submit" value="Отправить"></td>
</tr>
</table>
</form>
</body>
</html>
```

Как видно из этого примера, тег `<jsp:useBean>` обычно используется с тремя атрибутами. Атрибут `id` устанавливает идентификатор для бина таким образом, чтобы мы могли позже к нему обратиться, атрибут `class` указывает полностью определенное (квалифицированное) имя бина, а атрибут `scope` указывает контекст бина. В нашем примере контекстом бина является `page`. Этот контекст является специфичным для страниц JSP и не может использоваться с сервлетами. К объектам в этом контексте может получить доступ только страница JSP, которая их объявляет. Другими допустимыми значениями для атрибута `scope` являются `application`, `session` и `request`. Если кроме `page` указываются другие атрибуты, JSP ищет объект, присоединенный к указанному контексту с именем, соответствующим указанному идентификатору. Если страница обнаруживает этот объект, она его использует; в противном случае страница присоединяет бин к указанному контексту. Если не указан никакой контекст, то контекстом по умолчанию является `page`. Если объект, присоединенный к контексту, не является экземпляром ожидаемого класса, то генерируется исключение `ClassCastException`.

Свойства бина могут быть установлены путем использования тега `<jsp:setProperty>`. Атрибут `name` этого тега идентифицирует бин, для которого мы устанавливаем свойство. Его значение должно соответствовать значению атрибута `id` тега `<jsp:useBean>`. Значение атрибута `property` должно соответствовать имени одного из свойств бина. Атрибут `value` определяет значение, которое будет присвоено свойству бина. «За кулисами» вызывается сеттер свойства при помощи тега `<jsp:setProperty>`.

У тега `<jsp:getProperty>` имеются два атрибута: `name` и `property`. Атрибут `name` идентифицирует бин, из которого мы получаем значение. Его значение должно соответствовать атрибуту `id` тега `<jsp:useBean>` бина. Атрибут `property` идентифицирует, какое свойство бина нам нужно. Атрибут `<jsp:getProperty>` вызывает метод `get` для соответствующего свойства, указанного в его атрибуте `property`.

После упаковки и развертывания предыдущей JSP-страницы, указав в адресной строке обозревателя URL <http://localhost:8080/javabeansproperties/beanproperties1.jsp>, мы должны увидеть страницу, похожую на следующую:



Обратите внимание, что в форме показаны поля с уже заполненными свойствами бина, после того как мы встроили теги `<jsp:getProperty>` в атрибуты `value` HTML-тега `input`.

В предыдущем примере сама JSP устанавливает свойства бина из жестко закодированных значений и позже получает к ним доступ через тег `<jsp:getProperty>`. Чаще всего атрибуты бина устанавливаются из параметров запроса. Если мы возьмем предыдущую JSP и заменим фрагмент кода

```
<jsp:setProperty name="customer" property="firstName" param="Albert"/>
<jsp:setProperty name="customer" property="lastName" param="Chan"/>
```

следующим:

```
<jsp:setProperty name="customer" property="firstName" param="fNm"/>
<jsp:setProperty name="customer" property="lastName" param="lNm"/>
```

то JSP заполнит атрибуты бина из параметров запроса. Единственная разница между модифицированной и исходной JSP заключается в том, что атрибут `value` тега `<jsp:setProperty>` был заменен атрибутом `param`. Когда у тега `<jsp:setProperty>` есть атрибут `param`, он ищет название параметра, соответствующее этому значению, в запросе. При обнаружении такого названия параметра он устанавливает значение этого параметра запроса в соответствующее свойство бина.

Повторное развертывание приложения и указание в адресной строке обозревателя следующей строки: `http://localhost:8080/javabeans/beanproperties2.jsp?fNm=Albert&lNm=Chan` (предполагается, что модифицированная страница JSP была сохранена как `beanproperties2.jsp`) должно привести к отображению страницы, показанной на предыдущем снимке экрана.

Если названия параметра запроса соответствуют именам свойств бина, нет необходимости явно определять соответствие каждого имени свойства соответствующему атрибуту запроса. Имеется сокращение, которое будет устанавливать каждый атрибут бина в соответствующее ему значение в запросе. Если мы модифицируем JSP еще раз, на сей раз заменив фрагмент кода

```
<jsp:setProperty name="customer" property="firstName" param="fNm"/>
<jsp:setProperty name="customer" property="lastName" param="lNm"/>
```

следующим:

```
<jsp:setProperty name="customer" property="*"/>
```

тег <jsp:setProperty> будет теперь искать названия параметров запроса, соответствующие именам свойств бина, и устанавливать свойства бина в значения соответствующих параметров запроса. Указывая в адресной строке обозревателя <http://localhost:8080/javabeanproperties/beanproperties3.jsp?firstName=Albert&lastName=Chan> (предполагается, что модифицированная страница JSP была сохранена как beanproperties3.jsp), мы должны будем еще раз увидеть страницу наподобие той, что показана на предыдущем снимке экрана. Обратите внимание, что в этом случае названия параметров запроса соответствуют именам свойств бина.

Хотя в примерах этого раздела мы имели дело исключительно со строковыми свойствами, технологии, продемонстрированные здесь, работают и с числовыми свойствами. Значения свойств из запроса или из тега <jsp:setProperty> автоматически преобразуются в соответствующий тип.

Повторное использование JSP-контента

В большинстве веб-приложений определенные области дублируются на нескольких веб-страницах. Например, в верхней части каждой страницы может выводиться логотип (фирменный знак) или навигационное меню. Копирование и вставка кода для генерации этих общих зон – не очень удобный способ работы, поскольку, если в одной из этих зон производится некоторое изменение, его приходится переносить и в соответствующие зоны на других страницах.

При использовании JSP-страниц для разработки веб-приложения можно определить каждую из таких общих областей в единственной JSP-странице, а затем сделать ее частью других JSP-страниц. Например, у нас может быть одна JSP-страница, которая представляет собой навигационное меню сайта, и ряд других JSP-страниц, которые включают в себя навигационное меню в виде JSP-страницы. Если навигационное меню понадобится изменить, это достаточно сделать однократно. Страницы JSP, включающие в себя JSP-страницу с навигационным меню, изменять не придется.

Включить одну JSP-страницу в другую можно двумя способами: через тег <jsp:include> либо через директиву include.

Следующий пример поясняет использование директивы include для включения JSP-страницы в родительскую JSP-страницу:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
       pageEncoding="UTF-8" %>
<%! String pageName ="Main"; %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
      "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```



```
<title>Главная страница</title>
</head>
<body>
<table cellpadding="0" cellspacing="0" border="1" width="100%" height="100%>
<tr>
<td width="100">
<%@ include file="navigation.jspf" %>
</td>
<td>Это главная страница.</td>
</tr>
</table>
</body>
</html>
```

Как видно из этого примера, директива `include` очень проста в использовании. Она принимает единственный атрибут – `file`, значением которого является файл для включения. Включенный в нашем примере файл имеет расширение `.jspf` – это рекомендуемое расширение для фрагментов JSP, т. е. страниц JSP, которые не отображаются в соответствующую HTML-страницу.

Обратите внимание на следующую строку в верхней части разметки:

```
<%! String pageName = "Main"; %>
```

Эта строка является *декларацией JSP* (JSP declaration). Любые переменные (или методы), объявленные в декларации JSP, доступны объявившей их JSP-странице, а также любой странице JSP, включенной в нее через директиву `include`.

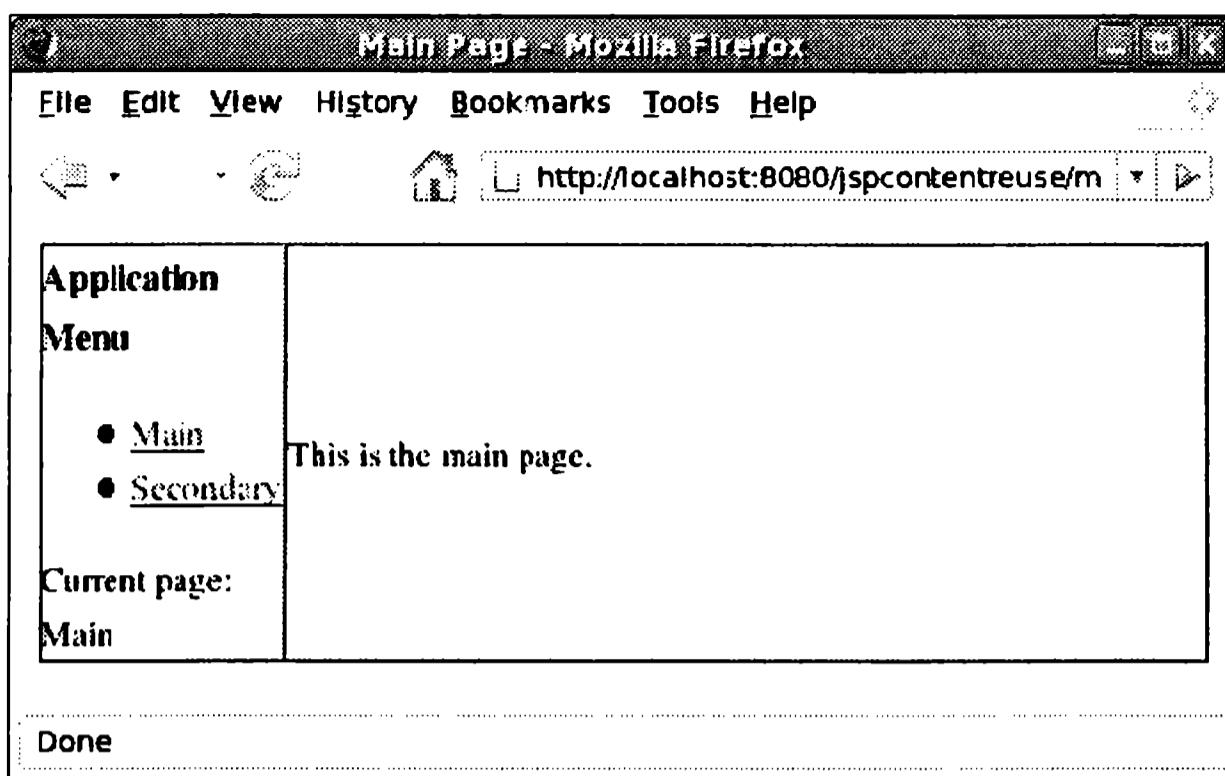
Код и разметка для `navigation.jspf` показаны ниже:

```
<b>Application Menu</b>
<ul>
<li> <a href="main.jsp">Main</a>
<li> <a href="secondary.jsp">Secondary</a>
</ul>
Current page: <%= pageName %>
```

Обратите внимание, что `navigation.jspf` получает доступ к переменной `pageName`, объявленной в родительской JSP-странице (для того чтобы это работало, любая JSP-страница, включающая `navigation.jspf`, должна объявлять переменную с именем `pageName`).

Есть и третий файл – `secondary.jsp`; он практически идентичен файлу `main.jsp`, поэтому не показан. Различие между `main.jsp` и `secondary.jsp` состоит в значении переменной `pageName`, хранящей название страницы, и в тексте во второй ячейке таблицы.

После упаковки и развертывания этих файлов в WAR-файл и указания в адресной строке обозревателя URL `http://localhost:8080/jspcontentreuse/main.jsp`, мы должны увидеть примерно такую страницу:



Меню с левой стороны представляется файлом `navigation.jspf`. Основная область представляется файлом `main.jsp`. Щелкнув по гиперссылке с меткой **Secondary** (Вторая), мы перейдем ко второй странице, которая фактически идентична главной.



Мы признаем тот факт, что предложенный нами веб-дизайн примитивен и не использует современные возможности его стилизации. Причина в том, что мы хотим показать как можно более простую HTML-разметку, чтобы можно было сосредоточиться на излагаемой теме, не отвлекаясь на сопутствующие детали.

Файлы JSP, подключенные через директиву страницы, включаются в основную страницу во время компиляции, т. е. когда наша JSP-страница преобразуется в сервлет. Именно по этой причине включенные JSP-страницы имеют доступ к переменным, объявленным в родительской JSP-странице.

При использовании тега `<jsp:include>` включенная JSP-страница добавляется во время выполнения. Поэтому у нее нет доступа ни к каким переменным, объявленным в родительской JSP-странице.

У тега `<jsp:include>` есть два атрибута: первый – `page`, который устанавливает страницу для включения, и второй, дополнительный, – `flush`, который определяет, должен ли какой-либо существующий буфер быть очищен перед считыванием включенной JSP-страницы. Допустимыми значениями для атрибута `flush` являются `true` и `false`; значением по умолчанию для него является `false`.

Предыдущие JSP-страницы могут быть легко модифицированы для использования тега `<jsp:include>`. Для этого нужно всего лишь заменить директиву `include` эквивалентным тегом `<jsp:include>` и, конечно, удалить JSP-выражения из `navigation.jspf`, поскольку фрагмент будет включен во время выполнения и не будет иметь доступа к странице.

Пользовательские теги JSP

Технология JSP позволяет разработчикам программного обеспечения создавать пользовательские теги. Пользовательские теги могут использоваться в JSP-страницах

наряду со стандартными HTML-тегами. Существует несколько способов разработки пользовательских тегов. В этом разделе мы обсудим два самых популярных способа: расширение класса `javax.servlet.jsp.tagext.SimpleTagSupport` и создание файла тега.

Расширение класса SimpleTagSupport

Один из способов, которым мы можем создавать пользовательские теги JSP, – расширение класса `javax.servlet.jsp.tagext.SimpleTagSupport`. Этот класс предоставляет реализации по умолчанию всех методов интерфейса `javax.servlet.jsp.tagext.SimpleTag`, а также некоторые методы, не определенные в этом интерфейсе. В большинстве случаев для создания пользовательского тега этим способом нам нужно всего лишь переопределить метод `SimpleTagSupport.doTag()`.

Давайте поясним данный подход на примере. Большинство HTML-форм имеют встроенную таблицу, содержащую несколько строк с полями ввода и их метками. Давайте создадим пользовательский тег JSP, который генерирует каждую из этих строк (для простоты наш тег будет генерировать только текстовые поля):

```
package net.ensode.glassfishbook.customtags;

import java.io.IOException;
import javax.servlet.jsp.JspContext;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.SimpleTagSupport;
public class LabeledTextField extends SimpleTagSupport
{
    private String label;
    private String value = "";
    private String name;
    @Override
    public void doTag() throws JspException, IOException
    {
        JspContext jspContext = getJspContext();
        JspWriter jspWriter = jspContext.getOut();
        jspWriter.print("<tr>");
        jspWriter.print("<td>");
        jspWriter.print("<b>");
        jspWriter.print(label);
        jspWriter.print("</b>");
        jspWriter.print("</td>");
        jspWriter.print("<td>");
        jspWriter.print("<input type=\"text\" name=\"");
        jspWriter.print(name);
        jspWriter.print("\" ");
        jspWriter.print("value=\"");
        jspWriter.print(value);
        jspWriter.print("\"");
        jspWriter.print("/>");
        jspWriter.print("</td>");
        jspWriter.println("</tr>");
    }
    public String getLabel()
    {
        return label;
    }
}
```

```
public void setLabel(String label)
{
    this.label = label;
}
public String getName()
{
    return name;
}
public void setName(String name)
{
    this.name = name;
}
public String getValue()
{
    return value;
}
public void setValue(String value)
{
    this.value = value;
}
}
```

Этот класс состоит из переопределенной версии метода `doTag()` и нескольких атрибутов. Наш метод `doTag()` получает ссылку на экземпляр `javax.servlet.jsp.JspWriter` методом `getJSPContext()`. Этот метод определяется в родительском классе тега и возвращает экземпляр `javax.servlet.jsp.JspContext`. Затем мы вызываем метод `getOut()` экземпляра `JspContext`. Данный метод возвращает экземпляр `javax.servlet.jsp.JspWriter`, который может использоваться для отправки вывода обозревателю с помощью его методов `print()` и `println()`. Остальная часть метода `doTag()` в основном отправляет вывод обозревателю через два вышенназванных метода.

Обратите внимание, что некоторые из вызовов метода `jspWriter.print()` в методе `doTag()` принимают переменные экземпляра в качестве их параметра. Эти атрибуты устанавливают теги, содержащиеся в JSP, с помощью файла *Дескриптора библиотеки тегов* (*Tag Library Descriptor* (TLD)).

Для того чтобы использовать пользовательские теги в наших JSP-страницах, должен быть создан файл Дескриптора библиотеки тегов. Файл библиотеки TLD-тегов для предыдущего пользовательского тега показан ниже:

```
<taglib xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                           web-jsptaglibrary_2_1.xsd"
                           xmlns="http://java.sun.com/xml/ns/javaee"
                           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                           version="2.1">
<tlib-version>1.0</tlib-version>
<uri>DemoTagLibrary</uri>
<tag>
    <name>labeledTextField</name>
    <tag-class>
        net.ensode.glassfishbook.customtags.LabeledTextField
    </tag-class>
    <body-content>empty</body-content>
    <attribute>
        <name>label</name>
        <required>true</required>
```



```

<rteprvalue>true</rteprvalue>
</attribute>
<attribute>
  <name>value</name>
  <rteprvalue>true</rteprvalue>
</attribute>
<attribute>
  <name>name</name>
  <required>true</required>
  <rteprvalue>true</rteprvalue>
</attribute>
</tag>
</taglib>

```

Файл TLD должен содержать элемент `<tlib-version>`, который указывает версию библиотеки тегов. Также он должен содержать элемент `<uri>`. Элемент `<uri>` используется в JSP-странице, содержащей тег, для однозначного определения библиотеки тегов. Последнее и наиболее важное: файл TLD должен содержать один или несколько элементов `<tag>`. Файлы TLD должны быть помещены в каталог WEB-INF WAR-файла приложения или в один из его подкаталогов. Как показано в предыдущем примере файла TLD, элемент `<tag>` содержит несколько подэлементов:

- подэлемент `<name>`, который присваивает логическое имя пользовательскому тегу;
- подэлемент `<tag-class>`, который идентифицирует полностью определенное (квалифицированное) имя пользовательского тега;
- один или более подэлементов `<attribute>`, которые определяют атрибуты пользовательского тега.

Подэлемент `<attribute>`, в свою очередь, также может содержать несколько подэлементов:

- подэлемент `<name>` определяет имя атрибута. Значение этого подэлемента должно соответствовать имени одной из переменных экземпляра тега с соответствующим сеттером (методом `set()`);
- необязательный подэлемент `<required>`, указывающий, является ли обязательным элемент, передающий значение атрибуту. Если этот элемент будет установлен в значение `true` и в JSP не будет никакого значения, отправляемого атрибуту, то на странице возникнет ошибка компиляции. Значение по умолчанию для этого элемента – `false`;
- необязательный тег `<rteprvalue>`, указывающий, может ли атрибут содержать выражение времени выполнения в качестве своего значения. Если данный элемент будет установлен в значение `true`, тег будет принимать выражения Унифицированного языка выражений (Unified Expression Language) в качестве своего значения. Унифицированный язык выражений будет подробно обсуждаться ниже в этой главе.

 Мы описали только наиболее часто используемые элементы файла TLD. Чтобы просмотреть полный список элементов файла TLD, обратитесь к спецификации JSP 2.1 по адресу: http://download.oracle.com/otn-pub/jcp/jsp-2.1-fr-spec-other-JSpec/jsp-2_1-fr-spec.pdf.

После того как у нас появился код тега и TLD, мы будем готовы к использованию тега в JSP:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
           pageEncoding="UTF-8" %>
<%@ taglib prefix="d" uri="DemoTagLibrary" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
           "http://www.w3.org/TR/html4/loose.dtd">

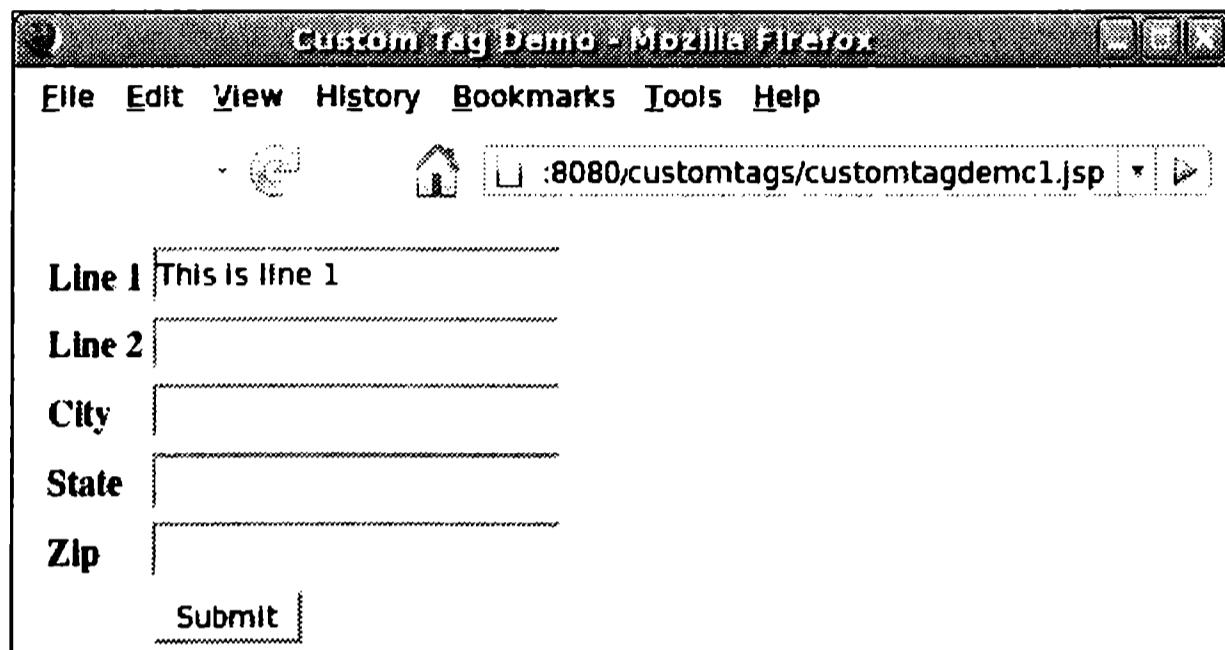
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
          charset=UTF-8">
    <title>Демонстрация пользовательского тега</title>
  </head>
  <body>
    <form>
      <table>
        <d:labeledTextField label="Улица 1" name="line1"
                           value="Где эта улица где этот дом">
        </d:labeledTextField>
        <d:labeledTextField label="Улица 2" name="line2">
        </d:labeledTextField>
        <d:labeledTextField label="Город" name="city">
        </d:labeledTextField>
        <d:labeledTextField label="Штат" name="state">
        </d:labeledTextField>
        <d:labeledTextField label="Zip код" name="zip">
        </d:labeledTextField>
      <tr>
        <td></td>
        <td><input type="submit" value="Отправить"></td>
      </tr>
    </table>
  </form>
</body>
</html>
```

Эта JSP-страница использует наш пользовательский тег для генерации элементарной формы ввода адресных сведений. Первое, что нам должно броситься в глаза на этой JSP-странице, – это использование директивы `taglib`. Эта директива уведомляет JSP о том, что мы будем использовать пользовательскую библиотеку тегов. Атрибут `uri` директивы `taglib` должен соответствовать значению элемента `<uri>` в файле библиотеки TLD-тегов. Значение атрибута `prefix` директивы `taglib` ожидается перед именем любого пользовательского тега из библиотеки, которую мы используем. В предыдущем примере все атрибуты `<d:labeledField>` используют пользовательский тег, который мы разработали. Символ `d` перед символом `:` в каждом из этих тегов соответствует значению атрибута `prefix`.

Следующее, на что мы обратим внимание в этом примере, – собственно использование пользовательского тега. Обратите внимание, что каждый раз, когда мы используем пользовательский тег, мы устанавливаем для него значение атрибутов `label` и `value`. Мы должны это сделать, потому что данные атрибуты были объявлены как `required` в файле тегов TLD. Значения, которые мы устанавливаем для атрибутов тега, автоматически используются для установки значений переменных экземпляра класса тега Java. Имя атрибута совпадает с именем соответствующей переменной

экземпляра. «За кулисами» вызывается сеттер-метод класса тега для соответствующей переменной экземпляра.

После того как мы упакуем JSP-код пользователяского тега в файл TLD в WAR-файле, развернем его и обратимся к нашей JSP-странице через адресную строку браузера, мы должны увидеть, что предыдущая JSP-страница отобразится в обозревателе примерно таким образом:



Обратите внимание, что предварительно заполнено только первое текстовое поле. Причина в том, что это поле было единственным, для которого мы установили атрибут `value`.

Если посмотреть на сгенерированную из нашей JSP-страницы HTML-разметку, то мы сможем увидеть разметку, которая была фактически сгенерирована из нашего пользовательского тега:

```
<table>
  <tr>
    <td><b>Line 1</b></td>
    <td><input type="text" name="line1" value="Где эта улица где этот
дом"/></td>
  </tr>
  <tr>
    <td><b>Line 2</b></td>
    <td><input type="text" name="line2" value="" /></td>
  </tr>
  <tr>
    <td><b>City</b></td>
    <td><input type="text" name="city" value="" /></td>
  </tr>
  <tr>
    <td><b>State</b></td>
    <td><input type="text" name="state" value="" /></td>
  </tr>
  <tr>
    <td><b>Zip</b></td>
    <td><input type="text" name="zip" value="" /></td>
  </tr>
  <tr>
    <td></td>
    <td><input type="submit" value="Отправить"></td>
  </tr>
</table>
```

Для простоты и краткости здесь показана только часть сгенерированной разметки. Все выделенные строки были сгенерированы пользовательским тегом.

Использование файлов тегов для создания пользовательских тегов JSP

Как было показано в предыдущем разделе, создание пользовательского тега путем расширения класса SimpleTagSupport включает в себя написание некоторого кода Java для генерации HTML-разметки. Код для достижения этой цели, как правило, трудно писать и трудно читать. Альтернативным способом создания пользовательских тегов JSP является использование файлов тегов. Этот альтернативный способ не требует написания кода Java.

Файл тега очень похож на JSP. Имя файла тега должно заканчиваться расширением .tag, а сам файл должен быть помещен в подкаталог, называемый tags в каталоге WEB-INF WAR-файла. Следующий файл тега генерирует полное (и менее примитивное) поле ввода адреса:

```
<%@ tag language="java" %>
<%@ attribute name="addressType" required="true" %>
<jsp:useBean id="address" scope="request"
              class="net.ensode.glassfishbook.customtags.AddressBean"/>
<table cellpadding="0" cellspacing="0" border="0">
    <tr>
        <td align="right" width="70"><b>Улица 1</b>&ampnbsp</td>
        <td><input type="text" name="${addressType}_line1" size="30"
                  maxlength="100" value="${address.line1}"></td>
    </tr>
    <tr>
        <td align="right"><b>Улица 2</b>&ampnbsp</td>
        <td><input type="text" name="${addressType}_line2" size="30"
                  maxlength="100" value="${address.line2}"></td>
    </tr>
    <tr>
        <td align="right"><b>Город</b>&ampnbsp</td>
        <td><input type="text" name="${addressType}_city" size="30"
                  value="${address.city}"></td>
    </tr>
    <tr>
        <td align="right"><b>Штат</b>&ampnbsp</td>
        <td>
            <select name="${addressType}_state">
                <option value=""></option>
                <option value="AL"
                       <% if(address.getState().equals("AL"))
                           out.print(" выбрано "); %>>Alabama
                </option>
                <option value="AK"
                       <% if(address.getState().equals("AK"))
                           out.print(" выбрано "); %>>Alaska
                </option>
                <option value="AZ"
                       <% if(address.getState().equals("AZ"))
                           out.print(" выбрано "); %>>Arizona
                </option>
                <option value="AR"
                       <% if(address.getState().equals("AR"))
                           out.print(" выбрано "); %>>Arizona
                </option>
            </select>
        </td>
    </tr>
```



```

        out.print(" выбрано ") ; %>>Arkansas
    </option>
    <option value="CA"
        <% if(address.getState().equals("CA"))
            out.print(" выбрано ") ; %>>California
    </option>
    <option value="CO"
        <% if(address.getState().equals("CO"))
            out.print(" выбрано ") ; %>>Colorado
    </option>
    <option value="CT"
        <% if(address.getState().equals("CT"))
            out.print(" выбрано ") ; %>>Conneticut
    </option>
    <option value="DC"
        <% if(address.getState().equals("DC"))
            out.print(" выбрано ") ; %>>District of Columbia
    </option>
    <option value="FL"
        <% if(address.getState().equals("FL"))
            out.print(" выбрано ") ; %>>Florida
    </option>
</select>
</td>
<tr>
    <td align="right"><b>Zip код</b>&ampnbsp</td>
    <td><input type="text" name="${addressType}_zip" size="5"
        value="${address.zip}">
    </td>
</tr>
</table>
```

Как видно из примера, файл тега очень похож на файл JSP. Точно так же, как JSP, он может содержать скриптлеты и свойства set и get JavaBean. Различие между файлом тега и JSP-страницей состоит в том, что файлы тега используют директиву tag вместо директивы page. Обычно используемым атрибутом директивы tag является атрибут import, который точно так же, как и в JSP-директиве page, используется для импорта отдельных классов или пакетов, которые будут использоваться в файле тега.

У файлов тега может быть директива attribute, генерирующая атрибут, который может быть установлен родительским файлом JSP. В предыдущем примере создается требуемый атрибут, называемый addressType.

Обратите внимание, что значение для атрибута name каждого поля ввода в файле тега в качестве примера содержит следующий текст: \${addressType}_line1. Первая часть этой строки (\${addressType}) является специальной нотацией для получения значения атрибута addressType. Эта нотация также может использоваться для получения значения свойств JavaBean. Синтаксис для получения свойства JavaBean с использованием этой нотации следующий: \${<имя бина>.<имя свойства>}. Атрибут value каждого поля ввода input в предыдущем примере использует эту нотацию для получения значений свойств бина address. Бин address является простым JavaBean, который объявляет несколько атрибутов наряду с соответствующими им методами сеттеров и геттеров.



Нотация \${} является частью Унифицированного языка выражений – нового языка выражений для спецификации JSP 2.1. Эта нотация совместима с языком выражений JSP, введенным в спецификации JSP 2.0. Однако Унифицированный язык выражений также поддерживает и нотацию #{}. Эта новая нотация не является совместимой с предыдущими версиями спецификации JSP. Нотация #{} будет подробно рассматриваться в главе 6, «JavaServer Faces».

Как видно из приведенного примера, файлы тега могут содержать скриплеты. Скриплеты в примере сравнивают значение атрибута state в бине State (Штат) с каждой из опций элемента выбора, после чего устанавливают соответствующий элемент в выбранное значение (для простоты и краткости использовалось только малое подмножество всех значений state).

Использование пользовательских тегов, определенных в файле тега, практически идентично использованию тега, определенного с помощью Java-кода:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
           pageEncoding="UTF-8" %>
<%@ taglib prefix="ct" tagdir="/WEB-INF/tags" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
           "http://www.w3.org/TR/html4/loose.dtd">

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Демонстрация пользовательского тега</title>
  </head>
  <body>
    <form>
      <h3>Адрес доставки</h3>
      <ct:address addressType="shipping"/>
    </form>
  </body>
</html>
```

Обратите внимание, что директива taglib используется для импорта библиотеки тегов в JSP. Однако в этом случае вместо атрибута uri используется атрибут tagdir для указания места расположения библиотеки тегов. Все файлы тегов, расположенные в том же каталоге, неявно являются частью библиотеки тегов; никакие TLD-файлы в этом случае не требуются. Однако имеется возможность добавления TLD-файла для библиотеки тегов, состоящей из файлов тегов. TLD-файл для такой библиотеки тегов следует назвать implicit.tld и поместить в тот же самый каталог, что и файлы тегов (в предыдущем примере это WEB-INF/tags; библиотеки тегов должны быть помещены в этот каталог или любой подкаталог каталога tags).

Для того чтобы предыдущая JSP-страница работала должным образом, экземпляр net.ensode.glassfishbook.customtags.AddressBean должен быть присоединен к запросу. Следующий сервлет создаст экземпляр этого класса. Заполните некоторые из его полей и переадресуйте запрос к предыдущей JSP-странице.

```
package net.ensode.glassfishbook.customtags;

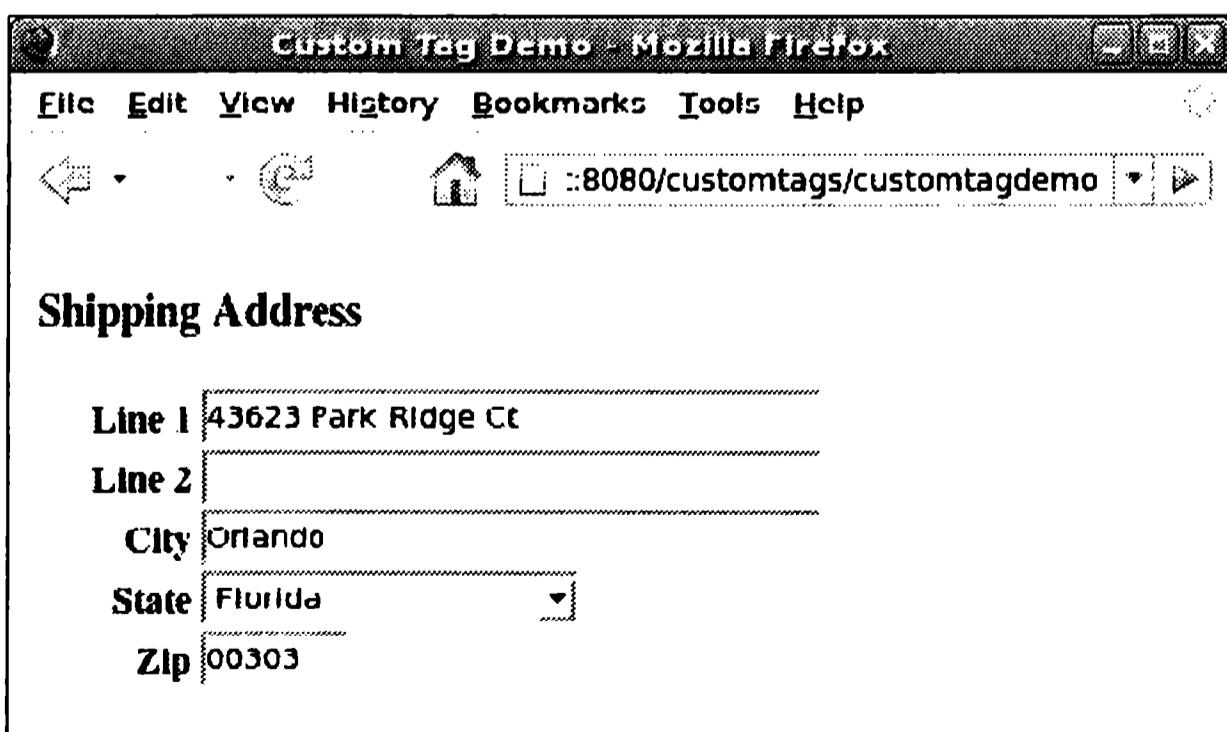
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```



```
public class CustomTagDemoServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response)
    {
        AddressBean addressBean = new AddressBean();
        addressBean.setLine1("43623 Park Ridge Ct");
        addressBean.setCity("Orlando");
        addressBean.setState("FL");
        addressBean.setZip("00303");
        request.setAttribute("address", addressBean);
        try
        {
            request.getRequestDispatcher("customtagdemo2.jsp").forward(request, response);
        }
        catch (ServletException e)
        {
            e.printStackTrace();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

Реальное приложение, вероятно, получило бы эту информацию из базы данных. Этот простой пример всего лишь создает экземпляр бина и заполняет его некоторыми произвольными атрибутами.

После упаковки этой JSP-страницы и файла тега в WAR-файле, развертывания WAR-файла и указания в адресной строке обозревателя URL сервлета (определенного в элементе `<servlet-mapping>` файла дескриптора развертывания приложения `web.xml`) мы должны увидеть страницу наподобие следующей:



Унифицированный язык выражений

В предыдущем разделе мы видели, как может использоваться унифицированный язык выражений для получения значений свойств из JavaBean. Когда к свойствам JavaBean получают доступ этим способом, веб-контейнер GlassFish ищет JavaBean с требуемым именем, присоединенный к контекстам: страницы, запроса, сеанса и приложения – в перечисленном порядке. Он использует первый из найденных для поиска и вызова метода геттера, соответствующего свойству, которое мы хотим получить.

Если нам известно, к какому контексту мы хотим присоединить бин, то мы можем получить его напрямую из этого контекста, поскольку у выражения JSP есть доступ к неявным объектам JSP. В следующем примере мы присоединяем несколько экземпляров JavaBean с именем CustomerBean к различным контекстам. Прежде чем пояснить JSP, давайте рассмотрим исходный код этого бина:

```
package net.ensode.glassfishbook.unifiedexprlang;
public class CustomerBean
{
    public CustomerBean()
    {
    }
    public CustomerBean(String firstName, String lastName)
    {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    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;
    }
    @Override
    public String toString()
    {
        StringBuffer fullNameBuffer = new StringBuffer();
        fullNameBuffer.append(firstName);
        fullNameBuffer.append(" ");
        fullNameBuffer.append(lastName);
        return fullNameBuffer.toString();
    }
}
```

Это довольно простой JavaBean, состоящий из двух свойств и соответствующих им методов сеттеров и геттеров. Для того чтобы этот класс можно было квалифицировать как JavaBean, он должен иметь конструктор с модификатором видимости public, который не принимает аргументов. В дополнение к этому конструктору мы для удобства добавили конструктор, который принимает два аргумента для инициализации свойств бина. Дополнительно класс переопределяет метод `toString()` так, чтобы его вывод представлял имя и фамилию пользователя.

Как мы упоминали ранее, нижеследующая JSP-страница получает экземпляры `CustomerBean` из различных контекстов с помощью унифициированного языка выражений и представляет соответствующий вывод в окне обозревателя.



Прежде чем данная JSP-страница будет выполнена, все экземпляры `CustomerBean` должны быть присоединены к соответствующему контексту. Мы написали сервлет, который это выполняет и затем переадресует запрос к JSP. Для краткости этот сервлет не показан, но он доступен как часть загрузки кода для этой книги.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
       pageEncoding="UTF-8" %>
<jsp:useBean scope="page" id="customer6"
       class="net.enseode.glassfishbook.unifiedexprlang.CustomerBean"/>
<jsp:setProperty name="customer6" property="firstName" value="David"/>
<jsp:setProperty name="customer6" property="lastName"
       value="Heffelfinger"/>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
       "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Демонстрация Унифициированного языка выражений</title>
  </head>
  <body>
    Заказчик присоединен к контексту приложения:
    ${applicationScope.customer1}
    <br/>
    <br/>
    Заказчик присоединен к контексту сеанса:
    ${sessionScope.customer2.firstName}
    ${sessionScope.customer2.lastName}
    <br/>
    <br/>
    Заказчик присоединен к контексту запроса:
    ${requestScope.customer3}
    <br/>
    <br/>
    Заказчик присоединен к контексту страницы:
    ${pageScope.customer6}
    <br/>
    <br/>
    Список заказчиков, присоединенных к сеансу:
    <br/>
    ${sessionScope.customerList[0]}
    <br/>
    ${sessionScope.customerList[1].firstName}
    ${sessionScope.customerList[1].lastName}
    <br/>
    <br/>
  </body>
</html>
```

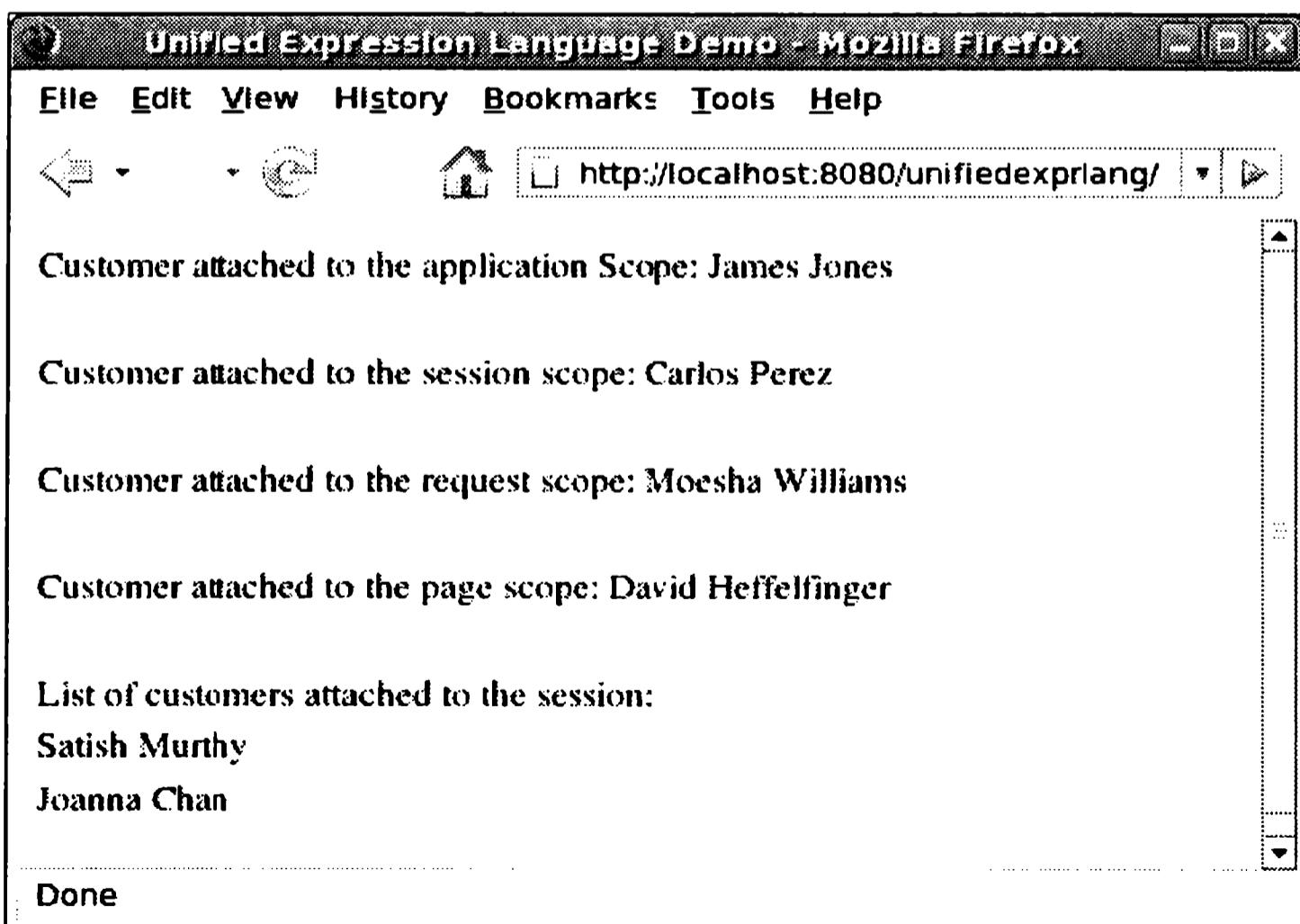
Первая выделенная строка в этой JSP-странице ищет бин, присоединенный к контексту приложения, с именем `customer1`. Поскольку мы не ссылаемся ни на одно из свойств бина, в этой точке кода вызывается метод `toString()`.

Следующие две выделенных строки ищут бин, присоединенный к контексту сеанса, с именем `customer2`. В этом случае мы получаем доступ к отдельным свойствам. Первая строка получает доступ к свойству `firstName`, а вторая строка – к свойству `lastName`. «За кулисами» веб-контейнер GlassFish вызывает соответствующие методы геттеров для каждого свойства.

Следующие две выделенных строки получают экземпляры `CustomerBean` из контекстов запроса и страницы соответственно. Поскольку мы не получаем доступа к отдельным свойствам бина, вновь вызывается метод `toString()`.

Последние три выделенных строки поясняют замечательную функцию унифицированного языка выражений. В этом случае экземпляры `CustomerBean` не были напрямую присоединены к сеансу. Вместо этого к сеансу был присоединен `ArrayList` с именем `customerList`, содержащий экземпляры `CustomerBean`. Как видно из этих трех строк, мы можем получить доступ к отдельным элементам `ArrayList` путем размещения нескольких элементов в скобках, подобно тому как мы поступили бы с массивом в регулярном коде Java. Между прочим, этот метод работает с массивами так же, как и с любыми другими классами, реализующими интерфейс `java.util.Collection`.

После упаковки предыдущей JSP-страницы в WAR-файл, развертывания WAR-файла и указания в адресной строке обозревателя соответствующего URL мы должны увидеть примерно такую страницу в окне обозревателя:



В данном конкретном случае метод `toString()` выводит имя и фамилию пользователя. Поэтому его вывод неотличим от отображения этих двух свойств рядом друг с другом.

Разумеется, методы, показанные в нашем примере, работают в любом контексте. Мы можем получить доступ к бину, присоединенному к любому контексту, не указывая никаких свойств. Точно так же мы можем получить доступ к свойствам бина в любом контексте и, конечно, доступ к отдельным элементам в коллекции или массиве, присоединенном к любому контексту.

XML-синтаксис JSP

В дополнение к использованию стандартного синтаксиса JSP, который мы обсуждали на протяжении этой главы, JSP-страница также может быть разработана с использованием XML-синтаксиса. Разработанные с использованием этого альтернативного синтаксиса JSP-страницы известны под названием *документов JSP* (*JSP documents*). В соответствии с соглашением имени файлов документа JSP заканчиваются расширением `.jspx`.

В следующей таблице сравниваются стандартный синтаксис JSP с эквивалентным ему XML-синтаксисом:

Свойство JSP	Пример стандартного синтаксиса	Пример XML-синтаксиса
Комментарий	<code><%-- comment --%></code>	<code><!-- comment --></code>
Объявление	<code><%! String s; %></code>	<code><jsp:declaration> String s; </jsp:declaration></code>
Выражение	<code><%= new java.util.Date() %></code>	<code><jsp:expression> new java.util.Date() </jsp:expression></code>
Скриптlet	<code><% x = 5 + y; %></code>	<code><jsp:scriptlet> ![CDATA[x = 5 + y;]]> </jsp:scriptlet></code>
Директива attribute	<code><%@ attribute name="addressType" required="true" %></code>	<code><jsp:directive.attribute name="addressType" required="true"/></code>
Директива include	<code><%@ include file="navigation.jspf" %></code>	<code><jsp:directive.include file="navigation.jspf"/></code>
Директива page	<code><%@ page import="java.util.Enumeration" %></code>	<code><jsp:directive.page import="java.util.Enumeration"/></code>
Директива tag	<code><%@ tag language="java" %></code>	<code><jsp:directive.tag language="java"/></code>
Директива taglib	<code><%@ taglib prefix="d" uri="DemoTagLibrary" %></code>	<code><jsp:root xmlns:d="DemoTagLibrary"></code>

Свойство JSP	Пример стандартного синтаксиса	Пример XML-синтаксиса
Директива variable	<%@ variable name-given="value" %>	<jsp:directive.variable name-given="value"/>

Как видно из таблицы, разработка JSP-страниц с использованием синтаксиса XML довольно проста, если нам уже известно, как разрабатывать JSP-страницы с использованием традиционного синтаксиса. Нужно отметить, что директивы tag и attribute, описанные в таблице, могут использоваться только в тегах JSP.

Чтобы разработать JSP с использованием XML-синтаксиса, мы должны просто использовать синтаксис XML для всех используемых нами функций JSP. Кроме того, поскольку документы JSP должны быть *XML-допустимыми* (valid XML), мы должны убедиться в том, что наши JSP-страницы имеют правильный формат – в частности, у каждого открывающегося тега имеется соответствующий ему закрывающийся тег и т. д.

Следующий документ JSP является модифицированной версией одного из примеров, рассмотренных нами в разделе «*JSP и JavaBeans*» (см. стр. 97):

```

<? xml version="1.0" encoding="UTF-8" ?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
    <jsp:directive.page language="java" contentType="text/html"
        pageEncoding="UTF-8"/>
    <jsp:useBean
        id="customer"
        class="net.ensode.glassfishbook.javabeanproperties.
            CustomerBean"
        scope="page">
    </jsp:useBean>
    <jsp:setProperty name="customer" property="firstName" param="fNm"/>
    <jsp:setProperty name="customer" property="lastName" param="lNm"/>
    <html>
        <head>
            <title>Свойства JavaBean</title>
        </head>
        <body>
            <form>
                <table cellpadding="0" cellspacing="0" border="0">
                    <tr>
                        <td align="right">Имя:&#160;</td>
                        <td><input type="text" name="firstName"
                            value='${customer.firstName}' />
                        </td>
                    </tr>
                    <tr>
                        <td align="right">Фамилия:&#160;</td>
                        <td><input type="text" name="lastName"
                            value='${customer.lastName}' />
                        </td>
                    </tr>
                    <tr>
                        <td></td>
                        <td><input type="submit" value="Отправить"/></td>
                    </tr>
                </table>
            </form>
        </body>
    </html>
</jsp:root>

```



```
</form>
</body>
</html>
</jsp:root>
```

Обратите внимание: для того чтобы сделать страницы XML-совместимыми при использовании синтаксиса XML, кроме внесения некоторых незначительных изменений нам понадобится всего лишь добавить элемент `<jsp:root>` и изменить директиву страницы под использование синтаксиса XML (см. предыдущую таблицу).

Резюме

В этой главе был рассмотрен ряд вопросов. Мы поговорили о том, как разработать и развернуть простые JSP-страницы. Также мы обсудили, как получить доступ из JSP-страниц к неявным объектам, таким как `request`, `session` и др. Было показано, как установить и получить значения свойств JavaBean с использованием тега `<jsp:useBean>`. В дополнение к этому мы рассмотрели способы включения одних JSP-страниц в другие во время выполнения при помощи тега `<jsp:include>` и во время компиляции с использованием директивы JSP `include`. Речь шла и о том, как написать пользовательские теги JSP путем расширения класса `javax.servlet.jsp.tagext.SimpleTagSupport` либо путем написания файлов тегов. Мы выяснили, как получить доступ к JavaBean и их свойствам с помощью Унифициированного языка выражений.

Наконец, был рассмотрен XML-синтаксис JSP, который позволяет нам разрабатывать XML-совместимые серверные страницы Java.

4

Библиотека стандартных тегов JSP

Библиотека стандартных тегов JSP (JSP Standard Tag Library (JSTL)) представляет собой набор стандартных тегов JSP, которые выполняют некоторые общие задачи. Она освобождает нас от необходимости разрабатывать пользовательские теги для этих задач или от необходимости использования «смеси» тегов нескольких различных разработчиков для решения нашей задачи.

JSTL содержит следующие теги:

- базовые теги, которые среди прочего выполняют условную логику и итерации по коллекциям;
- теги форматирования, которые выполняют строковое форматирование и интернационализацию;
- SQL-теги, которые взаимодействуют с базой данных;
- XML-теги для обработки XML-данных.

Кроме того, JSTL содержит ряд функций, выполняющих несколько задач, подавляющее большинство которых предназначено для работы со строками.

В этой главе мы рассмотрим каждую из JSTL-библиотек тегов, приводя примеры наиболее часто используемых тегов и функций. В главе будут затронуты следующие темы:

- JSTL-библиотека базовых тегов;
- JSTL-библиотека тегов форматирования;
- JSTL-библиотека SQL-тегов;
- JSTL-библиотека XML-тегов;
- JSTL-библиотека функций.

JSTL-библиотека базовых тегов

Базовые теги JSTL (Core JSTL tags) выполняют такие задачи, как написание вывода в обозреватель, условное отображение сегментов на странице и итерации по коллекциям. Большая часть того, что делают базовые теги JSTL, может быть выполнена с помощью скриптов. Однако страница будет намного более простой для вос-

приятия и потому более удобной в сопровождении, если вместо скриптов будут использоваться базовые теги JSTL.

В следующем примере показана JSP-страница, использующая некоторые из наиболее распространенных базовых тегов JSTL:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
           pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
           "http://www.w3.org/TR/html4/loose.dtd">
<%@ page import="java.util.ArrayList" %>
<html>
<%
    ArrayList<String> nameList = new ArrayList<String>(4);
    nameList.add("David");
    nameList.add("Raymond");
    nameList.add("Beth");
    nameList.add("Joyce");
    request.setAttribute("nameList", nameList);
%>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Демонстрация базовых тегов</title>
</head>
<body>
    <c:set var="name" scope="page" value="${param.name}"></c:set>
    <c:out value="Привет, "></c:out>
    <c:choose>
        <c:when test="${!empty name}">
            <c:out value="${name}"></c:out>
        </c:when>
        <c:otherwise>
            <c:out value="незнакомец!"></c:out>
            <br/>
            <c:out value="Вам нужно имя? Вот несколько вариантов:"/>
            <br />
            <ul>
                <c:forEach var="nameOption"
                           items="${requestScope.nameList}">
                    <li><c:out value="${nameOption}"></c:out>
                </c:forEach>
            </ul>
        </c:otherwise>
    </c:choose>
    <c:remove var="name" scope="page"/>
</body>
</html>
```

Говоря кратко, здесь выполняется поиск параметра запроса под названием name. Если такой параметр обнаружен, то на экран выводится сообщение «Привет, \${имя}». В обозревателе \${имя} заменяется фактическим значением параметра. Если параметр не найден, наш код выводит сообщение «Привет, незнакомец!» и пытается познакомиться с пользователем, предлагая несколько имен. Это можно увидеть на следующем снимке экрана:



Страница использует директиву `taglib`, чтобы объявить, что она использует библиотеку базовых тегов JSTL. Хотя для этой библиотеки можно использовать любой префикс, общепринятой практикой является использование префикса `c`¹.

У страницы имеется скриптлет, который, прежде чем что-либо будет сделано с JSTL, инициализирует экземпляр `java.util.ArrayList` некоторыми строками, содержащими имена, и присоединяет `ArrayList` к запросу (на практике это обычно делается в сервлете или некотором другом классе, а не непосредственно в JSP-странице; в данном примере это было сделано для простоты).

Первый тег JSTL, используемый на странице, – `<c:set>`. Этот тег устанавливает результат выражения, определенного в его атрибуте `variable`, и сохраняет его в переменной в указанном контексте. Имя переменной определяется в атрибуте `var`. Контекст переменной определяется в атрибуте тега `scope`; если не указан никакой контекст, то по умолчанию используется контекст страницы. Выражение, которое будет оценено, определяется в атрибуте `value`.



Контекст страницы всегда является значением по умолчанию

Многие теги JSTL содержат атрибут `var` для определения переменной в контексте, указанном атрибутом `scope`. В любом случае, если не указывается никакой контекст, по умолчанию используется контекст страницы.

В предыдущем примере выражение ищет значение параметра запроса с именем "name".param, являющееся неявной переменной, разрешающейся в карте (`map`), используя при этом названия параметров запросов в качестве ключей, а значения параметров запроса – в качестве значений. Эта неявная переменная эквивалентна вызову метода `getParameterMap()` в ответ на запрос. Значение после точки (имя в предыдущем примере) соответствует ключу, который мы хотим получить из карты параметров. Ключ, в свою очередь, соответствует имени параметра запроса.

Следующий базовый тег JSTL, который мы видим в примере, – это тег `<c:out>`. Он просто выводит на экран в обозревателе значение выражения, определенного в его атрибуте `value`. В данном случае выражение, определенное в атрибуте `value`, является константой, поэтому оно выводится на экран в обозревателе как есть.

Затем мы видим тег `<c:choose>`. Он позволяет нам выполнять конструкцию `if/then/else` как условие на странице. Тег `<c:choose>` должен содержать один или

¹ От слова core. – Прим. перев.

более тегов `<c:when>` и дополнительно – тег `<c:otherwise>`. Тег `<c:when>` содержит атрибут `test`, который должен содержать булево выражение. После того как значение выражения в одном из тегов `<c:when>`, вложенном в тег `<c:choose>`, будет оценено как `true` (истина), выполняется тело тега и атрибут `test` других тегов `<c:when>`, вложенных в тот же самый тег `<c:choose>`, не оценивается.

Следующий новый тег, который мы видим в примере, – `<c:otherwise>`. Тело этого необязательного тела выполняется, если ни одно из выражений в каком-либо из тегов `<c:when>` не будет оценено как истина (`true`). В примере тело тела выполнится, когда в запросе не будет никакого параметра запроса с именем "name" либо если значением этого параметра является пустая строка.

В предыдущем примере тег `<c:when>` содержит оператор `!`, который инвертирует булево выражение точно так же, как в Java. Тег также содержит оператор `empty`; этот оператор проверяет, имеет ли строковый объект `String` значение `null` или, возможно, нулевую длину. У атрибута `test` тега `<c:when>` может быть несколько логических операторов и/или операторов отношения, которые могут быть объединены для создания более сложного выражения. Все операторы отношений, которые могут использоваться в тестовом атрибуте (а также любые другие выражения Унифицированного языка выражений в этом отношении) перечислены в следующей таблице:

Операторы отношения	Описание
<code>==</code> или <code>eq</code>	<ul style="list-style-type: none"> Равно: истинно, если выражение слева от оператора совпадает с выражением справа от оператора
<code>></code> или <code>gt</code>	<ul style="list-style-type: none"> Больше чем: истинно, если выражение в левой части оператора больше, чем выражение в правой части от оператора
<code><</code> или <code>lt</code>	<ul style="list-style-type: none"> Меньше чем: истинно, если выражение в левой части оператора меньше, чем выражение в правой части от оператора
<code>>=</code> или <code>ge</code>	<ul style="list-style-type: none"> Больше или равно: будет истинным, если выражение слева от оператора больше или равно выражению справа от оператора
<code><=</code> или <code>le</code>	<ul style="list-style-type: none"> Меньше или равно: будет истинным, если выражение слева от оператора меньше или равно выражению справа от оператора
<code>!=</code> или <code>ne</code>	<ul style="list-style-type: none"> Не равно: истинно, если выражение в левой части от оператора не равно выражению в правой части от оператора

Все эти символьные операторы работают аналогично соответствующим операторам Java, поэтому их использование должно быть понятным любому разработчику Java. В дополнение к тому, что нам разрешается использовать символьные операторы в Унифицированном языке выражений, у всех символьных операторов имеются текстовые эквиваленты. Эти текстовые эквиваленты используются, если нам нужна синтаксически допустимая XML-страница, поскольку использование символьных операторов обычно приводит к возникновению недопустимой XML-разметки.

В дополнение к операторам отношения в выражении Унифицированного языка выражений также могут использоваться логические операторы. Допустимые логические операторы перечислены в следующей таблице:

Логические операторы	Описание
&& или and	<ul style="list-style-type: none"> И: истинно, если оба выражения и слева, и справа от оператора имеют значение <code>true</code>
или or	<ul style="list-style-type: none"> Или: истинно, если является истинным либо выражение слева от оператора, либо выражение справа от оператора, либо они оба
! или not	<ul style="list-style-type: none"> Не: отрицает выражение справа от оператора. Если выражение истинно, этот оператор делает его оценку ложной, и наоборот.
empty	<ul style="list-style-type: none"> Пустой: истинно, если значение справа от оператора является нулевым или пустым. Значение справа от оператора должно быть строкой или коллекцией
E1?E2:E3	<ul style="list-style-type: none"> Условное выражение: если выражение <code>E1 = true</code>, оценивается выражение <code>E2</code>; в противном случае оценивается выражение <code>E3</code>

Точно так же, как и операторы отношения, логические операторы работают аналогично их эквивалентам в Java. У каждого из них, кроме тройного оператора (условного выражения) и `empty`, имеются и символьная, и текстовая разновидности.

Унифицированный язык запросов также содержит арифметические операторы. Они перечислены в следующей таблице:

Арифметические операторы	Описание
+	<ul style="list-style-type: none"> Сложение: складывает значения слева и справа от оператора
- (бинарный)	<ul style="list-style-type: none"> Вычитание: вычитает значение справа от оператора из значения слева от оператора
*	<ul style="list-style-type: none"> Умножение: перемножает значения слева и справа от оператора
/ или div	<ul style="list-style-type: none"> Деление: делит значение слева от оператора (делимое) на значение справа от оператора (делитель)
% или mod	<ul style="list-style-type: none"> Деление по модулю: делит значение слева от оператора (делимое) на значение справа от оператора (делитель) и возвращает остаток
- (унарный)	<ul style="list-style-type: none"> Минус: умножает значение справа от оператора на <code>-1</code>

Все арифметические операторы должны использоваться с числовыми значениями.

После краткого обсуждения операторов Унифициированного языка выражений мы можем вернуться к обсуждению примера.

Следующий новый тег, который мы видим в примере, – `<c:forEach>`. Этот тег выполняет итерации по коллекциям (`collection`), массивам (`array`) или картам (`map`). В примере он выполняет итерации по экземплярам `java.util.ArrayList`, присоединенным к запросу в скриптлете, определенном ранее на странице. Атрибут `var` тела `<c:forEach>` определяет переменную, которая будет использоваться для получения доступа к текущему элементу в коллекции. Эта переменная видима только в теле тела. Атрибут `items` тела `<c:forEach>` указывает массив, коллекцию или карту, по которым выполняется итерация. У тела `<c:forEach>` имеются дополнительные атрибуты, которые не показаны в примере: атрибут `begin` указывает индекс первого элемента, с которого начинается выполнение итерации, и атрибут `end`, указывающий последний элемент, по которому будет выполнена итерация. Если атрибут `begin` не устанавливается, итерация начинается в первом элементе коллекции (`collection`), массива (`array`) или карты (`map`). Если не устанавливается атрибут `end`, итерация заканчивается в последнем элементе коллекции, массива или карты. Дополнительным атрибутом тела `<c:forEach>` является атрибут `step`. Он указывает «размер шага», на который увеличивается индекс при переходе от одного индекса к следующему. Его значением по умолчанию является 1. В дополнение к итерации по коллекции, массиву или карте тег `<c:forEach>` может использоваться для неоднократного выполнения его тела. В таком случае атрибут тела `<c:forEach>` `items` исключается, а его атрибуты `begin` и `end` добавляются.

Следующим новым тегом, который мы видим в примере, является тег `<c:remove>`. Он используется, чтобы удалить переменную, присоединенную к контексту, указанному в его атрибуте `scope`. Если никакой контекст не будет указан, тег `<c:remove>` по умолчанию использует контекст страницы.

Имеются некоторые дополнительные базовые теги JSTL, не показанные в примере. Эти оставшиеся теги будут объяснены далее.

Тег `<c:if>` аналогичен тегу `<c:when>`; его тело выполняется, если выражение, определенное его атрибутом `test`, является истиной. У тела `<c:if>` есть два дополнительных атрибута: атрибут `var`, который определяет имя логической (`Boolean`) переменной, хранящей результаты атрибута `test` тела, и атрибут `scope`, определяющий контекст атрибута `var`. Тег `<c:if>` не должен быть вложен в тег `<c:choose>`. В отличие от тега `<c:when>`, выражение, определенное в атрибуте `test` нескольких тегов `<c:if>`, оценивается независимо от того, оценено ли предыдущее выражение `<c:if>` как имеющее значение `true` (истина) или нет.

Тег `<c:forTokens>` выполняет итерации по элементам строки, отделенным друг от друга разделителями. У тела `<c:forTokens>` есть два обязательных атрибута: `items` и `delims`. Значением атрибута `items` должно быть разрешенное выражение в виде строки (`String`) или строковой константы. Значение атрибута `delims` должно быть выражением или строковой константой, указывающей символы, которые будут использоваться в качестве разделителей. Каждый отдельный символ в атрибуте `delims` будет использоваться в качестве разделителя для значения элемента,

подобно тому как работает класс `java.util.StringTokenizer`. Дополнительно у тега `<c:forTokens>` имеется атрибут `var`, который работает, по существу, таким же образом, как атрибут `var` тега `<c:forEach>`. Иными словами, он определяет имя для текущего элемента в его атрибуте `items`, что позволяет получить к нему доступ в теле тега `<c:forTokens>`.

Тег `<c:import>` аналогичен `<jsp:include>`. Он включает содержимое относительного или абсолютного URL в отображаемую JSP-страницу. При необходимости этот тег может хранить содержимое включаемого URL в строке (`String`) или в экземпляре `java.io.Reader`. У тега `<c:import>` есть один обязательный атрибут, называемый `url`; значение этого атрибута является строковым выражением, содержащим URL, который будет импортирован. Если мы хотим хранить содержимое включаемого URL в строке, то должен использоваться атрибут `var`. Значением этого атрибута является имя строки, которая будет содержать значение включаемого URL. Если мы хотим хранить значение включаемого URL в экземпляре `java.io.Reader`, для этого должен использоваться атрибут `varReader`. Значением данного атрибута является имя переменной, которая будет содержать значение включаемого (импортируемого) URL. У тега `<c:import>` есть дополнительный атрибут контекста, который определяет контекст переменной, определенной атрибутами `varReader` или `var`. Если этот атрибут не будет использоваться, то у переменной `var` или `varReader` контекстом по умолчанию будет контекст страницы.

Тег `<c:redirect>` перенаправляет обозреватель к URL, указанному в его атрибуте `url`. Он эквивалентен вызову метода `sendRedirect()` экземпляра `javax.servlet.http.HttpServletResponse`.

Тег `<c:url>` создает URL из значения его атрибута `url` и сохраняет его в строке, имя которой определяется атрибутом `var` тега. Контекстом переменной, определенной атрибутом `var`, по умолчанию является страница. Контекст может быть изменен путем использования атрибута `scope` тега.

Можно передать параметры к URL, определенному в атрибуте `url` тегов `<c:import>`, `<c:redirect>` или `<c:url>`. Это делается путем использования тега `<c:param>`. Он должен быть вложен в один из вышеупомянутых трех тегов. У тега `<c:param>` есть два атрибута: обязательный атрибут `name`, определяющий название параметра, и атрибут `value`, определяющий значение параметра.

Последним базовым тегом JSTL является тег `<c:catch>`. Он перехватывает любое исключение `java.lang.Throwable`, возникшее в его теле.



`java.lang.Throwable` является родительским классом для классов `java.lang.Exception` и `java.lang.Error`. Поэтому любое исключение (`Exception`) или ошибка (`Error`), возникшие в теле тега `<c:catch>`, также будут перехвачены.

Если прерывание (`Throwable`) возникает в теле тега `<c:catch>`, управление будет передано в строку, следующую непосредственно за закрытием тега `</c:catch>`. Будут обработаны любые строки в теле тега `<c:catch>`, которые оказались обработанными до возникновения прерывания. У тега `<c:catch>` имеется единственный дополнительный атрибут, называемый `var`. Этот атрибут определяет переменную,

содержащую прерывание, которое возникло в теле тега `<c:catch>`. Эта переменная всегда имеет контекст страницы.

В следующей таблице приведены все теги библиотеки базовых тегов JSTL:

Тег	Описание	Пример
<code><c:catch></code>	Перехватывает любые исключения <code>Error</code> или <code>Throwable</code> , возникшие внутри его тела	<pre><c:catch var="e"> <c:out value="1/0"/> <c:if test="e!=null"> <c:out value= "e.message"/> </c:if> </c:catch></pre>
<code><c:choose></code>	Используется для «обертки» тега <code><c:when></code> и (необязательно) тега <code><c:otherwise></code> . Выполняется тело первого тела <code><c:when></code> , содержащее тестовое выражение, значение которого оценено как истинное (<code>true</code>). Если ни один из тегов <code><c:when></code> , содержащих тестовое выражение, не будет оценен как имеющий значение <code>true</code> , то выполнится тело тега <code><c:otherwise></code>	<pre><c:choose> <c:when test="empty o"> <c:out value="o is empty"/> </c:when> <c:otherwise> <c:out value="o is not empty"/> </c:otherwise> </c:choose></pre>
<code><c:forEach></code>	Выполняет итерацию по массиву, коллекции или карте	<pre><c:forEach items="\${session.array OrCollection}" var="item"> <c:out value="item" = \${item}>
 </c:forEach></pre>
<code><c:if></code>	Его тело выполняется в том случае, если тестовое выражение (атрибут <code>test</code>) оценивается как истинное	<pre><c:if test="\${a>b}"> <c:out value="a is greater than b"/> </c:if></pre>
<code><c:import></code>	Импортирует содержимое из URL, указанного в атрибуте <code>url</code> , в отображаемую страницу	<pre><c:import url="http://foo.com/ somePage.jsp"> <c:param name="someName" value="some val"/> </c:import></pre>
<code><c:out></code>	Выводит значение выражения <code>value</code>	<pre><c:out value="> is the greater than symbol" escapeXml="true"/></pre>
<code><c:otherwise></code>	Его тело выполняется, если ни одно из тестовых выражений тегов <code><c:when></code> , вложенных в один и тот же тег <code><c:choose></code> , не оценивается как истинное	См. пример для <code><c:otherwise></code>

Тег	Описание	Пример
<c:param>	Устанавливает параметр для URL, определенного в тегах <c:url> или <c:import>	См. пример для <c:import>
<c:redirect>	Перенаправляет на указанный URL	<c:redirect url="http://ensode.net"/>
<c:remove>	Удаляет переменную из контекста страницы или указанного контекста	<c:remove var="varName" scope="session"/>
<c:set>	Устанавливает переменную в контексте страницы или указанном контексте	<c:set var="varName" value="foo" scope="session"/>
<c:url>	Создает переменную URL	<c:url value="http://foo.com" var="fooUrl"/>
<c:when>	Его тело выполняется в случае, когда его тестовое выражение оценивается как истинное	См. пример для <c:choose>

JSTL-библиотека тегов форматирования

Библиотека *тегов форматирования JSTL* (Formatting JSTL tags) предоставляет теги, которые упрощают интернационализацию и локализацию веб-приложений. Эта библиотека тегов позволяет выводить на экран веб-страницу на различных языках, на основе локали пользователя. Она также позволяет выполнять специфическое для локали форматирование даты и валюты.

Пример ниже поясняет использование библиотеки тегов форматирования JSTL:

```

<%@ page language="java" contentType="text/html; charset=UTF-8"
       pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
          "http://www.w3.org/TR/html4/loose.dtd">

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Демонстрация тегов форматирования</title>
  </head>
  <body>
    <jsp:useBean id="today" class="java.util.Date"/>
    <fmt:setLocale value="en_US"/>
    <fmt:bundle basename="ApplicationResources">
      <fmt:message key="greeting"/>,<br/>
      <fmt:message key="proposal"/>
      <fmt:formatNumber type="currency" value="42000"/>.<br/>
      <fmt:message key="offer_ends"/>
      <fmt:formatDate value="${today}" type="date" dateStyle="full"/>.
    </fmt:bundle>
    <br/>

```

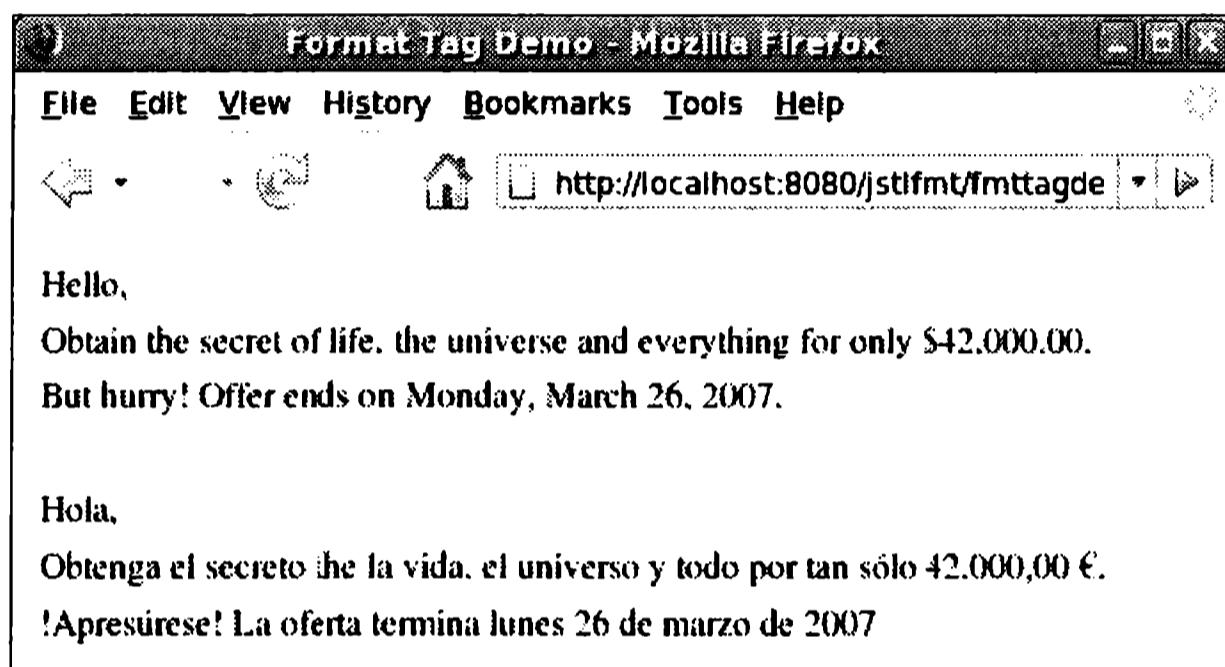


```

<br/>
<fmt:setLocale value="es_ES"/>
<fmt:bundle basename="ApplicationResources">
    <fmt:message key="greeting"/>,<br/>
    <fmt:message key="proposal"/>
    <fmt:formatNumber type="currency" value="42000"/>.<br/>
    <fmt:message key="offer_ends"/>
    <fmt:formatDate value="${today}" type="date" dateStyle="full"/>
</fmt:bundle>
</body>
</html>

```

Эта страница прежде всего отображает приветствие пользователю, а затем коммерческое предложение (коммерческую цель) с указанием цены и даты, до которой действительно данное предложение:



Поскольку эта страница интернационализирована, фактический текст страницы сохранен в файле свойств, названном *пакетом ресурса*. Пакет ресурса для страницы называется ApplicationResources.properties. Он устанавливается на странице с помощью тега <fmt:bundle>.

Поскольку страница отображает одно и то же сообщение на английском и испанском языках, необходимы два пакета ресурса – по одному для каждой локали. Используемая локаль определяется в атрибуте value тега <fmt:setLocale>.



Реальное приложение одновременно не вывело бы на экран одно и то же сообщение на двух различных языках. Вместо этого оно обнаружило бы локаль пользователя из запроса и использовало бы соответствующий пакет ресурса. Если локаль пользователя не соответствует ни одному из доступных пакетов ресурса, то используется значение по умолчанию.

Английская (по умолчанию) версия ApplicationResources.properties выглядит следующим образом:

```

greeting=Hello
proposal=Obtain the secret of life, the universe and
everything for only
offer_ends=But hurry! Offer ends on

```

Испанская версия пакета ресурса называется ApplicationResources_es_ES.properties и содержит следующий код:

```
greeting=Hola  
proposal=Obtenga el secreto de la vida, el universo y todo por tan  
sólo  
offer_ends=!Apresúrese! La oferta termina
```

Как видно из приведенного кода, пакет ресурса является всего лишь файлом свойств с ключами и значениями. Ключи для каждого локализованного пакета ресурса должны быть одинаковыми; значение же должно изменяться согласно локали. Чтобы быть доступными для JSP-страниц и Java-кода, пакеты ресурса должны быть помещены в любой из каталогов WEB-INF/classes или в любой из его подкаталогов в WAR-файле, в котором приложение развертывается. Если они помещаются в подкаталог каталога WEB-INF/classes, то атрибут basename тега <fmt:bundle> должен включать полный путь, содержащий все подкаталоги этого каталога, разделенные точками. Например, если бы ApplicationResources.properties и ApplicationResources_es_ES.properties были помещены в каталог WEB-INF/classes/net/ensode, то тег <fmt:bundle> выглядел бы следующим образом:

```
<fmt:bundle basename="net.ensode.ApplicationResources">
```

Как видно из приведенной строки, это похоже на полностью определенное (квалифицированное) имя класса, но в действительности мы указываем на пакет ресурса.

У имени пакета ресурса для каждой локали должно быть одно и тоже базовое имя, как и у основного пакета ресурса (в нашем случае ApplicationProperties); затем следуют подчеркивание и соответствующая локаль (в нашем случае es_ES). Локаль может указывать только язык (например, en или es) либо язык и страну (например, en_US или es_ES). Если в локали не указана ни одна страна, то любая страна, основной язык которой соответствует локали, будет использовать пакет ресурса для этого языка.

 Предыдущий пример использует es_ES в качестве локали, предполагая, что каждая страница на испанском языке предназначена для Испании. Очевидно, что это не будет работать в реальных приложениях и представлено таким образом для простоты.

Тег <fmt:message> ищет ключ в пакете ресурса, соответствующем его атрибуту key, и выводит на экран его значение на странице. Хотя это и не показано в примере, иногда значения элементов пакета ресурсов могут иметь параметры, которые заменяются во время работы соответствующими значениями. Параметры обозначаются целым числом между фигурными скобками. Это поясняется в следующем примере:

```
personalGreeting>Hello {0}
```

В этом свойстве {0} является параметром. Параметры можно заменить соответствующими значениями во время выполнения путем использования тега <fmt:param>. Этот тег должен быть вложен в тег <fmt:message>. У тега <fmt:param> есть атрибут value; значение этого атрибута может быть строковой константой или выражением Унифицированного языка выражений. Оно используется для замены параметра этим значением. У значений пакета ресурса может быть более одного параметра; в этом случае количество тегов <fmt:param>, вложенных внутрь

`<fmt:message>`, должно соответствовать числу параметров. Порядок определения тегов `<fmt:param>` показывает, какой параметр получает замену каким тегом. Первый тег `<fmt:param>` заменит параметр, индексированный как 0, второй тег `<fmt:param>` – параметр, индексированный как 1, и т. д.

Следующим тегом форматирования, который мы видим в примере, является тег `<fmt:formatNumber>`. Он форматирует число согласно локали. Некоторые локали используют запятые для разделения тысяч и точку в качестве десятичного разделителя, в то время как для других принято обратное. Как видно из предыдущего снимка экрана, тег `<fmt:formatNumber>` позаботится об этом за нас. Другим полезным атрибутом тега `<fmt:formatNumber>` является атрибут `type`. Для этого атрибута существуют три допустимых значения: `number`, `percent` или `currency`. Как видно из примера, если атрибут `type` устанавливается в значение `currency`, то соответствующее обозначение денежной единицы для локали автоматически добавляется к числу.

Следующий новый тег форматирования из нашего примера – `<fmt:formatDate>`. Этот тег получает объект `Date`, указанный его атрибутом `value`, и соответствующим образом форматирует его для данной локали. В дополнение к преобразованию даты для соответствующего языка этот тег поместит день недели, день месяца, месяц и год в предопределенное для соответствующей локали место. Он также будет использовать корректное написание прописными буквами для первой буквы месяца. У атрибута `dateStyle` тега `<fmt:formatDate>` имеются следующие допустимые значения: `full`, `long`, `medium`, `short` и `default`. Если значение не указано явно, то используется `default`.

Мы обсудили наиболее часто используемые теги библиотеки тегов форматирования. В следующей таблице приведены все теги библиотеки JSTL-тегов форматирования:

Тег	Описание	Пример
<code><fmt:bundle></code>	Загружает пакет ресурсов, который будет использоваться внутри его тела	<code><fmt:bundle basename="resbund"> <fmt:message key="greeting"> </fmt:bundle></code>
<code><fmt:formatDate></code>	Форматирует дату, указанную в качестве значения его атрибута, при необходимости используя указанный шаблон	<code><fmt:formatDate value="\${today}" pattern="MM/dd/yyyy"/></code>
<code><fmt:formatNumber></code>	Форматирует число, указанное в качестве значения его атрибута, в соответствии с текущей локалью. Может быть использован для форматирования числа в качестве валюты или процента, в зависимости от значения его дополнительного атрибута типа	<code><fmt:formatNumber value="42000"/></code>

Тег	Описание	Пример
<fmt:message>	Отображает локализованное сообщение, соответствующее определенному ключу в атрибуте ключа	<fmt:message key="offer_ends"/>
<fmt:param>	Заменяет параметр, заключенный в тег <fmt:message>	<fmt:param value="someVal"/>
<fmt:parseDate>	Выполняет разбор строки, содержащей дату, и преобразует результат в объект Date	<fmt:parseDate value="03/31/2007" pattern="MM/dd/yyyy" var="parsedDate"/>
<fmt:parseNumber>	Выполняет разбор числовой строки и преобразует результат в объект Long или Double	<fmt:parseNumber value="42,000.00" var="parsedNumber"/>
<fmt:requestEncoding>	Устанавливает кодировку запроса.	<fmt:requestEncoding key="ISO-8859-1"/>
<fmt:setBundle>	Устанавливает пакет ресурса для указанного контекста. Контекстом по умолчанию является страница	<fmt:setBundle basename="resbund" var="bundle" scope="session"/>
<fmt:setLocale>	Устанавливает локаль для использования в указанном контексте. Контекстом по умолчанию является страница	<fmt:setLocale value="en_US" />
<fmt:setTimeZone>	Устанавливает часовой пояс для использования в указанном контексте. Контекстом по умолчанию является страница	<fmt:setTimeZone value="EST" var="sessionTimeZone" scope="session"/>
<fmt:timezone>	Устанавливает часовой пояс для использования его внутри тела тега	<fmt:timezone value="EST"> <fmt:formatDate value="\${today}" /> </fmt:timezone>

JSTL-библиотека SQL-тегов

JSTL-библиотека SQL-тегов (SQL JSTL tag) позволяет нам выполнять SQL-запросы из JSP-страницы. В данной библиотеке тегов смешиваются элементы презентации и код доступа к базе данных. Эта библиотека должна использоваться только для создания прототипов и написания простых «одноразовых» приложений. Для более сложных приложений правилом хорошего тона будет следование шаблонам проектирования DAO и MVC.

Следующий пример поясняет обычно используемые теги из JSTL-библиотеки SQL-тегов:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
           pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
           "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Демонстрация SQL-тегов</title>
    </head>
    <body>
        <sql:setDataSource dataSource="jdbc/__CustomerDBPool"/>
        <sql:transaction>
            <sql:update>
                insert into CUSTOMERS (CUSTOMER_ID, FIRST_NAME, LAST_NAME)
                values ((select max(CUSTOMER_ID) from customers) + 1), ?, ?
                <sql:param value="${param.firstName}" />
                <sql:param value="${param.lastName}" />
            </sql:update>
        </sql:transaction>
        <p>В таблицу CUSTOMERS успешно вставлены следующие строки:</p>
        <sql:query
            var="selectedRows"
            sql="select FIRST_NAME, LAST_NAME from customers where
                  FIRST_NAME = ? and LAST_NAME = ?">
            <sql:param value="${param.firstName}" />
            <sql:param value="${param.lastName}" />
        </sql:query>
        <table border="1" cellpadding="0" cellspacing="0">
            <tr>
                <td>Имя</td>
                <td>Фамилия</td>
            </tr>
            <c:forEach var="currentRow" items="${selectedRows.rows}">
                <tr>
                    <td><c:out value="${currentRow.FIRST_NAME}" /></td>
                    <td><c:out value="${currentRow.LAST_NAME}" /></td>
                </tr>
            </c:forEach>
        </table>
    </body>
</html>
```

После упаковки этой JSP-страницы в WAR-файле, развертывания WAR-файла и указания в адресной строке обозревателя URL (с передачей параметров, которые ожидает страница) JSP, мы должны увидеть примерно такой результат:



Как и большинство наших примеров, предыдущая страница существенно упрощена и не обязательно выражает то, что было бы сделано в реальном приложении. Страница вставляет строку в таблицу CUSTOMERS (Заказчики), а затем запрашивает строки таблицы, соответствующие вставленным значениям. Реальное приложение (имейте в виду, что библиотекой SQL-тегов нужно пользоваться только для очень простых приложений!) обычно вставляло бы значения, полученные из параметров запроса, в базу данных. Маловероятен и тот факт, что та же самая страница будет выполнять запрос в базу данных для извлечения оттуда только что вставленных данных. Это, скорее всего, было бы сделано на отдельной странице.

Первый SQL-тег JSTL, который мы видим в примере, – `<sql:setDataSource>`. Этот тег устанавливает источник данных, который будет использоваться для доступа к базе данных. Источник данных может быть получен либо через JNDI при использовании его JNDI-имени в качестве значения атрибута `datasource` этого тега, либо путем указания URL JDBC, имени пользователя и пароля с помощью атрибутов `url`, `user` и `password`. В предыдущем примере используется первый подход. Для того чтобы этот подход работал правильно, элемент `<resource-ref>` должен быть добавлен к файлу дескриптора развертывания приложения `web.xml`.

```

<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="2.5"
          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_2_5.xsd" >
    <resource-ref>
        <res-ref-name>jdbc/_CustomerDBPool</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
</web-app>

```

Подэлемент `<res-ref-name>` элемента `<resource-ref>` содержит JNDI-имя источника данных JDBC. Источник данных должен быть создан на сервере приложений. Процедура создания источника данных JDBC подробно объясняется в следующей главе.

Подэлемент `<res-type>` элемента `<resource-ref>` содержит полностью определенное (квалифицированное) имя ресурса, который будет получен через JNDI. Для источников данных это всегда будет `javax.sql.DataSource`.

У подэлемента `<res-auth>` элемента `<resource-ref>` должно быть значение `Container` при использовании элемента `<resource-ref>` для определения источника данных в качестве ресурса. Это позволяет серверу приложений использовать учетные данные для входа в базу данных, соответствующие источнику данных, созданному в пуле соединений.



Исключение: «Отсутствует подходящий SQL-драйвер»

Иногда использование тега `<sql:setDataSource>` будет иметь результатом исключение `java.sql.SQLException`: «Отсутствует подходящий SQL-драйвер» при использовании его атрибута `datasource`, определяющего местоположение источника данных с помощью JNDI. Обычно это означает, что мы забыли модифицировать файл дескриптора развертывания приложения `web.xml`, как описано выше.

Как упоминалось выше, альтернативным способом использования тега `<sql:setDataSource>` является указание URL соединения с базой данных и учетных данных пользователя. Если бы мы использовали этот подход в предыдущем примере, тег `<sql:setDataSource>` выглядел бы следующим образом:

```
<sql:setDataSource
    url="jdbc:derby://localhost:1527/customerdb"
    user="dev"
    password="dev"/>
```

Используемые атрибуты очевидны. Атрибут `url` должен содержать URL соединения JDBC. Атрибуты `user` и `password` должны содержать соответственно имя и пароль пользователя, используемые для входа в базу данных.

Следующий SQL-тег JSTL в нашем примере – `<sql:transaction>`. Неудивительно, что этот тег является оберткой для любых тегов `<sql:query>` и `<sql:update>`, которые он содержит в транзакции.

Затем мы видим тег `<sql:update>`, используемый для выполнения любых запросов, которые модифицируют данные в базе данных. Он может использоваться для выполнения SQL-операторов `INSERT`, `UPDATE` или `DELETE`. Как видно из примера, у запросов в этом теге могут быть один или несколько параметров. Точно так же, как при использовании Подготовленных операторов JDBC (JDBC Prepared Statements), вопросительные знаки используются здесь в качестве местозаполнителей для параметров. Тег `<sql:param>` используется для установки значения любого параметра в запросе, определяемого тегом `<sql:update>` или `<sql:query>`. Тег `<sql:param>` устанавливает значение для содержащего его тега через свой атрибут `value`, который может содержать строковую константу или выражение на Унифицированном языке выражений.

Тег `<sql:query>` используется для запроса данных из базы данных через оператор `SELECT`. Набор результатов запроса сохраняется в переменной, определенной атрибутом `var` этого тега. По умолчанию атрибут `var` имеет контекст страницы. Это может быть изменено путем использования атрибута `scope` тега `<sql:query>` – установкой его значения в соответствующий контекст (`page`, `request`, `session` или `application`). Как видно из примера, мы можем выполнить итерацию по переменной, определенной атрибутом `var` этого тега, при использовании тега `<c:forEach>`.

В следующей таблице приведены все SQL-теги JSTL:

Тег	Описание	Пример
<sql:dateParam>	Устанавливает значение для параметров данных в теге <sql:query> или <sql:update>	См. пример для <sql:query>
<sql:param>	Устанавливает значение для текстового или числового параметра в теге <sql:query> или <sql:update>	См. пример для <sql:update>
<sql:query>	Выполняет SQL-запрос, определенный в атрибуте sql и, возможно, присоединяет полученный результатирующий набор к указанному контексту, используя указанное имя переменной	<pre><sql:query sql="select * from table where last_update < ?" var="selectedRows"> <sql:dateParam value="\${someDate}" /> </sql:query></pre>
<sql:setDataSource>	Определяет источник данных, который будет использоваться в указанном контексте. Если контекст не указан явно, то контекстом по умолчанию является страница. Источник данных может быть получен через JNDI-поиск или путем указания URL JDBC через атрибуты url, а также имя и пароль пользователя	<pre><sql:setDataSource dataSource="jdbc/ CustomerDBPool"/></pre>
<sql:transaction>	Обертывает любые теги <sql:query> и <sql:update>, помещая их внутрь тела транзакции	<pre><sql:transaction> <sql:update sql="update table set some_col = ?"> <sql:param value="someValue" /> </sql:update> <sql:update sql="update table2 set some_col = ?"> <sql:param value="someValue" /> </sql:update> </sql:transaction></pre>
<sql:update>	Выполняет SQL-операторы INSERT, UPDATE или DELETE	<pre><sql:update sql="update table set some_col = ?"> <sql:param value="someValue" /> </sql:update></pre>

JSTL-библиотека XML-тегов

JSTL-библиотека *XML-тегов* (XML JSTL tag) предоставляет легкий способ разбора (анализа) XML-документов и выполнения преобразований XML-документов с использованием Трансформаций XSL (eXtensible Stylesheet Language Transformations (XSLT))¹. Эта библиотека тегов использует выражения XPath для перемещения по элементам в XML-документе.



XPath – язык выражений, используемый для обнаружения информации в XML-документе или для того, чтобы выполнить вычисления на основе содержимого XML-документа. Для получения дополнительной информации о XPath обратитесь к следующему ресурсу: <http://www.w3.org/TR/xpath/>.

Нижеприведенный пример поясняет наиболее часто используемые теги XML из библиотеки тегов JSTL:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
           pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/xml" prefix="x" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
    <c:import url="customers.xml" var="xml"/>
    <x:parse doc="${xml}" var="doc"/>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
          "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Демонстрация XML-тегов</title>
    </head>
    <body>
        <table cellpadding="0" cellspacing="0" border="1">
            <tr>
                <td>Имя</td>
                <td>Фамилия</td>
                <td>Email</td>
            </tr>
            <x:forEach select="$doc/customers/customer">
                <tr>
                    <td>
                        <x:out select="firstName"/>
                    </td>
                    <td>
                        <x:out select="lastName"/>
                    </td>
                    <td>
                        <x:choose>
                            <x:when select="email">
                                <x:out select="email"/>
                            </x:when>
                            <x:otherwise>
                                <c:out value="N/A"/>
                            </x:otherwise>
                        </x:choose>
                    </td>
                </tr>
            </x:forEach>
        </table>
```

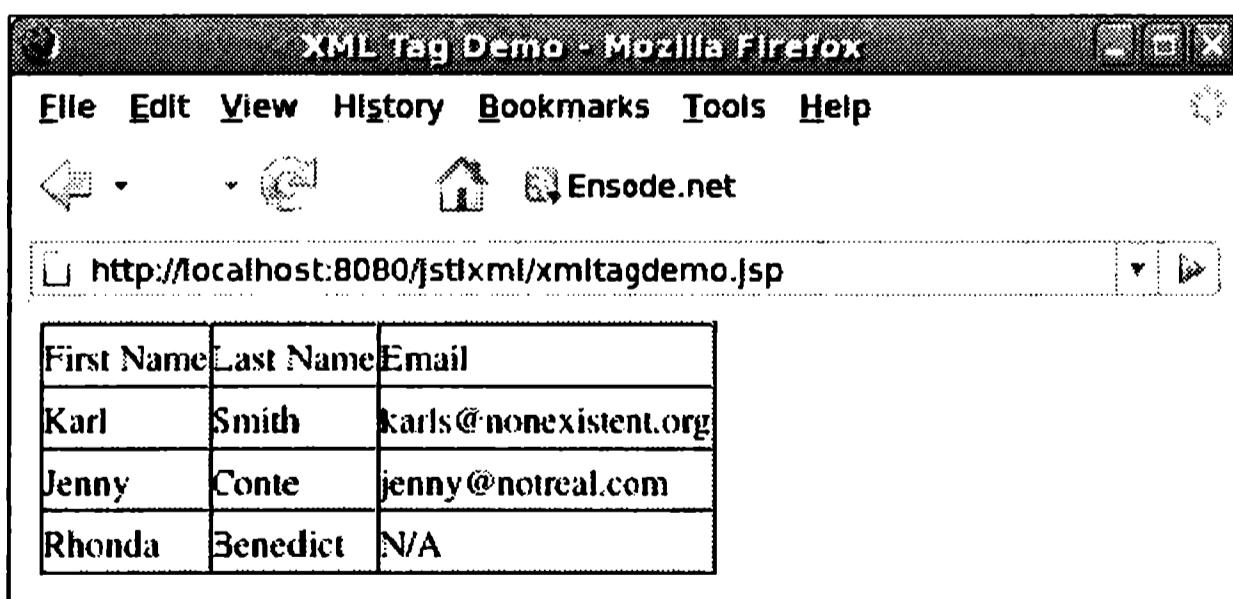
¹ См. http://www.w3schools.com/xsl/xsl_intro.asp. – Прим. перев.

```
</body>
</html>
```

Первое, что мы должны отметить в этом примере, – использование базового тега JSTL `<c:import>` для импорта XML-файла из URL. Значение атрибута URL определяет URL, где должен быть расположен XML-файл; это может быть относительный или абсолютный URL. В данном примере файл `customers.xml` находится в том же самом каталоге, что и JSP, поэтому используется относительный путь для его получения. Файл `customers.xml` содержит информацию о заказчиках, включая имя, фамилию и адрес электронной почты, как показано ниже:

```
<?xml version="1.0" encoding="UTF-8"?>
<customers>
  <customer>
    <firstName>Karl</firstName>
    <lastName>Smith</lastName>
    <email>karls@nonexistent.org</email>
  </customer>
  <customer>
    <firstName>Jenny</firstName>
    <lastName>Conte</lastName>
    <email>jenny@notreal.com</email>
  </customer>
  <customer>
    <firstName>Rhonda</firstName>
    <lastName>Benedict</lastName>
  </customer>
</customers>
```

После упаковки предыдущих двух файлов в WAR-файл, развертывания WAR-файла на сервере и посещения URL JSP-страницы мы должны увидеть в обозревателе примерно такой результат:



Первый XML-тег JSTL, который мы видим в примере, – это `<x:parse>`. Данный тег разбирает (анализирует) XML-документ и хранит его в переменной, определенной его атрибутом `var`. XML-документ для разбора определяется в атрибуте `doc` тега.

JSTL-библиотека XML-тегов содержит несколько тегов, которые походят на подобные теги в JSTL-библиотеке базовых тегов. К таковым относятся: `<x:if>`, `<x:choose>`, `<x:when>`, `<x:otherwise>`, `<x:forEach>`, `<x:param>` и `<x:set>`.

Использование этих тегов очень похоже на использование их «коллег» из библиотеки базовых тегов. Основное отличие этих тегов от их аналогов из библиотеки базовых тегов заключается в том, что они содержат атрибут `select`, содержащий выражение XPath для оценки, вместо атрибута `value`, который содержит соответствующие теги из библиотеки базовых тегов. Пример поясняет использование большинства из этих тегов.

Следующий XML-тег JSTL, который мы видим в примере, – `<x:forEach>`. Этот тег выполняет итерации по элементам XML-документа. Для выполнения итерации по элементам они указываются как выражение XPath с помощью атрибута `select`.

Далее в примере нам встречается XML-тег JSTL `<x:out>`, который выводит значение выражения XPath, определенного в его атрибуте `select`.

Затем мы видим тег `<x:choose>`, который является родительским для тегов `<x:when>` и (необязательно) `<x:otherwise>`. Тело первого вложенного тела `<x:when>` содержит выражение XPath для проверки на истинность выполнения выражения его атрибута `select`. Выражение `select` для последующих атрибутов `<x:when>` не оценивается после того, как выражение одного из них оценено как `true` (истина). Если никакие атрибуты `select` для какого-либо из тегов `<x:when>` не были оценены как истинные, выполняется тело необязательного тела `<x:otherwise>`.

Дополнительный XML-тег JSTL `<x:transform>` используется для XSLT-трансформаций XML-документов. Этот тег обычно используется с двумя атрибутами. Атрибут `xml` указывает расположение XML-документа для трансформации. Он может быть импортирован с помощью тела `<c:import>`, как показано в примере. Атрибут `xslt` указывает, что для трансформации документа используется таблица стилей XSL. Эта таблица стилей также может быть импортирована с помощью тела `<c:import>`.

В следующей таблице приведены все XML-теги JSTL:

Тег	Описание	Пример
<code><x:choose></code>	Используется для обертки тегов <code><x:when></code> и (необязательно) <code><x:otherwise></code> . Выполняется тело первого тела <code><x:when></code> , выражение <code>select</code> которого оценивается как истинное. Если ни один тег <code><x:when></code> , содержащий тестовое выражение, не будет оценен как истинный, то выполняется тело выражения <code><x:otherwise></code>	См. пример для <code><x:forEach></code>

Тег	Описание	Пример
<x:forEach>	Выполняет итерацию по элементам XML-документа. Элементы, по которым будет выполняться перебор, указываются с помощью атрибута select	<pre> <x:forEach select= "\$doc/customers/customer"> <tr> <td> <x:out select="firstName"/> </td> <td> <x:out select="lastName"/> </td> <td> <x:choose> <x:when select="email"> <x:out select="email"/> </x:when> <x:otherwise> <c:out value="N/A"/> </x:otherwise> </x:choose> </td> </tr> </x:forEach> </pre>
<x:otherwise>	Его тело выполняется, если ни одно из тестовых выражений в тегах <x:when>, вложенных в тег <x:choose>, не будет оценено как истинное	См. пример для <x:forEach>
<x:out>	Выводит выражение XPath, определенное в атрибуте select	См. пример для <x:forEach>
<x:param>	Добавляет параметр к содержимому тега <x:transform>	См. пример для <x:transform>
<x:parse>	Выполняет разбор XML-документа и сохраняет его в переменной, определенной атрибутом var	<pre> <x:parse doc="\${xml}" var="doc"/> </pre>
<x:set>	Сохраняет в переменной заданного контекста результат выражения XPath, определенного в его атрибуте select. Если контекст не определен, то используется контекст по умолчанию – контекст страницы	<pre> <x:set var="custEmail" select="email"/> </pre>

Тег	Описание	Пример
<x:transform>	Трансформирует XML-документ, определяемый атрибутом XML, с помощью стилей XSL, определяемых атрибутом XSLT	<pre><x:transform xml="\${someXmlDoc}" xslt="\${xslt}"> <x:param name="paramName" value="\${paramValue}" /> </x:transform></pre>
<x:when>	Его тело выполняется, когда выражение select оценивается как истинное	См. пример для <x:forEach>

Функции JSTL

JSTL содержит много функций, которые принимают выражения Унифицированного языка выражений в качестве параметров. Все функции JSTL, кроме одной, используются исключительно для манипуляции со строками. Исключением является функция `fn:length()`, которая может принимать в качестве параметра строку (`String`), коллекцию (`Collection`) или массив (`Array`). Она соответственно возвращает длину строки, размер коллекции или длину массива, в зависимости от того, какой параметр был ей передан. Следующий пример JSP-страницы поясняет использование функций JSTL:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Демонстрация тегов функций</title>
    </head>
    <body>
        <c:set var="nameArr" value="${fn:split('Kevin,Danielle,Alex,
            Beatrice','','')}" />
        У нас есть список ${fn:length(nameArr)} имен, среди которых:
        <br/>
        <ol>
            <c:forEach var="currentName" items="${nameArr}">
                <li>
                    ${fn:toUpperCase(currentName)}
                <br/>
            </c:forEach>
        </ol>
    </body>
</html>
```

После упаковки этой JSP-страницы в WAR-файл, его развертывания и указания URL этой страницы в адресной строке обозревателя мы должны будем увидеть примерно следующее:



Эта JSP-страница поясняет использование некоторых функций JSTL. Функция `fn:split()` разделяет строку (`String`) на массив строк, используя в качестве разделителя символ, указанный в ее втором параметре.



Обратите внимание, что строки в функции `fn:split()` заключены в апострофы. JSTL это допускает, поскольку использование двойных кавычек для строк вызвало бы ошибку -- «недопустимый синтаксис», в связи с тем что сама функция `fn:split()` и так уже заключена в двойные кавычки.

В нашем примере функция `fn:length()` возвращает число элементов в массиве, который мы создали при выполнении функции `fn:split()`. Как уже упоминалось ранее, функция `fn:length()` может также принимать в качестве параметра коллекцию или строку. При применении ее к коллекции функция возвращает число элементов в ней. При применении к строке функция возвращает число символов в строке.

Следующая функция, показанная в примере, – `fn:toUpperCase()`, просто преобразует каждый алфавитный символ в строке, которую она принимает в качестве параметра, к верхнему регистру раскладки клавиатуры. Имеется еще много других функций JSTL, но их назначение и использование интуитивно понятны.

В следующей таблице приведены все функции JSTL:

Функция	Описание	Пример
<code>fn:contains(String, String)</code>	Возвращает логическое значение, показывающее, содержится ли второй параметр в первом	<code> \${fn:contains("environment", "iron")}</code>
<code>fn:containsIgnoreCase(String, String)</code>	Не зависящая от регистра версия функции <code>fn:contains()</code>	<code> \${fn:containsIgnoreCase("environment", "Iron")}</code>
<code>fn:endsWith(String, String)</code>	Возвращает логическое значение, показывающее, заканчивается ли первый параметр строкой, содержащейся во втором параметре	<code> \${fn:endsWith("GlassFish", "Fish")}</code>

Функция	Описание	Пример
fn:escapeXml (String)	Возвращает в параметре строку со всеми XML-символами, взятыми в коде сущности соответствующего ей XML-символа	<code> \${fn:escapeXml ("<html>")}</code>
fn:indexOf (String, String)	Возвращает целое число с указанием индекса начала второго параметра в первом параметре. Возвращает -1, если второй параметр не является подстрокой первого параметра	<code> \${fn:endsWith ("GlassFish", "Fish")}</code>
fn:join (String[], String)	Возвращает строку, состоящую из элементов первого параметра, разделенных элементами второго параметра, используемыми в качестве разделителя	<code> \${fn:join(arrayVar, ", ")}</code>
fn:length (Object)	Возвращает длину массива, размер коллекции или длину строки – в зависимости от типа параметра	<code> \${fn:length("String, Collection ИЛИ Array")}</code>
fn:replace (String, String, String)	Возвращает строку, заменяя каждый экземпляр второго параметра в первом параметре третьим параметром	<code> \${fn:replace ("CrystalFish", "Crystal", "Glass")}</code>
fn:startsWith (String, String)	Возвращает логическое значение, показывающее, начинается ли первый параметр с содержимого второго параметра	<code> \${fn:startsWith ("GlassFish", "Glass")}</code>
fn:split (String, String)	Возвращает массив строк, содержащих элементы первого параметра, разделенные вторым параметром	<code> \${fn:split ("Eeny, meeny", ",")}</code>
fn:substring (String, int, int)	Возвращает строку, содержащую подстроку первого параметра, начиная с индекса, указанного вторым параметром, и заканчивая непосредственно перед индексом, указанным третьим параметром	<code> \${fn:substring ("0123456789", 3, 6)}</code>

Функция	Описание	Пример
fn:substringAfter (String, String)	Возвращает строку, содержащую подстроку первого параметра, начиная с места первого вхождения второго параметра и до конца первого параметра	<code> \${fn:substringAfter("GlassFish", "Glass")}</code>
fn:substringBefore (String, String)	Возвращает строку, содержащую подстроку первого параметра, начиная с начала первого параметра и заканчивая непосредственно перед первым появлением второго параметра	<code> \${fn:substringBefore("GlassFish", "Fish")}</code>
fn:toLowerCase(String)	Возвращает строку, содержащую версию параметра со всеми буквенными символами, приведенными к нижнему регистру	<code> \${fn:toLowerCase("GlassFish")}</code>
fn:toUpperCase(String)	Возвращает строку, содержащую версию параметра со всеми буквенными символами, приведенными к верхнему регистру	<code> \${fn:toUpperCase(" GlassFish ")}</code>
fn:trim(String)	Возвращает строку, содержащую модифицированную версию параметра со всеми удаленными пробельными символами в начале и конце строки	<code> \${fn:trim(" GlassFish ")}</code>

Резюме

В этой главе рассмотрены все теги Библиотеки стандартных тегов JSP (JSTL), включая базовые, форматирующие, SQL- и XML-теги. Дополнительно описаны функции JSTL. Были приведены примеры, поясняющие использование наиболее распространенных сочетаний тегов JSTL; также были упомянуты и описаны дополнительные сочетания JSTL-тегов.

5

Подключение к базе данных

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, Read, Update, Delete (Создание, Чтение, Обновление, Удаление)). Имеются два API Java EE, которые могут быть использованы для взаимодействия с реляционной базой данных: API Подключения к базе данных Java (Java Database Connectivity (JDBC)) и API Персистентности Java (Java Persistence API (JPA)). Оба они будут обсуждаться в этой главе.

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

- извлечение данных из базы данных посредством JDBC;
- вставка данных в базу данных посредством JDBC;
- обновление данных в базе данных посредством JDBC;
- удаление данных в базе данных посредством JDBC;
- извлечение данных из базы данных с помощью JPA;
- вставка данных в базу данных с помощью JPA;
- обновление данных в базе данных с помощью JPA;
- удаление данных в базе данных с помощью JPA;
- создание программных запросов с помощью API Критериев JPA 2.0;
- автоматизация проверки данных с помощью JPA 2.0 с поддержкой проверки со стороны бинов.

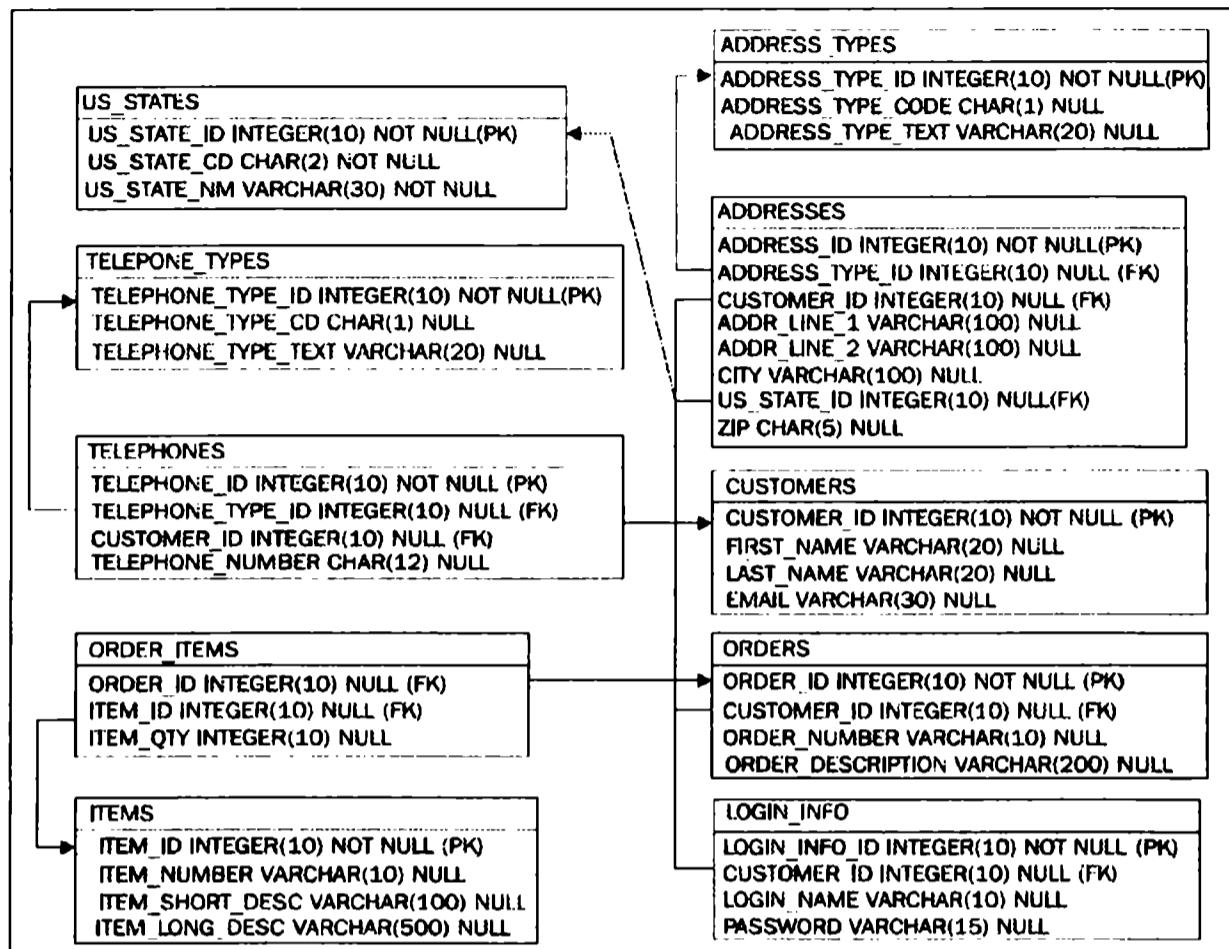
База данных CUSTOMERDB

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

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

добавлению пула соединений и источника данных для получения доступа к базе данных.

Схема базы данных CUSTOMERDB изображена на следующем рисунке:



Как видно из приведенного рисунка, база данных содержит таблицы для сохранения информации о клиентах, например такой, как имя, почтовый адрес и адрес электронной почты. Также в базу входят таблицы для сохранения информации о выставленных счетах и их содержимом.

Таблица ADDRESS_TYPES (типы адресов) хранит классификационные значения, такие как «Домашний», «Почтовый» и «Доставки», чтобы отличить тип адреса в таблице ADDRESSES. Аналогичным образом таблица TELEPHONE_TYPES (типы телефонов) хранит классификационные значения «Сотовый», «Домашний» и «Рабочий». При создании базы данных эти две таблицы заполняются предварительными данными, так же как таблица US_STATES.

 Для упрощения наша база данных имеет дело только с американскими адресами.

JDBC

API Подключения к базе данных Java (Java Database Connectivity (JDBC)) является стандартным API, используемым приложениями Java для взаимодействия с базой данных. Хотя JDBC не является частью спецификации Java EE, он используется очень часто в приложениях Java EE.

JDBC позволяет отправлять запросы к базе данных для выполнения операций выбора, вставки, обновления и удаления. Наиболее распространенный способ взаимодействия

с базой данных через JDBC – с помощью интерфейса `java.sql.PreparedStatement`. Использование *подготовленных операторов* (prepared statements) через этот интерфейс предоставляет много преимуществ по сравнению с использованием стандартных *объектов операторов* (statement objects) JDBC. Среди преимуществ подготовленных операторов, в частности, можно отметить следующие:

- подготовленные операторы компилируются в СУРБД при первом их выполнении и далее хранятся в откомпилированном виде, поэтому увеличивается производительность;
- они устойчивы к атакам с использованием инжекции SQL-кода;
- они освобождают нас от явного добавления апострофов (') к нашему SQL-оператору для обработки символьных значений.

У интерфейса `java.sql.PreparedStatement` имеются два метода, которые очень часто используются для отправки запросов к базе данных. Это `executeQuery()` и `executeUpdate()`. Метод `executeQuery()` используется для выполнения оператором `select` запроса к базе данных, и он возвращает экземпляр `java.sql.ResultSet`, содержащий строки, возвращаемые из запроса. Метод `executeUpdate()` используется для выдачи базе данных команд `insert`, `update` и `delete`. Он возвращает значение `int`, соответствующее числу строк, на которые повлиял запрос. В следующих разделах мы поясним взаимодействие с базой данных с помощью этих двух методов.

Извлечение данных из базы данных

Как мы упоминали в предыдущем разделе, метод `executeQuery()` интерфейса `java.sql.PreparedStatement` используется для отправки оператора `select` базе данных и получения данных от нее. Следующий пример кода поясняет этот процесс:

```
package net.ensode.glassfishbook.jdbcselect;

import java.io.IOException;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;
public class JDBCSelectServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException
    {
        String sql ="select us_state_nm, " +
            "us_state_cd from us_states order by us_state_nm";
        ArrayList<UsStateBean> stateList = new ArrayList<UsStateBean>();
        try
```

```
{  
    InitialContext initialContext = new InitialContext();  
    DataSource dataSource = (DataSource) initialContext.lookup(  
        "jdbc/_CustomerDBPool");  
    Connection connection = dataSource.getConnection();  
    PreparedStatement preparedStatement = connection.  
        prepareStatement(sql);  
    ResultSet resultSet = preparedStatement.executeQuery();  
    while (resultSet.next())  
    {  
        stateList.add(new UsStateBean(resultSet.getString("us_  
            state_nm"), resultSet.getString("us_state_cd")));  
    }  
    resultSet.close();  
    preparedStatement.close();  
    connection.close();  
    request.setAttribute("stateList", stateList);  
    request.getRequestDispatcher("us_states.jsp").forward(request,  
        response);  
}  
catch (NamingException namingException)  
{  
    namingException.printStackTrace();  
}  
catch (SQLException sqlException)  
{  
    sqlException.printStackTrace();  
}  
}  
}  
}
```

В этом сервлете мы создаем строку (`String`), содержащую оператор `select`, который мы отправим базе данных.

Затем мы создаем экземпляр `javax.naming.InitialContext`, который в дальнейшем используется для выполнения поиска *JNDI* (Java Naming and Directory Interface – Интерфейса именования и каталогов Java) интерфейса `javax.sql.DataSource`, соответствующего базе данных, к которой мы хотим подключиться. Это осуществляется вызовом метода `InitialContext.lookup()`. Строковый аргумент этого метода должен соответствовать имени источника данных, который мы установили в GlassFish (обратитесь к разделу «5. Подключение к базе данных» на странице 46). Этот метод возвращает экземпляр `java.lang.Object`. Возвращаемое им значение должно быть приведено к соответствующему типу (в нашем случае – `javax.sql.DataSource`).

После того как мы получим ссылку на объект `DataSource`, выполняя JNDI-поиск, мы можем получить соединение из пула соединений, вызывая метод `getConnection()`, определенный в интерфейсе `javax.sql.DataSource`. Этот метод возвращает экземпляр `java.sql.Connection`.

После получения соединения из пула соединений мы получим экземпляр класса, реализующего интерфейс `java.sql.PreparedStatement` для вызова метода `prepareStatement()` на экземпляре `java.util.Connection`, который мы получили на предыдущем шаге. Метод `prepareStatement()` принимает строку, содержащую SQL-запрос в качестве единственного аргумента.

После получения экземпляра класса, реализующего интерфейс `java.sql.PreparedStatement`, мы сможем наконец обратиться с запросом в базу данных, вызывая ее метод `executeQuery()`. Метод `PreparedStatement.executeQuery()` возвращает экземпляр класса, реализующего интерфейс `java.sql.ResultSet`. Этот экземпляр будет содержать результаты нашего запроса.

После этого сервер выполняет итерации по набору результатов и заполняет `ArrayList` экземплярами JavaBean типа `net.ensode.glassfishbook.jdbcselect.UsStateBean`.

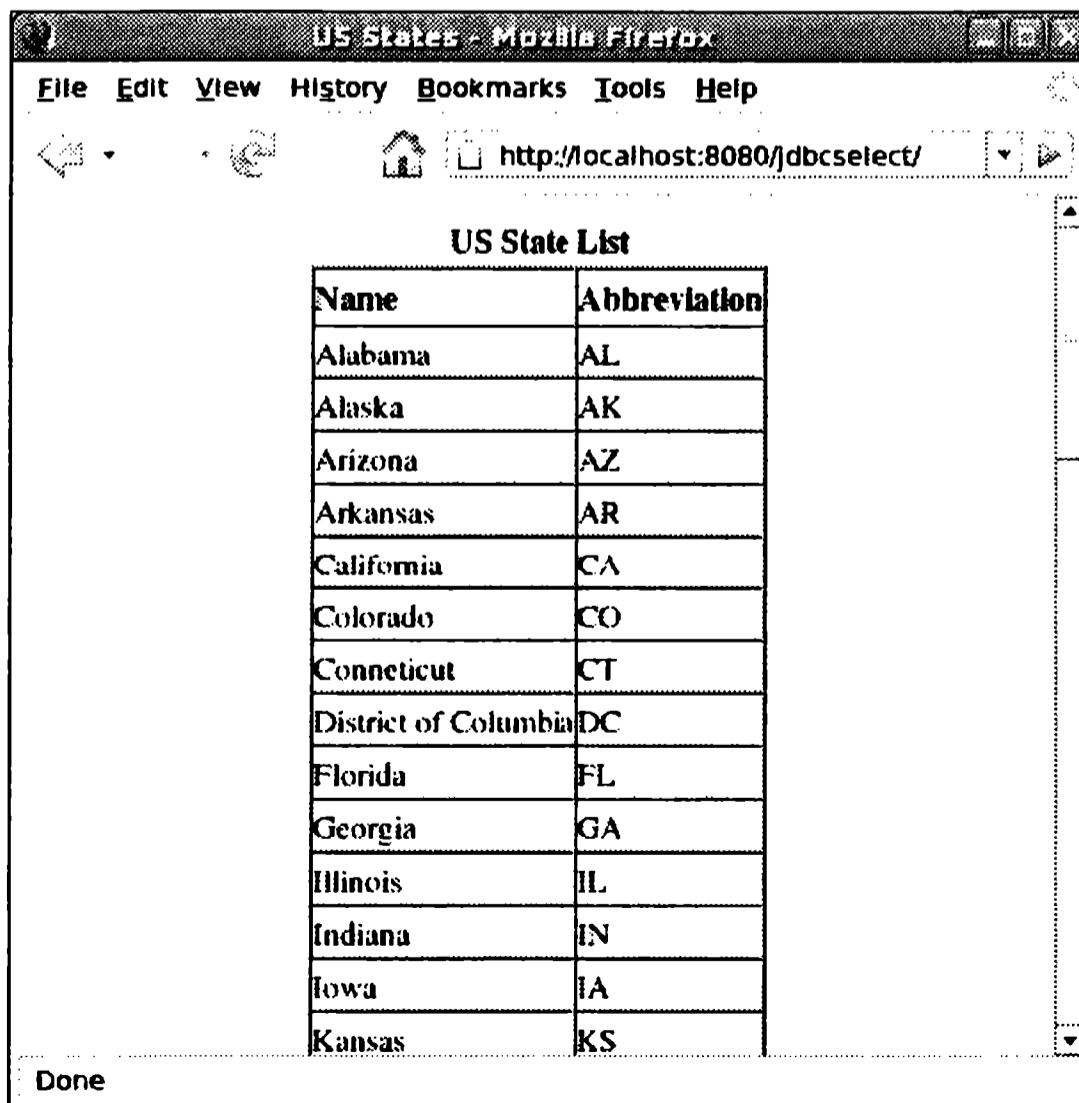
Наконец, мы закрываем набор результатов и подготовленный оператор, вызывая их методы `close()`. Соединение будет возвращено назад в пул соединений при вызове метода `close()` на экземпляре `java.sql.Connection`, который мы использовали.

 Вызов метода `close()` на соединении фактически не закрывает соединение; оно передается назад в пул соединений, так чтобы другие приложения могли его использовать.

Ранее заполненный `ArrayList` далее присоединяется к запросу, и запрос передается JSP-странице, названной `us_states.jsp`.

 Для краткости исходный код для `UsStateBean.java` и `us_states.jsp` не показан, поскольку эти файлы не поясняют ничего нового. Оба файла являются частью загрузки кода для этой книги.

После упаковки кода в WAR-файле, его развертывания и указания в адресной строке обозревателя соответствующего URL мы должны будем увидеть в окне обозревателя следующую страницу:



Все американские штаты, выведенные на экран на странице, были получены из базы данных.

Как видно из примера, у интерфейса `ResultSet` есть метод `next()`. Он возвращает булеву (`Boolean`) переменную, указывающую, есть ли еще строки у набора результата. У экземпляра класса, реализующего интерфейс `ResultSet`, есть курсор (`cursor`), указывающий на текущую строку. Перед любыми вызовами метода `next()` курсор устанавливается перед первой строкой. Когда метод `next()` вызывается впервые, курсор указывает на первую строку в наборе результатов. Последующие вызовы метода `next()` перемещают курсор к следующей строке. Когда курсор будет указывать на последнюю строку в `ResultSet`, вызов метода `next()` возвратит `false`, указывая тем самым, что больше нет строк в `ResultSet`. Метод `ResultSet.next()` обычно используется в качестве условия в цикле `while`. Цикл выполняется, пока этот метод не возвращает `false`. В цикле операции могут быть выполнены в текущей строке набора результатов. Пример использует этот метод для заполнения простого JavaBean значениями из текущей строки. Как видно из кода, интерфейс `ResultSet` содержит метод, называемый `getString()`. Метод `getString()` возвращает значение столбца, указанного его единственным параметром, являющееся строкой, соответствующей столбцу, из которого мы хотели бы получить значение.

В дополнение к методу `getString()` интерфейс `ResultSet` содержит ряд методов для получения других типов данных. В следующей таблице приведены наиболее часто используемые методы (для ознакомления с полным списком обратитесь к документации JavaDoc для интерфейса `ResultSet` по адресу: <http://docs.oracle.com/javase/6/docs/api/>):

Имя метода	Возвращаемый тип
<code>getBoolean()</code>	<code>boolean</code>
<code>getDate()</code>	<code>java.sql.Date</code>
<code>getDouble()</code>	<code>double</code>
<code>getFloat()</code>	<code>float</code>
<code>getInt()</code>	<code>int</code>
<code>getLong()</code>	<code>long</code>
<code>getShort()</code>	<code>short</code>
<code>getString()</code>	<code>java.lang.String</code>
<code>getTime()</code>	<code>java.sql.Time</code>
<code>getTimestamp()</code>	<code>java.sql.Timestamp</code>

Имеются две перегруженных версии каждого из методов, перечисленных в таблице. Одна версия принимает строку, указывающую имя столбца в качестве параметра; другая версия принимает число типа `int`, указывающее позицию столбца в запросе. Например, в следующем запросе:

```
select column1, column2, column3 from table
```

столбец, названный `column1`, имеет позицию 1, `column2` – позицию 2 и `column3` – позицию 3. Использование версии предыдущих методов, принимающих `int`, обычно

приводит к результирующему коду, который труднее читать и понимать, чем при использовании версии, принимающей строку, поэтому его использование не рекомендуется.

Экземпляр PreparedStatement, получаемый вызовом метода Connection.prepareStatement(), содержит, помимо обычного, еще и предварительно скомпилированный SQL-оператор. SQL-оператор передается экземпляру PreparedStatement и отправляется в СУРБД для компиляции. Это означает, что после того, как экземпляр PreparedStatement выполнится однажды, СУРБД может выполнить SQL-оператор PreparedStatement, не компилируя его, и последующие вызовы для выполнения будут более быстрыми. Это удобно для статических запросов, как в предыдущем примере, где запрос действительно превосходен; однако в случае, когда запросы создаются динамически, путем передачи им параметров, дело обстоит иначе. Следующий пример является модифицированной версией предыдущего сервлета, поясняющего данную концепцию:

```
package net.ensode.glassfishbook.jdbcselect;

import java.io.IOException;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import javax.annotation.Resource;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class JDBCSelectServlet2 extends HttpServlet
{
    @Resource(name = "jdbc/_CustomerDBPool")
    private javax.sql.DataSource dataSource;
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException
    {
        String sql = "select us_state_nm, us_state_cd " +
            "from us_states where us_state_nm like ? " +
            "or us_state_nm like ? order by us_state_nm";
        ArrayList<UsStateBean> stateList = new ArrayList<UsStateBean>();
        try
        {
            Connection connection = dataSource.getConnection();
            PreparedStatement preparedStatement = connection.
                prepareStatement(sql);
            preparedStatement.setString(1, "North%");
            preparedStatement.setString(2, "South%");
            ResultSet resultSet = preparedStatement.executeQuery();
            response.setContentType("text/html");
            while (resultSet.next())
            {
                stateList.add(new UsStateBean(resultSet.getString("us_state_
                    nm"), resultSet.getString("us_state_cd")));
            }
            resultSet.close();
            preparedStatement.close();
            connection.close();
        }
    }
}
```

```
    request.setAttribute("stateList", stateList);
    request.getRequestDispatcher("us_states.jsp").forward(request,
        response);
}
catch (SQLException sqlException)
{
    sqlException.printStackTrace();
}
}
}
```

В этой версии сервлета мы модифицировали SQL-запрос, чтобы ограничить набор результатов на соответствие некоторым параметрам. Обратите внимание на вопросительные знаки в SQL-операторах. Эти вопросительные знаки являются местозаполнителями для параметров запроса и фактически не отправляются базе данных.

В предыдущем примере метод `setString()` интерфейса `PreparedStatement` используется для замены параметров запроса действительными значениями, которые будут отправлены базе данных. Этот метод принимает два аргумента: первый является индексным указателем заменяемого параметра, а второй – значением для замены. После замены параметров значениями запрос, приведенный в предыдущем коде, получит данные для всех штатов, имена которых начинаются со слова «North» или «South».



В отличие от высказывания, для массивов или коллекций индекс первого параметра должен быть 1, а не 0.

После компиляции и упаковки кода в WAR-файл, а также его развертывания, указав в адресной строке обозревателя его URL, мы должны будем увидеть в окне обозревателя следующую таблицу:

US State List	
Name	Abbreviation
North Carolina	NC
North Dakota	ND
South Carolina	SC
South Carolina	SC
South Dakota	SD

Помимо метода `setString()` интерфейс `PreparedStatement` содержит много подобных методов, которые позволяют нам устанавливать параметры различных типов. Следующая таблица иллюстрирует наиболее часто используемые методы (для получения полного списка обратитесь к документации JavaDoc для интерфейса `PreparedStatement` по адресу: <http://docs.oracle.com/javase/6/docs/api/>):

Методы интерфейса PreparedStatement

```
setBoolean(int parameterIndex, boolean b)
setDate(int parameterIndex, java.sql.Date d)
setDouble(int parameterIndex, double d)
setFloat(int parameterIndex, float f)
setInt(int parameterIndex, int i)
setLong(int parameterIndex, long l)
setShort(int parameterIndex, short s)
setString(int parameterIndex, String s)
setTime(int parameterIndex, java.sql.Time t)
setTimeStamp(int parameterIndex, java.sql.Timestamp t)
```

Во всех этих методах первый аргумент определяет указатель параметра (начинается с 1), а второй аргумент содержит значение для параметра.

Помимо изменения запроса для принятия параметров мы произвели дополнительное, не связанное с ним изменение в сервлете. Вместо того чтобы создавать экземпляр `javax.naming.InitialContext` и выполнять JNDI-поиск для получения ссылки на `DataSource`, мы использовали *инжекцию зависимости* (*dependency injection*) для получения упомянутого экземпляра.



Инжекция зависимости является шаблоном проектирования, в котором зависимости объекта вводятся контейнером во время выполнения. Этот шаблон проектирования был популярным в мире Java в каркасе Spring. Java EE использует аннотацию `@Resource` для реализации этого шаблона.

Мы выполнили это, переместив объявление объекта `dataSource` из метода `doGet()` и сделав его полем. Далее мы декорировали его аннотацией `@Resource`. У аннотации `@Resource` имеется элемент, называемый `name`. Этот элемент используется для указания имени JNDI-ресурса, который мы хотим получить.

Аннотация `@Resource` может использоваться для поиска любого вида ресурсов, доступных через JNDI, а не только `DataSources`.

Изменение информации в базе данных

В предыдущем разделе мы узнали, как можно использовать метод `executeQuery()` интерфейса `java.sql.PreparedStatement` для чтения данных из базы данных. В этом разделе мы познакомимся с тем, как можно использовать метод `executeUpdate()` этого интерфейса для вставки, обновления или удаления данных в базе данных. Использование метода `executeUpdate()` пояснено на следующем примере:

```
package net.ensode.glassfishbook.jdbcupdate;

import java.io.IOException;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import javax.annotation.Resource;
import javax.servlet.ServletException;
```

```
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;
public class JdbcUpdateServlet extends HttpServlet
{
    @Resource(name = "jdbc/_CustomerDBPool")
    private DataSource dataSource;
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException
    {
        String insertCustomerSql = "insert into " +
            "customers (customer_id, first_name, " +
            "last_name, email) values (?, ?, ?, ?)";
        String updateCustomerLastNameSql = "update customers " +
            "set last_name = ? where customer_id = ?";
        String deleteCustomerSql = "delete from customers " +
            "where customer_id = ?";
        PreparedStatement insertCustomerStatement;
        PreparedStatement updateCustomerLastNameStatement;
        PreparedStatement deleteCustomerStatement;
        try
        {
            Connection connection = dataSource.getConnection();
            insertCustomerStatement =
                connection.prepareStatement(insertCustomerSql);
            updateCustomerLastNameStatement =
                connection.prepareStatement(updateCustomerLastNameSql);
            deleteCustomerStatement =
                connection.prepareStatement(deleteCustomerSql);
            insertCustomerStatement.setInt(1, 1);
            insertCustomerStatement.setString(2, "Leo");
            insertCustomerStatement.setString(3, "Smith");
            insertCustomerStatement.setString(4, "lsmith@fake.com");
            insertCustomerStatement.executeUpdate();
            insertCustomerStatement.setInt(1, 2);
            insertCustomerStatement.setString(2, "Jane");
            insertCustomerStatement.setString(3, "Davis");
            insertCustomerStatement.setString(4, null);
            insertCustomerStatement.executeUpdate();
            updateCustomerLastNameStatement.setString(1, "Jones");
            updateCustomerLastNameStatement.setInt(2, 2);
            updateCustomerLastNameStatement.executeUpdate();
            deleteCustomerStatement.setInt(1, 1);
            deleteCustomerStatement.executeUpdate();
            deleteCustomerStatement.close();
            updateCustomerLastNameStatement.close();
            insertCustomerStatement.close();
            connection.close();
            response.getWriter().println("База данных успешно обновлена.");
        }
        catch (SQLException e)
        {
            e.printStackTrace();
        }
    }
}
```

В этом сервлете все SQL-операторы модифицируют данные в базе данных. Так же, как в предыдущем примере, мы получаем ссылку на источник данных при использовании инъекции зависимости. Затем мы получаем соединение из пула соединений, вызывая метод `getConnection()`, определенный в интерфейсе `javax.sql.DataSource`.

Далее мы получаем экземпляр класса, реализующего интерфейс `javax.sql.PreparedStatement` для каждого SQL-оператора. Это осуществляется вызовом метода `prepareStatement()`, определенного в интерфейсе `java.sql.Connection`.

Точно так же, как ранее, мы устанавливаем значения для каждого параметра, вызывая соответствующие методы, определенные в интерфейсе `PreparedStatement` (в нашем примере это `setInt()` и `setString()`). После того как будет установлен каждый параметр, мы вызываем метод `executeUpdate()`. В этом месте кода оператор фактически выполняется в базе данных.

После выполнения всех четырех обновлений в базе данных сервлет просто выводит сообщение в обозревателе: «База данных успешно обновлена.».

API Персистентности Java

API Персистентности Java (Java Persistence API (JPA)) был введен в спецификацию Java EE в версии 5. Как явствует из его названия, он используется для сохранения данных в Системе управления реляционными базами данных (СУРБД). JPA заменил собой *сущностные бины* (entity beans) в Java EE 5. Конечно, для обратной совместимости сущностные бины все еще поддерживаются. *Сущности Java EE* являются регулярными классами Java. Контейнер Java EE узнает, что эти классы являются сущностями, поскольку они декорируются аннотацией `@Entity`. Давайте рассмотрим отображение сущности на таблицу CUSTOMER в базе данных CUSTOMERDB:

```
package net.ensode.glassfishbook.jpa;

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()
    {
        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() import javax.persistence.Id;
    {
        return lastName;
    }
    public void setLastName(String lastName)
    {
        this.lastName = lastName;
    }
}
```

В этом 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, на какую таблицу сущность отображается. Значение элемента `name` содержит имя таблицы базы данных, на которую отображается сущность. Эта аннотация является необязательной. Если имя класса совпадает с именем таблицы базы данных, то нет необходимости указывать, на какую таблицу отображается сущность.

Аннотация `@Id` указывает, что поле `customerId` отображается на первичный ключ.

Аннотация `@Column` отображает каждое поле на столбец в таблице. Если имя поля соответствует имени столбца базы данных, то эта аннотация не обязательна. По этой причине поле `email` в приведенном примере не аннотируется.

В общем и целом это все, что мы должны сделать для создания сущности Java EE. Сравните эту процедуру с созданием сущностных бинов, где бин должен был реализовывать много методов жизненного цикла, которые редко использовались! Мы также должны были написать локальный и/или удаленный интерфейс, локальный и/или удаленный домашний интерфейс плюс дескриптор развертывания – и все это только для того, чтобы разработать один-единственный сущностный бин.

Класс EntityManager используется для сохранения сущности в базе данных. Следующий пример поясняет его использование:

```
package net.ensode.glassfishbook.jpa;

import java.io.IOException;
import javax.annotation.Resource;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.PersistenceUnit;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.transaction.HeuristicMixedException;
import javax.transaction.HeuristicRollbackException;
import javax.transaction.NotSupportedException;
import javax.transaction.RollbackException;
import javax.transaction.SystemException;
import javax.transaction.UserTransaction;
public class JpaDemoServlet extends HttpServlet
{
    @PersistenceUnit
    private EntityManagerFactory entityManagerFactory;
    @Resource
    private UserTransaction userTransaction;
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException
    {
        EntityManager entityManager = entityManagerFactory.
            createEntityManager();
        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 (NotSupportedException e)
        {
            e.printStackTrace();
        }
    }
}
```

```
        catch (SystemException e)
        {
            e.printStackTrace();
        }
        catch (SecurityException e)
        {
            e.printStackTrace();
        }
        catch (IllegalStateException e)
        {
            e.printStackTrace();
        }
        catch (RollbackException e)
        {
            e.printStackTrace();
        }
        catch (HeuristicMixedException e)
        {
            e.printStackTrace();
        }
        catch (HeuristicRollbackException e)
        {
            e.printStackTrace();
        }
        response.getWriter().println("База данных успешно обновлена.");
    }
}
```

Этот сервлет получает экземпляр класса, реализующего интерфейс `javax.persistence.EntityManagerFactory` через инжекцию зависимости. Это делается путем декорирования переменной `EntityManagerFactory` аннотацией `@PersistenceUnit`. Экземпляр `EntityManagerFactory` используется для получения ссылки на экземпляр класса, реализующего интерфейс `javax.persistence.EntityManager`.

Экземпляр класса, реализующий интерфейс `javax.transaction.UserTransaction`, вводится далее с помощью аннотации `@Resource`. Этот объект необходим, поскольку без обертывания кода вызовов, сохраняющих сущности в базе данных в транзакцию, будет возбуждено исключение `javax.persistence.TransactionRequiredException`. `EntityManager` выполняет множество обязанностей, которые для сущностных бинов выполняет домашний интерфейс; сюда относятся такие обязанности, как поиск сущностей в базе данных, их обновление или удаление. Мы получаем экземпляр класса, реализующего интерфейс `EntityManager`, вызывая метод `createEntityManager()` на `EntityManagerFactory`.

Поскольку сущности JPA представляют собой старые добрые объекты Java (*plain old Java objects (POJO)*), их экземпляры могут быть созданы с помощью оператора `new`. Мы вызываем их методы напрямую, в отличие от сущностных бинов, где используются методы на экземпляре класса, реализующего их удаленный интерфейс.

 **Примечание** Вызов метода `setCustomerId()` пользуется возможностями автоматической упаковки, функции добавленной в язык Java JDK 1.5. Обратите внимание, что метод принимает экземпляр `java.lang.Long` в качестве его параметра, однако мы используем примитивный тип `long`. Код компилируется и выполняется должным образом благодаря этой функции.

Вызовы метода `persist()` на `EntityManager` должны быть осуществлены в рамках транзакции, поэтому ее нужно начинать, вызывая метод `begin()` на `UserTransaction`.

Далее мы вставляем две новых строки в таблицу `CUSTOMERS`, вызывая метод `persist()` на `entityManager` для двух экземпляров класса `Customer`, которые мы заполнили ранее в коде.

После сохранения данных, содержащихся в объектах `customer` и `customer2`, мы ищем в базе данных строку с первичным ключом 4 в таблице `CUSTOMERS`. Мы делаем это, вызывая метод `find()` на `entityManager`. Этот метод в качестве первого параметра принимает класс сущности, который мы ищем, а в качестве второго параметра – первичный ключ строки, соответствующей объекту, который мы хотим получить. Этот метод приблизительно эквивалентен методу `findByPrimaryKey()` в домашнем интерфейсе сущностного бина.

Первичный ключ, который мы установили для объекта `customer2`, был равен 4; по этой причине мы теперь имеем копию данного объекта. В фамилии этого заказчика была сделана орфографическая ошибка, когда мы заносили сведения о нем в базу данных. Теперь мы исправляем фамилию г-на Джонсона, вызывая метод `setLastName()` на `customer3`, а затем обновляем информацию в базе данных, вызывая метод `entityManager.persist()`.

После этого мы удаляем информацию для объекта `customer`, вызывая метод `entityManager.remove()` и передавая ему объект `customer` в качестве параметра.

Наконец, мы передаем изменения базе данных, вызывая метод `commit()` на `userTransaction`.

Для того чтобы предыдущий код работал как ожидается, конфигурационный XML-файл под названием `persistence.xml` должен быть развернут в WAR-файле, содержащем предыдущий сервлет. Этот файл должен быть помещен в каталог `WEB-INF/classes/META-INF/` в WAR-файле. Содержимое этого файла для предыдущего кода показано ниже:

```
<? xml version="1.0" encoding="UTF-8" ?>
<persistence version="1.0" 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 элемент `<persistence-unit>` должен предоставить значение для его атрибута `name` и содержать дочерний элемент `<jta-data-source>`, значение которого является JNDI-именем источника данных, который будет использоваться для модуля персистентности.

Причина, по которой разрешено иметь более чем один элемент `<persistence-unit>`, состоит в том, что приложение может получать доступ к более чем одной базе данных. Элемент `<persistence-unit>` требуется для каждой базы данных, к которой приложение получит доступ. Если приложение определяет больше чем один элемент `<persistence-unit>`, то аннотация `@PersistenceUnit`, используемая для инжекции интерфейса `EntityManagerFactory`, должна предоставить значение для своего элемента `unitName`. Значение для этого элемента должно соответствовать атрибуту имени 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 эту сущность на уровень бизнес-логики или пользовательского интерфейса, где она потенциально будет модифицирована и информация в базе данных, соответствующая сущности, позже будет обновлена. В случаях, подобных этому, вызов метода `EntityManager.persist()` приведет к возникновению исключения. Чтобы обновить JPA-сущность этим путем, мы должны вызвать метод `EntityManager.merge()`. Этот метод принимает экземпляр JPA-сущности в качестве его единственного аргумента и обновляет соответствующую строку в базе данных хранящимся в нем данными.

Отношения сущности

В предыдущем разделе мы видели, как получить одиночные сущности из базы данных, а также вставить их в базу данных, выполнить их обновление и удаление. 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` есть отношение «один к одному» между таблицами `LOGIN_INFO` (Учетные данные) и `CUSTOMERS` (Заказчики). Это означает, что каждому заказчику соответствует ровно одна строка в таблице `LOGIN_INFO`. Имеется также отношение «один ко многим» между таблицей `CUSTOMERS` и таблицей `ORDERS` (Заказы). Это связано с тем, что один заказчик может разместить много заказов. Кроме того, есть отношение «многие ко многим» между таблицей `ORDERS` и таблицей `ITEMS` (Элементы). Это связано с тем, что заказ может содержать много элементов, а элемент в свою очередь может входить в несколько заказов.

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

Отношения «один к одному»

Отношения «один к одному» встречаются, когда у экземпляра сущности может быть либо нуль, либо один соответствующий ему экземпляр другой сущности.

Отношения сущностей «один к одному» могут быть двунаправленными (каждый объект знает об отношении) или однонаправленными (только один из объектов знает об отношении). В базе данных `CUSTOMERDB` отношение один к одному между таблицами `LOGIN_INFO` и `CUSTOMERS` является однонаправленным, поскольку у таблицы

LOGIN_INFO есть внешний ключ к таблице CUSTOMERS, но не наоборот. Как мы скоро увидим, этот факт не мешает нам создать двунаправленное отношение «один к одному» между сущностью Customer и сущностью LoginInfo.

Исходный код для сущности LoginInfo, которая отображается на таблицу LOGIN_INFO, приведен ниже:

```
package net.ensode.glassfishbook.entityrelationships;

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;
    }
}
```

Код для этой сущности очень похож на код для сущности `Customer`. Он определяет поля, которые отображаются на столбцы базы данных. Каждое поле, имя которого не соответствует имени столбца базы данных, декорируется аннотацией `@Column`. В дополнение к этому первичный ключ декорируется аннотацией `@Id`.

То место кода, в котором он становится для нас действительно интересным, находится в объявлении поля `customer`. Как видно из кода, поле `customer` декорируется аннотацией `@OneToOne`. Оно позволяет серверу приложений (в данном случае GlassFish) узнать, что имеется отношение «один к одному» между этой сущностью и сущностью `Customer`. Поле `customer` также декорируется аннотацией `@JoinColumn`. Эта аннотация сообщает контейнеру, что столбец в таблице `LOGIN_INFO` является внешним ключом, соответствующим первичному ключу в таблице `CUSTOMER`. Поскольку у таблицы `LOGIN_INFO`, на которую отображается сущность `LoginInfo`, есть внешний ключ к таблице `CUSTOMER`, сущность `LoginInfo` является владельцем отношения. Если бы отношение было односторонним, то нам не нужно было бы производить изменения в сущности `Customer`. Однако, поскольку мы хотим иметь двунаправленное отношение между этими двумя сущностями, мы должны добавить поле `LoginInfo` к сущности `Customer` наряду с соответствующими методами геттеров и сеттеров.

Как уже упоминалось ранее, для того чтобы сделать отношение «один к одному» между сущностями `Customer` и `LoginInfo` двунаправленным, мы должны произвести несколько простых изменений в сущности `Customer`:

```
package net.ensode.glassfishbook.entityrelationships;

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;
    }
}

```

Для того чтобы сделать отношение «один к одному» двунаправленным, нам всего лишь нужно добавить в сущность Customer поле LoginInfo наряду с соответствующими методами геттеров и сеттеров. Поле loginInfo декорируется аннотацией @OneToOne. Поскольку сущности Customer не принадлежит отношение (таблица, на которую она отображается, не имеет внешнего ключа к соответствующей таблице), должен быть добавлен элемент mappedBy аннотации @OneToOne. Этот элемент указывает, какое поле, соответствующее сущности, содержит другой конец отношения. В данном случае поле заказчика (customer) в сущности LoginInfo соответствует другому концу этого отношения «один к одному».

Следующий сервлет поясняет использование этой сущности:

```

package net.ensode.glassfishbook.entityrelationships;

import java.io.IOException;
import javax.annotation.Resource;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.PersistenceUnit;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

```

```
import javax.transaction.HeuristicMixedException;
import javax.transaction.HeuristicRollbackException;
import javax.transaction.NotSupportedException;
import javax.transaction.RollbackException;
import javax.transaction.SystemException;
import javax.transaction.UserTransaction;
public class OneToOneRelationshipDemoServlet extends HttpServlet
{
    @PersistenceUnit(unitName = "customerPersistenceUnit")
    private EntityManagerFactory entityManagerFactory;
    @Resource
    private UserTransaction userTransaction;
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException
    {
        EntityManager entityManager = entityManagerFactory.
            createEntityManager();
        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();
            response.getWriter().println("База данных успешно обновлена.");
        }
        catch (NotSupportedException e)
        {
            e.printStackTrace();
        }
        catch (SystemException e)
        {
            e.printStackTrace();
        }
        catch (SecurityException e)
        {
            e.printStackTrace();
        }
        catch (IllegalStateException e)
        {
            e.printStackTrace();
        }
        catch (RollbackException e)
        {
            e.printStackTrace();
        }
        catch (HeuristicMixedException e)
        {
            e.printStackTrace();
        }
        catch (HeuristicRollbackException e)
        {
            e.printStackTrace();
        }
    }
}
```

В этом примере мы сначала создаем экземпляр сущности `LoginInfo` и заполняем его некоторыми данными. Затем мы получаем экземпляр сущности `Customer` из базы данных, вызывая метод `find()` на `EntityManager` (данные для этой сущности были вставлены в таблицу `CUSTOMERS` в одном из примеров JDBC). Затем мы вызываем метод `setCustomer()` на сущности `LoginInfo`, передавая сущность заказчика в качестве параметра. Наконец, мы вызываем метод `EntityManager.persist()` для сохранения данных в базе данных.

«За кулисами» происходит следующее: столбец `CUSTOMER_ID` таблицы `LOGIN_INFO` заполняется первичным ключом соответствующей строки в таблице `CUSTOMERS`. Это может быть легко проверено путем запроса в базу данных `CUSTOMERDB`.



Обратите внимание, что метод `EntityManager.find()` для получения сущности `Customer` вызывается в той же самой транзакции, в которой мы вызываем метод `EntityManager.persist()`. Это обязательно должно быть сделано, иначе база данных не будет успешно обновлена.

Отношения «один ко многим»

Отношения JPA-сущностей «один ко многим» могут быть двунаправленными (одна сущность содержит отношение «многие к одному», а соответствующая сущность содержит обратное отношение «один ко многим»).

SQL-отношения «один ко многим» определяются внешними ключами в одной из таблиц. Часть «многие» отношения содержит внешний ключ к части «один» отношения. Отношения «один ко многим», определенные в СУРБД, обычно односторонние, поскольку создание их двунаправленными обычно приводит к денормализации данных.

Подобно тому как определяется одностороннее отношение «один ко многим» в СУРБД, в JPA у части «многие» отношения имеется ссылка на часть «один» отношения. Поэтому аннотация, используемая для декорирования соответствующего сеттера, является аннотацией `@ManyToOne`.

В базе данных `CUSTOMERDB` есть одностороннее отношение «один ко многим» между заказчиками и заказами. Мы определяем это отношение в сущности `Order`:

```
package net.ensode.glassfishbook.entityrelationships;

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")
```

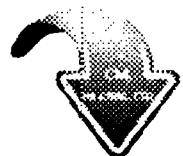
```
private String orderDescription;
@ManyToOne
@JoinColumn(name = "CUSTOMER_ID")
private Customer customer;
public Customer getCustomer()
{
    return customer;
}
public void setCustomer(Customer customer)
{
    this.customer = customer;
}
public String getOrderDescription()
{
    return orderDescription;
}
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 мы должны будем определить одностороннее отношение «многие к одному» между сущностью Orders и сущностью Customer, нам не нужно будет производить изменения в сущности Customer. Чтобы определить двунаправленное отношение «один ко многим» между этими двумя сущностями, нужно добавить новое поле, декорированное аннотацией @OneToMany, к сущности Customer:

```
package net.ensode.glassfishbook.entityrelationships;

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")
```



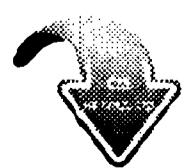
```
private Long customerId;
@Column(name = "FIRST_NAME")
private String firstName;
@Column(name = "LAST_NAME")
private String lastName;
private String email;
@OneToOne(mappedBy = "customer")
private LoginInfo loginInfo;
@OneToMany(mappedBy = "customer")
private Set<Order> orders;
public Long getCustomerId()
{
    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 версией состоит в добавлении поля заказов и связанных с ним методов сеттеров и геттеров. Особенno интересна аннотация `@OneToMany`, декорирующая это поле. Атрибут `mappedBy` должен совпадать с именем соответствующего поля в сущности, соответствующей части «многие» отношения. Проще говоря, значение атрибута `mappedBy` должно соответствовать имени поля, декорированного аннотацией `@ManyToOne` в бине на другой стороне отношения.

Следующий сервлет поясняет, как сохранить отношения «один ко многим» в базе данных:

```
package net.ensode.glassfishbook.entityrelationships;

import java.io.IOException;
import javax.annotation.Resource;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.PersistenceUnit;
import javax.persistence.PersistenceException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.transaction.HeuristicMixedException;
import javax.transaction.HeuristicRollbackException;
import javax.transaction.NotSupportedException;
import javax.transaction.RollbackException;
import javax.transaction.SystemException;
import javax.transaction.UserTransaction;
public class OneToManyRelationshipDemoServlet extends HttpServlet
{
    @PersistenceUnit(unitName = "customerPersistenceUnit")
    private EntityManagerFactory entityManagerFactory;
    @Resource
    private UserTransaction userTransaction;
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException
    {
        EntityManager entityManager = entityManagerFactory.
            createEntityManager();
        Customer customer;
        Order order1;
        Order order2;
        order1 = new Order();
        order1.setOrderId(1L);
        order1.setOrderNumber("SFX12345");
        order1.setOrderDescription("Незаполненный заказ.");
        order2 = new Order();
        order2.setOrderId(2L);
        order2.setOrderNumber("SFX23456");
        order2.setOrderDescription("Ещё один незаполненный заказ.");
        try
        {
            userTransaction.begin();
            customer = entityManager.find(Customer.class, 4L);
            order1.setCustomer(customer);
            order2.setCustomer(customer);
            entityManager.persist(order1);
```



```
entityManager.persist(order2);
userTransaction.commit();
response.getWriter().println("База данных успешно обновлена.");
}
catch (NotSupportedException e)
{
    e.printStackTrace();
}
catch (SystemException e)
{
    e.printStackTrace();
}
catch (SecurityException e)
{
    e.printStackTrace();
}
catch (IllegalStateException e)
{
    e.printStackTrace();
}
catch (RollbackException e)
{
    e.printStackTrace();
}
catch (HeuristicMixedException e)
{
    e.printStackTrace();
}
catch (HeuristicRollbackException e)
{
    e.printStackTrace();
}
}
}
```

Этот код очень похож на предыдущий пример. Он создает два экземпляра сущности Order, заполняет их некоторыми данными, а затем в транзакции помещается экземпляр сущности Customer и используется в качестве параметра метода setCustomer() для обоих экземпляров сущности Order. Далее мы сохраняем оба экземпляра сущности Order, вызывая метод EntityManager.persist() для каждого из них.

Точно так же, как в случае с отношениями «один к одному», «за кулисами» происходит следующее: столбец CUSTOMER_ID таблицы ORDERS в базе данных CUSTOMERDB заполняется первичным ключом, соответствующим связанной строке в таблице CUSTOMERS.

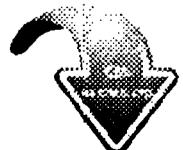
Поскольку отношение является двунаправленным, мы можем получить все заказы, связанные с заказчиком, вызывая метод getOrders() на сущности Customer.

Отношения «многие ко многим»

В базе данных CUSTOMERDB есть отношение «многие ко многим» между таблицей ORDERS и таблицей ITEMS. Мы можем отобразить это отношение, добавляя новое поле Collection<Item> к сущности Order и декорируя его аннотацией @ManyToMany.

```
package net.ensode.glassfishbook.entityrelationships;

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;
}
}
}

```

Как видно из этого кода, в дополнение к декорированию аннотацией @ManyToMany поле элементов также декорируется аннотацией @JoinTable. Как и предполагает ее название, эта аннотация сообщает серверу приложений, какая таблица используются в качестве объединяющей таблицы для создания отношения «многие ко многим» между этими двумя сущностями. У этой аннотации есть три соответствующих элемента: элемент name, который определяет имя объединяющей таблицы, а также элементы joinColumns и inverseJoinColumns, которые в свою очередь определяют столбцы, служащие внешними ключами в объединяющей таблице, указывающей на первичные ключи сущностей. Значения для элементов joinColumns и inverseJoinColumns являются еще одной аннотацией – @JoinColumn. У этой аннотации есть два взаимосвязанных элемента: элемент name, который определяет имя столбца в объединяющей таблице, и элемент referencedColumnName, который определяет имя столбца в таблице сущности.

Сущность Item является простой сущностью, отображающейся на таблицу ITEMS в базе данных CUSTOMERDB:

```

package net.ensode.glassfishbook.entityrelationships;

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 отношения «один ко многим», отношения «многие ко многим» могут быть односторонними или двунаправленными. Поскольку мы хотим, чтобы отношение «многие ко многим» между сущностями `Order` и `Item` было двунаправленным, мы добавили поле `Collection<Order>` и декорировали его аннотацией `@ManyToMany`. Поскольку соответствующему полю в сущности `Order` мы уже определили объединяющую таблицу, нет необходимости делать здесь это снова. Сущности, содержащей аннотацию `@JoinTable`, как говорят, принадлежит отношение. В отношении «многие ко многим» отношение может принадлежать любой сущности. В нашем примере оно принадлежит сущности `Order` и представлено ее полем `Collection<Item>`, декорированным аннотацией `@JoinTable`.

Точно так же, как в случае с отношениями «один к одному» и «один ко многим», аннотация `@ManyToMany` на стороне, не являющейся владельцем двунаправленного отношения «многие ко многим», должна содержать элемент `mappedBy`, указывающий, какое поле во владеющей сущности определяет это отношение.

Теперь, когда мы рассмотрели изменения, необходимые для установления двунаправленного отношения «многие ко многим» между сущностями Order и Item, в следующем примере мы можем рассмотреть отношение в действии:

```
package net.ensode.glassfishbook.entityrelationships;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Collection;
import javax.annotation.Resource;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.PersistenceUnit;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.transaction.HeuristicMixedException;
import javax.transaction.HeuristicRollbackException;
import javax.transaction.NotSupportedException;
import javax.transaction.RollbackException;
import javax.transaction.SystemException;
import javax.transaction.UserTransaction;
public class ManyToManyRelationshipDemoServlet extends HttpServlet
{
    @PersistenceUnit(unitName = "customerPersistenceUnit")
    private EntityManagerFactory entityManagerFactory;
    @Resource
    private UserTransaction userTransaction;
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException
    {
        EntityManager entityManager = entityManagerFactory.
            createEntityManager();
        Order order;
        Collection<Item> items = new ArrayList<Item>();
        Item item1 = new Item();
        Item item2 = new Item();
        item1.setItemId(1L);
        item1.setItemNumber("BCD1234");
        item1.setItemShortDesc("Мобильный компьютер");
        item1.setItemLongDesc("64-битный 4-х ядерный ЦПУ, 4ГБ ОЗУ");
        item2.setItemId(2L);
        item2.setItemNumber("CDF2345");
        item2.setItemShortDesc("Беспроводная мышь");
        item2.setItemLongDesc("Три кнопки, инфракрасный порт, "
            + "колёса прокрутки по вертикали и горизонтали");
        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();
            response.getWriter().println("База данных успешно обновлена.");
        }
```

```
        catch (NotSupportedException e)
        {
            e.printStackTrace();
        }
        catch (SystemException e)
        {
            e.printStackTrace();
        }
        catch (SecurityException e)
        {
            e.printStackTrace();
        }
        catch (IllegalStateException e)
        {
            e.printStackTrace();
        }
        catch (RollbackException e)
        {
            e.printStackTrace();
        }
        catch (HeuristicMixedException e)
        {
            e.printStackTrace();
        }
        catch (HeuristicRollbackException e)
        {
            e.printStackTrace();
        }
    }
}
```

Этот код создает два экземпляра сущности `Item` и заполняет их некоторыми данными. Затем он добавляет эти два экземпляра в коллекцию. После этого запускается транзакция, и два экземпляра `Item` сохраняются к базе данных. Далее выполняется получение экземпляра сущности `Order` из базы данных. Потом вызывается метод `setItems()` экземпляра сущности `Order`, передавая коллекцию, содержащую два экземпляра `Item` в качестве параметра. После этого экземпляр `Customer` сохраняется в базе данных. Далее «за кулисами» создаются две строки в таблице `ORDER_ITEMS`, которая объединяет таблицы `ORDERS` и `ITEMS`.

Составные первичные ключи

У большинства таблиц в базе данных `CUSTOMERDB` имеется столбец, предназначенный исключительно для использования его в качестве первичного ключа (этот тип первичного ключа иногда называют *первичным ключом-заместителем*, или *искусственным первичным ключом*). Однако некоторые базы данных не разрабатываются таким путем; вместо этого в них используется столбец базы данных, значение которого является уникальным для записей данных, использующих его в качестве первичного ключа. Если же нет никакого столбца, значение которого гарантированно будет уникальным для всех записей, то в качестве первичного ключа таблицы используется комбинация из двух или более столбцов. Имеется возможность отображения этой разновидности первичного ключа на сущности JPA путем использования класса первичного ключа.

В базе данных CUSTOMERDB есть одна таблица, у которой нет первичного ключа-заместителя; это таблица ORDER_ITEMS. Эта таблица является объединяющей таблицей для таблиц ORDERS и ITEMS. Помимо внешних ключей для этих двух таблиц у таблицы ORDER_ITEMS имеется дополнительный столбец под названием ITEM_QTY, который хранит количество каждого элемента в заказе. Поскольку эта таблица не имеет первичного ключа-заместителя, у JPA-сущности, отображающейся на нее, должен быть пользовательский класс первичного ключа. В таблице ORDER_ITEMS комбинация столбцов ORDER_ID и ITEM_ID должна быть уникальной. Таким образом, эта комбинация столбцов является хорошим кандидатом на роль составного первичного ключа.

```
package net.ensode.glassfishbook.compositekeys;

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 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:

- он имеет модификатор видимости `public`;
- он реализует интерфейс `java.io.Serializable`;
- у него имеется конструктор с модификатором видимости `public`, который не принимает аргументов;
- его поля имеют модификаторы видимости `public` или `protected`;
- имена его полей и типы соответствуют именам полей и типам из сущности;
- он переопределяет методы по умолчанию `hashCode()` и `equals()`, определенные в классе `java.lang.Object`.

Для класса `OrderPK` справедливо все вышеперечисленное. У него также есть конструктор, который для удобства принимает два объекта типа `Long`, предназначенные для инициализации его полей `orderId` и `itemId`. Этот конструктор был добавлен для удобства; он не является обязательным для класса, который будет использоваться в качестве класса первичного ключа.

Когда объект использует пользовательский класс первичного ключа, он должен быть декорирован аннотацией `@IdClass`. Поскольку класс `OrderItem` использует `OrderItemPK` в качестве своего пользовательского класса первичного ключа, он должен быть декорирован указанной выше аннотацией.

```
package net.ensode.glassfishbook.compositekeys;

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;
}
}

```

Имеется два различия между этой сущностью и предыдущими сущностями, которые мы видели. Первое различие заключается в том, что эта сущность декорируется аннотацией `@IdClass`, указывая класс первичного ключа, соответствующего ей; второе состоит в том, что у данной сущности имеется более одного поля, декорированного 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.compositekeys;

import java.io.IOException;
import java.io.PrintWriter;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.PersistenceUnit;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class CompositeKeyDemoServlet extends HttpServlet
{
    @PersistenceUnit(unitName = "customerPersistenceUnit")
    private EntityManagerFactory entityManagerFactory;

```

```
@Override  
protected void doGet(HttpServletRequest request, HttpServletResponse  
                      response) throws ServletException, IOException  
{  
    PrintWriter printWriter = response.getWriter();  
    EntityManager entityManager = entityManagerFactory.  
        createEntityManager();  
    OrderItem orderItem;  
    orderItem = entityManager.find(OrderItem.class, new  
        OrderItemPK(1L, 2L));  
    response.setContentType("text/html");  
    if (orderItem != null)  
    {  
        printWriter.println("Найден экземпляр элемента заказа для  
                           предоставленного первичного ключа:<br/>");  
        printWriter.println("Элемент заказа к заказу с id: " + orderItem.  
                           getOrderId() + "<br/>");  
        printWriter.println("Собственный id элемента заказа: " +  
                           orderItem.getItemId() + "<br/>");  
    }  
    else  
    {  
        printWriter.println("Для предоставленного первичного ключа  
                           экземпляров элементов заказа не найдено.");  
    }  
}  
}
```

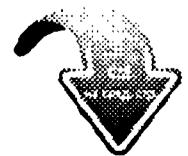
Как видно из этого примера, единственная разница между определением местоположения сущности с составным первичным ключом и сущности с первичным ключом в виде единственного поля заключается в том, что экземпляр пользовательского класса первичного ключа нужно передать в качестве второго параметра методу EntityManager.find(). Поля этого экземпляра должны быть заполнены адекватными значениями для всех полей, каждое из которых является частью первичного ключа.

Язык запросов персистентности Java

Все наши примеры, которые получают сущности из базы данных, до сих пор для упрощения предполагали, что первичный ключ сущности известен заранее. Однако все мы знаем, что зачастую дело обстоит далеко не так. Всякий раз, когда мы должны искать сущность по полю, отличному от первичного ключа сущности, мы должны использовать *Язык запросов персистентности Java (Java Persistence Query Language (JPQL))*.

JPQL является SQL-подобным языком, используемым для получения, обновления и удаления сущностей в базе данных. Следующий пример поясняет, как использовать JPQL для получения подмножества штатов из таблицы US_STATES в базе данных CUSTOMERDB:

```
package net.ensode.glassfishbook.jpaquerylang;  
  
import java.io.IOException;  
import java.io.PrintWriter;  
import java.util.List;  
import javax.persistence.EntityManager;
```



```
import javax.persistence.EntityManagerFactory;
import javax.persistence.PersistenceUnit;
import javax.persistence.Query;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class SelectQueryDemoServlet extends HttpServlet
{
    @PersistenceUnit(unitName = "customerPersistenceUnit")
    private EntityManagerFactory entityManagerFactory;
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException
    {
        PrintWriter printWriter = response.getWriter();
        List<UsState> matchingStatesList;
        EntityManager entityManager = entityManagerFactory.
            createEntityManager();
        Query query = entityManager.createQuery(
            "SELECT s FROM UsState s WHERE s.usStateNm " + "LIKE :name");
        query.setParameter("name", "New%");
        matchingStatesList = query.getResultList();
        response.setContentType("text/html");
        printWriter.println("Следующие штаты соответствуют " +
            "критерию:<br/>");
        for (UsState state : matchingStatesList)
        {
            printWriter.println(state.getUsStateNm() + "<br/>");
        }
    }
}
```

Данный код вызывает метод EntityManager.createQuery(), передавая строку, содержащую запрос JPQL в качестве параметра. Этот метод возвращает экземпляр javax.persistence.Query. Запрос получает все сущности UsState, имя которых начинается со слова «New».

Как видно из этого кода, JPQL подобен SQL. Однако есть и ряд отличий, которые могут смутить читателей, имеющих познания в области SQL. Код эквивалентного SQL-запроса был бы таким:

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

Первое различие между JPQL и SQL состоит в том, что в JPQL мы всегда ссылаемся на имена сущности, тогда как в SQL – на имена таблиц. Символ s после имени сущности в запросе JPQL является псевдонимом для сущности. Табличные псевдонимы являются необязательными в SQL, но псевдонимы сущности в JPQL обязательны. Если принять во внимание эти различия, запрос JPQL уже не столь обескураживает.

В запросе :name является именованным параметром. Именованные параметры предназначены для их замены действительными значениями. Это делается путем вызова метода setParameter() в экземпляре javax.persistence.Query, возвращаемого вызовом метода EntityManager.createQuery(). У запроса JPQL может быть несколько именованных параметров.

Для фактического выполнения запроса и получения сущности из базы данных должен быть вызван метод `getResultSet()` на экземпляре `javax.persistence.Query`, полученным из `EntityManager.createQuery()`. Этот метод возвращает экземпляр класса, реализующего интерфейс `java.util.List`. Полученный список будет содержать сущности, соответствующие критерию запроса. Если никакие сущности не соответствуют критерию, то будет возвращен пустой список.

Если мы уверены, что запрос возвратит единственный объект, то в качестве альтернативы может быть вызван метод `getSingleResult()` на экземпляре `Query`. Этот метод возвращает `Object`, который должен быть преобразован к типу соответствующей сущности.

Предыдущий пример использует оператор `LIKE`, чтобы обнаружить сущности, имя которых начинается со слова «New». Это достигается путем подстановки имени параметра запроса со значением `New%`. Знак процента в конце значения параметра означает, что любое число символов после слова «New» будет соответствовать выражению. Знак процента может использоваться где угодно в значении параметра. Например, значению `%Dakota` соответствовали бы любые сущности, имя которых заканчивается на «Dakota»; значение `A%a` будет соответствовать любым штатам, имя которых начинается с заглавной буквы «A» и заканчивается буквой «a» нижнего регистра. В значении параметра может быть более одного знака процента. Знак подчеркивания () может использоваться в качестве соответствия одиночному буквенно-цифровому символу. Все правила для знака процента также применимы и к знаку подчеркивания.

Кроме оператора `LIKE` существуют другие операторы, которые могут использоваться для получения сущностей из базы данных:

- `=` извлекает сущности, у которых значение поля слева от оператора точно соответствует значению справа;
- `>` извлекает сущности, у которых значение поля слева от оператора больше, чем значение справа;
- `<` извлекает сущности, у которых значение поля слева от оператора меньше, чем значение справа;
- `>=` извлекает сущности, у которых значение поля слева от оператора больше или равно значению справа;
- `<=` извлекает сущности, у которых значение поля слева от оператора меньше или равно значению справа.

Все эти операторы работают аналогично их эквивалентам в SQL. Точно так же как в SQL, эти операторы могут комбинироваться с помощью операторов `AND` и `OR`. Условия, объединенные с помощью оператора `AND`, соответствуют оператору, когда оба условия являются истинными, а условия, объединенные с помощью оператора `OR`, соответствуют оператору, когда по крайней мере одно из условий является истинным.

Если мы намереваемся использовать запрос много раз, он может быть сохранен в *именованном запросе* (named query). Именованный запрос может быть определен с помощью декорирования соответствующего класса сущности аннотацией @NamedQuery. Эта аннотация имеет два элемента: элемент name используется для определения имени запроса, а элемент query определяет собственно запрос. Чтобы выполнить именованный запрос, нужно вызвать метод createNamedQuery() на экземпляре EntityManager. Этот метод принимает строку (String), содержащую имя запроса, в качестве единственного параметра и возвращает экземпляр java. persistence.Query.

Помимо получения сущностей, JPQL может использоваться для их модификации или удаления. Однако операции изменения и удаления сущности могут быть выполнены программным путем через интерфейс EntityManager. Результат выполнения этого в коде будет более читабельным, чем при использовании JPQL. В связи с этим мы не будем описывать изменение и удаление сущности с помощью JPQL. Для читателей, интересующихся написанием запросов JPQL для модификации и удаления сущностей, равно как и для читателей, желающих больше узнать о JPQL, рекомендуется изучение Спецификации персистентность Java 2.0. Эта спецификация может быть загружена по адресу <http://jcp.org/en/jsr/detail?id=317>.

В примерах этой главы мы показали доступ к базе данных, выполняемый прямо из сервлетов. Мы поступили так для того, чтобы пояснить суть излагаемого материала, не вдаваясь в ненужные детали. Однако в общем и целом такая практика является неправильной – код доступа к базе данных должен быть инкапсулирован в *Объектах доступа к данным* (Data Access Objects (DAO)).



Для получения дополнительной информации о шаблоне проектирования DAO обратитесь к следующей веб-странице: <http://www.oracle.com/technetwork/java/dataaccessobject-138824.html>.

Кроме того, в наших примерах показаны сервлеты, которые в значительной степени выполняли только доступ к базе данных. Сервлеты обычно выполняют роль контроллеров, следуя шаблону проектирования *Модель-Представление-Контроллер* (Model-View-Controller (MVC)). Мы решили не добавлять какой-либо код пользовательского интерфейса для нашего примера, поскольку он не имеет отношения к данной теме. Однако в реальных приложениях мы, конечно, заполнили бы сущности из компонентов пользовательского интерфейса – скорее всего, с помощью полей ввода в JSP. Эти поля содержались бы в HTML-форме, которая, будучи заполненной и отправленной, передавала бы управление сервлету; затем он заполнил бы сущности данными, введенными пользователем, и передал бы эти сущности в DAO, а те в свою очередь сохранили бы данные в базе данных.



Для получения дополнительной информации о шаблоне проектирования MVC обратитесь к следующей веб-странице: <http://www.oracle.com/technetwork/java/mvc-140477.html>.

Новые функции, введенные в JPA 2.0

Версия 2.0 спецификации JPA вводит некоторые новые функции для того, чтобы еще более упростить работу с JPA. В следующих разделах мы обсудим некоторые из этих новых функций.

API Критериев

Одним из основных дополнений к спецификации JPA версии 2.0 является введение *API Критериев* (Criteria API). API Критериев предназначается в качестве дополнения к Языку запросов персистентности Java (JPQL).

Хотя JPQL очень гибок, имеются некоторые проблемы, которые затрудняют работу с ним. Во первых, запросы JPQL сохраняются как строки и у компилятора нет какого-либо способа проверить синтаксис JPQL. Во вторых, JPQL не безопасен с точки зрения типов. Мы можем написать запрос JPQL, в котором наше предложение `where` может иметь строковое значение для числового свойства, тем не менее наш код будет нормально скомпилирован и развернут.

Чтобы обойти ограничения JPQL, описанные в предыдущем абзаце, в спецификации JPA версии 2.0 был введен API Критериев. API Критериев позволяет нам писать программные запросы JPA без необходимости опираться на JPQL.

Следующий пример кода поясняет, как использовать API Критериев в нашем приложении Java EE 6:

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

import java.io.IOException;
import java.io.PrintWriter;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.PersistenceUnit;
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 javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet(urlPatterns = {" /criteriaapi"})
public class CriteriaApiDemoServlet extends HttpServlet
{
    @PersistenceUnit(unitName = "customerPersistenceUnit")
    private EntityManagerFactory entityManagerFactory;
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException
```



```
{  
    PrintWriter printWriter = response.getWriter();  
    List<UsState> matchingStatesList;  
    EntityManager entityManager =  
        entityManagerFactory.createEntityManager();  
    CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();  
    CriteriaQuery<UsState> criteriaQuery =  
        criteriaBuilder.createQuery(UsState.class);  
    Root<UsState> root = criteriaQuery.from(UsState.class);  
    Metamodel metamodel = entityManagerFactory.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();  
    response.setContentType("text/html");  
    printWriter.println("Следующие штаты соответствуют критерию:<br/>");  
    for (UsState state : matchingStatesList)  
    {  
        printWriter.println(state.getUsStateNm() + "<br/>");  
    }  
}
```

Этот пример эквивалентен примеру JPQL, который приводился выше в этой главе. Однако этот пример использует преимущества API Критериев вместо того, чтобы полагаться на JPQL.

При написании кода с использованием API Критериев мы прежде всего должны получить экземпляр класса, реализующего интерфейс `javax.persistence.criteria.CriteriaBuilder`. Как видно из предыдущего примера, мы должны получить упомянутый экземпляр, вызывая метод `getCriteriaBuilder()` на нашем `EntityManager`.

Из нашей реализации CriteriaBuilder мы должны получить экземпляр класса, реализующего интерфейс javax.persistence.criteria.CriteriaQuery. Делается это путем вызова метода createQuery() в нашей реализации CriteriaBuilder. Обратите внимание, что CriteriaQuery имеет обобщенный тип. Аргумент обобщенного типа диктует результату тип, который наша реализация CriteriaQuery возвращает после выполнения. Таким образом, используя преимущества обобщения, API Критериев позволяет нам писать безопасный с точки зрения типов код.

После того как мы получим реализацию CriteriaQuery, из нее мы сможем получить экземпляр класса, реализующего интерфейс javax.persistence.criteria.Root. Реализация Root диктует, какие JPA-сущности мы будем запрашивать. Она походит на запрос FROM JPQL (и SQL).

Следующие две строки в нашем примере используют преимущества другого нововведения в спецификацию JPA – *API Метамодели* (Metamodel API). Чтобы использовать возможности API Метамодели, мы должны получить реализацию интерфейса

`javax.persistence.metamodel.Metamodel`, вызывая метод `getMetamodel()` на нашем `EntityManagerFactory`.

Из нашей реализации `Metamodel` мы можем получить экземпляр интерфейса обобщенного типа `javax.persistence.metamodel.EntityType`. Обобщенный тип аргумента указывает JPA-сущность, которой соответствует наша реализация `EntityType`. `EntityType` позволяет нам просматривать персистентные (сохраняемые) атрибуты наших JPA-сущностей во время выполнения. Это именно то, что мы делаем в следующей строке нашего примера. В нашем случае мы получаем экземпляр `SingularAttribute`, который отображается на простой обособленный атрибут в нашей JPA-сущности. У `EntityType` имеются методы для получения атрибутов, которые отображаются на коллекции, наборы, списки и карты. Получение этих типов атрибутов очень похоже на получение `SingularAttribute`, поэтому мы не будем рассматривать их непосредственно. Обратитесь к документации по API Java EE 6 (<http://docs.oracle.com/javase/6/docs/api/>) для получения дополнительной информации.

Как видно из нашего примера, `SingularAttribute` содержит два аргумента обобщенного типа. Первый аргумент определяет JPA-сущность, с которой мы работаем, а второй указывает тип атрибута. Мы получаем наш `SingularAttribute`, вызывая метод `getDeclaredSingularAttribute()` на нашей реализации `EntityType` и передавая наименование атрибута (объявленного в нашей JPA-сущности) как строку.

После того как мы получим нашу реализацию `SingularAttribute`, мы должны получить реализацию `import javax.persistence.criteria.Path`, вызывая метод `get()` в нашем экземпляре `Root` и передавая наш `SingularAttribute` в качестве параметра.

В нашем примере мы получим список всех «новых» штатов Соединенных Штатов Америки (т. е. всех штатов, имена которых начинаются с «New»). Конечно, это работа для `like`-условия. Мы можем сделать это с помощью API Критериев, вызывая метод `like()` на нашей реализации `CriteriaBuilder`. Метод `like()` принимает нашу реализацию `Path` в качестве своего первого параметра и значение для поиска – в качестве своего второго параметра.

У `CriteriaBuilder` есть много методов, которые походят на предложения SQL и JPQL, в частности `equals()`, `greaterThan()`, `lessThan()`, `and()`, `or()` и др. (для ознакомления с полным списком обратитесь к документации Java EE 6 по адресу: <http://docs.oracle.com/javaee/6/api/>). Эти методы могут быть объединены для создания сложных запросов с помощью API Критериев.

Метод `like()` в `CriteriaBuilder` возвращает реализацию интерфейса `javax.persistence.criteria.Predicate`, который мы должны передать методу `where()` в нашей реализации `CriteriaQuery`. Этот метод возвращает новый экземпляр `CriteriaBuilder`, который мы присваиваем переменной `criteriaBuilder`.

Теперь мы готовы создать запрос. Работая с API Критериев, мы имеем дело с интерфейсом `javax.persistence.TypedQuery`, который может считаться безопасной (с точки зрения типа) версией интерфейса `Query`, который мы используем с JPQL.

Мы получаем экземпляр TypedQuery, вызывая метод `createQuery()` на EntityManager, и передаем ему нашу реализацию CriteriaQuery в качестве параметра.

Для получения результатов нашего запроса в виде списка мы просто вызываем метод `getResultSet()` на нашей реализации TypedQuery. Следует еще раз подчеркнуть, что API Критериев безопасен с точки зрения типов. Поэтому попытка присвоения результатов `getResultSet()` списку неправильного типа имела бы результатом ошибку компиляции.

После построения, упаковки и развертывания кода и указания в адресной строке обозревателя URL нашего сервлета мы должны будем увидеть список всех «новых» штатов в окне обозревателя.

Поддержка проверки допустимости со стороны бинов

Другая новая функция, введенная в JPA 2.0, – поддержка стандартом JSR 303 проверки допустимости со стороны бинов. Поддержка проверки допустимости бинами позволяет нам декорировать наши JPA-сущности аннотациями проверки допустимости со стороны бинов. Эти аннотации позволяют нам легко проверять введенные пользователем данные и выполнять их «санитарную обработку».

Использовать преимущества проверки допустимости со стороны бинов очень просто. Для этого нам нужно всего лишь декорировать наши поля JPA-сущности или методы геттеров любой из аннотаций проверки допустимости, определенных в пакете `javax.validation.constraints`. После того как наши поля будут соответствующим образом декорированы, EntityManager будет препятствовать сохранению данных, не прошедших проверку допустимости.

Следующий пример кода представляет собой модифицированную версию JPA-сущности Customer, которую мы рассматривали выше в этой главе. Она модифицирована для использования преимуществ проверки допустимости со стороны бинов в некоторых ее полях.

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

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 мы использовали аннотацию `@NotNull`, чтобы предотвратить сохранение `firstName` и `lastName` нашей сущности со значениями `null`. Также была использована аннотация `@Size` для ограничения минимальной и максимальной длины этих полей.

Это все, что необходимо для того, чтобы воспользоваться преимуществами проверки допустимости со стороны бинов в JPA. Если код пытается сохранить или обновить экземпляр нашей сущности, не прошедшей декларированную проверку допустимости, будет выдано исключение типа `javax.validation.ConstraintViolationException` и сущность не будет сохранена.

Как мы видим, проверка допустимости со стороны бинов в значительной степени автоматизирует подтверждение правильности данных, освобождая нас от необходимости вручную писать код валидации.

Кроме двух аннотаций, обсужденных в предыдущем примере, пакет `javax.validation.constraints` содержит несколько дополнительных аннотаций, которые мы можем использовать, чтобы автоматизировать проверку допустимости наших JPA-сущностей. Для ознакомления с полным списком обратитесь к документации по API Java EE 6: <http://docs.oracle.com/javaee/6/api/>.

Резюме

В этой главе мы рассмотрели, как получить доступ к данным в базе данных через API Подключения к базе данных Java (Java Database Connectivity (JDBC)) и API Персистентности Java (Java Persistence API (JPA)).

Было показано, как получить данные из базы данных путем использования JDBC с помощью метода `executeQuery()`, определенного в интерфейсе `java.sql.PreparedStatement`. Также мы выяснили, как вставить, обновить и удалить данные в базе данных с помощью метода `executeUpdate()`, определенного в том же интерфейсе. Кроме того, было рассмотрено использование инъекции зависимости для внедрения `DataSource` в сущность.

Мы узнали, как установить класс Java в качестве сущности путем декорирования его аннотацией `@Entity`. Было показано, как отобразить сущность на таблицу базы данных через аннотацию `@Table` и как отобразить поля сущности на столбцы таблицы базы данных с помощью аннотации `@Column`. Мы обсудили объявление первичного ключа сущности через аннотацию `@Id`.

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

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

Дополнительно мы обсудили, как использовать составные первичные ключи JPA-сущности путем разработки пользовательских классов первичного ключа.

Также мы показали, как получить сущности из базы данных путем использования Языка запросов персистентности Java (Java Persistence Query Language (JPQL)).

Затем были рассмотрены новые средства JPA 2.0, такие как API Критериев, который позволяет нам создавать программные запросы JPA, API Метамодели, который позволяет нам использовать преимущества безопасности типов Java при работе с JPA, и Проверка допустимости со стороны бинов, которая позволяет нам легко проверять вводимые данные, просто аннотируя поля ввода нашей JPA-сущности.

6

JavaServer Faces

В этой главе мы расскажем о *JavaServer Faces (JSF)* – каркасе стандартных компонентов платформы Java EE. Java EE 6 включает JSF 2.0 (самую последнюю на момент написания книги версию JSF) в качестве каркаса стандартных компонентов пользовательского интерфейса. Читатели, знакомые с более ранними версиями JSF, заметят, что JSF 2.0 включает много новых функций для упрощения разработки приложений JSF. Примечательно, что в основе JSF 2.0 лежит несколько соглашений по конфигурации. Если мы будем следовать этим JSF-соглашениям, то нам не придется записывать много конфигурационной информации. В большинстве случаев запись конфигурации не понадобится вообще. Если учесть, что файл дескриптора развертывания `web.xml` является необязательным в Сервлете 3.0, это значит, что во многих случаях мы можем создать завершенное веб-приложение, не написав ни единой строчки XML-конфигурации. В связи с этим больше нет необходимости в создании файла `web.xml` или `faces-config.xml`.

Введение в JSF 2.0

JSF 2.0 представляет ряд улучшений для упрощения разработки приложений JSF. В следующих разделах мы объясним некоторые из этих новых функций.



Читатели, которые не знакомы с более ранними версиями JSF, возможно, не поймут следующие несколько разделов полностью; однако нет повода для беспокойства, поскольку к концу этой главы все встанет на свои места.

Фэйслеты

Одно из примечательных различий между JSF 2.0 и более ранними версиями состоит в том, что *Фэйслеты (Facelets)* теперь являются предпочтительной технологией представления для JSF. Более ранние версии JSF в качестве своей технологии представления по умолчанию использовали 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 несоответствие представляло некоторые трудности для разработчиков JSF 1.x-приложений.

Каркас JSF с самого начала был разработан для поддержки нескольких технологий представления. Чтобы воспользоваться этой возможностью, Джейкоб Хуком (Jacob Hookom) написал новую технологию представления специально для JSF. Он назвал ее *Фэйслетом* (Facelet). Фэйслет оказался настолько удачным, что де-факто стал стандартом для JSF. Экспертная группа JSF 2.0 приняла во внимание популярность Фэйслета и сделала его официальной технологией представления JSF 2.0.

Необязательный файл faces-config.xml

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

Стандарт Java EE 5 способствовал значительному уменьшению объема XML-конфигурации. Java EE 6 еще больше уменьшил объем конфигурационной информации, сделав конфигурационный файл JSF `faces-config.xml` вообще не обязательным в самой последней версии JSF.

В JSF 2.0 управляемые бины JSF могут быть сконфигурированы с помощью новой аннотации `@ManagedBean`, устранив потребность в их конфигурировании с помощью дескриптора `faces-config.xml`.

Кроме прочего, имеется соглашение по навигации JSF. Если значение атрибута `action` команды-ссылки или команды-кнопки JSF 2.0 соответствует имени фэйслета (минус расширение `xhtml`), то в соответствии с соглашением приложение перейдет к фэйслету, соответствующему имени действия. Это соглашение позволяет нам избегать необходимости конфигурировать навигацию приложения в файле дескриптора `faces-config.xml`.

Для большинства JSF 2.0-приложений файл `faces-config.xml` совершенно не нужен.

Стандартное расположение ресурсов

JSF 2.0 вводит стандартное расположение ресурсов. *Ресурсы* являются артефактами, которые страница или компонент JSF должны отобразить необходимым образом. Примерами ресурсов являются таблицы стилей CSS, файлы JavaScript и изображения.

В JSF 2.0 ресурсы могут быть помещены в подкаталог каталога `resources`, который в свою очередь помещается или в корне WAR-файла, или в его подкаталоге `META-INF`. В соответствии с соглашением компоненты JSF «знают», что могут получить ресурсы из одного из этих двух месторасположений.

Во избежание загромождения каталога `resources` ресурсы обычно помещаются в подкаталог. Этот подкаталог упоминается в атрибуте `library` компонентов JSF.

Например, мы хотим поместить таблицу стилей CSS под названием `styles.css` в `/resources/css/styles.css`. В нашей JSF странице мы можем получить этот CSS-файл, используя тег `<h:outputStylesheet>`:

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

Значение атрибута `library` должно соответствовать подкаталогу, где располагается наша таблица стилей. Точно так же у нас могут быть JavaScript-файлы в `/resources/scripts/somescript.js` и изображения в `/resources/images/logo.png`. Мы можем получить доступ к этим ресурсам следующим образом:

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

и

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

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

Разработка нашего первого JSF 2.0-приложения

Чтобы пояснить основные понятия JSF, мы разработаем простое приложение, состоящее из двух страниц фэйслетов и одного управляемого бина.

Фэйслеты

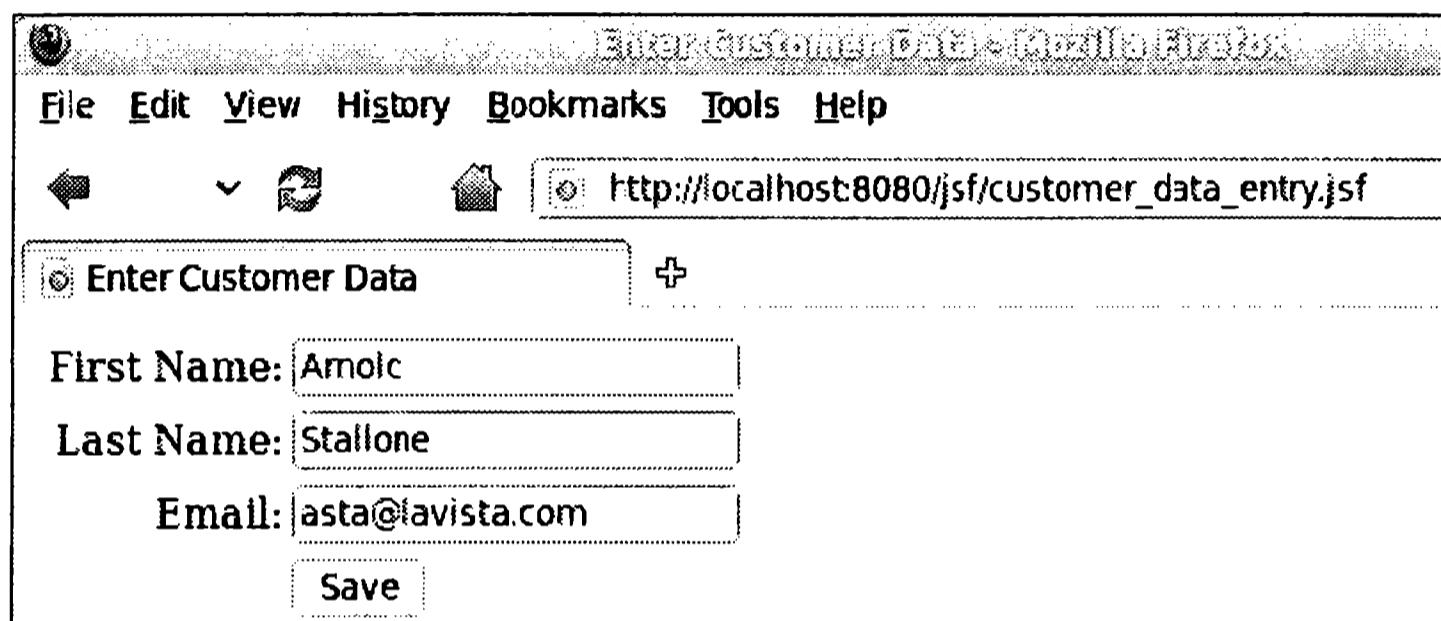
Как мы упоминали во введении к этой главе, технологией представления по умолчанию для JSF 2.0 являются фэйслеты. Фэйслеты должны быть написаны с использованием стандартного XML. Самый популярный способ разработки страниц фэйслетов – использование XHTML в сочетании с определенными пространствами XML-имен JSF. Следующий пример показывает, на что похожа типичная страница фэйслета:

```
<? 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>Ввод информации о заказчике</title>
    </h:head>
    <h:body>
        <h:outputStylesheet library="css" name="styles.css" target="body"/>
        <h:form>
            <h:messages></h:messages>
            <h:panelGrid columns="2" columnClasses="rightAlign, leftAlign">
                <h:outputText value="Имя:></h:outputText>
                <h:inputText label="Имя" value="#{customer.firstName}"
                    required="true">
                    <f:validateLength minimum="2" maximum="30">
                        </f:validateLength>
                </h:inputText>
                <h:outputText value="Фамилия:></h:outputText>
                <h:inputText label="Фамилия" 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:validateLength minimum="3" maximum="30">
        </f:validateLength>
    </h:inputText>
    <h:panelGroup></h:panelGroup>
    <h:commandButton action="confirmation" value="Сохранить">
        </h:commandButton>
    </h:panelGrid>
</h:form>
</h:body>
</html>
```

Следующий снимок экрана показывает, как эта страница представлена в обозревателе:



Конечно, этот снимок экрана был получен после редактирования некоторых данных в каждом текстовом поле; первоначально все текстовые поля были пустыми.

Любая фэйслет-страница JSF будет включать два пространства имен, показанные в примере. Первое пространство имен (`xmlns:h="http://java.sun.com/jsf/html"`) предназначено для тегов, 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 специфическими JSF-тегами в предыдущем примере являются теги `<h:head>` и `<h:body>`. Они напоминают стандартные HTML-теги `<head>` и `<body>` и представляются как таковые, когда страница выводится на экран в обозревателе.

Тег `<h:outputStylesheet>` является новым тегом JSF 2.0. Он используется для загрузки таблицы стилей CSS из известного расположения (JSF 2.0 стандартизирует расположение 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 мы хотим загрузить.

Следующий тег, который мы видим, – `<h:form>`. Этот тег генерирует HTML-форму для представления страницы. Как видно из примера, нет никакой потребности в указании атрибутов `action` или `method` для этого тега. Фактически для него нет ни атрибута `action`, ни атрибута `method`. Атрибут `action` для представленной HTML-формы будет сгенерирован автоматически, а атрибут `method` всегда будет иметь тип `post`.

Следующий тег в нашем примере – `<h:messages>`. Как явствует из его имени, он используется для вывода на экран любых сообщений. Как мы увидим в ближайшее время, JSF может автоматически генерировать сообщения проверки допустимости, которые будут выведены на экран в этом теге. Кроме того, программно могут быть добавлены произвольные сообщения с помощью метода `addMessage()`, определенного в `javax.faces.context.FacesContext`.

Еще один тег JSF в рассматриваемом нами примере – `<h:panelGrid>`. Этот тег является абсолютным эквивалентом HTML-таблицы, но работает немного по-другому. Вместо того чтобы объявлять строки и столбцы, тег `<h:panelGrid>` располагает атрибутом `columns`. Значение этого атрибута указывает число столбцов в таблице, представленной данным тегом. Поскольку мы помещаем в этом теге компоненты, они будут помещаться последовательно, пока число столбцов, определенных в атрибуте `columns`, не достигнет предела, после чего следующий компонент будет помещен в следующую строку. В приведенном примере значение атрибута `columns` равняется двум. Поэтому первые два 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 нашем примере два класса CSS (разделенные запятой) используются в качестве значения этого атрибута. Результат – присвоение первого класса CSS первому столбцу и второго класса – второму столбцу. Если бы было три или больше столбцов, третий получил бы первый класс CSS, четвертый – второй класс CSS и т. д., с чередованием первого и второго. Чтобы прояснить, как это работает, следующий фрагмент кода иллюстрирует часть исходного кода HTML-разметки, сгенерированного предыдущей страницей.

```
<table>
    <tbody>
        <tr>
            <td class="rightAlign">
                Имя:
            </td>
            <td class="leftAlign">
                <input type="text" name="j_idt8:j_idt12"/>
            </td>
        </tr>
        <tr>
            <td class="rightAlign">
                Фамилия:
            </td>
            <td class="leftAlign">
```



```

        <input type="text" name="j_idt8:j_idt14"/>
    </td>
</tr>
<tr>
    <td class="rightAlign">
        Email:
    </td>
    <td class="leftAlign">
        <input type="text" name="j_idt8:j_idt16"/>
    </td>
</tr>
<tr>
    <td class="rightAlign"></td>
    <td class="leftAlign">
        <input type="submit" name="j_idt8:j_idt18" value="Сохранить"/>
    </td>
</tr>
</tbody>
</table>

```

Обратите внимание, что каждый тег `<td>` имеет альтернативный тег CSS "rightAlign" или "leftAlign". Мы добились этого, присваивая значение "rightAlign, leftAlign" атрибуту `columnClasses` тега `<h:panelGrid>`. Следует отметить, что классы CSS, импользуемые в нашем примере, определяются в таблице стилей CSS, которую мы загрузили с помощью вышеописанного тега `<h:outputStylesheet>`.

С этого места в примере мы начинаем добавлять компоненты внутрь тега `<h:panelGrid>`. Эти компоненты будут отображаться в таблице, представленной `<h:panelGrid>`. Как уже упоминалось ранее, число столбцов, отображаемых в таблице, определяется атрибутом `columns` тега `<h:panelGrid>`. Поэтому нам не нужно беспокоиться о столбцах или строках – мы только добавляем компоненты, а они автоматически помещаются в правильное место.

Следующий тег, который мы видим, – `<h:outputText>`. Он похож на базовый тег JSTL `<c:out>` и выводит текст или выражение, которое будет отображено на странице, из своего атрибута `value`.

Далее у нас показан тег `<h:inputText>`. Он генерирует текстовое поле в отображаемой странице. Его атрибут `label` используется для любых сообщений проверки допустимости. Он сообщает пользователю, к какому полю относится выведенное сообщение.

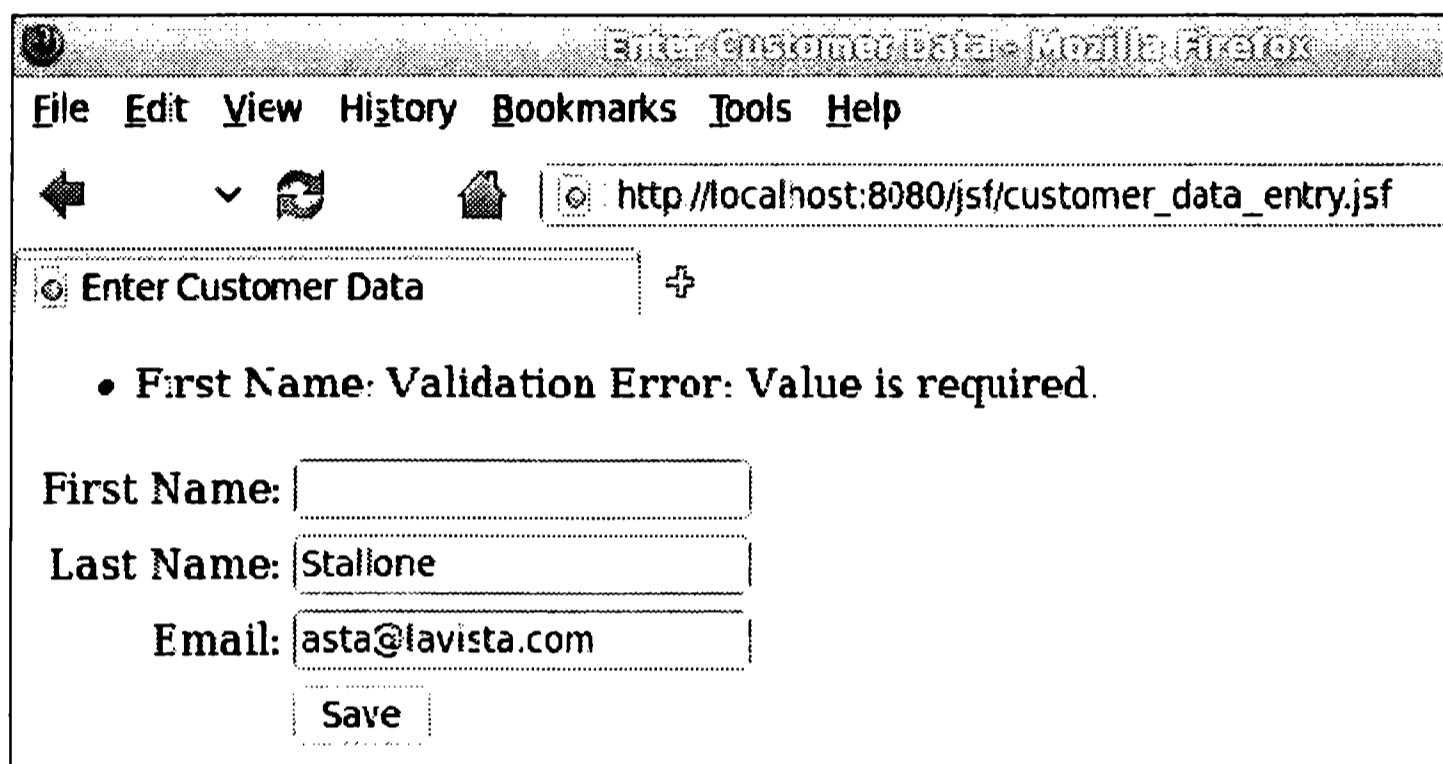


Хотя и не требуется, чтобы значение атрибута `label` тега `<h:inputText>` соответствовало метке, выведенной на экран на странице, настоятельно рекомендуется использовать это значение. В случае возникновения ошибки пользователю будет ясно, к какому полю относится сообщение.

Особенный интерес представляет атрибут `value` тега. Мы видим, что в качестве значения данного атрибута выступает значение выражения привязки (value binding expression). Это означает, что значение связано со свойством одного из управляемых бинов приложения. В нашем примере это конкретное текстовое поле привязывается к свойству, названному `firstName`, в управляемом бине `customer`.

Когда пользователь вводит значение этого текстового поля и отправляет форму, соответствующее свойство в управляемом бине обновляется этим значением. Атрибут `required` тега является необязательным; его допустимые значения – `true` или `false`. Если атрибут будет установлен в значение `true`, контейнер не позволит пользователю отправлять форму до тех пор, пока пользователь не введет некоторые данные в текстовое поле. Если пользователь попытается отправить форму, не вводя требуемые значения, то страница будет перезагружена и на экран будет выведено сообщение об ошибке в теге `<h:messages>`.

Это можно увидеть на следующем снимке экрана:



Здесь представлено сообщение об ошибке по умолчанию, которое появляется, когда пользователь пытается сохранить форму в примере, не вводя значение для имени заказчика. Первая часть сообщения («First Name») получена из значения атрибута `label` соответствующего тега `<h:inputTextField>`. Текст сообщения может редактироваться 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 генерируют вывод в журнал сервера GlassFish. Распространенная ошибка, которую делают новички при разработке на JSF, заключается в том, что они не добавляют теги `<h:messages>` на свои страницы. В результате, если проверка допустимости не прошла, кажется, что навигация, перестала работать без всякой причины (на экране отображается та же самая страница, и при этом без тега `<h:messages>` никакие сообщения об ошибках не выводятся).

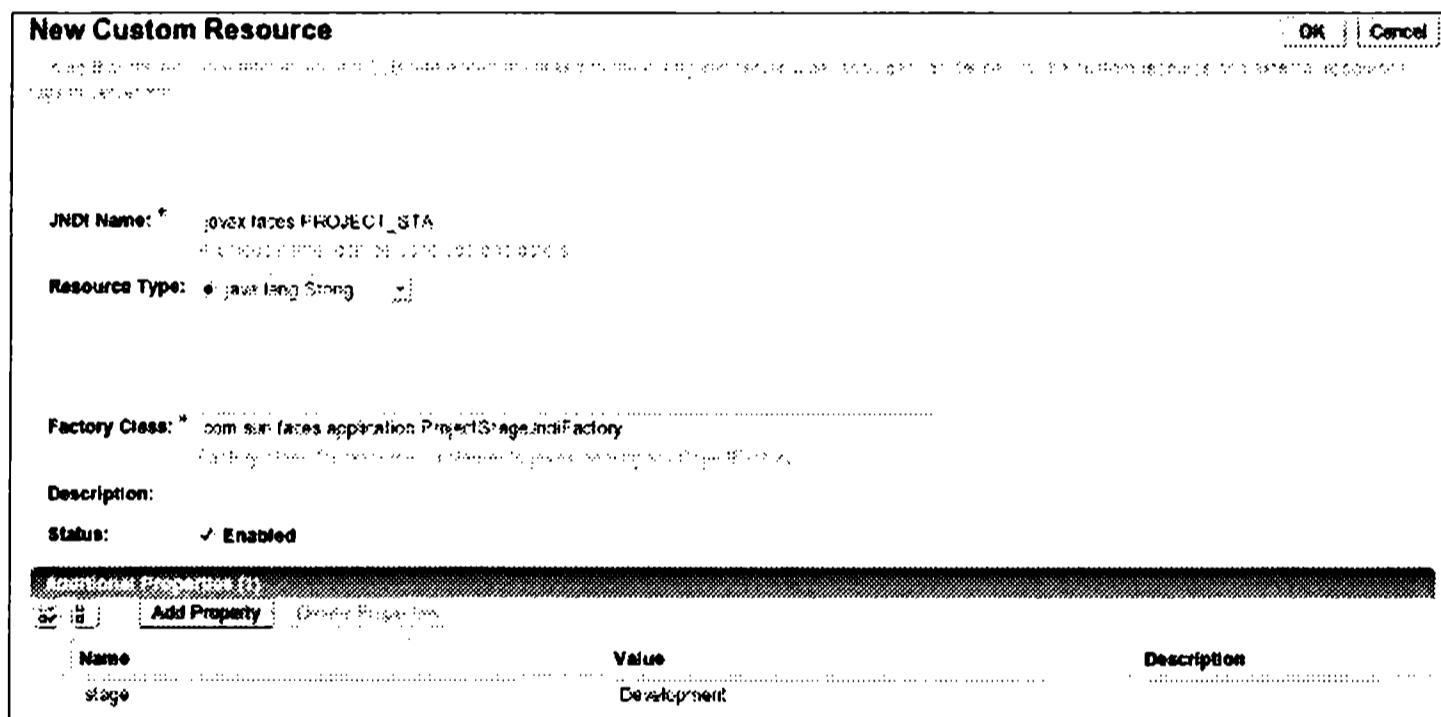
Во избежание подобной ситуации JSF 2.0 вводит понятие *этапов проекта* (*project stages*).

В JSF 2.0 определяются следующие этапы проекта:

- Производство (Production);
- Разработка (Development);
- Модульное тестирование (UnitTest);
- Системное тестирование (SystemTest).

Мы можем определить этап проекта как параметр инициализации фэйсис-сервлета в файле web.xml или как пользовательский ресурс JNDI. Поскольку web.xml теперь является необязательным и его изменение относительно легко выполнить при использовании неправильной стадии проекта или в случае, если мы забываем его модифицировать при перемещении нашего кода из одной среды в другую, то предпочтительным способом установки этапа проекта является использование пользовательского ресурса JNDI.

В GlassFish можно сделать это, войдя в веб-консоль и переместившись к **JNDI | Пользовательские ресурсы (Custom Resources)**, затем щелкнув по кнопке **Новый... (New...)**.



На странице, появившейся на экране после вышеописанных действий, мы должны ввести следующую информацию:

Наименование поля	Значение
Имя JNDI (JNDI Name)	javax.faces.PROJECT_STAGE
Тип ресурса (Resource Type)	java.lang.String
Класс фабрики (Factory Class)	com.sun.faces.application.ProjectStageJndiFactory

А затем добавить новое свойство с именем stage (этап) и значением, соответствующим этапу проекта, которое мы хотим использовать.

Установка этапа проекта позволяет нам выполнять некоторую логику, только если мы работаем в рамках конкретного этапа. Например, в одном из наших управляемых бинов может быть код, который выглядит следующим образом:

```
FacesContext facesContext = FacesContext.getCurrentInstance();
Application application = facesContext.getApplication();
if (application.getProjectStage().equals(ProjectStage.Production))
{
    // Исполняется заглушка этапа производства
}
else if (application.getProjectStage().equals(ProjectStage.Development))
{
    // Исполняется заглушка этапа разработки
}
else if (application.getProjectStage().equals(ProjectStage.UnitTest))
{
    // Исполняется заглушка модульного теста
}
else if (application.getProjectStage().equals(ProjectStage.SystemTest))
{
    // Исполняется заглушка системного теста
}
```

Как видно из приведенного фрагмента кода, этапы проекта позволяют нам модифицировать поведение нашего кода для различного окружения. Более того, установка стадии проекта позволяет механизму JSF вести себя немного по-разному в зависимости от установленного этапа проекта. Применительно к обсуждаемому нами случаю установка этапа проекта в значение `development` приводит к появлению дополнительных операторов журналирования событий в журнале сервера приложений. Поэтому, если мы забудем добавить тег `<h:messages>` к нашей странице, а этапом нашего проекта будет `development` и возникнут сбои проверки допустимости, мы увидим примерно такие записи в журнале сервера GlassFish:

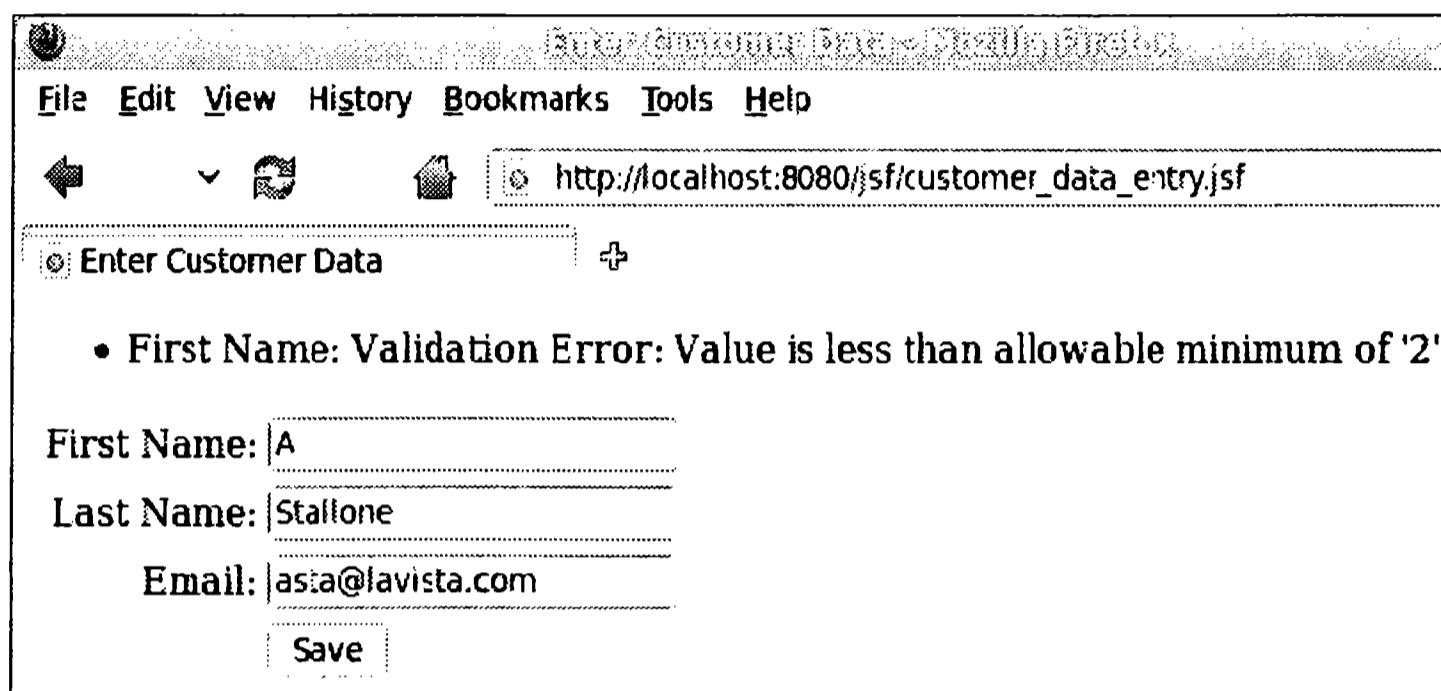
```
INFO: WARNING: FacesMessage(s) have been enqueued, but may not have
been displayed.
sourceId=j_idt8:j_idt11[severity=(ERROR 2), summary=(Имя:
Validation Error: Value is required.), detail=(Имя: Validation
Error: Value is required.)]
sourceId=j_idt8:j_idt13[severity=(ERROR 2), summary=(Фамилия:
Validation Error: Value is required.), detail=(Фамилия: Validation
Error: Value is required.)]
sourceId=j_idt8:j_idt15[severity=(ERROR 2), summary=(Email: Validation
Error: Value is less than allowable minimum of '3'), detail=(Email:
Validation Error: Value is less than allowable minimum of '3')]
```

На этапе Производства этот вывод по умолчанию не отправляется в журнал, оставляя нас в недоумении относительно того, почему навигация на нашей странице не работает.

Проверка допустимости

Обратите внимание, что у каждого тега `<h:inputField>` имеется вложенный тег `<f:validateLength>`. Как явствует из его имени, он проверяет значение, вводимое

в текстовое поле, на соответствие условию его длины между минимальным и максимальным значением. Минимальное и максимальное значения определяются атрибутами `minimum` и `maximum` тега. Тег `<f:validateLength>` является одним из стандартных блоков проверки допустимости, включенных в JSF. Точно так же, как атрибут `required` тега `<h:inputText>`, JSF автоматически выведет на экран сообщение об ошибке по умолчанию, когда пользователь попытается отправить форму с недопустимым значением.



Сообщения по умолчанию и их стиль могут быть переопределены. Мы рассмотрим, как это сделать, в следующем разделе.

В дополнение к `<f:validateLength>` JSF включает другие стандартные блоки проверки допустимости. Они перечислены в следующей таблице:

Тег проверки допустимости	Описание
<code><f:validateBean></code>	Проверка допустимости со стороны бина позволяет проверять значения управляемых бинов при использовании аннотаций в наших управляемых бинах без необходимости добавления блоков проверки допустимости в наши теги JSF. Этот тег позволяет нам выполнять тонкую настройку проверки допустимости со стороны бина в случае необходимости
<code><f:validateDoubleRange></code>	Проверяет вводимое вещественное число двойной точности (<code>Double</code>) на допустимость, связанную с его нахождением между двумя значениями включительно, указанными атрибутами <code>minimum</code> и <code>maximum</code> тега
<code><f:validateLength></code>	Проверяет длину вводимого текстового значения на предмет нахождения ее между значениями указанными атрибутами <code>minimum</code> и <code>maximum</code> тега

Тег проверки допустимости	Описание
<f:validateLongRange>	Проверяет вводимое целое число двойной точности (Long) на допустимость, связанную с его нахождением между двумя значениями включительно, указанными атрибутами minimum и maximum тега
<f:validateRegex>	Проверяет вводимое значение на предмет соответствия его регулярному выражению, образец которого установлен атрибутом pattern тега
<f:validateRequired>	Проверяет, что ввод не является пустым (empty). Этот тег эквивалентен установке атрибута required в значение true в родительском поле ввода

Обратите внимание, что в описании для тега <f:validateBean> мы кратко упоминали проверку допустимости со стороны бина. *Запрос на спецификацию Java (Java Specification Requests (JSR))* стремится стандартизировать проверку допустимости бином JavaBean. JavaBean используются в нескольких других API, которые до недавнего времени должны были реализовывать свою собственную логику проверки допустимости. Аналогично JPA 2.0 принятие JSF 2.0 в качестве стандарта - проверки допустимости со стороны бина помогает проверять свойства управляемых бинов.

Если мы собираемся использовать преимущества, предоставляемые проверкой допустимости бином, нам нужно просто декорировать требуемое поле соответствующей аннотацией *Проверки допустимости со стороны бина (bean validation)* – без необходимости явного использования блока проверки допустимости JSF.



Для ознакомления с полным списком аннотаций Проверки допустимости со стороны бина обратитесь к пакету javax.validation.constraints в API Java EE 6: <http://docs.oracle.com/javaee/6/api/>.

Группировка компонентов

Следующий новый тег в приведенном выше примере – <h:panelGroup>. Как правило, он используется для группировки нескольких компонентов таким образом, чтобы они заняли одну ячейку в <h:panelGrid>. Этого можно добиться путем добавления компонентов внутрь тега <h:panelGroup> и последующего его добавления к тегу <h:panelGrid>. Как видно из примера, у нашего экземпляра <h:panelGroup> нет никаких дочерних компонентов. В данном случае тег <h:panelGroup> предназначен для получения «пустой» ячейки и для того, чтобы заставить компонент <h:commandButton> выровняться со всеми другими полями ввода в форме.

Отправка формы

Отображением тега <h:commandButton> на код HTML является кнопка отправки (Submit), отображаемая в обозревателе. Точно так же, как и в случае со стандартным HTML, ее целью является отправка формы. Ее атрибут value просто

устанавливает метку кнопки. Атрибут `action` данного тега используется для навигации. Следующая страница для отображения основана на значении этого атрибута. У атрибута `action` могут быть строковая константа или *выражение метода связывания* (*method binding expression*), означающее, что он может указывать на метод в управляемом бине, который возвращает строку. Позже в этой главе мы увидим пример тега `<h:commandButton>`, атрибутом `action` которого является выражение метода связывания.

JSF 2.0 вводит новое соглашение. Если базовое имя страницы в нашем приложении соответствует значению атрибута `action` тега `<h:commandButton>`, то при нажатии кнопки мы перемещаемся к этой странице. Эта новая возможность JSF 2.0 освобождает нас от необходимости определять правила навигации, что мы обязательно должны были делать в JSF 1.x. В нашем примере страница подтверждения имеет название `confirmation.xhtml`. Поэтому, в соответствии с соглашением, эта страница будет показана при щелчке по кнопке, поскольку значение ее атрибута `action` соответствует базовому имени страницы ("confirmation").



Даже при том, что кнопка называется **Сохранить** (Save), в нашем простом примере щелчок по кнопке фактически не будет сохранять данные. Позже в этой главе мы увидим более перспективный вариант этого приложения, который реализует эту функциональность фактически.

Управляемые бины

В более ранних версиях JSF, используемых нами, мы должны были определять наши управляемые бины в конфигурационном файле под названием `faces-config.xml`. JSF 2.0 вводит новые аннотации, которые мы можем использовать в наших управляемых бинах, освобождающие нас от необходимости поддерживать отдельный конфигурационный файл. Следующий код является управляемым бином для нашего примера:

```
package net.enseode.glassfishbook.jsf;

import javax.faces.bean.ManagedBean;
@ManagedBean
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 @ManagedBean класса определяет этот бин в качестве управляемого бина JSF. У этой аннотации есть дополнительный атрибут name, который мы можем использовать, чтобы дать нашему бину логическое имя для использования его в нашем JSF. Однако, в соответствии с соглашением, значение этого атрибута идентично имени класса (в нашем случае Customer), за исключением того что первый его символ переключен в нижний регистр. В нашем примере мы устанавливаем это поведение заданным по умолчанию, поэтому получаем доступ к свойствам нашего бина через его логическое имя customer. Обратите внимание на значение любого из полей ввода, чтобы увидеть это логическое имя в действии. Также отметьте, что кроме аннотации @ManagedBean нет ничего особенного в этом бине – он является стандартным JavaBean со свойствами, объявленными как private, и соответствующими им методами геттеров и сеттеров.

Контексты управляемых бинов

У управляемых бинов всегда есть контекст. Контекст управляемых бинов определяет продолжительность их жизни. Он определяется аннотацией уровня класса. В следующей таблице приводятся все допустимые контексты управляемых бинов:

Аннотации контекстов управ- ляемых бинов	Описание
@ApplicationScoped	Один и тот же экземпляр управляемого бина в контексте приложения доступен всем клиентам нашего приложения. Если один из клиентов изменит значение управляемого бина в контексте приложения, то это изменение отразится на всех клиентах
@SessionScoped	Каждый экземпляр управляемого бина в контексте сеанса присваивается каждому из клиентов нашего приложения. Управляемый бин в контексте сеанса используется для хранения специфических данных клиента между запросами
@RequestScoped	Управляемый бин в контексте запроса «живет» только в течение одиночного HTTP-запроса

Аннотации контекстов управ- ляемых бинов	Описание
@ViewScoped	Управляемый бин в контексте представления ассоциируется с определенным представлением (страницей). Он будет уничтожен, как только пользователь перейдет к другому представлению
@NoneScoped	Управляемый бин без контекста создается, когда к нему получает доступ другой управляемый бин, обычно как к управляемому свойству
@CustomScoped	В JSF 2.0 у нас появилась возможность создавать пользовательские контексты для наших управляемых бинов. Атрибут value аннотации @CustomScoped должен разрешаться на карте контекста сеанса

Если никакой контекст не указывается в управляемом бине (как в нашем примере), то по умолчанию используется контекст запроса.

Навигация

Как видно из нашей страницы ввода, при щелчке по кнопке **Сохранить** (Save) на странице `customer_data_entry.xhtml` приложение перемещается к странице с названием `confirmation.xhtml`. Это происходит потому, что мы используем возможности соглашения по конфигурации JSF 2.0, в соответствии с которым, если значение атрибута `action` кнопки или ссылки соответствует базовому имени другой страницы, навигация перемещает нас к этой странице.



При щелчке по кнопке или ссылке, которая должна переместить нас к другой странице, отображается та же самая страница?

Если JSF не распознает значение атрибута `action` кнопки или ссылки, он по умолчанию перемещает нас к той же самой странице, которая отображалась в обозревателе на момент, когда пользователь щелкнул по кнопке или ссылке, предназначеннной для перемещения к другой странице.

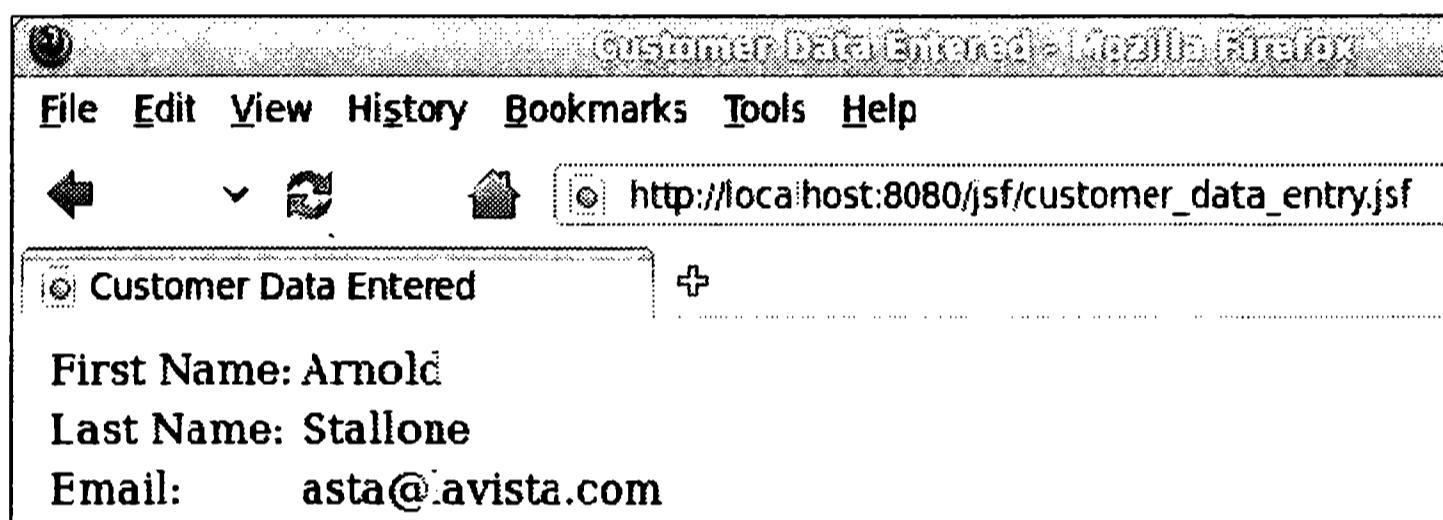
Если кажется, что навигация не работает должным образом, скорее всего, в значение этого атрибута вкрадлась опечатка. Помните, что по соглашению JSF будет искать страницу, базовое имя которой соответствует значению атрибута `action` кнопки или ссылки.

Исходный код для страницы `confirmation.xhtml` выглядит следующим образом:

```
<?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>Ввод информации о заказчике</title>
    </h:head>
    <h:body>
```

```
<h:panelGrid columns="2" columnClasses="rightAlign, leftAlign">
    <h:outputText value="Имя:></h:outputText>
    <h:outputText value="#{customer.firstName}"></h:outputText>
    <h:outputText value="Фамилия:></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>
```

Здесь нет никаких тегов, которых бы мы не видели ранее в этой странице. Единственное, что обращает на себя внимание, – это то, что здесь используются выражения привязки значений в качестве значения для всех тегов `<h:outputText>`. Поскольку эти выражения привязки значений являются теми же выражениями, которые используются в предыдущей странице для тегов `<h:inputText>`, их значения будут соответствовать данным, которые ввел пользователь.



В традиционных веб-приложениях Java мы определяем шаблоны URL, которые будут обработаны определенным сервлетом. Конкретно для JSF обычно использовались суффиксы `.jsf` или `.faces`. Другим обычно используемым URL, отображаемым на JSF, был префикс `/faces`. По умолчанию GlassFish автоматически добавляет все три этих отображения к фэйсис-сервлету. Поэтому, если мы хотим использовать одно из этих отображений, мы вообще не должны указывать никакой URL отображения. Если по какой-либо причине мы должны указать другое отображение, то следует добавить конфигурационный файл `web.xml` к нашему приложению. Однако значения по умолчанию в большинстве случаев удовлетворят нас.

URL, который мы использовали для страниц нашего приложения, представляет собой имя страницы фэйслета, в котором суффикс `.xhtml` заменен суффиксом `.jsf`. Он использует возможность отображения URL по умолчанию. Мы также можем получить доступ к нашей странице путем использования расширения `.faces` или префикса `/faces/`.

Пользовательская проверка допустимости данных

JSF не только предоставляет нам стандартные блоки проверки допустимости, но и позволяет создавать нестандартные (пользовательские) элементы верификации.

Для этого предусмотрены два способа: создание класса нестандартного элемента верификации и добавление метода проверки допустимости в наши управляемые бины.

Создание нестандартных элементов верификации

В дополнение к стандартным блокам проверки допустимости JSF позволяет нам создавать нестандартные (пользовательские) элементы верификации путем создания класса Java, реализующего интерфейс `javax.faces.validator.Validator`.

Следующий класс реализует блок проверки допустимости электронной почты, который мы будем использовать для проверки поля текстового ввода электронной почты на нашей 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() + ": задан неверный формат email");
                throw new ValidatorException(facesMessage);
            }
        }
    }
}
```

Аннотация `@FacesValidator` регистрирует наш класс как класс нестандартного элемента верификации JSF. Значением его атрибута `value` является логическое имя, которое страницы JSF могут использовать для обращения к нему.

Как видно из приведенного примера, единственным методом, который мы должны реализовать при реализации интерфейса `Validator`, 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 мы проверяем; третий представляет собой фактическое значение. В примере мы приводим uiComponent к типу javax.faces.component.html.HtmlInputText. Таким образом, мы получаем доступ к его методу getLabel(), который сможем использовать в качестве части сообщения об ошибке.

Если введенное значение имеет недопустимый формат адреса электронной почты, создается новый экземпляр javax.faces.application.FacesMessage, передающий сообщение об ошибке в качестве параметра его конструктора, которое будет выведено на экран в обозревателе. Затем мы вызываем новое исключение javax.faces.validator.ValidatorException. После этого сообщение об ошибке выводится на экран в обозревателе. То, как оно туда попадает, происходит «за кулисами» API JSF.



Общий блок проверки допустимости Apache

Предыдущий блок проверки допустимости использует общий блок проверки допустимости Apache для выполнения фактической проверки допустимости. Эта библиотека включает много общих проверок допустимости, таких как даты, номера кредитных карт, ISBN и электронные письма. При реализации нестандартного элемента верификации вначале стоит посмотреть, нет ли в этой библиотеке такого блока проверки допустимости, который подходит для нашего случая.

Чтобы использовать наш блок проверки допустимости в странице, мы должны воспользоваться тегом JSF <f:validator>. Следующая страница фэйслета является модифицированной версией экрана ввода данных о клиентах. Эта версия использует тег <f:validator> для проверки поля электронной почты:

```
<?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>Ввод информации о клиенте</title>
  </h:head>
  <h:body>
    <h:outputStylesheet library="css" name="styles.css" target="body"/>
    <h:form>
      <h:messages></h:messages>
      <h:panelGrid columns="2" columnClasses="rightAlign, leftAlign">
        <h:outputText value="Имя:></h:outputText>
        <h:inputText
            label="Имя"
            value="#{customer.firstName}"
            required="true">
          <f:validateLength minimum="2" maximum="30">
          </f:validateLength>
        </h:inputText>
        <h:outputText value="Фамилия:></h:outputText>
        <h:inputText
            label="Фамилия"
            value="#{customer.lastName}"
            required="true">
          <f:validateLength minimum="2" maximum="30">
          </f:validateLength>
        </h:inputText>
        <h:outputText value="Email:></h:outputText>
        <h:inputText
```

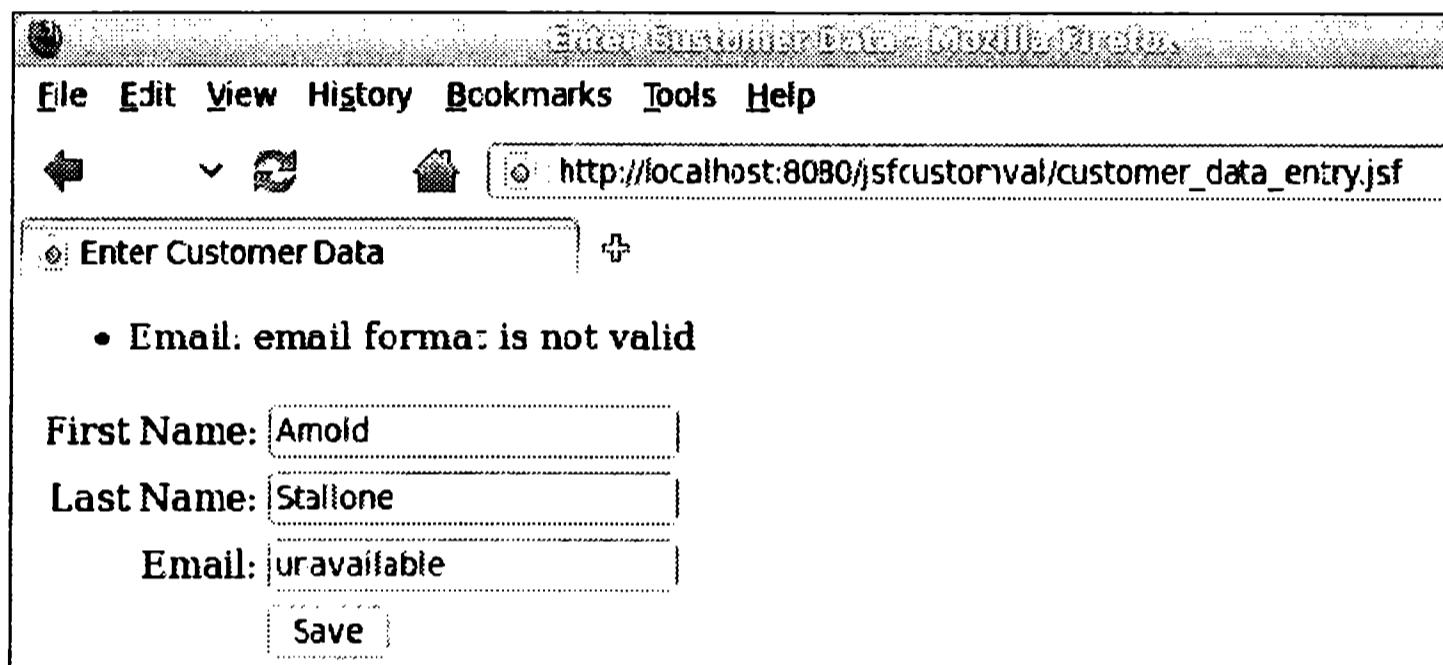


```

        label="Email"
        value="#{customer.email}"
        <f:validator validatorId="emailValidator"/>
    </h:inputText>
    <h:panelGroup></h:panelGroup>
    <h:commandButton action="confirmation" value="Сохранить">
    </h:commandButton>
</h:panelGrid>
</h:form>
</h:body>
</html>

```

После написания нашего нестандартного элемента верификации и изменения нашей страницы для использования его возможностей мы сможем увидеть наш блок проверки допустимости в действии:



Методы блока проверки допустимости

Мы можем иначе реализовать пользовательскую проверку допустимости, добавляя методы проверки допустимости в один или более управляемых бинов приложения. Следующий класс Java поясняет использование методов блока проверки допустимости, выполняющих проверку допустимости JSF:

```

package net.ensode.glassfishbook.jsfcustomval;

import javax.faces.application.FacesMessage;
import javax.faces.bean.ManagedBean;
import javax.faces.component.UIComponent;
import javax.faces.component.html.HtmlInputText;
import javax.faces.context.FacesContext;
import javax.faces.validator.ValidatorException;
import org.apache.commons.lang.StringUtils;
@ManagedBean
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() + ": допустимы только буквенные символы.");
        throw new ValidatorException(facesMessage);
    }
}
```

В этом примере класс содержит только метод блока проверки допустимости. Мы можем дать нашему методу блока проверки допустимости любое имя на свое усмотрение. Однако возвращаемое им значение должно быть пустым; кроме того, три входных параметра, показанные в примере, должны использоваться в указанном порядке. Другими словами, за исключением имени метода сигнатура метода блока проверки допустимости должна быть идентичной сигнатуре метода validate(), определенного в интерфейсе javax.faces.validator.Validator.

Тело этого метода блока проверки допустимости почти идентично телу нашего метода validate() в нестандартном элементе верификации. Мы проверяем значение, вводимое пользователем, чтобы удостовериться, что оно содержит только буквенные символы и/или пробелы. Если это не так, мы вызываем исключение ValidatorException, передавая экземпляр FacesMessage, содержащий соответствующую строку сообщения об ошибке.



StringUtils

В примере мы использовали валидатор org.apache.commons.lang.StringUtils для выполнения фактической логики проверки допустимости. В дополнение к методу, используемому в примере, данный класс содержит несколько методов, позволяющих выяснить, какой является строка – числовой или алфавитно-цифровой. Этот класс, являющийся частью библиотеки Apache commons-lang, очень полезен при написании нестандартных элементов верификации.

Поскольку каждый метод блока проверки допустимости должен находиться в управляемом бине, мы должны удостовериться, что класс, содержащий наш метод блока проверки допустимости, декорируется аннотацией @ManagedBean, как поясняется в нашем примере.

Последнее, что нам необходимо сделать для использования нашего метода блока проверки допустимости, – связать его с нашим компонентом с помощью атрибута validator тега:

```
<?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>Ввод информации о заказчике</title>
    </h:head>
    <h:body>
      <h:outputStylesheet library="css" name="styles.css" target="body"/>
      <h:form>
        <h:messages></h:messages>
        <h:panelGrid columns="2" columnClasses="rightAlign, leftAlign">
          <h:outputText value="Имя:></h:outputText>
          <h:inputText
            label="Имя"></h:inputText>
        </h:panelGrid>
      </h:form>
    </h:body>
  </html>
```



```

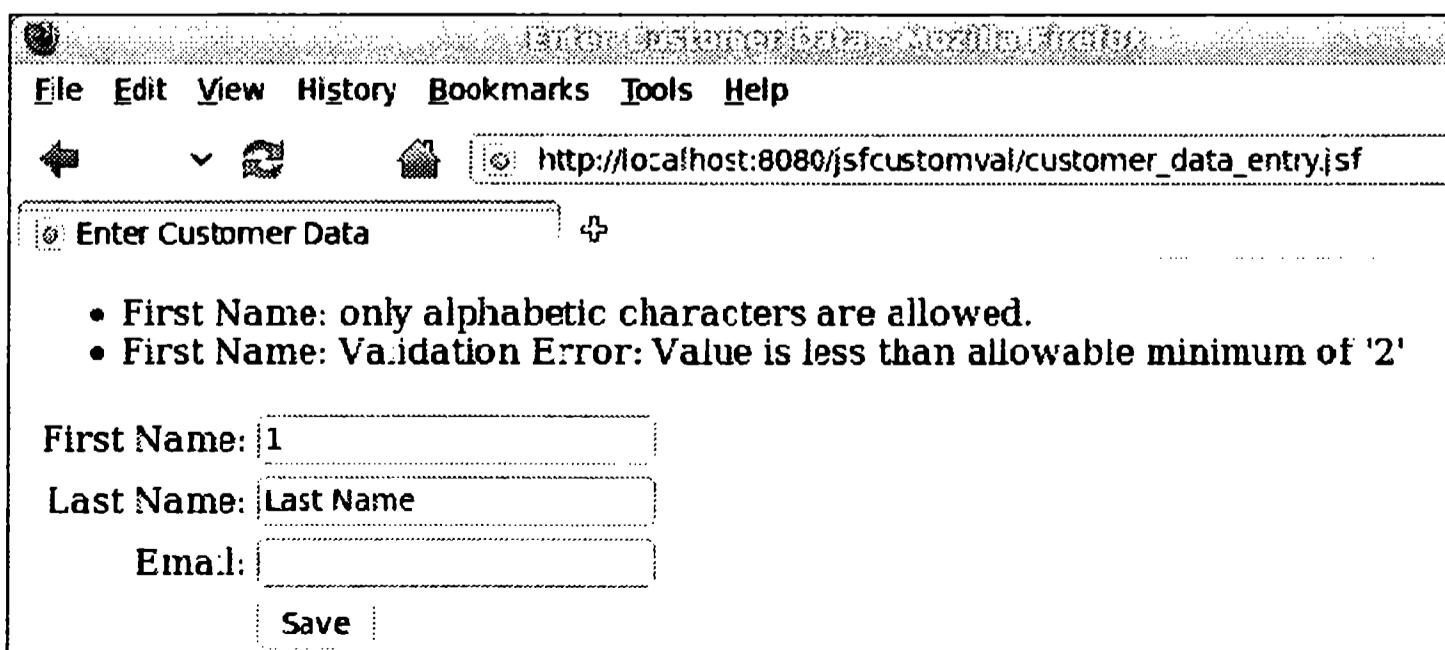
        value="#{customer.firstName}"
        required="true"
        validator="#{alphaValidator.validateAlpha}">
    <f:validateLength minimum="2" maximum="30">
    </f:validateLength>
</h:inputText>
<h:outputText value="Фамилия:></h:outputText>
<h:inputText
    label="Фамилия"
    value="#{customer.lastName}"
    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="Сохранить">
</h:commandButton>
</h:panelGrid>
</h:form>
</h:body>
</html>

```

Ни поле **Имя** (First Name), ни поле **Фамилия** (Last Name) не должны принимать что-либо кроме буквенных символов или пробелов, поскольку мы добавили наш метод нестандартного элемента верификации в оба этих поля.

Обратите внимание, что значение атрибута `validator` блока проверки допустимости тега `<h:inputText>` является языком выражений JSF, по умолчанию использующим имя управляемого бина для бина, содержащего наш метод проверки допустимости. Наш бин имеет имя `alphaValidator`, а метод нашего блока проверки допустимости называется `validateAlpha`.

После изменения страницы для использования нашего нестандартного элемента верификации мы сможем увидеть его в действии:



Обратите внимание, что для поля **Имя** (First Name) оказались выполненными: и наше сообщение нестандартного элемента верификации, и стандартный блок проверки допустимости длины.

Реализация методов блока проверки допустимости имеет преимущество, связанное с отсутствием издержек, возникающих при создании целого класса только для единственного метода блока проверки допустимости (в нашем примере делается только это, но во многих случаях методы блока проверки допустимости добавляются к существующему управляемому бину, содержащему другие методы). Тем не менее у такого подхода есть и недостаток: каждый компонент может быть проверен только единственным методом блока проверки допустимости. При использовании классов блока проверки допустимости несколько тегов `<f:validator>` могут быть вложены в тег, который будет проверен, поэтому для поля может быть выполнено несколько проверок допустимости – как пользовательских, так и стандартных.

Настройка сообщений JSF по умолчанию

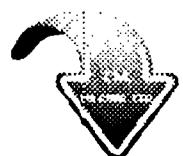
Как мы упоминали в предыдущем разделе, можно настроить стиль (шрифт, цвет, текст и т. д.) сообщений проверки допустимости JSF по умолчанию. Кроме того, можно отредактировать текст сообщений проверки допустимости JSF по умолчанию. В следующих разделах мы объясним, как изменить текст и форматирование сообщения об ошибке.

Настройка стилей сообщения

Настройка стилей сообщения может быть выполнена через *Каскадные таблицы стилей* (Cascading Style Sheets (CSS)). Она может производиться путем использования атрибутов `style` или `styleClass` тега `<h:message>`. Атрибут `style` используется, когда мы хотим объявить встроенный стиль CSS, а атрибут `styleClass` – когда мы хотим использовать предопределенный стиль в таблице стилей CSS или в теге `<style>` на нашей странице.

Следующая разметка поясняет использование атрибута `styleClass` для изменения стиля сообщений об ошибках. Это модифицированная версия страницы ввода, которую мы видели 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>Ввод информации о заказчике</title>
    </h:head>
    <h:body>
```



```

<h:outputStylesheet library="css" name="styles.css" target="body"/>
<h:form>
    <h:messages styleClass="errorMsg"></h:messages>
    <h:panelGrid columns="2" columnClasses="rightAlign, leftAlign">
        <h:outputText value="Имя:"></h:outputText>
        <h:inputText
            label="Имя"
            value="#{customer.firstName}"
            required="true"
            validator="#{alphaValidator.validateAlpha}">
            <f:validateLength minimum="2" maximum="30">
            </f:validateLength>
        </h:inputText>
        <h:outputText value="Фамилия:"></h:outputText>
        <h:inputText
            label="Фамилия"
            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="Сохранить">
        </h:commandButton>
    </h:panelGrid>
</h:form>
</h:body>
</html>

```

Как видно из приведенной разметки, единственная разница между этой страницей и предыдущей состоит в использовании атрибута `styleClass` тега `<h:messages>`. Как мы упоминали ранее, значение атрибута `styleClass` должно соответствовать имени стиля CSS, определенного в каскадной таблице стилей, к которой у нашей страницы есть доступ.

В нашем случае мы определили стиль CSS для сообщений следующим образом:

```

errorMsg
{
    color: red;
}

```

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

Следующий снимок экрана поясняет, как сообщения об ошибках проверки допустимости выглядят после реализации этого изменения:

File Edit View History Bookmarks Tools Help

← → ⌛ ⌂ http://localhost:8080/jsfcustommess/customer_data_entry.jsf

Enter Customer Data +

- First Name: Validation Error: Value is less than allowable minimum of '2'
- Last Name: Validation Error: Value is less than allowable minimum of '2'
- Email: email format is not valid

First Name: A

Last Name: B

Email: C

Save

В данном случае мы изменили лишь цвет текста сообщения об ошибке – с черного на красный; но на самом деле в установке стиля сообщений об ошибках мы ограничены только возможностями CSS.



У любого стандартного компонента JSF есть и атрибут `style`, и атрибут `styleClass`, который может использоваться для изменения его стиля. Первый используется для встроенных стилей CSS, последний – для предопределенных стилей CSS.

Настройка текста сообщения

Иногда желательно переопределить сообщения выдаваемые по умолчанию об ошибках проверки допустимости JSF. Ошибки проверки допустимости выдаваемые по умолчанию, определяются в пакете ресурса `Messages.properties`. Этот файл можно найти внутри файла `jsf-api.jar`, который в свою очередь находится в каталоге [Каталог установки *Glassfish*] `/glassfish/modules`. Файл `Messages.properties` находится внутри JAR-файла `jsf-api.jar`, в его каталоге `javax.faces`. Файл содержит несколько сообщений; здесь нас интересуют только сообщения об ошибках проверки допустимости. По умолчанию они определяются следующим образом:

```
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 выдаваемые по умолчанию, мы должны создать наш собственный пакет ресурса, используя те же самые ключи, что и используемые 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 мы переопределяем сообщение об ошибке для случая, когда введенное в поле значение, проверенное тегом <f:validateLength>, окажется короче допустимого минимума. Чтобы сообщить нашему приложению, что у нас есть пользовательский пакет ресурса для свойств сообщений, мы должны модифицировать файл конфигурации приложения faces-config.xml.

```

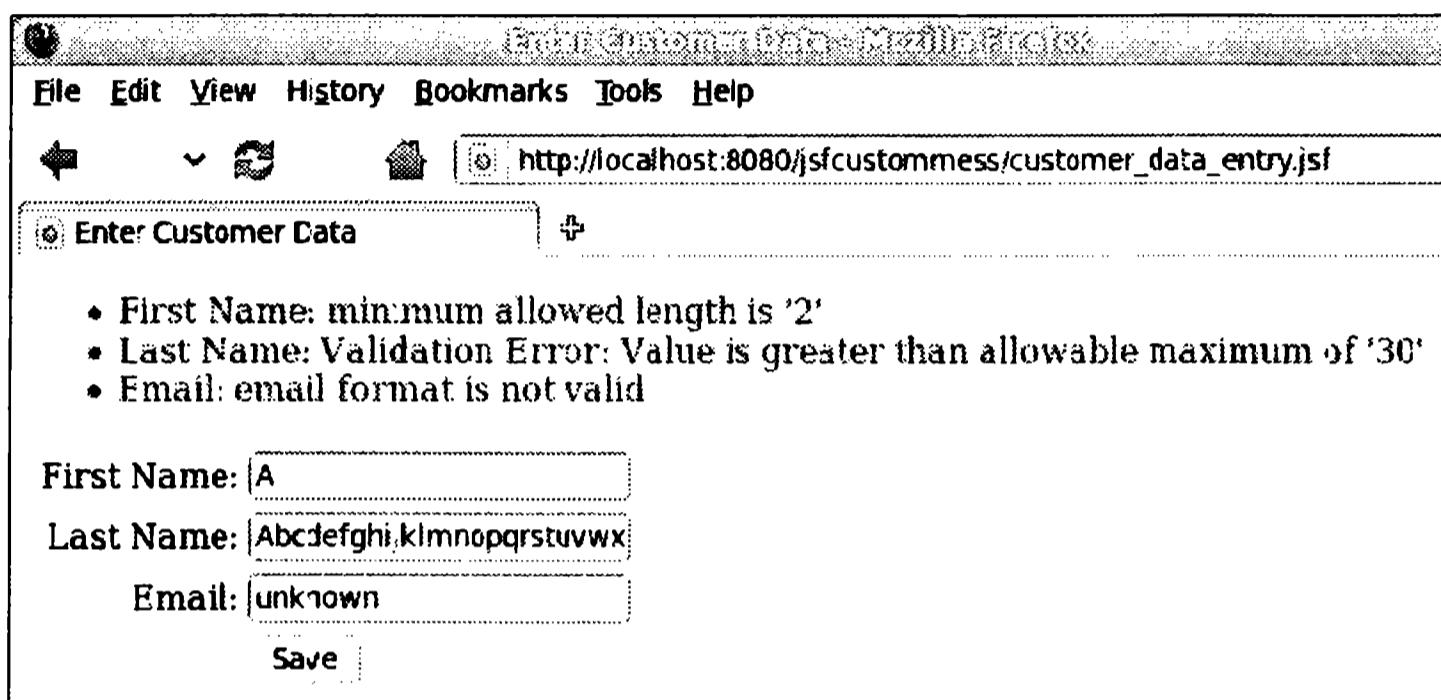
<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="2.0"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd">
    <application>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

Как видно из приведенной разметки, нам нужно всего лишь добавить к файлу конфигурации приложения faces-config.xml элемент <message-bundle>, который указывает название и местоположение пакета ресурса, содержащего наши пользовательские сообщения.



Определение пользовательских текстовых сообщений об ошибках – один из тех немногих случаев, когда мы все еще должны создавать файл faces-config.xml для приложения JSF 2.0. Однако заметьте, насколько прост наш файл faces-config.xml. Он сильно отличается от типичного файла faces-config.xml для JSF 1.x, который обычно содержит определения управляемых бинов, правила навигации, определения блока проверки допустимости JSF и т. д.

После добавления нашего пользовательского пакета ресурса сообщений и изменения конфигурационного файла приложения faces-config.xml мы сможем увидеть сообщения пользовательской проверки допустимости в действии:



Как видно из приведенного снимка экрана, если мы не переопределили сообщения проверки допустимости, то на экран все еще будет выведено сообщение об ошибке по умолчанию. В нашем пакете ресурса мы переопределяли только сообщение об ошибке проверки допустимости минимальной длины ввода, поэтому наше пользовательское сообщение об ошибке будет показано для текстового поля **Имя** (First Name). Поскольку мы не переопределяли сообщение об ошибке для ввода данных, превышающего максимально допустимую длину, то показано сообщение об ошибке по умолчанию. Блок проверки допустимости электронной почты является нестандартным элементом верификации, который мы разработали выше в этой главе. Поскольку это пользовательский элемент верификации, на его сообщения об ошибке наш пакет ресурса влияния не оказывает.

Интеграция JSF и JPA

К настоящему моменту мы рассмотрели многие из функций JSF. Однако наше приложение, приведенное в качестве примера, все еще фактически не сохраняет данные. В этом разделе мы рассмотрим, как JavaServer Faces (JSF) и API Персистентности Java (Java Persistence API (JPA)) могут быть легко интегрированы для сохранения в базе данных информации, введенной пользователем.

Мы также рассмотрим дополнительные функции JSF и JPA. Применительно к JSF мы обсудим, как выполнить немного логики на сервере прежде, чем переместиться к другой странице, и как автоматически заполнить свойство управляемого бина с

помощью аннотации `@ManagedProperty`. Применительно к JPA мы обсудим, как автоматически генерировать первичные ключи.

Как уже было показано в этой главе, управляемый бин JSF является всего лишь стандартным бином JavaBean. В главе 5 мы узнали, что JPA использует стандартные бины JavaBean для объектно-реляционного отображения. Поскольку и управляемые бины JSF, и бины JPA являются стандартными бинами JavaBean, ничто не мешает нам использовать JPA-бины в качестве управляемых бинов JSF.

Как мы уже говорили, теги JSF могут содержать значения выражений связывания, которые используются для автоматического заполнения управляемых бинов при отправке формы. Если мы используем JPA-бин в качестве управляемого бина, свойства бина будут заполнены этим способом. Затем мы можем просто вызвать метод `EntityManager.persist()`, чтобы сохранить данные в базе данных.

Первое, что нам нужно сделать, – задействовать JPA-бин в качестве управляемого бина, который будет использоваться для значения выражения связывания.

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

import java.io.Serializable;
import javax.faces.bean.ManagedBean;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
@ManagedBean
@Entity
@Table(name = "CUSTOMERS")
public class Customer implements Serializable
{
    @Id
    @GeneratedValue
    @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()
    {
        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;
}
@Override
public String toString()
{
    Long localCustomerId = customerId;
    String localFirstName = firstName;
    String localLastName = lastName;
    String localEmail = email;
    if (localCustomerId == null)
    {
        localCustomerId = 0L;
    }
    if (localEmail == null)
    {
        localEmail = "";
    }
    if (localFirstName == null)
    {
        localFirstName = "";
    }
    if (localLastName == null)
    {
        localLastName = "";
    }
    String toString = "customerId = " + customerId + "\n";
    toString += "firstName = " + localFirstName + "\n";
    toString += "lastName = " + localLastName + "\n";
    toString += "email = " + localEmail;
    return toString;
}
}
```

Этот класс является почти точной копией бина `Customer`, который мы видели в главе 5; отличия от упомянутого бина заключаются в том, что он принадлежит другому пакету, и в том, что он декорируется аннотацией `@ManagedBean`. Аннотирование нашей JPA-сущности как управляемого бина JSF позволяет нам использовать его в значении выражения связывания на нашей странице. Поэтому мы можем заполнить его вводимыми пользователем данными, которые затем сможем сохранить в базе данных.

Обратите внимание, что мы декорировали первичный ключ нашей сущности аннотацией `@GeneratedValue`. Благодаря этому поле первичного ключа генерируется автоматически, освобождая нас от необходимости заполнять его вручную.

У JPA имеется несколько стратегий генерации первичного ключа. Если мы не указем ни одну из них, автоматически будет использоваться значение по умолчанию. Стратегия генерации первичного ключа по умолчанию изменяется в зависимости от СУРБД, которую мы используем. В случае с JavaDB, к которой мы прибегли в нашем примере, стратегией генерации первичного ключа по умолчанию будет TABLE, которая используется для генерации первичных ключей с автоматически сгенерированной таблицей базы данных.

Если нас устраивает стратегия генерации первичного ключа по умолчанию, нам нужно только декорировать поле первичного ключа аннотацией `@GeneratedValue` для того, чтобы иметь первичные ключи. Если по какой-либо причине мы хотим использовать другую стратегию генерации, то мы должны указать ее с помощью атрибута `strategy` аннотации `@GeneratedValue`.

В следующей таблице перечислены все поддерживающие стратегии генерации первичного ключа. Обратите внимание, что не все стратегии поддерживаются всеми системами СУРБД.

Стратегия генерации	Описание
<code>GenerationType.AUTO</code>	Стратегия генерации для нас выбирается автоматически. Стратегия по умолчанию зависит от системы СУРБД и реализации JPA
<code>GenerationType.IDENTITY</code>	Используется столбец идентичности базы данных для генерации первичных ключей. Не все системы СУРБД поддерживают идентичность
<code>GenerationType.SEQUENCE</code>	Используется последовательность базы данных для генерации первичных ключей
<code>GenerationType.TABLE</code>	Первичный ключ получается из таблицы базы данных. Реализация JPA будет автоматически управлять этой таблицей, удостоверяясь, что мы получаем уникальное значение из нее каждый раз, когда генерируется первичный ключ

Для использования стратегии генерации первичного ключа, отличной от той, что предусмотрена по умолчанию, требуется просто указать соответствующую стратегию в качестве значения атрибута `strategy` аннотации `@GeneratedValue`. Например, если мы хотим использовать последовательность базы данных для генерации первичных ключей, нам нужно указать такое значение атрибута:

```
@GeneratedValue(strategy=GenerationType.SEQUENCE)
```

В результате используемая нами стратегия генерации первичного ключа будет изменена соответствующим образом.

Наш бин `Customer` выступает в роли модели (данных) в нашем приложении; наши страницы выступают в роли представления. Нам нужно добавить дополнительный управляемый бин, который будет использоваться в качестве контроллера, поскольку

мы будем придерживаться ранее принятой практики следования шаблону проектирования Модель-Представление-Контроллер.

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

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.annotation.Resource;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.ManagedProperty;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.PersistenceUnit;
import javax.sql.DataSource;
import javax.transaction.UserTransaction;
@ManagedBean
public class CustomerController
{
    @Resource(name = "jdbc/_CustomerDBPool")
    private DataSource dataSource;
    @PersistenceUnit(unitName = "customerPersistenceUnit")
    private EntityManagerFactory entityManagerFactory;
    @Resource
    private UserTransaction userTransaction;
    @ManagedProperty(value = "#{customer}")
    private Customer customer;
    public String saveCustomer()
    {
        String returnValue = "customer_saved";
        EntityManager entityManager = entityManagerFactory.
            createEntityManager();
        try
        {
            userTransaction.begin();
            entityManager.persist(customer);
            userTransaction.commit();
        }
        catch (Exception e)
        {
            e.printStackTrace();
            returnValue = "error_saving_customer";
        }
        return returnValue;
    }
    public Customer getCustomer()
    {
        return customer;
    }
    public void setCustomer(Customer customer)
    {
        this.customer = customer;
    }
}
```

Метод `saveCustomer()` в этом классе будет вызываться всякий раз, когда пользователь щелкает по кнопке **Сохранить** (Save) в HTML-форме. На странице, содержащей форму, должно быть произведено небольшое изменение; мы рассмотрим его коротко. Этот метод просто сохраняет данные, содержащиеся в бине `Customer` –

в базе данных. Обратитесь к главе 5 для получения подробной информации.

Особенно интересными для нас являются методы `setCustomer()` и `getCustomer()`. Эти методы не предназначены для прямого вызова разработчиками приложений. Вместо этого они должны быть вызваны JSF-реализацией GlassFish с соответствующим экземпляром бина `Customer`. Мы должны объявить свойство `customer` предыдущего контроллера как *управляемое свойство* (*managed property*). JSF 2.0 вводит аннотацию `@ManagedProperty`, которая может использоваться для объявления управляемого свойства бина.

Как видно из предыдущего примера, для того чтобы сделать свойство управляемым, мы должны декорировать его аннотацией `@ManagedProperty` и указать логическое имя бина для выполнения привязки к свойству как выражение языка выражений JSF, соответствующее имени бина. В нашем случае, поскольку мы не указывали имя для нашего бина `Customer`, будет иметь место поведение по умолчанию, заключающееся в использовании имени класса бина с переводом его первого символа в нижний регистр. Таким образом, наш бин `Customer`, значение которого мы использовали для атрибута `value` аннотации `@ManagedBeanProperty`, будет называться `customer`.

Наконец, в отношении метода `saveCustomer()`, который будет вызываться всякий раз, когда пользователь отправляет форму и все поля проходят проверку допустимости, мы должны внести небольшое изменение на странице ввода сведений о клиентах:

```
<?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>Сохранение информации о заказчике</title>
  </h:head>
  <h:body>
    <h:form>
      <h:messages></h:messages>
      <table cellpadding="0" cellspacing="0" border="0">
        <tr>
          <td align="right">Имя:</td>
          <td align="left">
            <h:inputText
                label="Имя"
                value="#{customer.firstName}"
                required="true">
              <f:validateLength minimum="2" maximum="30">
                </f:validateLength>
            </h:inputText>
          </td>
        </tr>
        <tr>
          <td align="right">Фамилия:</td>
          <td align="left">
            <h:inputText
                label="Фамилия"
                value="#{customer.lastName}"
                required="true">
          </td>
        </tr>
      </table>
    </h:form>
  </h:body>
</html>
```

```
<f:validateLength minimum="2" maximum="30">
    </f:validateLength>
</h:inputText>
</td>
</tr>
<tr>
    <td align="right">Email:</td>
    <td align="left">
        <h:inputText
            label="Email"
            value="#{customer.email}">
            <f:validateLength minimum="2" maximum="30">
                </f:validateLength>
        </h:inputText>
    </td>
</tr>
<tr>
    <td></td>
    <td align="left">
        <h:commandButton
            action="#{customerController.saveCustomer}"
            value="Сохранить">
        </h:commandButton>
    </td>
</tr>
</table>
</h:form>
</h:body>
</html>
```

Единственная значительная разница между этой и предыдущей версией страницы состоит в том, что атрибут `action` тега `<h:commandButton>` был изменен для указания на метод `saveCustomer()` управляемого бина `CustomerController`. Как видно из исходного кода для этого бина (см. выше в этом разделе), этот метод возвращает строку `customer_saved`, если данные были сохранены успешно, или `error_saving_customer`, если возникла какая-либо проблема при сохранении данных. Этим двум значениям соответствуют базовые имена страниц, к которым нам понадобится перейти, когда данные заказчика будут успешно сохранены и когда возникнет проблема при сохранении данных.

В этой версии страницы было произведено еще несколько не связанных с задачей обработки изменений. Во-первых, для простоты мы удалили ряд функций, которые рассматривались выше в этой главе (нестандартные элементы верификации, стилизация сообщения об ошибке и др.). Дополнительное, несколько более интересное изменение – замена компонента `<h:panelGrid>` стандартной HTML-таблицей. Большинство разработчиков серверных Java-приложений по крайней мере немного знакомы с HTML. Поэтому использование стандартных компонентов HTML, вероятнее всего, улучшит это знакомство и потенциально сделает разметку страницы более читаемой. В предыдущих версиях спецификации JSF не рекомендовалось смешивать теги стандартного HTML и теги JSF на странице JSF, поскольку подобные действия иногда приводили к неожиданным результатам. Это ограничение было снято в JSF 1.2, поскольку JSP-теги JSF были модифицированы во избежание неожиданных проблем, возникавших в более ранних версиях JSF. У фэйслетов же с самого начала никаких проблем не было. Поскольку фэйслеты теперь явля-

ются предпочтительной технологией представления JSF, это вносит больше смысла в использование стандартных компонентов HTML, чем когда бы то ни было.

Включение Ajax в приложения JSF 2.0

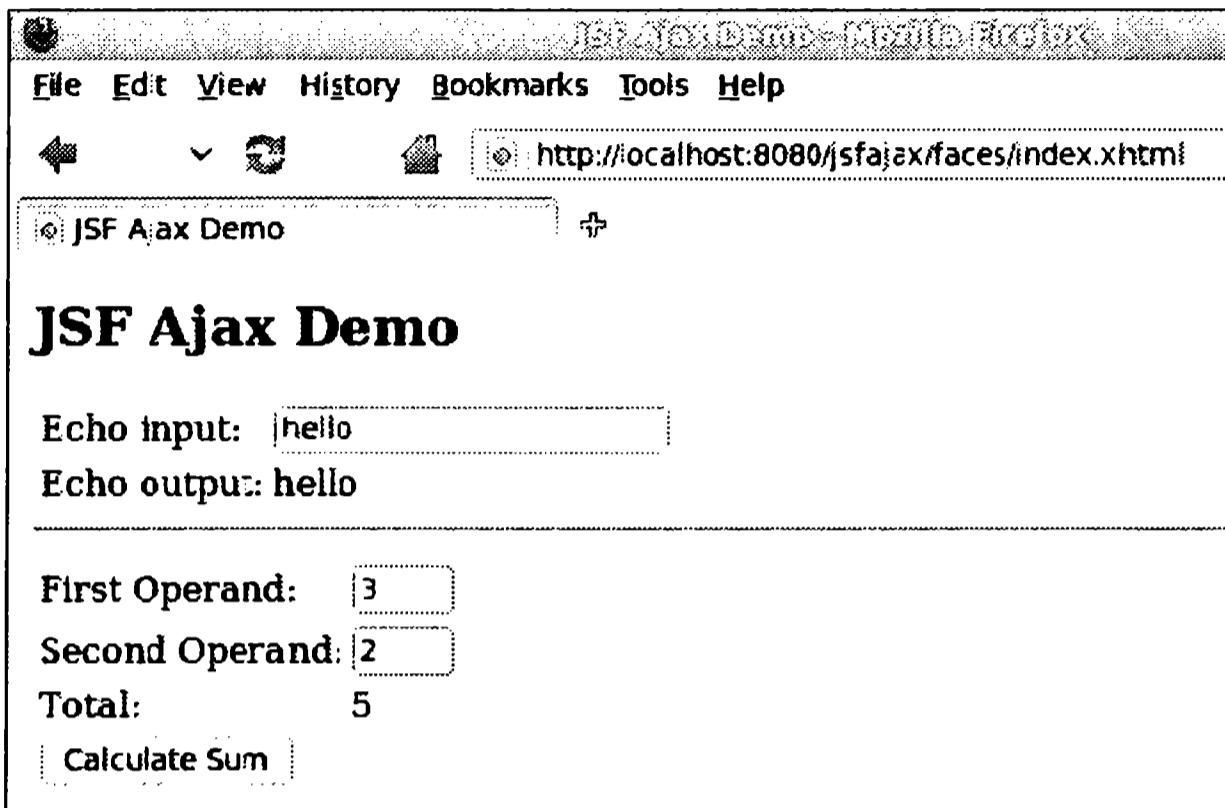
JSF 1.x не включал собственную поддержку Ajax. Пользовательские библиотеки JSF были вынуждены реализовывать Ajax своим собственным способом. К сожалению, такое положение дел приводило к несовместимости между библиотеками компонентов JSF. JSF 2.0 стандартизирует поддержку Ajax благодаря введению нового тега `<f:ajax>`.

Следующий код поясняет 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>Демонстрация Ajax JSF</title>
  </h:head>
  <h:body>
    <h2>Демонстрация Ajax JSF</h2>
    <h:form>
      <h:messages/>
      <h:panelGrid columns="2">
        <h:outputText value="Эхо ввода:"/>
        <h:inputText
            id="textInput"
            value="#{controller.text}">
          <f:ajax render="textVal" event="keyup"/>
        </h:inputText>
        <h:outputText value="Эхо вывода:"/>
        <h:outputText
            id="textVal"
            value="#{controller.text}" />
      </h:panelGrid>
      <hr/>
      <h:panelGrid columns="2">
        <h:panelGroup/>
        <h:panelGroup/>
        <h:outputText value="Первый операнд:"/>
        <h:inputText
            id="first"
            value="#{controller.firstOperand}"
            size="3"/>
        <h:outputText value="Второй операнд:"/>
        <h:inputText
            id="second"
            value="#{controller.secondOperand}"
            size="3"/>
        <h:outputText value="Итог:"/>
        <h:outputText id="sum" value="#{controller.total}" />
        <h:commandButton
            actionListener="#{controller.calculateTotal}"
            value="Вычисление суммы">
          <f:ajax execute="first second" render="sum"/>
        </h:commandButton>
      </h:panelGrid>
    </h:form>
  </h:body>
</html>
```

```
</h:panelGrid>
</h:form>
</h:body>
</html>
```

После развертывания нашего приложения этот код будет отображаться в окне обозревателя, как показано на следующем снимке экрана:



Предыдущий код иллюстрирует двукратное использование тега `<f:ajax>`. В первый раз, когда мы используем этот тег, мы реализуем типичный пример – Эхо Ajax, в котором у нас имеется компонент `outputText`, обновляющий себя значением компонента `inputText`. Всякий раз, когда любой символ вводится в поле ввода, значение компонента `outputText` автоматически обновляется.

Реализуя вышеописанную функциональность, мы помещаем тег `<f:ajax>` в тег `<h:inputText>`. Значение атрибута `render` тега `<f:ajax>` должно соответствовать идентификатору компонента, который мы хотим обновить после завершения запроса Ajax. В нашем конкретном примере мы хотим обновить компонент `outputText` идентификатором `"textVal"`, поэтому данное значение мы используем для атрибута `render` нашего тега `<f:ajax>`.



В некоторых случаях мы, возможно, должны отобразить более одного компонента JSF после того, как событие Ajax заканчивается. В целях удовлетворения этой потребности мы можем добавить несколько ID в качестве значения атрибута `render`; при этом значения ID нужно просто разделить пробелами.

Другим атрибутом `<f:ajax>`, который мы использовали в этом экземпляре, является атрибут `event`. Он указывает событие JavaScript, которое вызывает событие Ajax. В данном случае мы должны инициировать событие в любое время, при отпускании клавиши, во время ввода данных пользователем в поле ввода. Поэтому соответствующим используемым событием является событие `"keyup"`.

В следующей таблице перечислены все поддерживаемые события JavaScript:

Событие	Описание
blur	Компонент теряет фокус
change	Компонент теряет фокус, и его значение изменяется
click	По компоненту щелкнули
dblclick	По компоненту выполнили двойной щелчок
focus	Компонент получает фокус
keydown	Нажата клавиша в тот момент, когда компонент имеет фокус
keypress	Нажата или удерживается клавиша в тот момент, когда компонент имеет фокус
keyup	Клавиша отпущена в тот момент, когда компонент имеет фокус
mousedown	Кнопка мыши нажата в тот момент, когда компонент имеет фокус
mousemove	Перемещение указателя мыши над компонентом
mouseout	Указатель мыши покидает компонент
mouseover	Указатель мыши находится над компонентом
mouseup	Кнопка мыши отпущена в тот момент, когда компонент имеет фокус
select	Выделен текст компонента
valueChange	Эквивалентно изменению (change), компонент теряет фокус и его значение получает изменение

Ниже в странице мы используем тег `<f:ajax>` еще раз – с целью включения Ajax для компонента кнопки-команды. В этом случае мы хотим повторно вычислить значение на основе значений двух компонентов ввода. Для того чтобы иметь на сервере значение, обновленное самыми последними данными, вводимыми пользователем, мы использовали атрибут `execute` тега `<f:ajax>`. Этот атрибут принимает список разделенных пробелами идентификаторов компонентов для использования их в качестве ввода. Далее мы используем атрибут `render` точно так же, как и ранее – для указания того, какие компоненты должны быть повторно перерисованы после завершения запроса Ajax.

Обратите внимание, что мы используем атрибут `actionListener` тега `<h:commandButton>`. Этот атрибут обычно используется всякий раз, когда после нажатия кнопки мы не должны перемещаться к другой странице. Значением для этого атрибута является *метод действия слушателя* (*action listener method*), который мы написали в одном из наших управляемых бинов. Методы действия слушателя, не должны возвращать данных (имеют тип `void`) и принимают экземпляр `javax.faces.event.ActionEvent` в качестве единственного параметра.

Управляемый бин для нашего приложения выглядит следующим образом:

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

```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.ViewScoped;
import javax.faces.event.ActionEvent;
@ManagedBean
@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 нам не нужно было делать ничего особенного в нашем управляемом бине для включения Ajax в нашем приложении. Все это выполняется тегами <f:ajax>, размещенными на странице.

Как мы видим из этого примера, включение Ajax в приложениях JSF 2.0 – очень простое действие. Мы всего лишь должны использовать единственный тег для включения Ajax на нашей странице, без необходимости написания какого-либо кода на JavaScript, JSON или XML.

Стандартные компоненты JSF

JSF включает целый ряд стандартных компонентов, но до настоящего момента мы рассмотрели только некоторые из них. В следующих разделах рассмотрены все доступные компоненты JSF.

Базовые компоненты JSF

Базовыми компонентами JSF являются компоненты, которые не привязываются к рендерингу HTML или какому-либо другому механизму отображения. Среди прочего они предоставляют такую функциональность, как преобразование типов и проверка допустимости. В этом разделе мы рассмотрим все базовые компоненты JSF.

Тег <f:actionListener>

Выполняет метод действия слушателя `processAction()`, определяемый атрибутом `type` тега. Значение атрибута `type` должно быть полностью определенным (квалифицированным) именем класса, реализующего интерфейс `javax.faces.event.ActionListener`. Этот тег обычно является дочерним тегом `<h:commandButton>` или `<h:commandLink>`. Когда пользователь щелкает по родительскому компоненту, метод `processAction()` объявленной реализации `ActionListener` выполняется автоматически. Следующий фрагмент разметки поясняет, как этот тег обычно используется:

```
<h:commandButton action="save" value="Сохранить">
    <f:actionListener type="net.ensode.CustomActionListener"/>
</h:commandButton>
```

Тег <f:ajax>

Включает поведение Ajax. Этот тег обычно вкладывается в другой тег JSF, такой как `<h:commandButton>`. Когда присутствует этот тег, родительский компонент автоматически «аяксифицируется». Например кнопка, поддерживающая Ajax, не обновляет целую страницу – вместо этого она инициирует частичное обновление страницы. Такое поведение делает наше приложение намного более динамичным и отзывчивым.

```
<h:commandButton actionListener="#{controller.calculateTotal}"
    value="Вычисление суммы">
    <f:ajax execute="first second" render="sum"/>
</h:commandButton>
```

Значением атрибута `execute` является разделенный пробелами список идентификаторов (ID) компонентов, которые будут использоваться в качестве входных при запуске запроса Ajax. Значением атрибута `render` является разделенный пробелами список компонентов для повторного отображения после того, как запрос Ajax будет завершен.

Тег <f:attribute>

Устанавливает атрибут родительского компонента с ключом, определенным атрибутом name тега и значением, определенным атрибутом value тега. Все атрибуты компонента позже могут быть получены программно, как структура данных – карта (мап), вызовом метода getAttributes() соответствующего экземпляра javax.faces.component.UIComponent. Этот тег часто используется в сочетании с классом <f:actionListener>, чтобы передать параметры методу действия слушателя.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:commandButton action="save" value="Сохранить">
    <f:actionListener type="net.ensode.CustomButtonListener"/>
    <f:attribute name="someAttribute" value="someValue"/>
</h:commandButton>
```

Метод processAction() нашего класса CustomActionListener выглядел бы примерно так:

```
public void processAction(ActionEvent actionEvent)
{
    String attribute = (String) actionEvent.getComponent().
        getAttributes().get("attrname1");
    // Обработка продолжается...
}
```

Тег <f:convertDateTime>

Преобразует значение родительского компонента в экземпляр java.util.Date. Этот тег позволяет присваивать правильно отформатированную строку, вводимую пользователем, полю даты в управляемом бине. Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:inputText value="#{customer.birthDate}">
    <f:convertDateTime dateStyle="short"/>
</h:inputText>
```

Тег <f:convertNumber>

Преобразует значение родительского компонента в экземпляр java.lang.Number. Этот тег позволяет правильно отформатированную строку, вводимую пользователем, присваиваемую в последствии числовому полю в управляемом бине. Поскольку класс java.lang.Number является родительским для классов java.lang.Integer, java.lang.Long, java.lang.Float и java.lang.Double (наряду с другими числовыми типами), этот тег может использоваться, чтобы преобразовать практически любой тип поля ввода числовых данных в соответствующий тип.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:inputText value="#{customer.age}">
    <f:convertNumber/>
</h:inputText>
```

Тег <f:converter>

Регистрирует пользовательский преобразователь, указанный атрибутом converterId тега, в родительском теге. Указанный преобразователь должен представлять собой класс, реализующий интерфейс javax.faces.convert.Converter, и быть или декорированным аннотацией @FacesConverter, или зарегистрированным в дескрипторе конфигурации приложения, файле faces-config.xml, с помощью тега <converter>.

Предположим, что мы создали пользовательский класс под названием TelephoneNumber для сохранения телефонного номера; при этом управляемый бин под названием Customer имеет поле с названием telephone типа TelephoneNumber. Мы можем создать пользовательский элемент верификации для преобразования вводимого пользователем телефонного номера в экземпляр класса TelephoneNumber.

```
<h:inputText value="#{customer.telephone}">
    <f:converter converterId="TelephoneConverter"/>
</h:inputText>
```

Класс TelephoneConverter должен реализовывать интерфейс javax.faces.convert.Converter.

Тег <f:event>

Позволяет вызывать метод управляемого бина всякий раз, когда встречается конкретное событие. Этот тег должен быть вложен в другой тег JSF. Регистрируемое событие указывается в качестве значения атрибута type тега <f:event>. Для него допустимы следующие значения:

- **preRenderComponent** – инициируется непосредственно перед тем, как родительский компонент будет прорисован;
- **PostAddToView** – инициируется после того, как родительский компонент будет добавлен к представлению;
- **preValidate** – инициируется непосредственно перед тем, как значение родительского компонента будет проверено;
- **postValidate** – инициируется сразу после того, как значение родительского компонента будет проверено.

Значение атрибута listener тега <f:event> должно быть значением выражения привязки, которое разрешается в метод управляемого бина, удовлетворяющего следующим требованиям: имеет модификатор видимости public, не возвращает значения (имеет тип void) и принимает экземпляр класса javax.faces.event.ComponentSystemEvent в качестве единственного параметра. Этот метод будет автоматически вызван при запуске события.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:outputText>
    <f:event type="preRenderComponent"
        action="#{myManagedBean.doSomething}"/>
</h:outputText>
```

Тег <f:facet>

Регистрирует фасет на родительском компоненте. *Фасет* (facet) – это специальный дочерний компонент, к которому можно получить доступ с помощью метода UI-Component.getFacet(). Этот метод может быть переопределен для пользовательских компонентов; данное обстоятельство позволяет иначе обрабатывать компоненты в фасете. Например, у стандартного тега <h:dataTable> может быть фасет, называемый "header", который будет использоваться для представления всех компонентов в теге <f:facet> в качестве заголовка представленной HTML-таблицы.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:dataTable value="{Order.items}" var="item">
    <h:column>
        <f:facet name="header">
            <h:outputText value="Номер элемента"/>
        </f:facet>
        <h:outputText value="#{item.itemNumber}"/>
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="Описание элемента"/>
        </f:facet>
        <h:outputText value="#{item.itemShortDesc}"/>
    </h:column>
</h:dataTable>
```

Тег <f:loadBundle>

Загружает пакет ресурса в контекст запроса. Имя пакета ресурса указывается атрибутом basename тега. Переменная для использования с целью получения доступа к свойствам пакета ресурса определяется атрибутом var тега.

Следующий фрагмент разметки поясняет типичное использование этого тега:

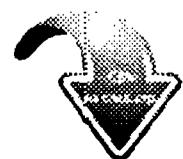
```
<f:view locale="#{facesContext.getExternalContext.request.
    locale}">
    <f:loadBundle basename="net.ensode.Messages" var="mess"/>
    <h:outputText value="#{mess.greeting}"/>
</f:view>
```

Тег <f:metadata>

Этот тег прежде всего используется в качестве родительского тега для <f:viewParam>, который в свою очередь используется для отображения параметров GET-запроса на значения управляемого бина.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<f:metadata>
    <f:viewParam name="someParam" value="#{someBean.
        someProperty}"/>
```



```
</f:metadata>
```

Тег <f:param>

Когда этот тег является дочерним элементом тега `<h:commandLink>`, он генерирует параметр запроса, определенный его атрибутами `name` и `value`. Когда же он является дочерним элементом тега `<h:outputFormat>`, он заменяется в обозревателе параметром, определенным атрибутом `value` тега `<h:outputFormat>`.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:outputFormat value="Привет, {0}">
    <f:param value="#{customer.firstName}" />
</h:outputFormat>
```

Тег <f:phaseListener>

Регистрирует *слушателя этапа* (phase listener) на текущей странице. Слушатель этапа должен быть экземпляром класса, реализующего интерфейс `javax.faces.event.PhaseListener`. Этот класс определяется атрибутом `type` тега.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<f:view>
    <f:phaseListener type="net.ensode.CustomPhaseListener"/>
</f:view>
```

Тег <f:selectItem>

Добавляет выбираемый элемент, принадлежащий родительскому компоненту. Способ, которым этот компонент прорисовывается, зависит от родительского компонента. Он может использоваться в качестве дочернего компонента для тегов `<h:selectManyCheckBox>`, `<h:selectManyListBox>`, `<h:selectManyMenu>`, `<h:selectOneListBox>`, `<h:selectOneMenu>` и `<h:selectOneRadio>`.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:selectManyCheckBox value="#{order.items}">
    <f:selectItem itemValue="#{item1}"
                  itemLabel= "Беспроводная клавиатура"/>
    <f:selectItem itemValue="#{item1}"
                  itemLabel= "Беспроводная мышь"/>
</h:selectManyCheckBox>
```

Тег <f:selectItems>

Добавляет серию выбираемых элементов, принадлежащих родительскому тегу. Атрибут `value` этого тега должен быть *значением отложенного выражения* (deferred value expression), разрешающимся в массив или список объектов `javax.faces.model.SelectItem`.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:selectManyCheckBox value="#{order.items}">
    <f:selectItems value="#{valueContainer.allItems}" />
</h:selectManyCheckBox>
```

Тег **<f:setPropertyActionListener>**

Может быть дочерним тегом тегов `<h:commandLink>` или `<h:commandButton>`. Когда производится щелчок по кнопке или ссылке, этот тег устанавливает атрибут в управляемом бине, определенном атрибутом `target` тега с помощью значения атрибута `value` тега.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:commandButton value="Сохранить" action="#{controller.save}">
    <f:setPropertyActionListener
        target="#{order.lastUpdUserId}"
        value="#{user.userId}" />
</h:commandButton>
```

Тег **<f:subview>**

Этот тег предназначен для использования только в случае использования JSP в качестве технологии представления JSF. Включение любой JSP-страницы выполняется с помощью тега `<jsp:include>` или JSTL-тега `<c:import>`, который должен быть вложен в тег `<f:subview>`.

Следующий фрагмент разметки поясняет типичное использование тега `<f:subview>`:

```
<f:view>
    <table>
        <tr>
            <td width="30%">
                <f:subview>
                    <jsp:include page="menu.jsp">
                </f:subview>
            </td>
            <td>
                Дополнительное содержимое здесь.
            </td>
        </tr>
    </table>
</f:view>
```

Тег **<f:validateBean>**

Этот тег используется для тонкой настройки Проверки допустимости со стороны бина (JSR 303). Он может использоваться для отключения Проверки допустимости

со стороны бина в индивидуальном порядке или быть присвоенным компонентам спецификации JSR 303 группы Проверки допустимости со стороны бина, которые будут проверяться.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:inputText value="#{someManagedBean.someProperty}">
    <f:validateBean disabled="#{anotherManagedBean.
        booleanProperty}"/>
</h:inputText>
```

В этом примере предполагается, что свойство `someProperty` бина `someManagedBean` декорируется аннотацией Проверки допустимости со стороны бина (`bean validation`), такой как `@NotNull` или `@Pattern`. В этом случае тег `<f:validateBean>` используется для отключения Проверки допустимости со стороны бина, если атрибут `booleanProperty` в бине `anotherManagedBean` разрешается в значение `true` (истина).

Тег `<f:validateDoubleRange>`

Проверяет, что значение для родительского компонента является экземпляром `java.lang.Double`, которое находится между значениями, определенными атрибутами `minimum` и `maximum` тега.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:inputText value="#{item.price}">
    <f:validateDoubleRange minimum="1.0" maximum="100.0"/>
</h:inputText>
```

Тег `<f:validateLength>`

Проверяет, что значение для родительского компонента является строкой, длина которой находится в диапазоне между значениями, определенными атрибутами `minimum` и `maximum` тега (для обоих значений включительно).

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:inputText label="Имя"
    value="#{customer.firstName}" required="true">
    <f:validateLength minimum="2" maximum="30">
    </f:validateLength>
</h:inputText>
```

Тег `<f:validateLongRange>`

Проверяет, что значение для родительского компонента является экземпляром `java.lang.Long`, величина которого находится в диапазоне между значениями, определенными атрибутами `minimum` и `maximum` тега.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:inputText value="#{orderItem.quantity}">
    <f:validateDoubleRange minimum="1" maximum="100"/>
</h:inputText>
```

Тег <f:validateRegex>

Проверяет, что значение родительского тега соответствует регулярному выражению, указанному в его атрибуте pattern.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:inputText value="#{someBean.phoneNumber}">
    <f:validateRegex pattern="\d{3}-\d{3}-\d{4}"/>
</h:inputText>
```

Тег <f:validateRequired>

Проверяет, что пользователь ввел значение в родительский компонент ввода. Этот тег эквивалентен установке атрибута required в значение true (истина) в компоненте ввода.

Следующий фрагмент разметки поясняет типичное использование данного тега:

```
<h:inputText value="#{someBean.someProperty}">
    <f:validateRequired/>
</h:inputText>
```

Тег <f:validator>

Проверяет значения родительского компонента с использованием пользовательского элемента верификации, реализующего интерфейс javax.faces.validator.Validator. Пользовательский элемент проверки допустимости должен быть или декорирован аннотацией @FacesValidator, или объявлен в дескрипторе конфигурации приложения – файле faces-config.xml.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:inputText label="Email" value="#{customer.email}">
    <f:validator validatorId="emailValidator"/>
</h:inputText>
```

Тег <f:valueChangeListener>

Регистрирует экземпляр класса, реализующего интерфейс javax.faces.event.ValueChangeListener в родительском компоненте. Реализация ValueChangeListener реализует метод processValueChange(), который может выполнить действие, если значение родительского компонента изменяется.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:inputText value="#{orderItem.quantity}">
    <f:valueChangeListener type="net.ensode.
        CustomValueChangeListener"/>
</h:inputText>
```

Тег <f:verbatim>

Содержимое этого тега передается «как есть» в отображаемую страницу. До JSF 1.2 не рекомендовалось помещать HTML-теги внутри JSF-тега `<f:view>`, так как они иногда неправильно отображались. Общим обходным решением для этого ограничения было помещение стандартных HTML-тегов в теги `<f:verbatim>`. Начиная с JSF 1.2, а также при использовании фэйслетов этот тег является избыточным, поскольку теперь можно безопасно помещать стандартные HTML-теги в JSF-теги страницы.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<f:view>
    <f:verbatim><p></f:verbatim>
        Этот текст будет отображен внутри тега &lt;p&gt; HTML.
    <f:verbatim></p></f:verbatim>
</f:view>
```

Тег <f:view>

Этот тег является родительским тегом для всех тегов JSF – и стандартных, и пользовательских.

Следующий фрагмент разметки поясняет типичное использование данного тега:

```
<f:view>
    <h:outputText
        escape="true"
        value="Все компоненты JSF должны быть внутри тега
            <f:view>"/>
</f:view>
```

Тег <f:viewParam>

Этот тег используется для отображения параметра HTTP-запроса GET на свойство управляемого бина. Значение его атрибута `name` должно соответствовать наименованию параметра, а значение его параметра `value` должно быть значением выражения связывания, соответствующим значению управляемого бина, которое будет заполнено параметром запроса.

```
<f:metadata>
    <f:viewParam name="someParam" value="#{someBean.
        someProperty}"/>
</f:metadata>
```

HTML-компоненты JSF

В предыдущих примерах мы рассмотрели только подмножество стандартных HTML-компонентов JSF. В этом разделе мы перечислим все стандартные HTML-компоненты JSF.

Тег <h:body>

Отображает тело страницы. Этот тег походит на стандартный HTML-тег <body>.

```
<h:body>
    <! - здесь находится тело страницы ->
</h:body>
```

Тег <h:button>

Подобен тегу <h:commandButton>. Тем не менее кнопки, отображаемые с помощью этого тега, генерирует HTTP-запрос GET при переходе на целевую страницу.

```
<h:button value="Щелкните" action="next_page"/>
```

Тег <h:column>

Этот тег обычно вкладывается в тег <h:dataTable>. Любые компоненты в этом теге будут отображаться как одиночный столбец в таблице, отображаемой тегом <h:dataTable>.

Следующий фрагмент разметки поясняет типичное использование данного тега:

```
<h:dataTable value="{Order.items}" var="item">
    <h:column>
        <f:facet name="header">
            <h:outputText value="Номер элемента"/>
        </f:facet>
        <h:outputText value="#{item.itemNumber}"/>
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="Описание элемента"/>
        </f:facet>
        <h:outputText value="#{item.itemShortDesc}"/>
    </h:column>
</h:dataTable>
```

Тег <h:commandButton>

Отображает HTML-кнопку отправки (Submit) на отображаемой странице.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:form>
    <h:inputText
```

```

        label="Имя"
        value="#{customer.firstName}"/>
<h:commandButton
    action="save"
    value="Сохранить">
</h:commandButton>
</h:form>
```

Тег <h:commandLink>

Отображает ссылку, которая отправляет форму, определенную родителем этого тега – тегом <h:form>.

Следующий фрагмент разметки поясняет типичное использование данного тега:

```

<h:form>
    <h:inputText
        label="Имя"
        value="#{customer.firstName}"/>
    <h:commandLink
        action="save"
        value="Сохранить">
    </h:commandLink>
</h:form>
```

Тег <h:dataTable>

Выполняет динамическое построение таблицы на основе коллекции. Коллекция, содержащая значения, должна быть определена атрибутом value тега.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```

<h:dataTable value="{Order.items}" var="item">
    <h:column>
        <f:facet name="header">
            <h:outputText value="Номер элемента"/>
        </f:facet>
        <h:outputText value="#{item.itemNumber}"/>
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="Описание элемента"/>
        </f:facet>
        <h:outputText value="#{item.itemShortDesc}"/>
    </h:column>
</h:dataTable>
```

Тег <h:form>

Отображает HTML-форму на сгенерированной странице.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:form>
    <h:inputText
        label="Имя"
        value="#{customer.firstName}"/>
    <h:commandLink
        action="save"
        value="Сохранить">
    </h:commandLink>
</h:form>
```

Тег **<h:graphicImage>**

Отображает HTML-тег img.

Следующие фрагменты разметки поясняют типичное использование этого тега.

Если мы хотим загрузить изображение из URL:

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

Если изображение помещается в стандартном каталоге ресурсов JSF 2.0, можно использовать тег, например, таким образом:

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

Тег **<h:head>**

Специфичная для JSF версия стандартного HTML-тега <head>.

Следующий фрагмент разметки поясняет, как использовать этот тег:

```
<h:head>
    <title>Заголовок страницы</title>
</h:head>
```

Тег **<h:inputHidden>**

Отображает скрытое поле HTML.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:inputHidden value="#{customer.id}"/>
```

Тег **<h:inputSecret>**

Отображает поле ввода HTML input password для ввода пароля.

Следующий фрагмент разметки поясняет типичное использование данного тега:

```
<h:inputSecret
    redisplay="false"
    value="#{user.password}"/>
```

Тег <h:inputText>

Отображает поле ввода HTML `input text`.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:inputText  
    label="Имя"  
    value="#{customer.firstName}"/>
```

Тег <h:inputTextarea>

Отображает поле ввода HTML `textarea`.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:inputTextarea  
    label="Комментарии"  
    value="#{order.comments}"/>
```

Тег <h:link>

Подобен тегу `<h:commandLink>`. Однако ссылка, используемая этим тегом, генерирует HTTP-запрос GET при перемещении к целевой странице.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:link value="Щелкните" action="next_page"/>
```

Тег <h:message>

Отображает сообщения для одного компонента. Компонент для отображения сообщений должен использовать свой атрибут `id` для установки собственного идентификатора, который затем должен использоваться в качестве атрибута `for` этого элемента.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<table>  
    <tr>  
        <td align="right">  
            <h:outputLabel  
                value="Имя входа в систему:"  
                for="loginField"/>  
        </td>  
        <td>  
            <h:inputText  
                id="loginField"  
                value="#{user.login}"  
                required="true"/>  
        </td>  
        <td>  
            <h:message for="loginField"/>  
        </td>
```

```
</tr>
</table>
```

Тег <h:messages>

Выводит сообщения для всех компонентов или глобальных сообщений. Если атрибут globalOnly тега будет установлен в значение true, на экран будут выведены только глобальные сообщения (сообщения, не относящиеся к какому-либо компоненту).

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<f:view>
    <h:messages/>
        <h:form>
            <h:inputText
                label="Имя"
                value="#{customer.firstName}"/>
            <h:commandButton
                action="save"
                value="Сохранить"/>
        </h:form>
</f:view>
```

Тег <h:outputFormat>

Отображает параметризованный текст. Параметры в атрибуте value этого тега определяются способом, подобным тому, которым они определяются в пакете ресурса, т. е. путем размещения целых чисел между фигурными скобками, в местах, где должны быть расположены параметры. Параметры заменяются значениями, определенными в любом дочернем элементе <f:param>.

Следующий фрагмент разметки поясняет типичное использование этого тега:

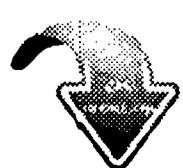
```
<h:outputFormat value="Привет, {0}">
    <f:param value="#{customer.firstName}"/>
</h:outputFormat>
```

Тег <h:outputLabel>

Отображает HTML-поле label.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<table>
    <tr>
        <td align="right">
            <h:outputLabel
                value="Имя входа в систему:"
                for="loginField"/>
        </td>
        <td>
```



```

<h:inputText
    id="loginField"
    value="#{user.login}"
    required="true"/>
</td>
</tr>
</table>

```

Тег <h:outputLink>

Отображает HTML-ссылку как anchor (a)-элемент (якорь) с атрибутом href.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```

<h:outputLink value="http://ensode.net">
    <h:outputText value="Enso&#dce"/>
</h:outputLink>

```

Тег <h:outputScript>

Используется для загрузки файла JavaScript из стандартного расположения ресурса.

Следующий фрагмент разметки поясняет, как использовать этот тег:

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

Тег <h:outputStylesheet>

Используется для загрузки таблицы стилей CSS из стандартного расположения ресурса.

Следующий фрагмент разметки поясняет типичное использование этого тега:

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

Тег <h:outputText>

Если атрибуты dir, lang, style или styleClass определены, этот тег отображает элемент HTML span, содержащий атрибут value. В противном случае значение, определенное атрибутом value тега, отображается так, чтобы избежать любых символов XML/HTML (чтобы они отображались должным образом). Если атрибут escape тега устанавливается в значение false, то символы XML/HTML не игнорируются.

Следующий фрагмент разметки поясняет типичное использование данного тега:

```
<h:outputText value="#{customer.firstName}"/>
```

Тег <h:panelGrid>

Отображает статическую HTML-таблицу. Число столбцов в таблице указывается в атрибуте columns тега. Дочерние компоненты будут добавляться к последующей строке, как только к текущей строке будет добавлено число элементов, определенное в атрибуте columns.

Следующий фрагмент разметки поясняет типичное использование данного тега:

```
<h:panelGrid columns="2"
    columnClasses="rightAlign, leftAlign">
    <h:outputText value="Имя:></h:outputText>
    <h:inputText
        label="Имя"
        value="#{customer.firstName}"
        required="true">
        <f:validateLength minimum="2" maximum="30">
        </f:validateLength>
    </h:inputText>
    <h:outputText value="Фамилия:></h:outputText>
    <h:inputText
        label="Фамилия"
        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:validateLength minimum="3" maximum="30">
        </f:validateLength>
    </h:inputText>
    <h:panelGroup></h:panelGroup>
    <h:commandButton
        action="save"
        value="Сохранить">
    </h:commandButton>
</h:panelGrid>
```

Тег <h:panelGroup>

Используется для группировки его дочерних компонентов в единую ячейку родительского тега <h:panelGrid> или <h:dataTable>. Может также использоваться для создания «пустой» ячейки в родительском теге <h:panelGrid>.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:panelGrid columns="2" columnClasses="rightAlign, leftAlign">
    <h:outputText value="Имя:></h:outputText>
    <h:inputText
        label="Имя"
        value="#{customer.firstName}"
        required="true">
        <f:validateLength minimum="2" maximum="30">
        </f:validateLength>
    </h:inputText>
```



```
<h:outputText value="Фамилия:"></h:outputText>
<h:inputText
    label="Фамилия"
    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:validateLength minimum="3" maximum="30">
    </f:validateLength>
</h:inputText>
<h:panelGroup></h:panelGroup>
<h:commandButton
    action="save"
    value="Сохранить">
</h:commandButton>
</h:panelGrid>
```

Тег <h:selectBooleanCheckbox>

Отображает единичное поле ввода HTML input checkbox. Атрибут value для этого тега обычно устанавливается в значение выражения связывания, отображаемое на булево свойство в управляемом бине.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:selectBooleanCheckBox value="#{customer.newsletterOk}"/>
<h:outputText value="Хотите получать наши новости?">
```

Тег <h:selectManyCheckbox>

Отображает серию связанных флажков. Значения для выбора пользователем определяются в любом дочернем теге <f:selectItem> или <f:selectItems>.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:selectManyCheckBox value="#{order.items}">
    <f:selectItems value="#{valueContainer.allItems}"/>
</h:selectManyCheckBox>
```

Тег <h:selectManyListbox>

Отображает HTML-поле select переменного размера, которое допускает множественные выборы. Значения для выбора пользователем определяются в любом дочернем теге <f:selectItem> или <f:selectItems>. Число элементов, одновременно выводимое на экран, устанавливается атрибутом size тега.

Следующий фрагмент разметки поясняет типичное использование данного тега:

```
<h:selectManyListBox value="#{order.items}">
    <f:selectItems value="#{valueContainer.allItems}" />
</h:selectManyListBox>
```

Тег **<h:selectManyMenu>**

Отображает HTML-поле `select`, которое допускает множественные выборы. Значения для выбора пользователем определяются в любом дочернем теге `<f:selectItem>` или `<f:selectItems>`. Этот тег идентичен тегу `<h:selectManyListBox>` за исключением того, что он всегда выводит на экран один элемент за один раз. Поэтому у него нет атрибута `size`.

Следующий фрагмент разметки поясняет типичное использование данного тега:

```
<h:selectManyMenu value="#{order.items}">
    <f:selectItems value="#{valueContainer.allItems}" />
</h:selectManyMenu>
```

Тег **<h:selectOneListbox>**

Отображает HTML-поле выбора переменного размера, не допускающее множественных выборов. Значения для выбора пользователем определяются в любом дочернем теге `<f:selectItem>` или `<f:selectItems>`. Число элементов, выведенное на экран одновременно, устанавливается атрибутом `size` тега, который является необязательным. Если атрибут `size` не установлен, то все элементы выводятся на экран одновременно.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:selectOneListBox value="#{order.selectedItem}">
    <f:selectItems value="#{valueContainer.allItems}" />
</h:selectOneListBox>
```

Тег **<h:selectOneMenu>**

Отображает «выпадающее меню» HTML, предназначенное для отображения HTML- поля `select`, которое не допускает множественных выборов. Только один элемент выводится на экран за один раз. Значения для выбора пользователем определяются в любом дочернем теге `<f:selectItem>` или `<f:selectItems>`.

Следующий фрагмент разметки поясняет типичное использование этого тега:

```
<h:selectOneMenu value="#{order.selectedItem}">
    <f:selectItems value="#{valueContainer.allItems}" />
</h:selectOneMenu>
```

Тег **<h:selectOneRadio>**

Отображает серию связанных переключателей. Значения для выбора пользователем определяются в любом дочернем теге `<f:selectItem>` или `<f:selectItems>`.

Следующие фрагменты разметки поясняют типичное использование этого тега:

```
<h:selectOneRadio value="#{order.selectedItem}">
    <f:selectItems value="#{valueContainer.allItems}"/>
</h:selectOneRadio>
```

Дополнительные библиотеки компонентов JSF

В дополнение к стандартным библиотекам компонентов JSF имеется много доступных библиотек тегов JSF сторонних разработчиков. В следующей таблице приводятся некоторые из самых популярных. Пожалуйста, отметьте, что во время написания этой книги не все перечисленные библиотеки компонентов были обновлены до поддержки JSF 2.0.

Библиотека тегов	Поставщик	Лицензия	URL
MyFaces Trinidad	Apache	Apache 2.0	http://myfaces.apache.org/trinidad/
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://primefaces.org

Резюме

В этой главе мы рассмотрели, как разработать веб-приложение с использованием технологии JavaServer Faces, являющейся каркасом стандартных компонентов для платформы Java EE 6. Было показано, как написать простое приложение путем создания JSP-страниц, содержащих теги JSF и управляемые бины. Мы также обсудили, как проверить вводимые пользователем данные, используя стандартные блоки проверки допустимости JSF или наши собственные блоки проверки допустимости, а также как написать методы блока проверки допустимости. Речь шла и о том, как настроить стандартные сообщения об ошибках JSF – в частности, отредактировать текст и стиль сообщений (шрифт, цвет и т. д.). Наконец, мы рассмотрели, как написать приложения, интегрирующие JSF и API Персистентности Java (JPA).

7

Служба обмена сообщениями Java

API Системы обмена сообщениями Java (Java Messaging System API (JMS)) предоставляет приложениям Java EE механизм для отправки сообщений друг другу. Приложения JMS не взаимодействуют напрямую – вместо этого они обмениваются сообщениями, продюсеры отправляют сообщения пункту назначения, а потребители получают (потребляют) сообщение из пункта назначения.

Пунктом назначения (destination) сообщения является *очередь* (queue) сообщений, когда используется домен обмена сообщениями «точка-точка» (point-to-point (PTP)), или *тема* (topic) сообщения, когда используется домен обмена сообщениями «публикация/подписка» (publish/subscribe (pub/sub)).

В этой главе мы затронем следующие темы:

- настройка GlassFish для использования JMS;
- работа с очередями сообщений;
- работа с темами сообщения.

Настройка GlassFish для использования JMS

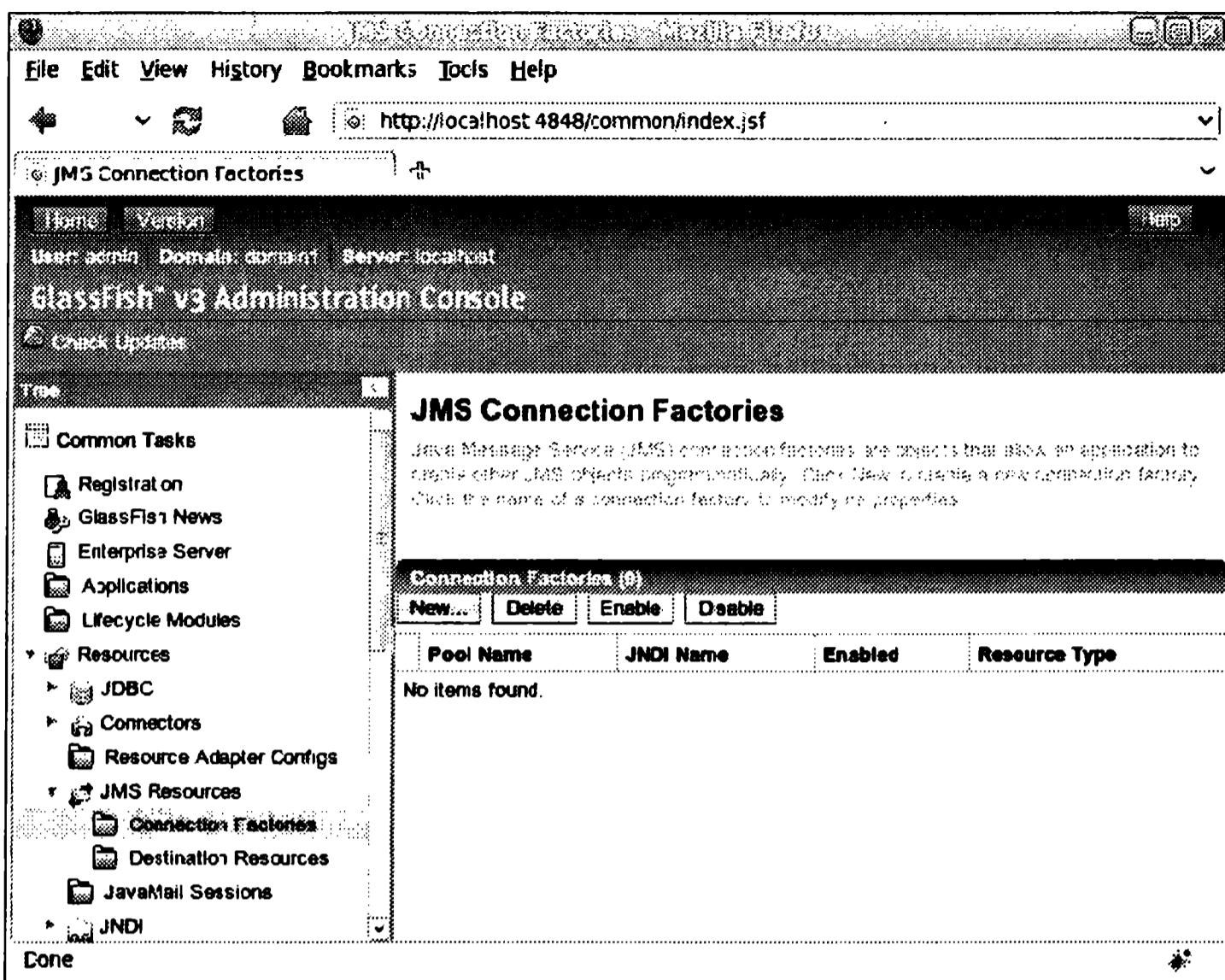
Прежде чем мы начнем писать код для использования возможностей API JMS, следует сконфигурировать некоторые ресурсы GlassFish. В частности, нам необходимо создать фабрику JMS-соединений, очереди сообщений и темы сообщений.

Создание фабрики JMS-соединений

Самый простой способ создания фабрики JMS-соединений – использование веб-консоли сервера GlassFish. Напомним из главы 1, что к веб-консоли можно получить доступ после запуска нашего домена вводом следующей команды в командной строке:

```
asadmin start-domain domain1
```

Затем нужно указать в адресной строке обозревателя URL <http://localhost:4848> и войти в систему:



Для добавления фабрики соединений следует развернуть узел **Ресурсы** (Resources) в дереве панели навигации, находящемся в левой части веб-консоли, далее развернуть вершину **Ресурсы JMS** (JMS Resources), затем щелкнуть по узлу **Фабрика соединений** (Connection Factories) и, наконец, щелкнуть по кнопке **Новая...** (New...) в панели основного содержания веб-консоли.

New JMS Connection Factory

The creation of a new Java Message Service (JMS) connection factory will create a corresponding resource pool for the factory and a connector resource.

General Settings	OK	Cancel
Pool Name: *	jms/GlassFishBookConnectionFactory	
Resource Type: *	javax.jms.ConnectionFactory	
Description:	Used for book examples	
Status:	<input checked="" type="checkbox"/> Enabled	
Pool Settings		
Initial and Minimum Pool Size:	8	Connections
Min number and initial number of connections maintained in the pool		
Maximum Pool Size:	32	Connections
Maximum number of connections that can be created to satisfy client requests		
Pool Resize Quantity:	2	Connections
Number of connections to be removed when pool idle time has expired		
Idle Timeout:	300	Seconds
Maximum time that connections can remain idle in the pool		

Для наших целей мы можем принять большинство значений по умолчанию; нам понадобится всего лишь ввести **Имя пула** (Pool Name) и выбрать **Тип ресурса** (Resource Type) для нашей фабрики соединений.



При выборе имени для ресурсов JMS будет целесообразно использовать **Имя пула (Pool Name)**, начинающееся с `jms/`. Ресурсы JMS, таким образом, всегда могут быть легко идентифицированы при просмотре дерева JNDI.

В текстовом поле **Имя пула** нужно ввести `jms/GlassFishBookConnectionFactory`. Примеры кода, рассматриваемые в этой главе, будут использовать это JNDI-имя для получения ссылки на данную фабрику соединений.

В раскрывающемся меню **Тип ресурса** имеются три опции:

- **javax.JMS.TopicConnectionFactory** – для создания фабрики соединений, в свою очередь создающей темы JMS для клиентов JMS, использующих домен обмена сообщениями «публикация/подписка»;
- **javax.JMS.QueueConnectionFactory** – для создания фабрики соединений, которая используется для создания очереди JMS для клиентов JMS, использующих домен обмена сообщениями PTP;
- **javax.JMS.ConnectionFactory** – для создания фабрики соединений, которая создает или темы JMS, или очереди JMS.

Для нашего примера мы выберем **javax.jms.ConnectionFactory**. Таким образом мы сможем использовать одну и ту же фабрику соединений для всех наших примеров – и для тех, которые используют PTP-домен обмена сообщениями, и для тех, которые используют домен обмена сообщениями «публикация/подписка».

После ввода **Имени пула** для нашей фабрики соединений нужно выбрать тип фабрики соединений и дополнительно ввести описание этой фабрики соединений, после чего щелкнуть по кнопке **OK**, чтобы изменения вступили в силу.

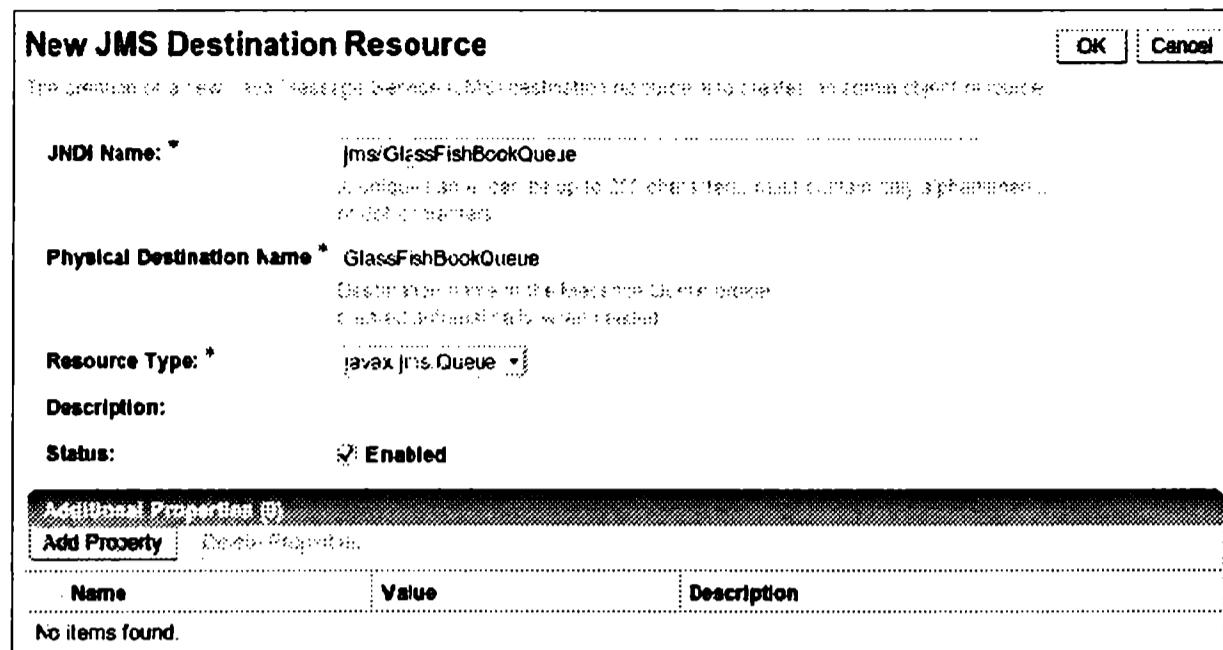
JMS Connection Factories			
Создание новых соединений для вашего приложения или для других приложений, использующих JMS. Вы можете создать соединение для каждого из ваших клиентов JMS, а также для каждого темы или очереди JMS, которую вы хотите использовать в своем приложении.			
Соединение создано (1)			
<input type="button" value="Edit"/>	<input type="button" value="New..."/>	<input type="button" value="Delete"/>	<input type="button" value="Enable"/>
Pool Name	JNDI Name	Enabled	Resource Type
<code>jms/GlassFishBookConnectionFactory</code>	<code>jms/GlassFishBookConnectionFactory</code>	true	<code>javax.jms.ConnectionFactory</code>

Теперь мы можем убедиться, что созданная нами новая фабрика соединений находится в списке панели содержимого веб-консоли сервера GlassFish.

Создание очереди JMS-сообщений

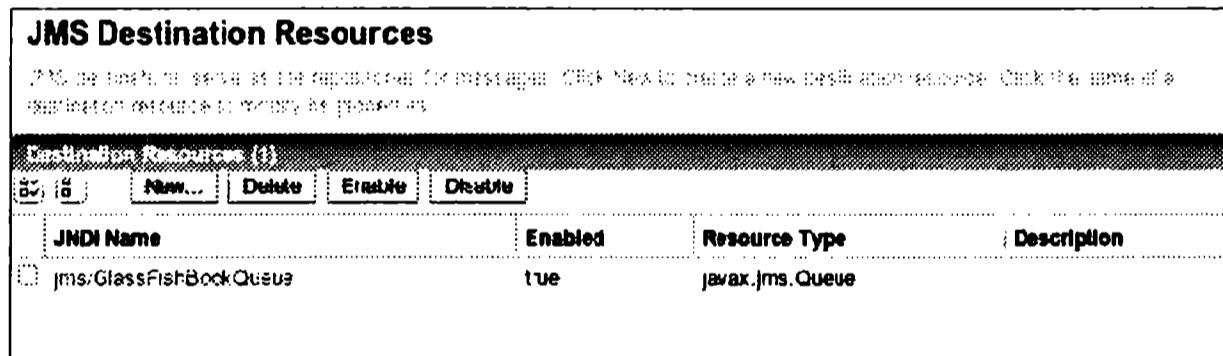
Для добавления очереди JMS-сообщений мы должны развернуть узел **Ресурсы (Resources)** в дереве панели навигации в левой части веб-консоли, затем развернуть узел **Ресурсы JMS (JMS Resources)**, щелкнуть по узлу **Ресурсы пунктов назначения**

(Destination Resources) и, наконец, нажать кнопку **Новый... (New...)** в основной панели веб-консоли.



В нашем примере JNDI-именем очереди сообщений является **jms/GlassFishBookQueue**. Типом ресурса для создаваемой очереди сообщений должен быть **javax.jms.Queue**. Дополнительно нужно ввести **Имя физического пункта назначения** (Physical Destination Name). В нашем примере в качестве значения этого поля используется **GlassFishBookQueue**.

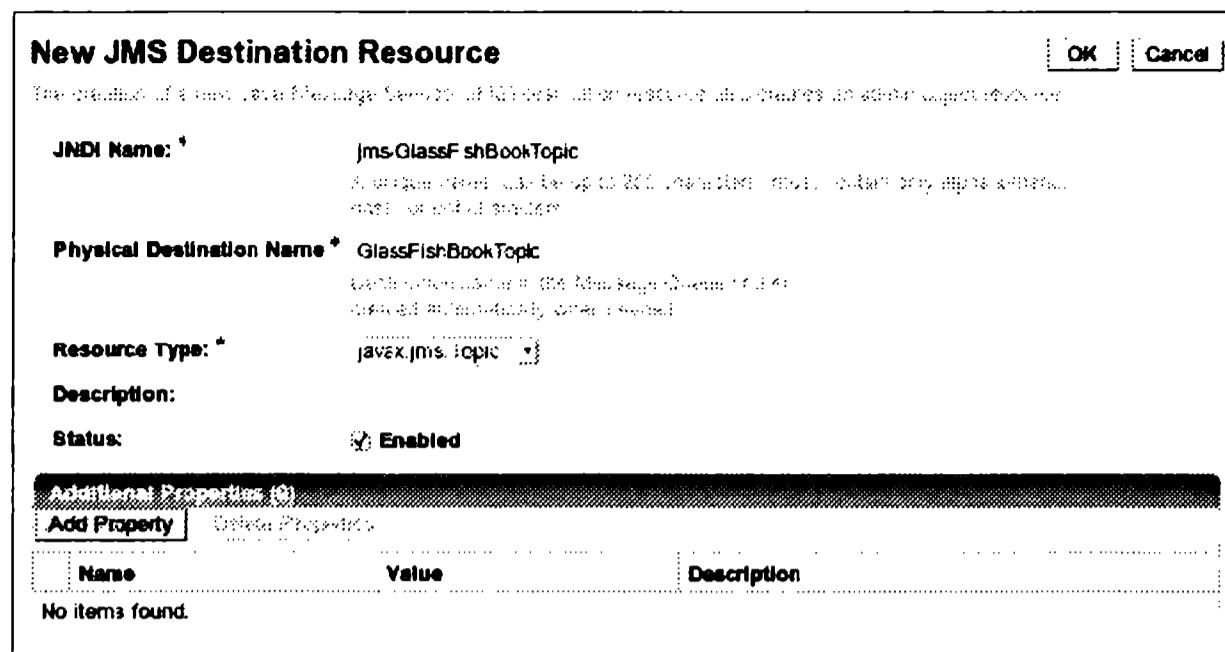
После щелчка по кнопке **Новая... (New...)**, ввода соответствующей информации для нашей очереди сообщений и нажатия кнопки **OK** мы увидим созданную нами новую очередь в списке:



Создание темы JMS-сообщений

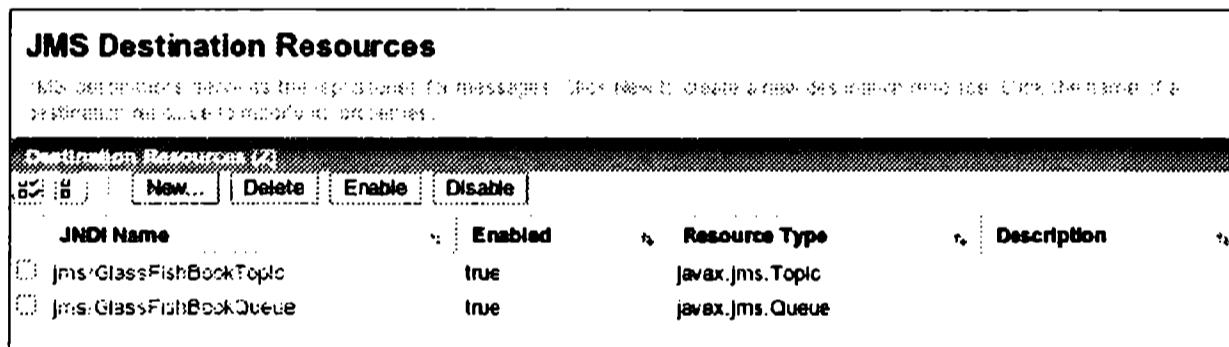
Создание темы JMS-сообщения в GlassFish очень похоже на создание очереди сообщений.

В веб-консоли GlassFish нужно развернуть узел **Ресурсы (Resources)** в дереве панели навигации в левой части веб-консоли, затем развернуть узел **Ресурсы JMS (JMS Resources)**, щелкнуть по узлу **Ресурсы пунктов назначения (Destination Resources)** и, наконец, по кнопке **Новая... (New...)** в основной панели веб-консоли.



В наших примерах будет использоваться **JNDI-имя** (JNDI Name) **jms/GlassFish-BookTopic**. Поскольку это тема сообщений, то в поле **Тип ресурса** (Resource Type) должен быть установлен тип **javax.jms.Topic**. Поле **Описание** (Description) заполнять не обязательно. **Имя физического пункта назначения** (Physical Destination Name) требуется указать; для нашего примера мы будем использовать имя **Glass-FishBookTopic**.

После щелчка по кнопке **OK** мы увидим созданную нами новую тему сообщений в списке:



Теперь, когда у нас есть созданные фабрика соединений, очередь сообщений и тема сообщений, мы готовы начать писать код с использованием API JMS.

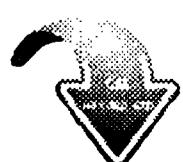
Очереди сообщений

Как уже упоминалось ранее, очереди сообщений используются, когда наш код JMS использует домен обмена сообщениями типа «точка-точка» (PTP). У домена обмена сообщениями PTP обычно имеется один *продюсер сообщений* (message producer) и один *потребитель сообщений* (message consumer). Продюсер и потребитель сообщения не обязаны работать одновременно для того, чтобы взаимодействовать. Сообщение, помещенное в очередь сообщений продюсером сообщений, останется в очереди до тех пор, пока потребитель сообщений не выполнит запрос сообщения из очереди.

Отправка сообщений в очередь сообщений

Следующий пример поясняет, как добавить сообщения в очередь сообщений:

```
package net.ensode.glassfishbook;
import javax.annotation.Resource;
import javax.jms.Connection;
```



```
import javax.jms.ConnectionFactory;
import javax.jms.JMSEException;
import javax.jms.MessageProducer;
import javax.jms.Queue;
import javax.jms.Session;
import javax.jms.TextMessage;
public class MessageSender
{
    @Resource(mappedName = "jms/GlassFishBookConnectionFactory")
    private static ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/GlassFishBookQueue")
    private static Queue queue;
    public void produceMessages()
    {
        MessageProducer messageProducer;
        TextMessage textMessage;
        try
        {
            Connection connection = connectionFactory.createConnection();
            Session session = connection.createSession(false, Session.
                AUTO_ACKNOWLEDGE);
            messageProducer = session.createProducer(queue);
            textMessage = session.createTextMessage();
            textMessage.setText("Тестирование, 1, 2, 3. Меня слышно?");
            System.out.println("Отправка следующего сообщения: " +
                textMessage.getText());
            messageProducer.send(textMessage);
            textMessage.setText("Скопировал?");
            System.out.println("Отправка следующего сообщения: " +
                textMessage.getText());
            messageProducer.send(textMessage);
            textMessage.setText("До свидания!");
            System.out.println("Отправка следующего сообщения: " +
                textMessage.getText());
            messageProducer.send(textMessage);
            messageProducer.close();
            session.close();
            connection.close();
        }
        catch (JMSEException e)
        {
            e.printStackTrace();
        }
    }
    public static void main(String[] args)
    {
        new MessageSender().produceMessages();
    }
}
```

Прежде чем мы углубимся в детали этого кода, отметим, что это приложение относится к классу автономных приложений Java в связи с тем, что оно содержит метод `main`. Поскольку это автономное приложение, оно выполняется вне сервера приложений. Несмотря на это мы видим, что в него инжектируются (вводятся) некоторые ресурсы, в частности фабрика соединений и очередь. Причина, по которой мы можем инжектировать ресурсы в этот код – даже при том, что он работает вне сервера приложений, – состоит в том, что сервер GlassFish включает утилиту, называемую `appclient`.

Эта утилита позволяет нам «оберывать» исполняемый JAR-файл и позволяет ему иметь доступ к ресурсам сервера приложений. Для выполнения предыдущего кода, принимая во внимание сказанное, его нужно упаковать в исполняемый JAR-файл `jmsptpproducer.jar`. Для этого нам необходимо ввести следующую команду в командной строке:

```
appclient -client jmsptpproducer.jar
```

Затем, после внесения некоторых записей в журнал GlassFish, мы увидим следующий вывод на консоль:

Отправка следующего сообщения: Тестирование, 1, 2, 3. Меня слышно?

Отправка следующего сообщения: Скопировал?

Отправка следующего сообщения: До свидания!

Исполняемую программу `appclient` можно найти в [*Каталог установки GlassFish*]/GlassFish/bin. Предыдущий пример предполагает, что этот каталог находится в нашей переменной PATH. Если это не полный путь к исполняемой программе `appclient`, то он должен быть введен в командной строке.

После этого небольшого отступления мы можем объяснить код.

Метод `produceMessages()` выполняет все необходимые действия для отправки сообщения очереди сообщений.

Первое, что делает этот метод, – получает JMS-соединение, вызывая метод `createConnection()` на инжектированном экземпляре `javax.jms.ConnectionFactory`. Обратите внимание, что атрибут `mappedName` аннотации `@Resource`, декорирующий объект фабрики соединений, соответствует JNDI-имени фабрики соединений, которое мы установили в веб-консоли сервера GlassFish. «За кулисами» выполняется JNDI-поиск, использующий это имя для получения объекта фабрики соединений.

После получения соединения следующим шагом будет получение JMS-сессии из названного соединения. Это может быть выполнено путем вызова метода `createSession()` на объекте `Connection`. Как видно из предыдущего кода, метод `createSession()` принимает два параметра.

Первый параметр метода `createSession()` является логической переменной (`Boolean`), указывающей, выполняет ли сеанс транзакцию. Если его значением является `true` (истина), несколько сообщений могут быть отправлены как часть транзакции путем вызова метода `commit()` на объекте сеанса. Точно так же они могут откатываться вызовом его метода `rollback()`.

Второй параметр метода `createSession()` указывает, каким образом получатель сообщений будет подтверждать получение сообщения. Допустимые значения этого параметра определены как константы в интерфейсе `javax.jms.Session`:

- **Session.AUTO_ACKNOWLEDGE**: указывает, что сеанс автоматически подтвердит получение сообщения;
- **Session.CLIENT_ACKNOWLEDGE**: указывает, что получатель сообщения должен явно вызвать метод `acknowledge()` на сообщении.

- **Session.DUPS_OK_ACKNOWLEDGE**: указывает, что сеанс использует «левивое» (по требованию) подтверждение получения сообщений. Использование этого значения может привести к доставке некоторых сообщений более одного раза.

После получения сеанса JMS мы получаем экземпляр `javax.jms.MessageProducer`, вызывая метод `createProducer()` на объекте сеанса. Объект `MessageProducer` является фактическим отправителем сообщения очереди сообщений. Инжектированный экземпляр `Queue` передается в качестве параметра методу `createProducer()`. Значение атрибута `mappedName` для аннотации `@Resource`, декорирующей этот объект, должно опять же соответствовать JNDI-имени, которое мы дали нашей очереди сообщений при ее создании в веб-консоли GlassFish.

После получения экземпляра `MessageProducer` код создает ряд текстовых сообщений, вызывая метод `createTextMessage()` на объекте сеанса. Данный метод возвращает экземпляр класса, реализующего интерфейс `javax.jms.TextMessage`. Этот интерфейс определяет метод, называемый `setText()`, который используется для фактической установки текста в сообщении. После создания каждого текстового сообщения и установки их текста они отправляются очереди путем вызовов метода `send()` на объекте `MessageProducer`.

После отправки сообщений код отсоединяется от очереди JMS, вызывая метод `close()` на объектах `MessageProducer`, `Session` и `Connection`.

Хотя предыдущий пример отправляет только текстовые сообщения очереди, мы не ограничены этим типом сообщений. API JMS предоставляет несколько типов сообщений, которые могут быть отправлены и получены приложениями JMS. Все типы сообщения определяются как интерфейсы в пакете `javax.jms`.

В следующей таблице перечислены все доступные типы сообщений:

Тип сообщения	Описание
<code>BytesMessage</code>	Позволяет отправить массив байтов в виде сообщения
<code>MapMessage</code>	Позволяет отправить реализацию <code>java.util.Map</code> в виде сообщения
<code>ObjectMessage</code>	Позволяет отправить любой объект Java <code>java.io.Serializable</code> в виде сообщения
<code>StreamMessage</code>	Позволяет отправить массив байтов в виде сообщения. Отличие от <code>BytesMessage</code> состоит в том, что он сохраняет тип каждого примитивного типа, добавляя его к потоку
<code>TextMessage</code>	Позволяет отправить <code>java.lang.String</code> в виде сообщения

Для получения дополнительной информации по всем этим типам сообщений обратитесь к их JavaDoc-документации, доступной по адресу: <http://docs.oracle.com/javaee/6/api/>.

Извлечение сообщений из очереди сообщений

Нет никакого смысла в отправке сообщений в очередь, если никто не собирается их получать. Следующий пример поясняет, как извлечь сообщения из очереди сообщений JMS:

```
package net.ensode.glassfishbook;

import javax.annotation.Resource;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSEException;
import javax.jms.MessageConsumer;
import javax.jms.Queue;
import javax.jms.Session;
import javax.jms.TextMessage;
public class MessageReceiver
{
    @Resource(mappedName = "jms/GlassFishBookConnectionFactory")
    private static ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/GlassFishBookQueue")
    private static Queue queue;
    public void getMessages()
    {
        Connection connection;
        MessageConsumer messageConsumer;
        TextMessage textMessage;
        boolean goodByeReceived = false;
        try
        {
            connection = connectionFactory.createConnection();
            Session session = connection.createSession(false, Session.
                AUTO_ACKNOWLEDGE);
            messageConsumer = session.createConsumer(queue);
            connection.start();
            while (!goodByeReceived)
            {
                System.out.println("Ожидание сообщений... ");
                textMessage = (TextMessage) messageConsumer.receive();
                if (textMessage != null)
                {
                    System.out.print("Получено следующее сообщение: ");
                    System.out.println(textMessage.getText());
                    System.out.println();
                }
                if (textMessage.getText() != null &&
                    textMessage.getText().equals("До свидания!"))
                {
                    goodByeReceived = true;
                }
            }
            messageConsumer.close();
            session.close();
            connection.close();
        }
        catch (JMSEException e)
        {
            e.printStackTrace();
        }
    }
}
```



```
public static void main(String[] args)
{
    new MessageReceiver().getMessages();
}
```

Как и в предыдущем примере, экземпляр `javax.jms.ConnectionFactory` и экземпляр `javax.jms.Queue` инжектируются с использованием аннотации `@Resource`. Получение соединения и сеанса JMS осуществляется аналогично предыдущему примеру.

В этом примере мы получаем экземпляр `javax.jms.MessageConsumer`, вызывая метод `createConsumer()` на объекте сеанса JMS. Когда мы будем готовы запустить прием сообщений из очереди сообщений, мы должны вызвать метод `start()` на объекте JMS-соединения.



Код не принимает сообщений?

Распространенная ошибка при записи JMS-сообщений состоит в том, что мы не вызываем метод `start()` на объекте JMS-соединения. Если наш код не получает сообщений (а он их должен получать), то мы должны убедиться, что не забыли вызвать этот метод.

Сообщения принимаются вызовом метода `receive()` на экземпляре `MessageConsumer`, полученном из сеанса JMS. Этот метод возвращает экземпляр класса, реализующего интерфейс `javax.jms.Message`. Он должен быть преобразован к соответствующему типу для получения фактического сообщения.

В данном примере этот метод вызова помещен в цикле, чтобы мы смогли убедиться, что никакие другие сообщения больше не поступают. В частности, мы ищем сообщение, содержащее текст «До свидания!». Как только мы получаем упомянутое сообщение, мы выходим из цикла и продолжаем обработку. В данном конкретном случае для обработки больше ничего нет. Поэтому мы просто вызываем метод `close()` на объекте потребителя сообщений, на объекте сеанса и на объекте соединения.

Так же, как в предыдущем примере, использование утилиты `appclient` позволяет нам инжектировать ресурсы в код и предотвращает необходимость добавления любых библиотек в CLASSPATH. После выполнения кода через утилиту `appclient` мы должны увидеть следующий вывод в командной строке:

```
appclient -client target/jmsptpconsumer.jar
```

Ожидание сообщений... Получено следующее сообщение: Тестирование, 1, 2, 3. Меня слышно?

Ожидание сообщений... Получено следующее сообщение: Скопировал?

Ожидание сообщений... Получено следующее сообщение: До свидания!

При этом, конечно, предполагается, что предыдущий пример уже выполнялся и сообщения были помещены в очередь сообщений.

Асинхронный прием сообщений из очереди сообщений

Недостаток метода `MessageConsumer.receive()` состоит в том, что он блокирует дальнейшее выполнение, пока сообщение не будет получено из очереди. Мы можем

избежать этого, получая сообщения асинхронно с помощью реализации интерфейса `javax.jms.MessageListener`.

Интерфейс `javax.jms.MessageListener` содержит единственный метод, называемый `onMessage()`. Он принимает экземпляр класса, реализующего интерфейс `javax.jms.Message` в качестве единственного параметра. Следующий пример поясняет типичную реализацию этого интерфейса:

```
package net.ensode.glassfishbook;

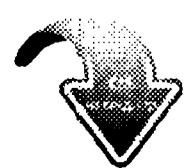
import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;
public class ExampleMessageListener implements MessageListener
{
    @Override
    public void onMessage(Message message)
    {
        TextMessage textMessage = (TextMessage) message;
        try
        {
            System.out.print("Получено следующее сообщение: ");
            System.out.println(textMessage.getText());
            System.out.println();
        }
        catch (JMSEException e)
        {
            e.printStackTrace();
        }
    }
}
```

В этом случае метод `onMessage()` просто выводит текст сообщения в консоли.

Наш основной код теперь может делегировать извлечение сообщения пользовательской реализации `MessageListener`:

```
package net.ensode.glassfishbook;

import javax.annotation.Resource;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSEException;
import javax.jms.MessageConsumer;
import javax.jms.Queue;
import javax.jms.Session;
public class AsynchMessReceiver
{
    @Resource(mappedName = "jms/GlassFishBookConnectionFactory")
    private static ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/GlassFishBookQueue")
    private static Queue queue;
    public void getMessages()
    {
        Connection connection;
        MessageConsumer messageConsumer;
        try
        {
            connection = connectionFactory.createConnection();
            Session session = connection.createSession(false, Session.
```



```

        AUTO_ACKNOWLEDGE);
messageConsumer = session.createConsumer(queue);
messageConsumer.setMessageListener(new ExampleMessageListener());
connection.start();
System.out.println("Вышеприведенная строка позволяет "
+ "с помощью реализации MessageListener организовать "
+ "прием и обработку сообщений из очереди.");
Thread.sleep(1000);
System.out.println("Теперь наш код не блокируется "
+ "до тех пор, пока сообщение не будет получено.");
Thread.sleep(1000);
System.out.println("Он может делать и другие вещи - "
+ "возможно, что-то более полезное, "
+ "чем отправка глупого вывода на консоль. :)");
Thread.sleep(1000);
messageConsumer.close();
session.close();
connection.close();
}
catch (JMSException e)
{
    e.printStackTrace();
}
catch (InterruptedException e)
{
    e.printStackTrace();
}
}
public static void main(String[] args)
{
    new AsynchMessReceiver().getMessages();
}
}
}

```

Единственное различие между этим примером и примером из предыдущего раздела состоит в том, что в данном случае мы вызываем метод `setMessageListener()` на экземпляре `javax.jms.MessageConsumer`, полученном из JMS-сеанса. Мы передаем этому методу экземпляр нашей пользовательской реализации `javax.jms.MessageListener`. Его метод `onMessage()` автоматически вызывается всякий раз, когда имеется сообщение, ожидающее в очереди. При использовании такого подхода основной код не блокируется, ожидая получения сообщения.

Выполнение предыдущего примера (конечно, с использованием утилиты GlassFish `appclient`) имеет результатом следующий вывод:

```
appclient -client target/jmsptpasynchconsumer.jar
```

Вышеприведенная строка позволяет с помощью реализации `MessageListener` организовать прием и обработку сообщений из очереди.

Ожидание сообщений... Получено следующее сообщение: Тестирование, 1, 2, 3. Меня слышно?

Ожидание сообщений... Получено следующее сообщение: Скопировал?

Ожидание сообщений... Получено следующее сообщение: До свидания!

Теперь наш код не блокируется до тех пор, пока сообщение не будет получено.

Он может делать и другие вещи - возможно, что-то более полезное, чем отправка глупого вывода на консоль. :)

Обратите внимание на то, что сообщения были получены и обработаны, в то время как основной поток выполнялся. Мы можем утверждать, что дело обстоит именно так, потому что вывод метода `onMessage()` нашего `MessageListener` можно увидеть между вызовами `System.out.println()` в основном классе.

Просмотр очередей сообщений

JMS предоставляет способ просмотра очереди сообщений, не удаляя фактически сообщения из очереди. Следующий пример поясняет, как это сделать:

```
package net.ensode.glassfishbook;

import java.util.Enumeration;
import javax.annotation.Resource;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSEException;
import javax.jms.Queue;
import javax.jms.QueueBrowser;
import javax.jms.Session;
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;
            Connection connection = connectionFactory.createConnection();
            Session session = connection.createSession(false, Session.
                AUTO_ACKNOWLEDGE);
            QueueBrowser browser = session.createBrowser(queue);
            messageEnumeration = browser.getEnumeration();
            if (messageEnumeration != null)
            {
                if (!messageEnumeration.hasMoreElements())
                {
                    System.out.println("В очереди нет сообщений.");
                }
                else
                {
                    System.out.println("В очереди находятся следующие
                        сообщения:");
                    while (messageEnumeration.hasMoreElements())
                    {
                        textMessage = (TextMessage) messageEnumeration.
                            nextElement();
                        System.out.println(textMessage.getText());
                    }
                }
            }
            session.close();
            connection.close();
        }
        catch (JMSEException e)
```



```
        {
            e.printStackTrace();
        }
    }
    public static void main(String[] args)
    {
        new MessageQueueBrowser().browseMessages();
    }
}
```

Как видно из кода, процедура просмотра сообщений в очереди сообщений элементарна. Мы получаем JMS-соединение и JMS-сессию обычным путем, затем вызываем метод `createBrowser()` на объекте сеанса JMS. Этот метод возвращает реализацию интерфейса `javax.jms.QueueBrowser`. Данный интерфейс содержит метод `getEnumeration()`, который мы можем вызвать для получения перечисления, содержащего все сообщения в очереди. Чтобы исследовать сообщения в очереди, мы просто перебираем это перечисление и получаем сообщения одно за другим. В предыдущем примере мы просто вызываем метод `getText()` каждого сообщения в очереди.

Темы сообщений

Темы сообщений используются, когда наш код JMS использует домен обмена сообщениями типа «публикация/подписка». При использовании этого домена обмена сообщениями одно и то же сообщение может быть отправлено всем подписчикам темы.

Отправка сообщений теме сообщений

Следующий пример поясняет, как отправить сообщение теме сообщений:

```
package net.ensode.glassfishbook;

import javax.annotation.Resource;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSEException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
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()
    {
        MessageProducer messageProducer;
        TextMessage textMessage;
        try
        {
            Connection connection = connectionFactory.createConnection();
            Session session = connection.createSession(false,
                Session.AUTO_ACKNOWLEDGE);
            messageProducer = session.createProducer(topic);
            textMessage = session.createTextMessage();
            textMessage.setText("Тестирование, 1, 2, 3. Меня слышно?");
        }
    }
}
```

```
System.out.println("Отправка следующего сообщения: " +
    textMessage.getText());
messageProducer.send(textMessage);
textMessage.setText("Скопировал?");
System.out.println("Отправка следующего сообщения: " +
    textMessage.getText());
messageProducer.send(textMessage);
textMessage.setText("До свидания!");
System.out.println("Отправка следующего сообщения: " +
    textMessage.getText());
messageProducer.send(textMessage);
messageProducer.close();
session.close();
connection.close();
}
catch (JMSEException e)
{
    e.printStackTrace();
}
}
public static void main(String[] args)
{
    new MessageSender().produceMessages();
}
}
```

Этот код практически идентичен классу `MessageSender`, который мы рассматривали, когда обсуждали обмен сообщениями типа «точка-точка». На самом деле отличаются только несколько выделенных строк кода. API JMS был разработан таким образом, чтобы разработчикам приложений не нужно было изучать два различных API для двух доменов обмена сообщениями РТР и «публикация/подписка».

Поскольку код практически идентичен соответствующему примеру из раздела «*Очереди сообщений*» (см. стр. 245), мы объясним только различия между этими двумя примерами. В данном примере вместо объявления экземпляра класса, реализующего интерфейс `javax.jms.Queue`, мы объявляем экземпляр класса, реализующего интерфейс `javax.jms.Topic`. Так же, как в предыдущих примерах, мы используем инжекцию зависимости для инициализации объекта `Topic`. После получения JMS-соединения и JMS-сессии мы передаем объект `Topic` методу `createProducer()` объекта `Session`. Этот метод возвращает экземпляр `javax.jms.MessageProducer`, который мы сможем использовать для отправки сообщений теме JMS.

Получение сообщений от темы сообщений

Подобно тому как отправка сообщений теме сообщений практически идентична отправке сообщений очереди сообщений, получение сообщений от темы сообщений практически идентично извлечению сообщений из очереди сообщений.

```
package net.ensode.glassfishbook;

import javax.annotation.Resource;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
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/GlassFishBookConnectionFactory")
    private static ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/GlassFishBookTopic")
    private static Topic topic;
    public void getMessages()
    {
        Connection connection;
        MessageConsumer messageConsumer;
        TextMessage textMessage;
        boolean goodByeReceived = false;
        try
        {
            connection = connectionFactory.createConnection();
            Session session = connection.createSession(false, Session.
                AUTO_ACKNOWLEDGE);
            messageConsumer = session.createConsumer(topic);
            connection.start();
            while (!goodByeReceived)
            {
                System.out.println("Ожидание сообщений...");
                textMessage = (TextMessage) messageConsumer.receive();
                if (textMessage != null)
                {
                    System.out.print("Получено следующее сообщение: ");
                    System.out.println(textMessage.getText());
                    System.out.println();
                }
                if (textMessage.getText() != null
                    && textMessage.getText().equals("До свидания!"))
                {
                    goodByeReceived = true;
                }
            }
            messageConsumer.close();
            session.close();
            connection.close();
        }
        catch (JMSException e)
        {
            e.printStackTrace();
        }
    }
    public static void main(String[] args)
    {
        new MessageReceiver().getMessages();
    }
}
```

И опять же, различия между этим кодом и соответствующим кодом для РТР три-виальны. Вместо того чтобы объявить экземпляр класса, реализующего интерфейс `javax.jms.Queue`, мы объявляем класс, реализующий интерфейс `javax.jms.Topic`. Мы используем аннотацию `@Resource`, чтобы инжектировать экземпляр этого класса в наш код, применяя JNDI-имя, которое мы использовали при его создании в веб-консоли сервера GlassFish. После получения JMS-соединения и JMS-сесии мы передаем объект `Topic` методу `createConsumer()` объекта `Session`.

Этот метод возвращает экземпляр `javax.jms.MessageConsumer`, который мы можем использовать для получения сообщений от темы JMS.

Использование домена обмена сообщениями «публикация/подписка», как поясняется в этом разделе, имеет то преимущество, что сообщения могут быть отправлены нескольким потребителям. Это можно легко протестировать, одновременно запустив на выполнение два экземпляра класса `MessageReceiver`, который мы разработали в этом разделе, а затем выполняя экземпляр класса `MessageSender`, который был разработан в предыдущем разделе. Мы должны будем увидеть консольный вывод для каждого экземпляра, свидетельствующий о том, что оба они получали все сообщения.

Так же, как и в случае с очередями сообщений, сообщения от темы сообщений могут быть получены асинхронно. Соответствующая процедура настолько похожа на версию очереди сообщений, что мы не будем приводить пример. Чтобы преобразовать асинхронный пример, представленный выше в этой главе, для использования темы сообщений, просто замените переменную `javax.jms.Queue` экземпляром `javax.jms.Topic` и инжектируйте соответствующий экземпляр использованием `"jms/GlassFishBookTopic"` в качестве значения атрибута `mappedName` аннотации `@Resource`, декорирующей экземпляр `javax.jms.Topic`.

Создание долговременных подписчиков

Недостаток использования домена обмена сообщениями типа «публикация/подписка» состоит в том, что потребители сообщения должны выполняться в то время, когда сообщения отправляются теме. Если потребитель сообщения не выполняется в это время, он не будет получать сообщения, тогда как в PTP сообщения сохраняются в очереди, пока потребитель сообщения не выполнится и не извлечет их. К счастью, API JMS предоставляет способ использовать домен обмена сообщениями типа «публикация/подписка» с сохранением сообщения в теме, пока все подписанные на него потребители не выполнятся и не получат это сообщение. Это может быть достигнуто путем создания долговременных подписчиков на тему JMS.

Чтобы иметь возможность обслуживать долговременных подписчиков, мы должны установить свойство `ClientId` нашей фабрики JMS-соединения. У каждого долговременного подписчика должен быть уникальный клиентский идентификатор, поэтому должна быть объявлена уникальная фабрика соединений для каждого потенциального долговременного подписчика.

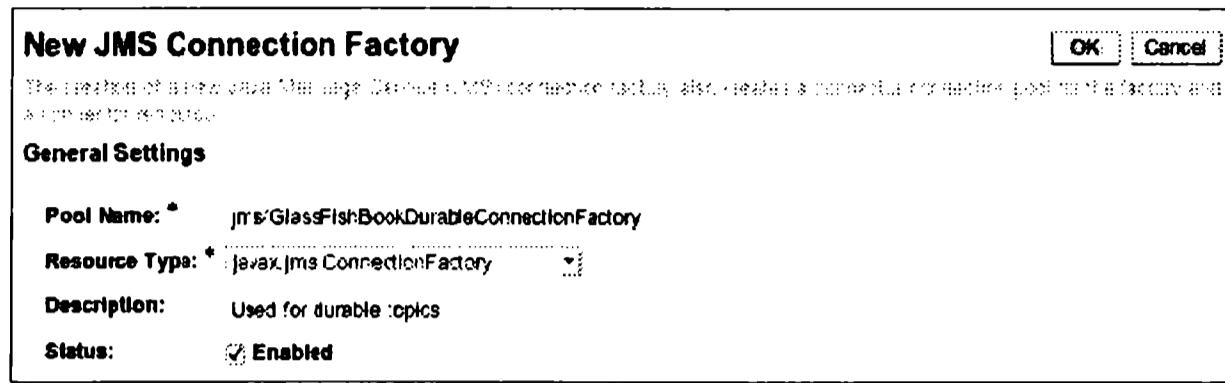


Возникло исключение `InvalidClientIdException`?

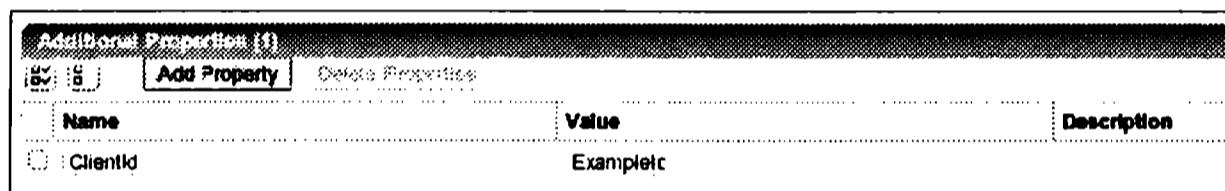
Только одному клиенту JMS с конкретным клиентским идентификатором позволено соединиться с темой. Если несколько клиентов JMS попытаются получить JMS-соединение, используя одну и ту же фабрику соединений, возникнет исключение `JMSException`, уведомляющее, что такой клиентский идентификатор уже используется. Решение этой проблемы заключается в том, чтобы создать фабрику соединений для каждого потенциального клиента, который будет на долговременной основе получать сообщения из темы.

Как мы упоминали ранее, самым простым способом добавления фабрики соединений является использование веб-консоли сервера GlassFish. Напомним, что для добавления соединений JMS через веб-консоль GlassFish мы должны развернуть узел

Ресурсы (Resources) в панели навигации с левой стороны, затем развернуть узел **Ресурсы JMS (JMS Resources)**, далее щелкнуть по вершине **Фабрики соединений (Connection Factories)** и, наконец, по кнопке **Новая... (New...)** в основной панели страницы. Наш следующий пример будет использовать настройки, показанные на этом снимке экрана:



Прежде чем нажать кнопку **OK**, мы должны прокрутить страницу до конца, щелкнуть по кнопке **Добавить свойства (Add Property)** и ввести новое свойство под названием `ClientId`. Наш пример будет использовать `ExampleId` в качестве значения этого свойства.



Теперь, когда у нас есть настройка GlassFish, способная предоставить долговременных подписчиков, мы готовы написать некоторый код для того, чтобы воспользоваться их преимуществами:

```
package net.ensode.glassfishbook;

import javax.annotation.Resource;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
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()
    {
        Connection connection;
        MessageConsumer messageConsumer;
        TextMessage textMessage;
        boolean goodByeReceived = false;
        try
        {
```

```
connection = connectionFactory.createConnection();
Session session = connection.createSession(false, Session.
    AUTO_ACKNOWLEDGE);
messageConsumer = session.createDurableSubscriber(topic,
    "Subscriber1");
connection.start();
while (!goodByeReceived)
{
    System.out.println("Ожидание сообщений...");
    textMessage = (TextMessage) messageConsumer.receive();
    if (textMessage != null)
    {
        System.out.print("Получено следующее сообщение: ");
        System.out.println(textMessage.getText());
        System.out.println();
    }
    if (textMessage.getText() != null
        && textMessage.getText().equals("До свидания!"))
    {
        goodByeReceived = true;
    }
}
messageConsumer.close();
session.close();
connection.close();
}
catch (JMSEException e)
{
    e.printStackTrace();
}
}
public static void main(String[] args)
{
    new MessageReceiver().getMessages();
}
}
```

Как видно из приведенного кода, он практически не отличается от предыдущих примеров, целью которых было получение сообщений. В нем имеется только два отличия от предыдущих примеров: инжектированный нами экземпляр Connection-Factory является экземпляром, созданным нами ранее в этом разделе для обработки долговременных подписчиков, а вместо того, чтобы вызывать метод create-Subscriber() на объекте сеанса JMS, мы вызываем метод createDurable-Subscriber(). Этот метод принимает два аргумента: первым является объект Topic (тема) JMS для извлечения сообщений, а вторым – String (строка), определяющая имя создаваемого подписчика. Второй параметр должен быть уникальным для каждого из долговременных подписчиков темы.

Резюме

В этой главе мы рассмотрели, как настроить фабрики JMS-соединений, очереди и темы JMS сообщений на сервере GlassFish, используя его веб-консоль.

Мы также обсудили, как отправить сообщения очереди сообщений через интерфейс `javax.jms.MessageProducer`.

Было показано, как извлечь сообщения из очереди сообщений через интерфейс `javax.jms.MessageConsumer` и как асинхронно извлечь сообщения из очереди сообщений путем реализации интерфейса `javax.jms.MessageListener`.

Мы также узнали, как использовать эти интерфейсы для отправки и получения сообщения в темы и от тем сообщений JMS.

Мы выяснили, как просмотреть сообщения в очереди сообщений через интерфейс `javax.jms.QueueBrowser`, не удаляя сообщения из очереди.

Наконец, мы показали, как настраивать долговременных подписчиков на темы JMS и взаимодействовать с ними.

8

Безопасность

В этой главе мы рассмотрим, как защитить приложения 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 минимального кодирования. По большей части безопасность приложения достигается путем создания пользователей и групп безопасности (security groups) в областях безопасности (security realms) на сервере приложений; затем выполняется конфигурирование нашего 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

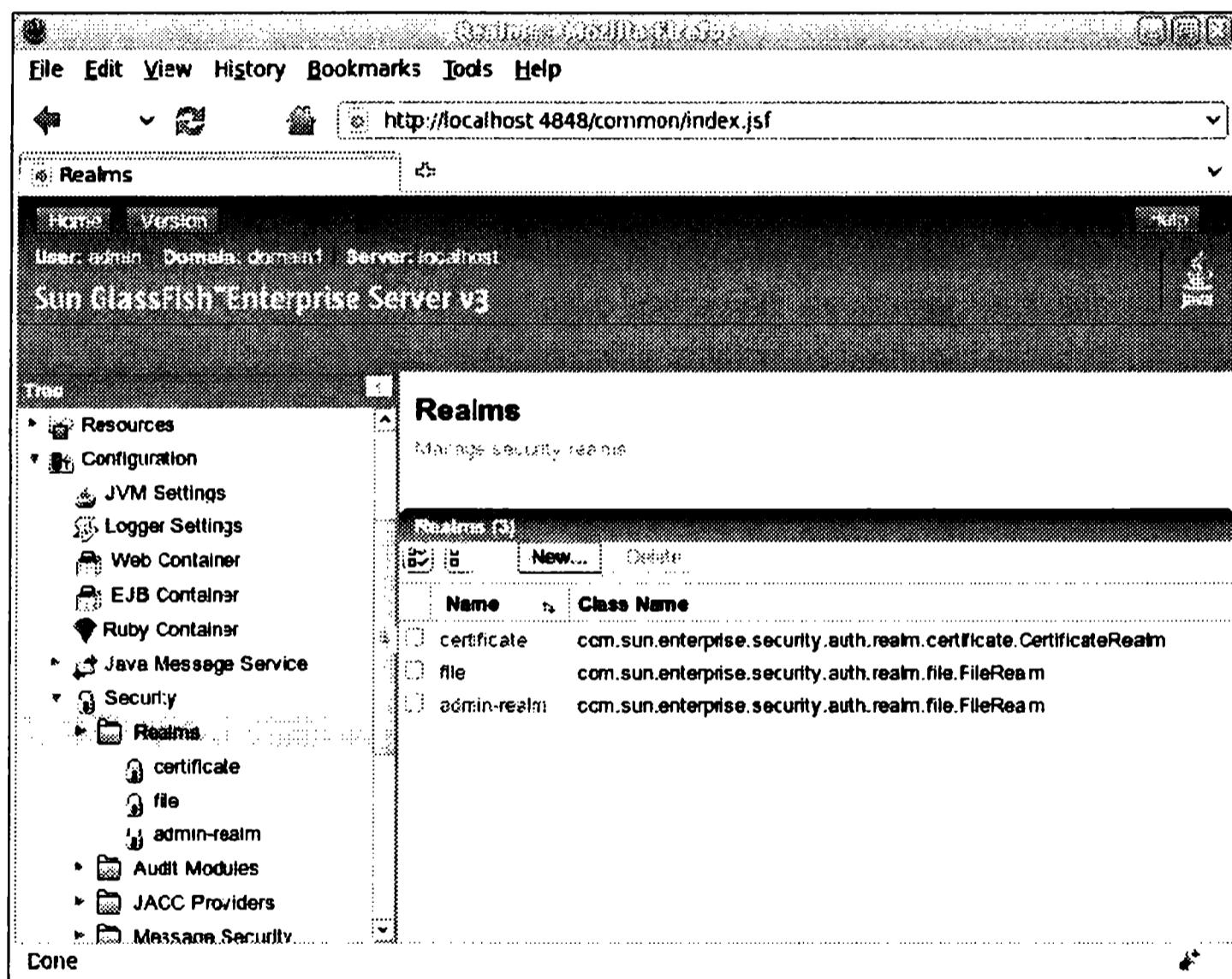
Область безопасности (security realm) представляет собой коллекцию пользователей и связанных с пользователями групп безопасности. Здесь пользователями называются пользователи приложений; пользователь может принадлежать к одной или более группам безопасности. В зависимости от группы, к которой принадлежит пользователь, система позволяет ему выполнять те или иные действия. Например, у приложения могут быть обычные пользователи, которые могут использовать только основную функциональность приложения, и могут быть администраторы, которые, помимо использования основной функциональности приложения, наделены правом добавлять в систему других пользователей.

Области безопасности хранят пользовательскую (учетную) информацию, в частности имя пользователя, пароль и группы безопасности. Приложениям не нужно

реализовывать эту функциональность; они просто могут быть сконфигурированы для получения данной информации из области безопасности. Одна область безопасности может использоваться несколькими приложениями.

Предопределенные области безопасности

GlassFish поставляется с тремя предварительно сконфигурированными предопределенными областями безопасности: *областью администратора* (admin realm), *областью файла* (file realm) и *областью 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.



В дополнение к предопределенным областям безопасности мы можем добавить дополнительные области, прикладывая минимум усилий. Мы рассмотрим, как это сделать, чуть ниже, но сначала обсудим предопределенные области безопасности GlassFish.

Область администратора

Чтобы пояснить, как добавлять пользователей в область, давайте добавим нового пользователя в область администратора. Это позволит данному пользователю входить в систему к веб-консоли сервера GlassFish. Для добавления пользователя в область администратора нужно войти в систему к веб-консоли сервера GlassFish, развернуть узел **Конфигурация** (Configuration) в панели навигации в левой части веб-

страницы, затем развернуть узел **Безопасность** (Security), узел **Области** (Realms) и щелкнуть по области администратора (admin realm). Справа, в области панели основного содержания, должна отобразиться страница, похожая на следующую:

Edit Realm

Save | Cancel

Realm Name: admin-realm

Class Name: com.sun.enterprise.security.auth.realm.FileRealm

Properties specific to this Class

JAAS Context: * fileRealm
Identifier for the login module to use for this realm.

Key File: * \$com.sun.ee.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:

Add Additional Properties | Add Property | Create Properties

Name	Value	Description
No items found.		

Save | Cancel

Чтобы добавить пользователя в область, щелкните по кнопке с называнием **Управление пользователями** (Manage Users) в верхнем левом углу. Панель основного содержания страницы теперь должна выглядеть следующим образом:

Admin Users

Manage user accounts for the currently selected security realm

Realm Name: admin-realm

Admin Users (1)

New... Delete

User ID	Group List
admin	asadmin

Back

Чтобы добавить пользователя в область, просто щелкните по кнопке **Новый...** (New...) в верхнем левом углу экрана, затем введите данные нового пользователя.

New File Realm User

Create new user account for the currently selected security realm.

Realm Name: admin-realm

User ID: * root
Grant this user the granted access to this realm. Please enter up to 100 characters. Root users can edit themselves, or change other users' passwords.

Group List: esadmin

New Password: *****

Confirm New Password: *****

OK | Cancel

На последнем снимке экрана видно, что мы добавили нового пользователя с именем **root** в группу **asadmin**, а также ввели пароль для этого пользователя.



Веб-консоль сервера GlassFish позволяет только пользователям группы **asadmin** входить в систему. Возникновение ошибки при добавлении пользователя в эту группу безопасности будет препятствовать его входу в консоль.

User ID	Group List
admin	asadmin
root	asadmin

Теперь, когда мы успешно включили нового пользователя в область администратора, мы можем протестировать новую учетную запись, вводя в веб-консоли GlassFish учетные данные добавленного пользователя.

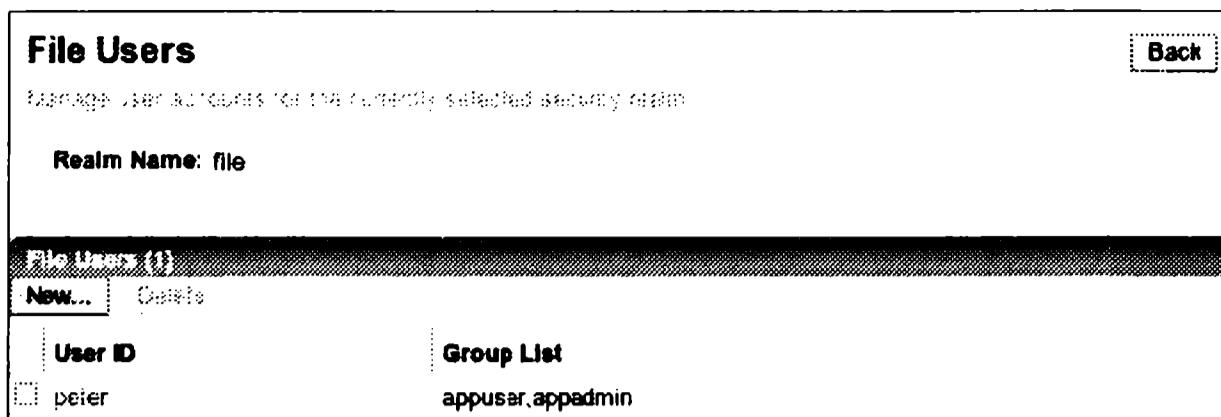
Область файла

Вторая предопределенная область сервера GlassFish – это область файла. Она хранит зашифрованную пользовательскую информацию в текстовом файле. Добавление пользователей к этой области практически аналогично добавлению пользователей к области администратора. Мы можем добавить пользователя, разворачивая узел **Конфигурация** (Configuration), затем – узел **Безопасность** (Security) и узел **Области** (Realms), далее щелкая по пункту **файл** (file), затем по кнопке **Управление пользователями** (Manage Users) и, наконец, по кнопке **Новая...** (New...).

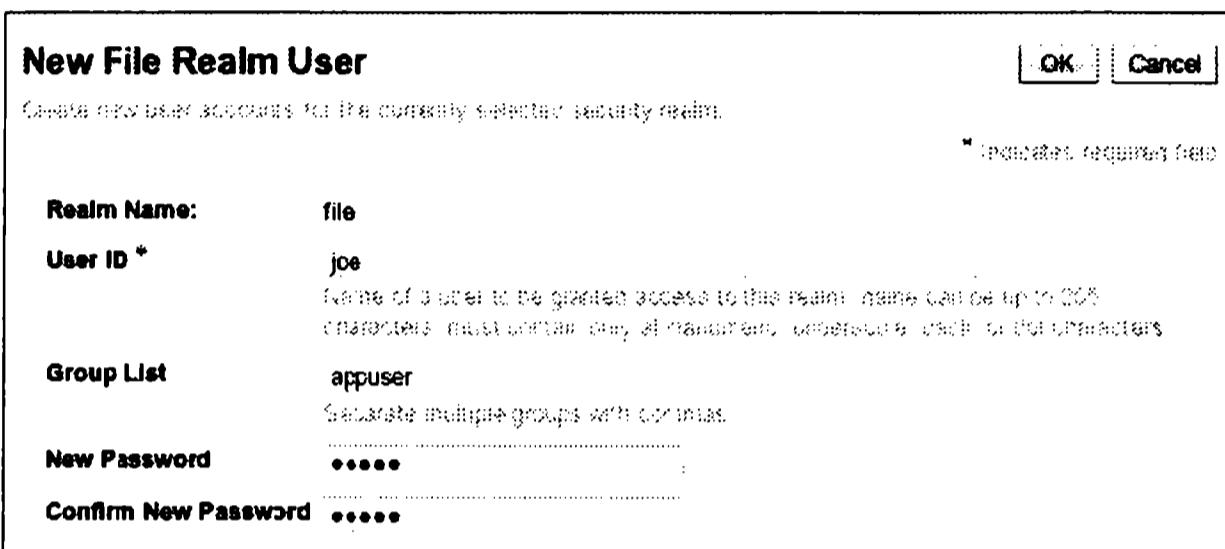
Realm Name:	file
User ID *	peter
Group List	appuser, appadmin
New Password	*****
Confirm New Password	*****

Поскольку эта область предназначается для использования нашими приложениями, мы можем придумать наши собственные группы. В этом примере мы добавили пользователя с **Идентификатором пользователя** (User ID) Peter в группы **appuser** и **appadmin**.

Щелчок по кнопке **OK** должен сохранить нового пользователя и вернуть нас к списку пользователей данной области.



Щелкая по кнопке **Новый...** (New...), можно добавлять других пользователей в область. Давайте добавим дополнительного пользователя с именем **joe**, принадлежащего только группе **appuser**:



Как мы уже убедились, добавить пользователей в область файла очень просто. Теперь поясним, как аутентифицировать и авторизовать пользователей с помощью области файла.

Стандартная аутентификация через область файла

В предыдущем разделе мы рассмотрели, как добавить пользователей в область файла и как занести этих пользователей в группы. В этом разделе будет показано, как защитить веб-приложение, чтобы только пользователи, должным образом прошедшие аутентификацию и авторизацию, могли получить к нему доступ. Это веб-приложение будет использовать область файла для управления доступом пользователей.

Приложение будет состоять из нескольких очень простых JSP-страниц. Обо всей логике аутентификации заботится сервер приложений. Таким образом, единственное место, куда нам требуется внести изменения для обеспечения безопасности приложения, находится в дескрипторах развертывания `web.xml` и `sun-web.xml`. Начнем с `web.xml`, который показан ниже:

```
<?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>
</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 определенному шаблону URL. Шаблон URL страницы определяется элементом `<url-pattern>`, который, как показано в примере, должен быть вложен в элемент `<web-resource-collection>`. Роли, которым разрешен доступ к странице, определяются элементом `<role-name>`, который должен быть вложен в элемент `<auth-constraint>`.

В этом примере мы определяем два набора страниц, которые должны быть защищены. В первый набор входит любая страница, URL которой начинается с `/admin`. К этим страницам могут получить доступ только пользователи с ролью администратора. Во второй набор входят все страницы, определенные шаблоном URL `/*`. Только пользователи с ролью `user` могут получить доступ к этим страницам. Стоит отметить, что второй набор страниц является надмножеством первого набора, т. е. любая страница, URL которой соответствует `/admin/*`, также соответствует шаблону `/*`. В подобных случаях «побеждает» наиболее конкретный вариант. В данном примере пользователи с ролью `user` (не наделенные ролью администратора) не смогут получать доступ к страницам, URL которых начинается с `/admin`.

Следующий элемент, который мы должны добавить в файл `web.xml` для защиты наших страниц, – `<login-config>`. Он должен содержать элемент `<auth-method>`, который определяет метод аутентификации для приложения. Допустимые значения данного элемента: `BASIC`, `DIGEST`, `FORM` и `CLIENT-CERT`.

Значение `BASIC` (стандартная) указывает, что будет использоваться стандартная аутентификация. Этот тип аутентификации имеет результатом сгенерированное обозревателем всплывающее приглашение пользователю ввести имя и пароль, которое будет отображаться при первой попытке пользователя получить доступ к защищенной странице. Если используется протокол HTTPS, при использовании стандартной (`BASIC`) аутентификации учетные данные пользователя будут представлены в кодировке Base64, в незашифрованном виде. Злоумышленнику не составит особого

труда расшифровать эти учетные данные, поэтому использовать стандартную аутентификацию не рекомендуется.

Значение DIGEST (отпечаток) подобно стандартной аутентификации за исключением того, что используется дайджест MD5 для шифрования учетных данных пользователя вместо их отправки в формате Base64.

Значение FORM (форма) использует пользовательскую HTML- или JSP-страницу, содержащую HTML-форму с полями имени пользователя и пароля. Значения, введенные в эту форму, затем проверяются по области безопасности для пользовательской аутентификации и авторизации. Если используется протокол HTTPS, учетные данные пользователя не отправляются открытым текстом при использовании аутентификации на основе формы. В связи с этим рекомендуется использование именно протокола HTTPS, поскольку он шифрует данные. Мы рассмотрим настройку GlassFish для использования HTTPS позже в этой главе.

Значение CLIENT-CERT (клиентский сертификат) использует клиентские сертификаты для аутентификации и авторизации пользователя.

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 мы приступим к успешной аутентификации наших пользователей, необходимо связать роли пользователей, определенные в файле `web.xml`, с группами, определенными в области. Это реализуется в файле дескриптора развертывания `sun-web.xml`:

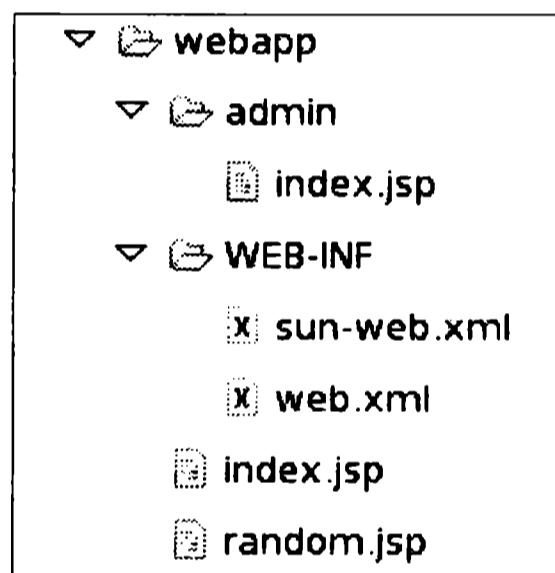
```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD
    Application Server 9.0 Servlet 2.5//EN"
    "http://www.sun.com/software/appserver/dtds/sun-web-app_2_5-0.dtd">
<sun-web-app>
    <security-role-mapping>
        <role-name>admin</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>
</sun-web-app>
```

Как видно из примера, дескриптор развертывания `sun-web.xml` может иметь один или более элементов `<security-role-mapping>`. Для каждой роли, определенной в файле `web.xml`, необходимо наличие одного такого элемента. Подэлемент

<role-name> указывает роль для отображения. 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.

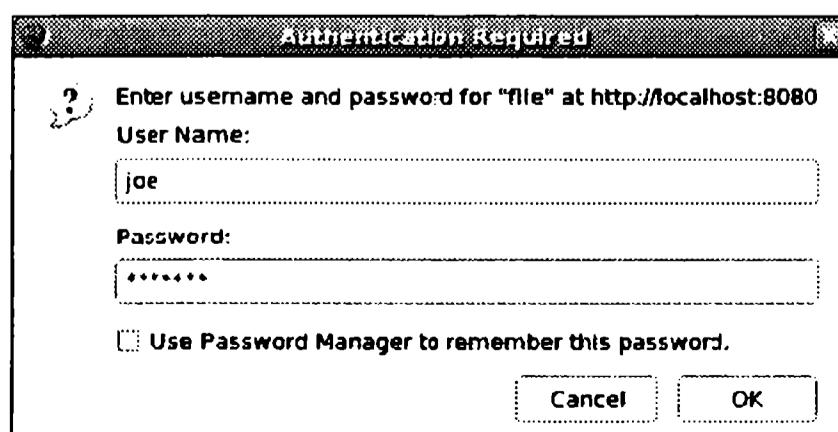
В нашем примере первый элемент <security-role-mapping> отображает роль admin, определенную в дескрипторе развертывания приложения web.xml в группе appadmin, которую мы создали ранее, добавляя пользователей в область файла. Второй элемент <security-role-mapping> отображает роль user в файле web.xml на группу appuser в области файла.

Как мы уже упоминали, нам не нужно ничего делать в нашем коде для аутентификации и авторизации пользователей. Все, что мы должны сделать, – это модифицировать дескрипторы развертывания приложения, как описано в данном разделе. Поскольку наше приложение состоит только из нескольких простых JSP-страниц, мы не будем показывать исходный код для них. Структура нашего приложения показана на следующем снимке экрана:

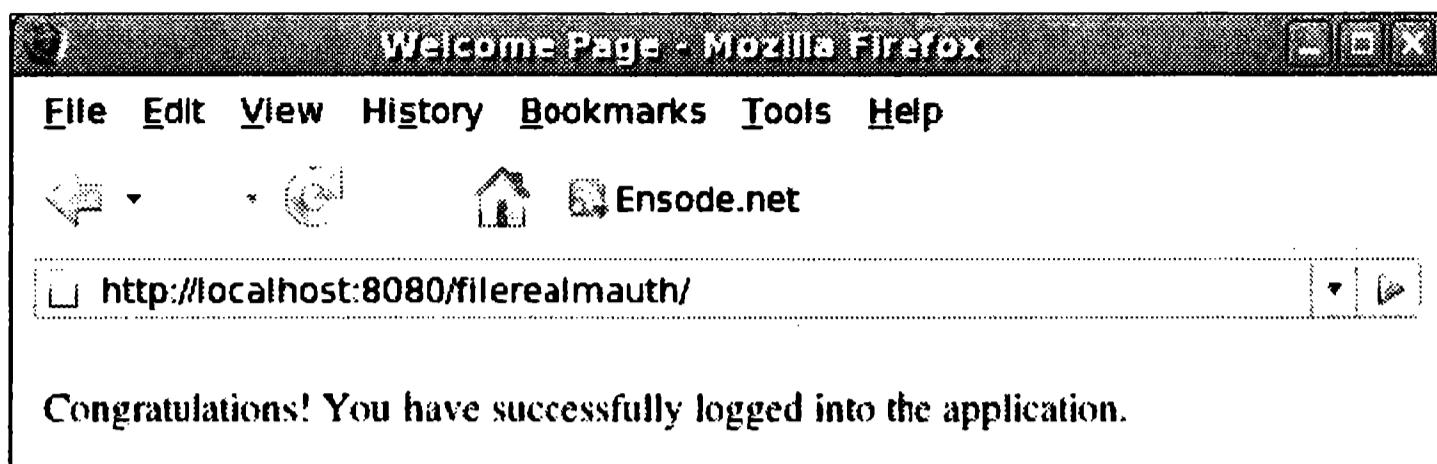


В соответствии с созданным нами дескриптором развертывания для нашего приложения пользователи с ролью user смогут получить доступ к двум JSP-страницам в корне приложения (index.jsp и random.jsp). Только пользователи с ролью admin смогут получить доступ к любой странице в каталоге admin, где в данном случае находится единственная JSP-страница с названием index.jsp.

После упаковки и развертывания нашего приложения и указания в адресной строке обозревателя URL любой из страниц мы должны увидеть всплывающее диалоговое окно с запросом о вводе имени пользователя и пароля.

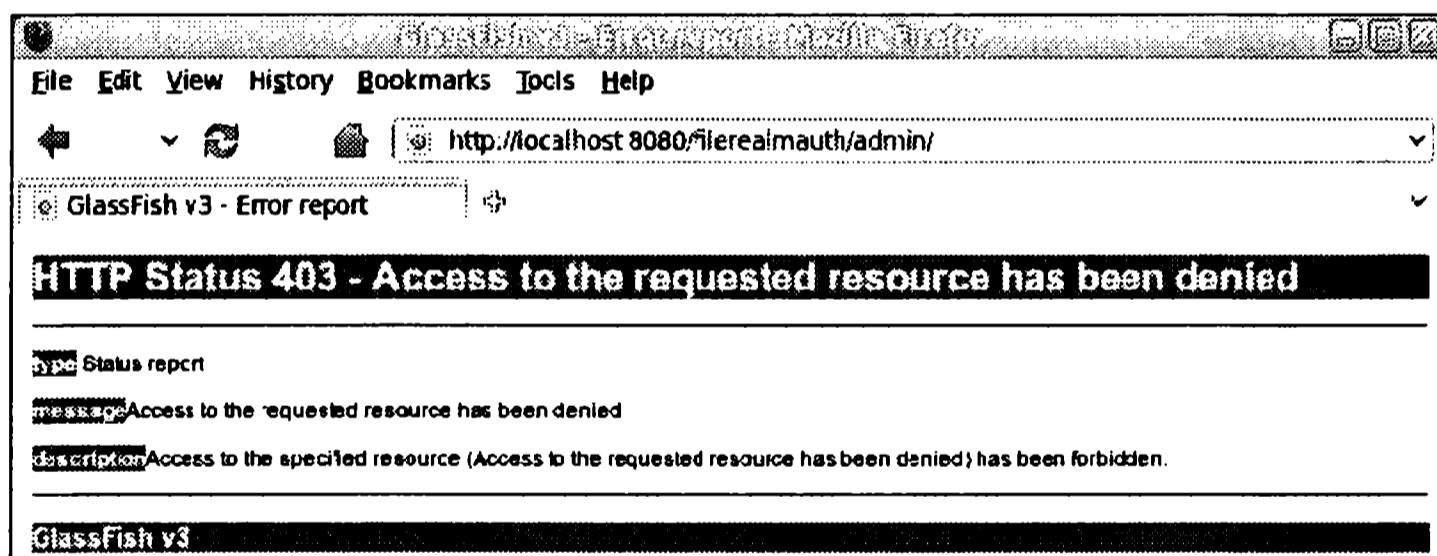


После ввода корректного имени пользователя и пароля мы будем направлены на страницу, которую мы пытались просмотреть:



Теперь пользователь может переместиться к любой из страниц из числа тех, к которым ему разрешено получить доступ в приложении, – либо по ссылкам, либо вводя URL в адресной строке обозревателя, без необходимости повторного ввода имени пользователя и пароля.

Обратите внимание, что мы входили в систему как пользователь **joe** (Джо). Этот пользователь принадлежит только роли user, поэтому у него нет доступа к любой странице, которая начинается, например с URL /admin. Если Джо попытается получить доступ к одной из этих страниц, он увидит в обозревателе сообщение об ошибке:



Только пользователи с ролью admin смогут увидеть страницы, соответствующие данному URL. Добавляя пользователей в область файла, мы включили туда пользователя по имени Peter, у которого была эта роль. Если мы войдем в систему как Peter, то сможем увидеть требуемую страницу. В случае стандартной аутентификации единственный способ выйти из приложения – закрыть обозреватель. Поэтому, чтобы войти в систему как Peter, нам придется закрыть и вновь открыть обозреватель.



Как мы упоминали выше, один из недостатков стандартного метода аутентификации, который использовался нами в этом примере, состоит в том, что информация о входе в систему не шифруется. Один из способов обойти эту проблему – использование протокола HTTPS (HTTP по SSL). В таком случае вся информация, передаваемая между обозревателем и сервером, шифруется.

Самый простой способ задействовать HTTPS заключается в модификации дескриптора развертывания приложения – файла 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 видно из приведенного кода, для получения приложения, к которому доступ выполняется только через HTTPS, нам нужно всего лишь добавить элемент <user-data-constraint>, содержащий вложенный элемент <transport-guarantee>, к каждому набору страниц, которые мы хотим зашифровать. Наборы страниц, которые должны быть защищены, объявляются в элементе <security-constraint> в файле дескриптора развертывания web.xml.

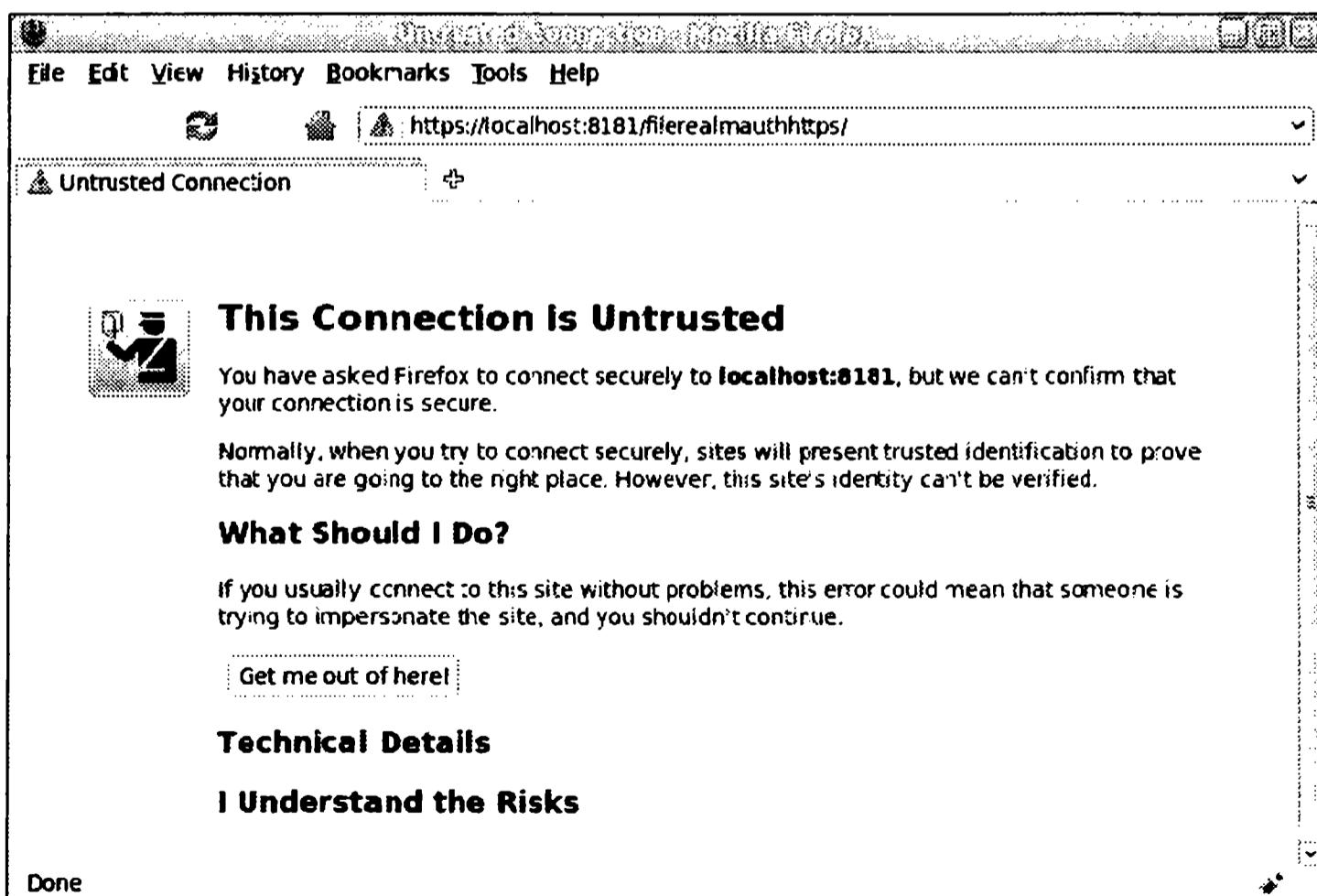
Теперь, когда мы будем получать доступ к приложению через небезопасный порт HTTP (по умолчанию – 8080), запрос будет автоматически переадресован к безопасному порту HTTPS (значение по умолчанию – 8181).

В данном примере мы назначаем элементу `<transport-guarantee>` значение `CONFIDENTIAL`. Оно имеет эффект шифрования всех данных, передаваемых между обозревателем и сервером. Кроме того, если с запросом обращаются через небезопасный порт HTTP, запрос автоматически переадресуется защищенному порту HTTPS.

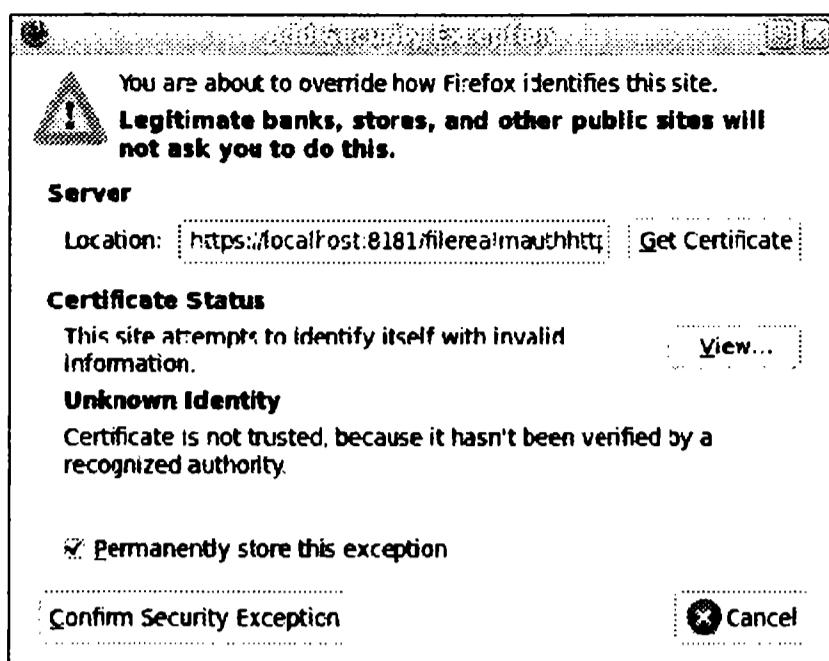
Другим допустимым значением для элемента `<transport-guarantee>` является `INTEGRAL`. При использовании этого значения гарантируется целостность данных, передаваемых между обозревателем и сервером. Другими словами, данные не могут быть изменены при их транспортировке. При использовании этого значения запросы, переданные по HTTP, не переадресуются автоматически порту HTTPS. Если пользователь попытается получить доступ к защищенной странице через HTTP, когда используется это значение, то обозреватель отклонит запрос и возвратит ошибку 403 – «Доступ запрещен».

Третьим и последним допустимым значением элемента `<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>` не присутствует в дескрипторе развертывания приложения – `web.xml`.

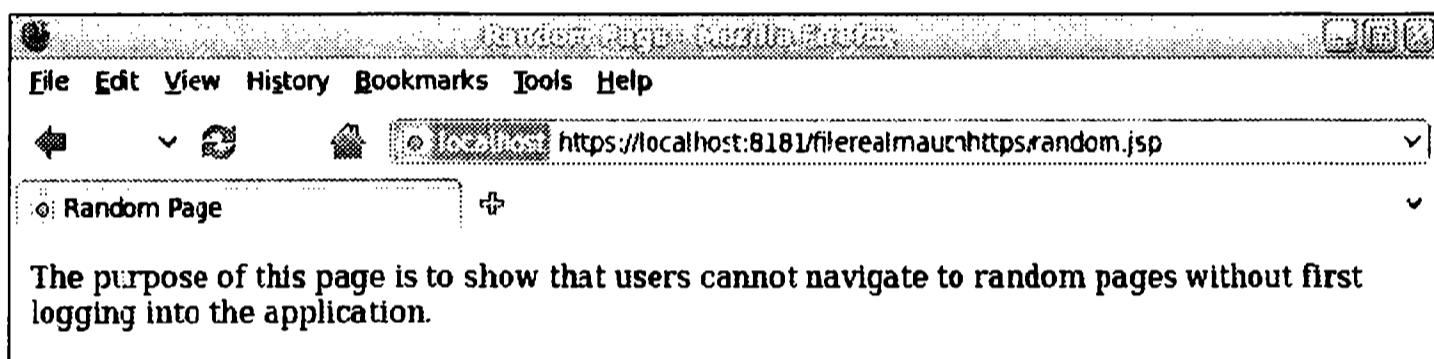
После внесения вышеперечисленных изменений в дескриптор развертывания приложения `web.xml`, повторного развертывания приложения и указания в адресной строке обозревателя URL любой страницы приложения мы увидим следующую страницу предупреждения при обращении к нашему приложению с помощью Firefox:



После развертывания пункта **Я осознаю риски** (*I Understand the Risks*) и щелчка по кнопке **Добавить исключение...** (*Add Exception...*) мы должны увидеть окно наподобие следующего:



После щелчка по кнопке **Подтверждаю исключение безопасности** (Confirm Security Exception) у нас будут запрошены имя пользователя и пароль. После ввода соответствующих учетных данных нам будет предоставлен доступ к требуемой странице.



Это предупреждение появляется на экране постольку, поскольку для того, чтобы сервером использовался протокол HTTPS, у него должен быть сертификат SSL. Как правило, сертификаты SSL выпускаются центрами сертификации, такими как Veri-sign или Thawte. Эти центры сертификации в цифровой форме подписывают сертификат; таким образом, они удостоверяют, что сервер принадлежит той сущности, принадлежность 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 предварительно сконфигурированным с самоподписанным сертификатом SSL. Но поскольку этот сертификат не был подписан центром сертификации, обозреватель показывает окно предупреждения, когда мы пытаемся получить доступ к защищенной странице через протокол HTTPS.

Обратите внимание на URL предыдущего снимка экрана: протокол устанавливается в HTTPS, а значение порта – в 8181, при том, что в адресной строке обозревателя мы указали URL `http://localhost:8080/filerealmauthhttps/random.jsp`. Из-за изменений, которые мы произвели в дескрипторе развертывания приложения `web.xml` запрос был автоматически переадресован данному URL. Конечно, пользователи могут напрямую ввести безопасный URL и он будет работать без проблем.

Любые данные, переданные по HTTPS, шифруются, включая имя пользователя и пароль, введенные во всплывающем окне, сгенерированном обозревателем. Протокол

HTTPS позволяет нам безопасно использовать стандартную аутентификацию, однако у нее есть еще один недостаток: пользователь сможет выйти из приложения единственным способом – закрыв обозреватель. Если мы должны обеспечить пользователям выход из приложения без закрытия обозревателя, нам следует использовать аутентификацию на основе формы.

При использовании аутентификации на основе формы нам потребуется произвести некоторые изменения в файле дескриптора 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>
    </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>FORM</auth-method>
        <realm-name>file</realm-name>
        <form-login-config>
            <form-login-page>/login.jsp</form-login-page>
            <form-error-page>/loginerror.jsp</form-error-page>
        </form-login-config>
    </login-config>
    <servlet>
        <servlet-name>LogoutServlet</servlet-name>
        <servlet-class>
            net.ensode.glassfishbook.LogoutServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>LogoutServlet</servlet-name>
        <url-pattern>/logout</url-pattern>
    </servlet-mapping>
</web-app>
```

При использовании аутентификации на основе формы мы просто используем FORM в качестве значения элемента `<auth-method>` в файле web.xml. При использовании этого метода аутентификации необходимо предоставить страницу входа в систему, а также страницу ошибки входа в систему. Мы указываем URL страницы входа в систему и страницы ошибки входа в систему как значения элементов

<form-login-page> и <form-error-page> соответственно. Как видно из примера, эти элементы должны быть вложены в элемент <form-login-config>.

Разметка страницы входа в систему для нашего приложения показана ниже:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Login</title>
    </head>
    <body>
        <p>Введите имя пользователя и пароль для доступа к приложению</p>
        <form method="POST" action="j_security_check">
            <table cellpadding="0" cellspacing="0" border="0">
                <tr>
                    <td align="right">Имя пользователя: </td>
                    <td>
                        <input type="text" name="j_username">
                    </td>
                </tr>
                <tr>
                    <td align="right">Пароль: </td>
                    <td>
                        <input type="password" name="j_password">
                    </td>
                </tr>
                <tr>
                    <td></td>
                    <td><input type="submit" value="Войти"></td>
                </tr>
            </table>
        </form>
    </body>
</html>
```

Страница приложения для входа в систему (регистрации), использующая аутентификацию на основе формы, должна содержать форму, методом которой является "POST", а ее элемент `action` имеет значение "`j_security_check`". Нам не нужно реализовывать сервлет или что-либо еще для обработки этой формы. Код для ее обработки предоставляется сервером приложений.

Форма на странице входа в систему должна содержать текстовое поле, называемое `j_username`; оно предназначено для хранения введенного пользователем имени пользователя. Кроме того, форма должна содержать поле `j_password`, предназначенное для пароля пользователя, и, безусловно, кнопку отправки формы – для передачи данных серверу.

Единственное требование, предъявляемое к странице входа в систему, – наличие формы, атрибуты которой должны соответствовать таковым из предыдущего примера: полям ввода `j_username` и `j_password`, описанным в предыдущем абзаце.

К странице ошибки никаких специальных требований не предъявляется. Конечно, она должна показывать сообщение об ошибке, сообщая пользователю о том, что

вход в систему оказался неудачным. Тем не менее она может содержать все, что мы хотим. Страница ошибки для нашего приложения просто сообщает пользователю, что произошла ошибка входа в систему, и предоставляет ссылку на страницу входа, чтобы дать пользователю возможность попытаться войти еще раз.

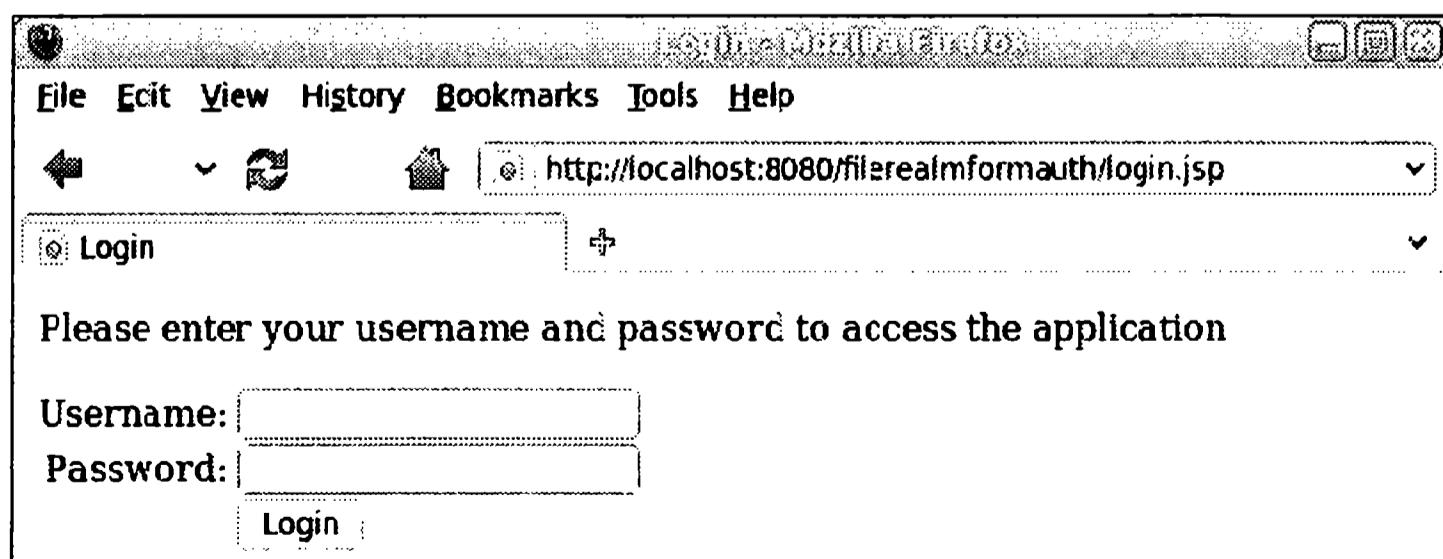
В дополнение к странице входа в систему и странице ошибки входа в систему мы добавили в наше приложение сервлет. Он позволяет нам реализовывать функциональность выхода из системы, что было невозможно в случае, когда мы использовали стандартную аутентификацию.

```
package net.ensode.glassfishbook;

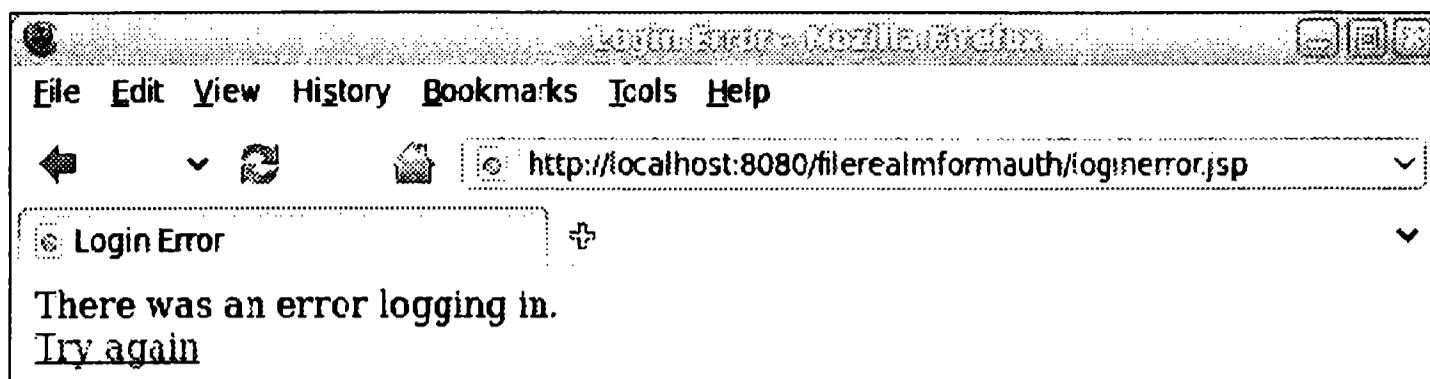
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet(urlPatterns = {"/*logout"})
public class LogoutServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException
    {
        request.getSession().invalidate();
        response.sendRedirect("index.jsp");
    }
}
```

Как видно из приведенного кода, для выхода из системы нам нужно всего лишь сделать недействительным пользовательский сеанс. В нашем сервлете мы перенаправляем отклик на страницу `index.jsp`, поскольку сеанс для этой точки недопустим, механизм безопасности «вытолкнет» его и автоматически переадресует пользователя на страницу входа в систему.

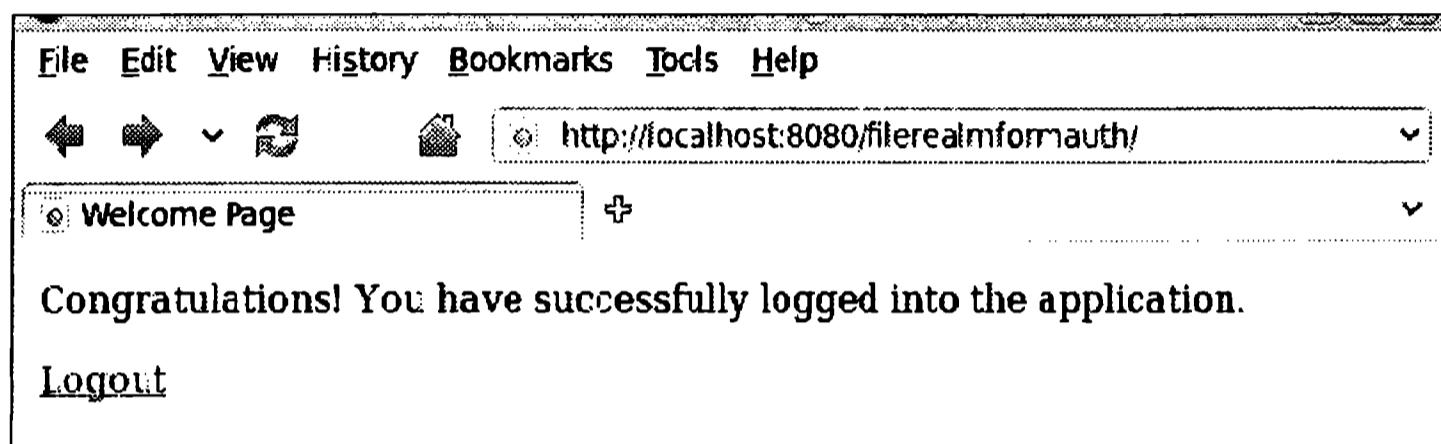
Теперь мы готовы к испытанию аутентификации на основе формы. После создания нашего приложения, его развертывания и указания в адресной строке обозревателя адреса любой из страниц мы должны будем увидеть нашу страницу входа в систему, представленную в обозревателе:



Если мы введем недопустимые учетные данные, то автоматически будем переадресованы на страницу ошибки входа в систему:



Мы можем щелкнуть по ссылке **Попытаться снова** (Try again), чтобы попробовать еще раз. После ввода допустимых учетных данных нам будет разрешен доступ к приложению:



Как видно на снимке экрана, мы добавили к странице ссылку **Выход из системы** (Logout). Эта страница переадресует пользователя к сервлету выхода из системы, который, как мы упоминали ранее, просто делает недействительным сеанс. С точки зрения пользователя эта ссылка для него будет просто выполнять выход из системы и направлять его на страницу входа в систему.

Область сертификата

Область сертификата использует клиентские сертификаты для аутентификации. Точно так же как серверные сертификаты, клиентские сертификаты обычно получаются из центра сертификации, например Verisign или Thawte. Эти центры сертификации проверяют, что сертификат действительно принадлежит той сущности, которая декларирует, что является таковой.

Получение сертификата от центра сертификации стоит денег и занимает время. Непрактично будет получать сертификат одного из центров сертификации, когда мы разрабатываем и/или тестируем наше приложение. К счастью, у нас есть возможность создать самоподписанные сертификаты для тестирования.

Создание самоподписанных сертификатов

Мы можем создать самоподписанные сертификаты, приложив минимум усилий, с помощью утилиты keytool, включенной в *Комплект разработчика для Java (JDK)*.



Мы только вкратце рассмотрим часть функциональности утилиты keytool. В частности, мы рассмотрим то, что необходимо для создания и импорта самоподписанных сертификатов в GlassFish и в обозреватель. Чтобы узнать больше об утилите keytool, обратитесь к следующему ресурсу: <http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html>.

Генерирование самоподписанного сертификата может быть выполнено путем ввода в командной строке следующей команды (она записывается в одну строку):

```
keytool -genkey -v -alias selfsignedkey -keyalg RSA -storetype PKCS12  
-keystore client_keystore.p12 -storepass wonttellyou -keypass  
wonttellyou
```

Эта команда предполагает, что утилита keytool находится в системном пути. Данный инструмент может быть найден в каталоге bin соответствующего каталога, в котором установлен Комплект разработчика для Java (Java Development Kit (JDK)).

Мы заменим значения аргументов -storepass и -keypass своим собственным паролем. Оба этих пароля должны быть одинаковыми, чтобы можно было успешно использовать сертификат для аутентификации клиента. Мы вправе выбрать любое значение аргумента -alias. Можно также выбрать любое значение аргумента -keystore, однако оно должно заканчиваться на .p12, поскольку эта команда генерирует файл, который должен быть импортирован в веб-обозреватель; данный файл не будет распознан, если у него не будет расширения p12.

После ввода предыдущей команды в командной строке keytool запросит некоторую информацию:

Ваше имя и фамилия?

[Неизвестно] : Дэвид Хеффельфингер

Наименование Вашего подразделения?

[Неизвестно] : Отдел написания книг

Название Вашей организации?

[Неизвестно] : Ensoode Technology, LLC

Ваш город или район?

[Неизвестно] : Fairfax

Ваш штат или провинция?

[Неизвестно] : Вирджиния

Двухбуквенный код страны, где находится этот штат или провинция?

[Неизвестно] : US

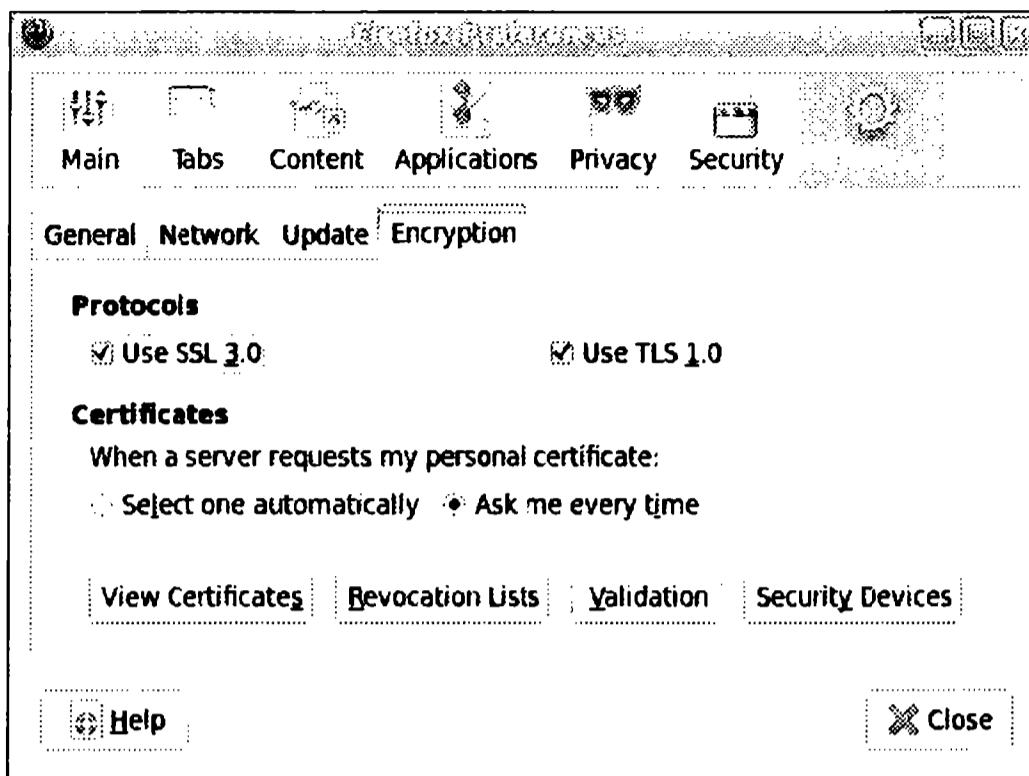
CN=Дэвид Хеффельфингер, OU=Отдел написания книг, O=Ensoode Technology, LLC, L=Fairfax, ST=Вирджиния, C=US корректно?

[нет] : да

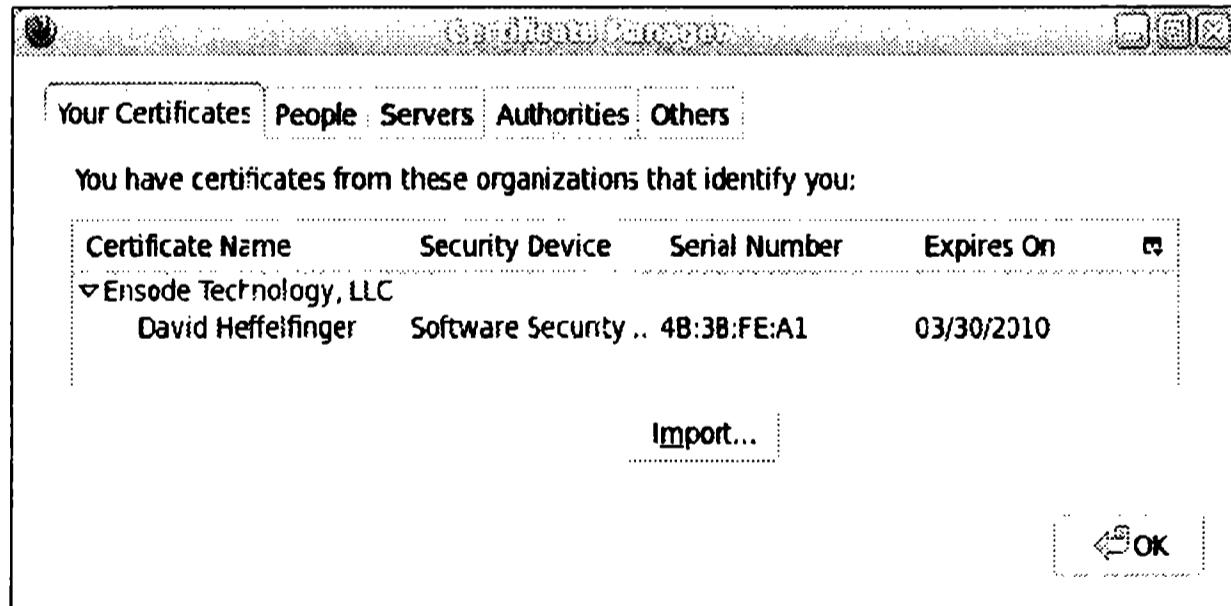
После ввода ответов на каждый вопрос keytool сгенерирует сертификат; он будет сохранен в текущем каталоге, и имя файла будет значением, которое мы использовали для аргумента -keystore (в нашем примере – client_keystore.p12). Чтобы можно было использовать этот сертификат для нашей аутентификации, мы должны импортировать его в обозреватель. Данная процедура схожа для всех обозревателей, но все-таки имеются некоторые отличия. В Firefox¹ следует перейти в меню Инструменты (Tools) | Настройки (Settings), затем щелкнуть по значку Дополнительные

¹ Приведены сведения для версии Firefox v.17. – Прим. перев.

(Advanced) в верхней части всплывающего окна и выбрать вкладку **Шифрование** (Encryption):



Теперь нужно щелкнуть по кнопке **Просмотр сертификатов** (View Certificates), далее, в открывшемся окне, – по кнопке **Импортировать** (Import), а затем переместиться и выбрать наш сертификат из каталога, в котором мы его создали. В этом месте Firefox запросит пароль, используемый для шифрования сертификата. В нашем примере мы использовали «wonttellyou» в качестве пароля. После ввода пароля мы должны увидеть во всплывающем окне подтверждение того, что наш сертификат успешно импортирован. Затем он появится в списке сертификатов:



Итак, мы добавили наш сертификат в Firefox для того, чтобы его можно было использовать в целях нашей аутентификации. При использовании другого обозревателя процедура будет похожей (за подробностями обратитесь к документации своего обозревателя).

Сертификат, который мы создали на предыдущем шаге, должен быть экспортирован в формат, понятный для GlassFish¹:

¹ Вся команда записывается в одной строке. – Прим. перев.

```
keytool -export -alias selfsignedkey -keystore client_keystore.p12  
-storetype PKCS12 -storepass wonttellyou -rfc -file selfsigned.cer
```

Значения аргументов `-alias`, `-keystore` и `-storepass` должны соответствовать значениям, используемым в предыдущей команде. Можно выбрать любое значение для аргумента `-file`, но рекомендуется завершить его расширением `.cer`.

Поскольку наш сертификат не был выпущен центром сертификации, GlassFish по умолчанию не будет признавать его допустимым. GlassFish знает, каким сертификатам, созданным центрами сертификации, можно доверять. Это реализуется за счет того, что сертификаты различных органов сертификации сохраняются в хранилище ключей под названием `cacerts.jks`. Это хранилище ключей можно найти в следующем месте:

[Каталог установки GlassFish]/GlassFish/domains/domain1/config/
`cacerts.jks`.

Соответственно, чтобы GlassFish принимал наш сертификат, мы должны импортировать его в хранилище ключей `cacerts`. Это может быть выполнено вводом следующей команды²:

```
keytool -import -file selfsigned.cer -keystore [Каталог установки  
GlassFish]/glassfish/domains/domain1/config/cacerts.jks -keypass  
changeit -storepass changeit
```

После этого `keytool` выведет на экран информацию о сертификате в командной строке и спросит нас, хотим ли мы доверять ему.

Владелец: CN=Дэвид Хеффельфингер, OU=Отдел написания книг, O=Ensoode
Technology, LLC, L=Fairfax, ST=Вирджиния, C=US

Издатель: CN=Дэвид Хеффельфингер, OU=Отдел написания книг, O=Ensoode
Technology, LLC, L=Fairfax, ST=Вирджиния, C=US

Порядковый номер: 4b3bfeal

Действителен до: Wed Dec 30 20:30:09 EST 2009 **until:** Tue Mar 30
21:30:09 EDT 2010

Цифровой отпечаток сертификата:

MD5: CD:77:45:77:5F:30:F1:A2:AE:3F:E3:6F:B5:7F:D1:A2

SHA1: 8C:2B:53:A7:92:5F:21:17:6F:DD:B2:F0:84:66:DC:83:8F:B7:10:47

Наименование алгоритма подписи: SHA1withRSA

Версия: 3

Доверять этому сертификату? [нет]: да

Сертификат был добавлен в хранилище ключей.

После того как мы добавим сертификат в хранилище ключей `cacerts.jks`, мы должны перезапустить домен, чтобы изменения вступили в силу.

Фактически здесь мы добавляем себя в качестве центра сертификации, которому GlassFish будет доверять. Конечно, этого не нужно делать в производственной сис-

² Вся команда записывается в одной строке. – Прим. перев.

теме.

Значение аргумента `-file` должно соответствовать значению, которое мы использовали для этого же аргумента, когда экспорттировали сертификат.



changeit является паролем по умолчанию для аргументов `-keypass` и `-storepass`, для аргумента хранилища ключей `sacerts.jks`. Это значение может быть изменено путем выполнения следующей команды: [Каталог установки GlassFish]/GlassFish/bin/asadmin change-master-password --savemasterpassword=true. Эта команда запросит существующий основной пароль и новый основной пароль. Аргумент `--savemasterpassword=true` не обязательен; он сохраняет основной пароль в файле с названием `master-password` в корневом для домена каталоге. Если мы не будем использовать этот аргумент, изменения основной пароль, то мы должны будем вводить основной пароль каждый раз, когда хотим запустить домен.

Теперь, когда мы создали самоподписанный сертификат, импортировали его в наш обозреватель и установили себя в качестве центра сертификации, которому будет доверять GlassFish, мы готовы разработать приложение, которое будет использовать клиентские сертификаты для аутентификации.

Конфигурирование приложений для использования области сертификата

Поскольку мы используем возможности средств защиты Java EE, нам вообще не нужно модифицировать код для использования областей безопасности. Все, что мы должны сделать, – это модифицировать конфигурацию приложения в ее дескрипторах развертывания - `web.xml` и `sun-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>
```

Основным различием между этим дескриптором развертывания `web.xml` и тем, который мы видели в предыдущем разделе, является содержимое элемента `<login-config>`. В этом случае мы объявили, что методом авторизации является `CLIENT-CERT`, а используемой для аутентификации областью – `certificate`. В результате

GlassFish будет запрашивать у обозревателя клиентский сертификат прежде, чем позволить пользователю получить доступ к приложению.

При использовании аутентификации на основе клиентского сертификата запрос должен всегда выполняться через HTTPS. Поэтому целесообразно добавить элемент `<transport-guarantee>` со значением CONFIDENTIAL к дескриптору развертывания `web.xml`. Напомним из предыдущего раздела, что таким образом достигается эффект передачи любых запросов через порт HTTP к порту HTTPS. Если мы не добавим этого значения в дескриптор развертывания `web.xml`, то любые запросы через порт HTTP перестанут работать, поскольку аутентификация на основе клиентского сертификата не может быть выполнена по протоколу HTTP.

Обратите внимание на следующее: мы объявили, что только пользователи в роли `users` смогут получить доступ к любой странице в системе, добавив роль `users` в элементе `<role-name>`, вложенном в элемент `<auth-constraint>` элемента `<security-constraint>` в дескрипторе развертывания `web.xml`. Чтобы предоставить доступ авторизованным пользователям, мы должны добавить их в данную роль. Это делается в дескрипторе развертывания `sun-web.xml`:

```
<? xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD
    Application Server 9.0 Servlet 2.5//EN"
    "http://www.sun.com/software/appserver/dtds/
    sun-web-app_2_5-0.dtd">
<sun-web-app error-url="">
    <context-root>/certificaterealm</context-root>
    <security-role-mapping>
        <role-name>users</role-name>
        <principal-name>CN=Дэвид Хеффельфингер, OU=Отдел написания книг,
            O=Ensoode Technology, LLC, L=Fairfax, ST=Вирджиния, C=US
        </principal-name>
    </security-role-mapping>
</sun-web-app>
```

Такое присвоение выполняется путем отображения принципала (участника) на роль в элементе `<security-role-mapping>` дескриптора развертывания `sun-web.xml`. Его подэлемент `<principal-name>` должен содержать имя роли, а подэлемент `<principal-name>` должен содержать имя пользователя, которое берется из сертификата.

Если Вы сомневаетесь в том, какое имя нужно использовать, это имя может быть получено из сертификата с помощью утилиты `keytool`:

Владелец: CN=Дэвид Хеффельфингер, OU=Отдел написания книг, O=Ensoode Technology, LLC, L=Fairfax, ST=Вирджиния, C=US
Издатель: CN=Дэвид Хеффельфингер, OU=Отдел написания книг, O=Ensoode Technology, LLC, L=Fairfax, ST=Вирджиния, C=US

Порядковый номер: 4b3bfeal

Действителен до: Wed Dec 30 20:30:09 EST 2009 until: Tue Mar 30
21:30:09 EDT 2010

Цифровой отпечаток сертификата:

MD5: CD:77:45:77:5F:30:F1:A2:AE:3F:E3:6F:B5:7F:D1:A2

SHA1: 8C:2B:53:A7:92:5F:21:17:6F:DD:B2:F0:84:66:DC:83:8:B7:10:47

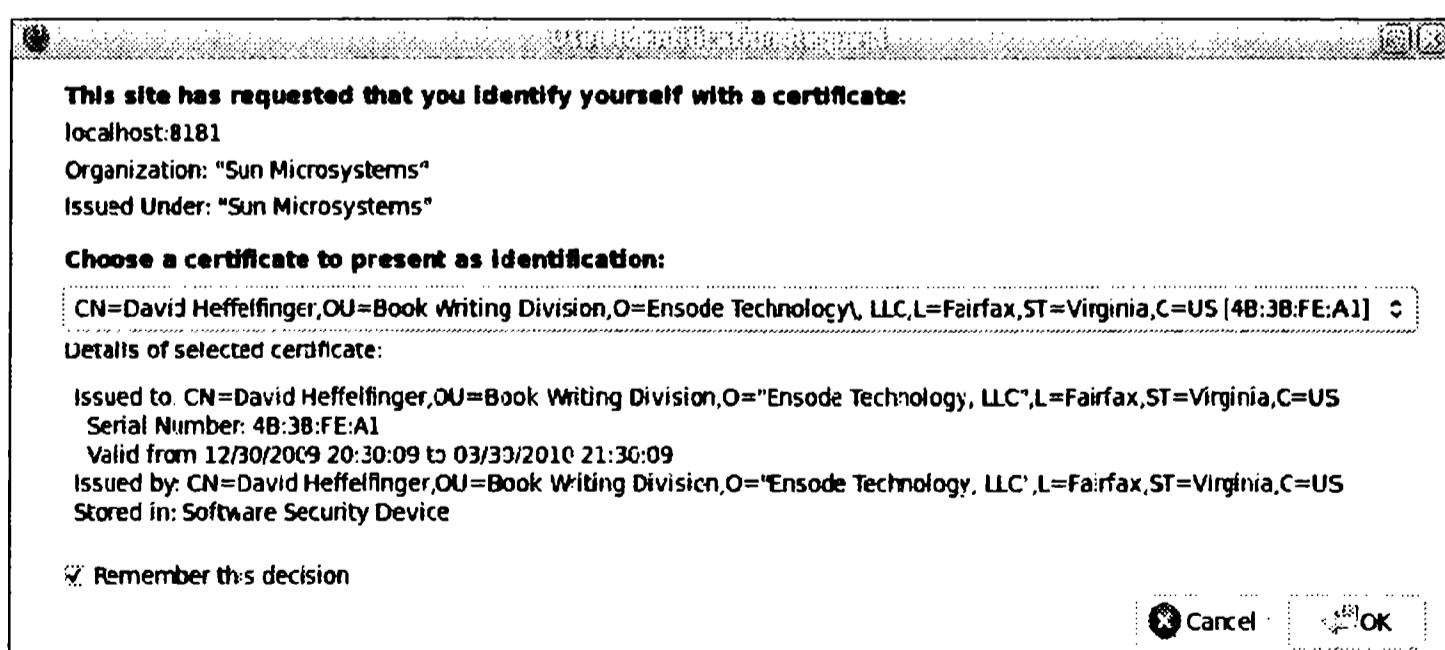
Наименование алгоритма подписи: SHA1withRSA

Версия: 3

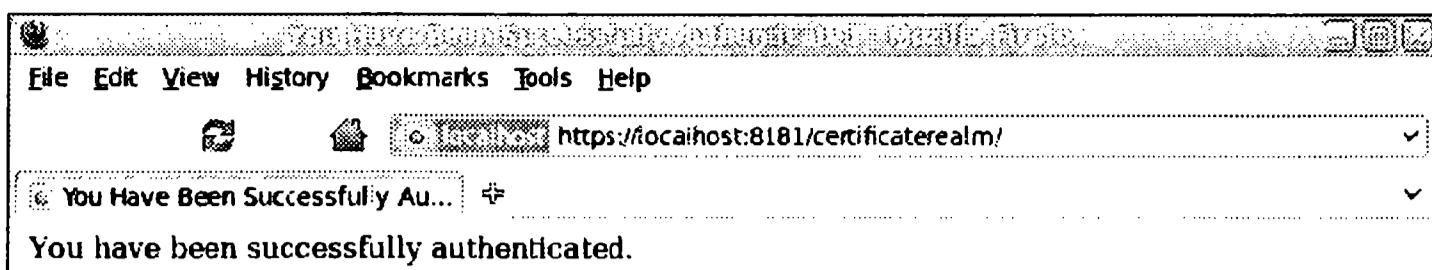
Значением для использования в качестве <principal-name> является строка, следующая за строкой «Владелец» (в данном случае это строка «Издатель»). Пожалуйста, обратите внимание, что значение <principal-name> должно быть в той же самой строке, что и его начальный и заключительный элементы (<principal-name> и </principal-name>). Если имеются символы новой строки или возврата каретки перед значением или после него, они будут интерпретироваться как являющиеся частью значения и проверка перестает работать.

Поскольку у нашего приложения имеются один пользователь и одна роль, мы готовы развернуть его. Если бы у нас было больше пользователей, то мы должны были бы добавить дополнительный элемент <security-role-mapping> к нашему дескриптору развертывания sun-web.xml; по крайней мере по одному элементу на пользователя. Если бы у нас были пользователи, которые принадлежат нескольким ролям, нам следовало бы добавить элемент <security-role-mapping> для каждой роли, которой принадлежит пользователь, используя значение <principal-name>, соответствующее сертификату пользователя для каждой роли.

Теперь мы готовы протестировать наше приложение. После того как мы развернем его и укажем в адресной строке обозревателя любую страницу приложения, мы должны увидеть экран наподобие следующего (предполагается, что обозреватель не был сконфигурирован на предоставление сертификата по умолчанию при каждом его запросе сервером):



После щелчка по кнопке **OK** нам будет разрешено получить доступ к приложению:



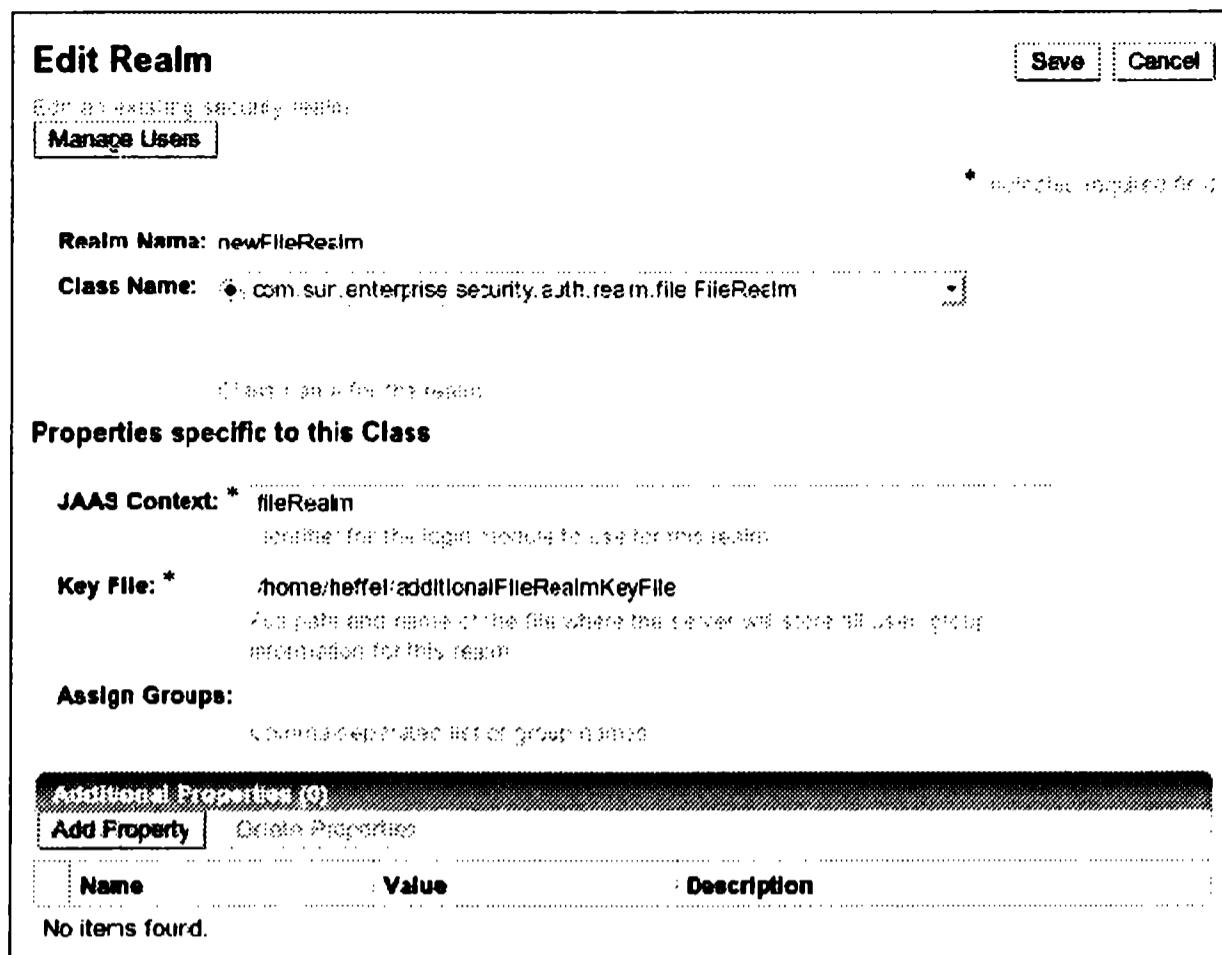
Прежде чем предоставить доступ к приложению, GlassFish проверяет центр сертификации, который выпустил сертификат и проверяет эти данные по списку доверенных центров сертификации. В предоставленном нами самоподписанном сертификате владелец сертификата и центр сертификации – это одно и то же, поскольку мы добавили себя как доверяемый орган, импортируя свой самоподписанный сертификат в хранилище ключей `cacerts.jks`. GlassFish признает центр сертификации допустимым; затем он получает основное имя из сертификата и сравнивает его с записями в дескрипторе развертывания приложения `sun-web.xml`. Поскольку мы добавили себя в этот дескриптор развертывания и назначили себе допустимую роль, нам будет разрешен доступ к приложению.

Определение дополнительных областей

В дополнение к трем предварительно сконфигурированным областям безопасности, которые мы обсуждали в предыдущем разделе, мы можем создать дополнительные области аутентификации для приложения. Мы можем создать области, которые ведут себя точно так же, как область администратора или область файла, а кроме того, области, которые ведут себя как область сертификата. Наконец, можно создать области, которые используют другие методы аутентификации. Мы можем аутентифицировать пользователей через базу данных LDAP, а также через реляционную базу данных. Когда GlassFish устанавливается на сервере Solaris, мы можем использовать аутентификацию Solaris в рамках GlassFish. Если же ни один из этих механизмов аутентификации не соответствует нашим потребностям, мы вправе реализовать наш собственный.

Определение дополнительных областей файла

В консоли администрирования необходимо развернуть узел **Конфигурация** (Configuration), а затем узел **Безопасность** (Security), после чего щелкнуть по узлу **Области** (Realms) и по кнопке **Новая...** (New...) на открывшейся странице в основной панели веб-консоли. Здесь мы должны будем увидеть экран наподобие следующего:



Для создания дополнительной области нам всего лишь нужно ввести ее уникальное имя в поле **Имя области** (Realm Name), выбрать **com.sun.enterprise.security.auth.realm.file.FileRealm** для поля **Имя класса** (Class Name) и ввести значение для полей **Контекст JAAS** (JAAS context) и **Файл ключей** (Key File). Значение поля **Файл ключей** (Key File) должно быть абсолютным путем к файлу, где будет храниться пользовательская информация. Для областей файла значением поля **Контекст JAAS** всегда должно быть **fileRealm**.

После ввода этой информации мы можем щелкнуть по кнопке **OK**, и наша новая область будет создана. Затем мы можем использовать ее точно так же, как предопределенную область файла. Приложения, желающие аутентифицироваться через эту новую область, должны использовать ее имя в качестве значения элемента `<realm-name>` в дескрипторе развертывания приложения `web.xml`.



На момент написания этой книги в GlassFish v.3 имеется проблема, которая препятствует успешному созданию пользовательской области файла. Обходное решение этой проблемы заключается в предварительном создании файла ключей (как пустого файла) до создания области.

Либо пользовательская область файла может быть добавлена из командной строки через утилиту `asadmin`¹:

```
asadmin create-auth-realm --classname com.sun.enterprise.
security.auth.realm.file.FileRealm --property file=/home/heffel/
additionalFileRealmKeyFile:jaas-context=fileRealm newFileRealm
```

Команда `create-auth-realm` сообщает `asadmin`, что мы хотим создать новую область безопасности. Значение параметра `--classname` соответствует имени класса области безопасности; обратите внимание, что оно соответствует значению, которое

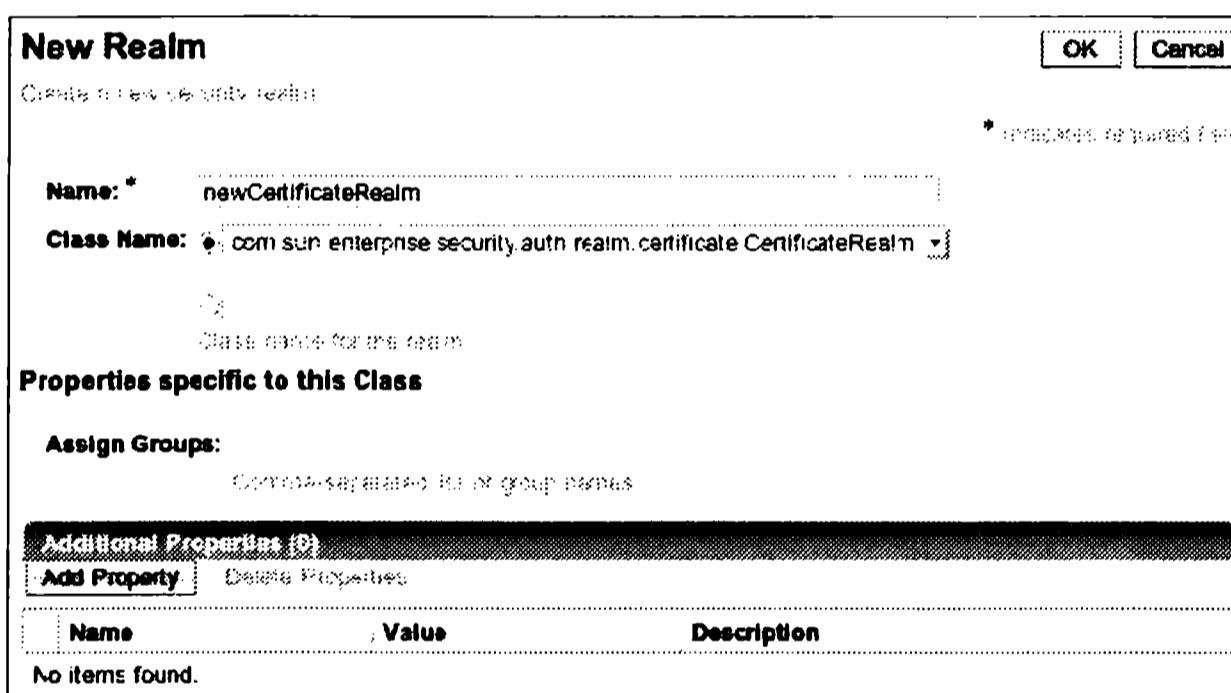
¹ Вся команда записывается в одной строке. – Прим. перев.

мы выбрали ранее в веб-консоли. Параметр `--property` позволяет нам передавать свойства и их значения. Значение данного параметра должно представлять собой разделенный двоеточием список свойств и их значения. Последним параметром команды `create-auth-realm` является имя, которое мы хотим дать нашей области безопасности.

Хотя проще создать область безопасности через веб-консоль, выполнение этой процедуры через утилиту командной строки `asadmin` имеет то преимущество, что ее удобно использовать в сценариях, позволяющих нам сохранить данную команду в сценарии и легко сконфигурировать несколько экземпляров GlassFish.

Определение дополнительных областей сертификата

Чтобы определить дополнительную область сертификата, мы просто должны ввести ее имя в поле **Имя (Name)** и выбрать `com.sun.enterprise.security.auth.realm.certificate.CertificateRealm` в качестве значения поля **Имя класса (Class Name)**, а затем щелкнуть по кнопке **OK**. В результате будет создана наша новая область.



Приложения, желающие использовать эту новую область для аутентификации, должны использовать ее имя в качестве значения элемента `<realm-name>` в дескрипторе развертывания `web.xml` и указать `CLIENT-CERT` в качестве значения элемента `<auth-method>`. Конечно, клиентские сертификаты должны присутствовать и быть сконфигурированными, как было показано в разделе «Конфигурирование приложений для использования области сертификата» (см. стр. 280).

 На момент написания этой книги у GlassFish имеется проблема, которая препятствует созданию пользовательской области безопасности через веб-консоль администрирования. В качестве обходного решения этой проблемы пользовательские области на основе сертификата могут создаваться через утилиту командной строки `asadmin`.

Либо пользовательская область сертификата может быть создана с помощью утилиты командной строки `asadmin`²:

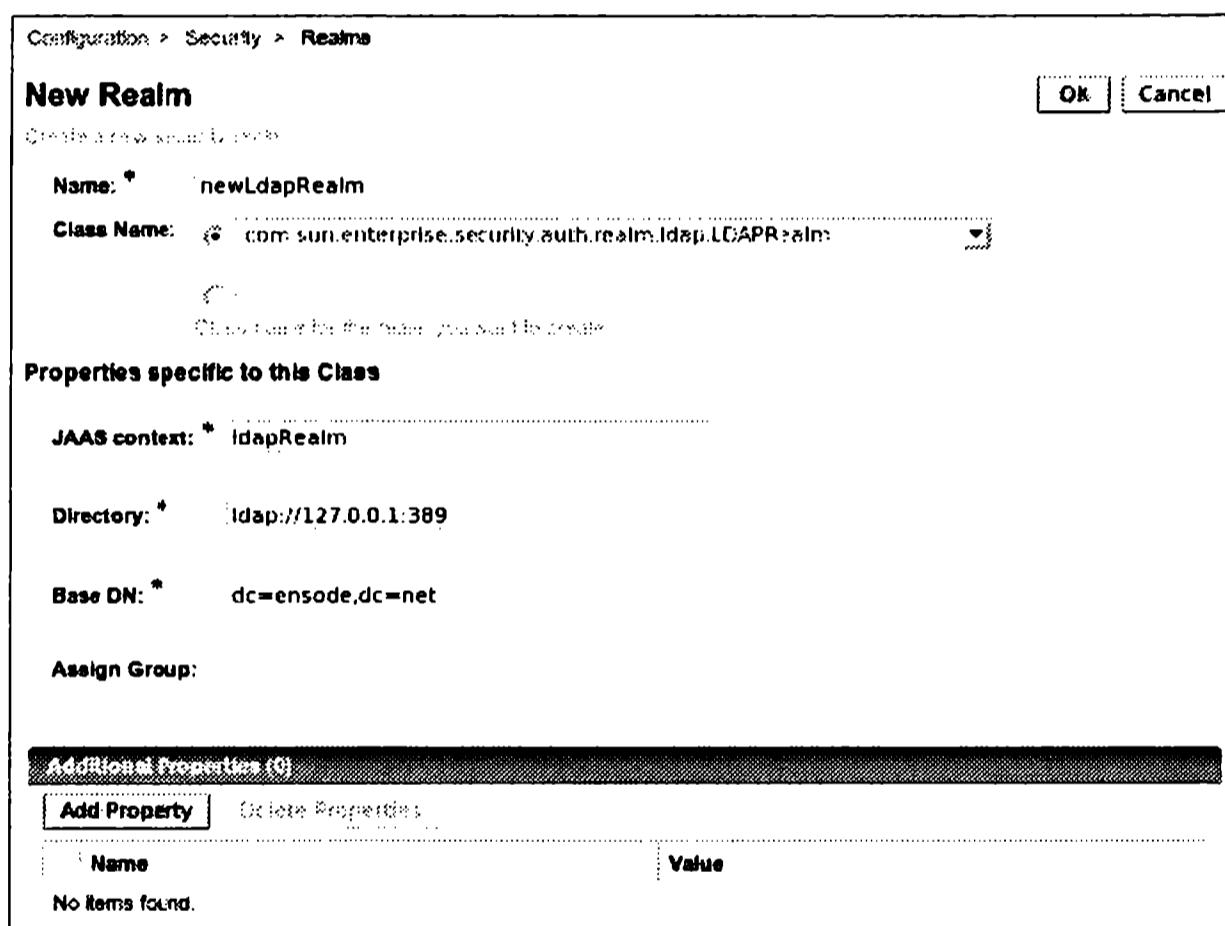
² Вся команда записывается в одной строке. – Прим. перев.

```
asadmin create-auth-realm --classname com.sun.enterprise.security.
auth.realm.certificate.CertificateRealm newCertificateRealm
```

В этом случае мы не должны передавать свойства, как при создании пользовательской области файла. Нам нужно всего лишь передать соответствующее значение параметру `--classname` и указать новое имя области безопасности.

Определение области LDAP

Мы можем легко создать область для аутентификации через базу данных *Облегченного протокола доступа к каталогам* (Lightweight Directory Access Protocol (LDAP)). Чтобы это сделать, нам нужно, в дополнение к очевидному шагу ввода имени для области, выбрать `com.sun.enterprise.security.auth.realm.ldap.LDAPRealm` в качестве значения **Имени класса** (Class Name) для новой области:



Затем мы должны ввести URL для сервера каталогов в поле **Каталог** (Directory) и основное отличительное имя, которое будет использоваться для поиска информации пользователя в качестве значения поля **Основное DN** (Base DN).

После создания области LDAP приложения могут использовать ее для аутентификации через базу данных LDAP. Имя области должно использоваться в качестве значения элемента `<realm-name>` в дескрипторе развертывания приложения `web.xml`. Значение элемента `<auth-method>` должно быть BASIC или FORM. Пользователи и роли в базе данных LDAP могут быть отображены на группы в дескрипторе развертывания приложения `sun-web.xml` путем использования элементов `<principal-name>`, `<role-name>` и `<group-name>`, как было показано выше в этой главе.

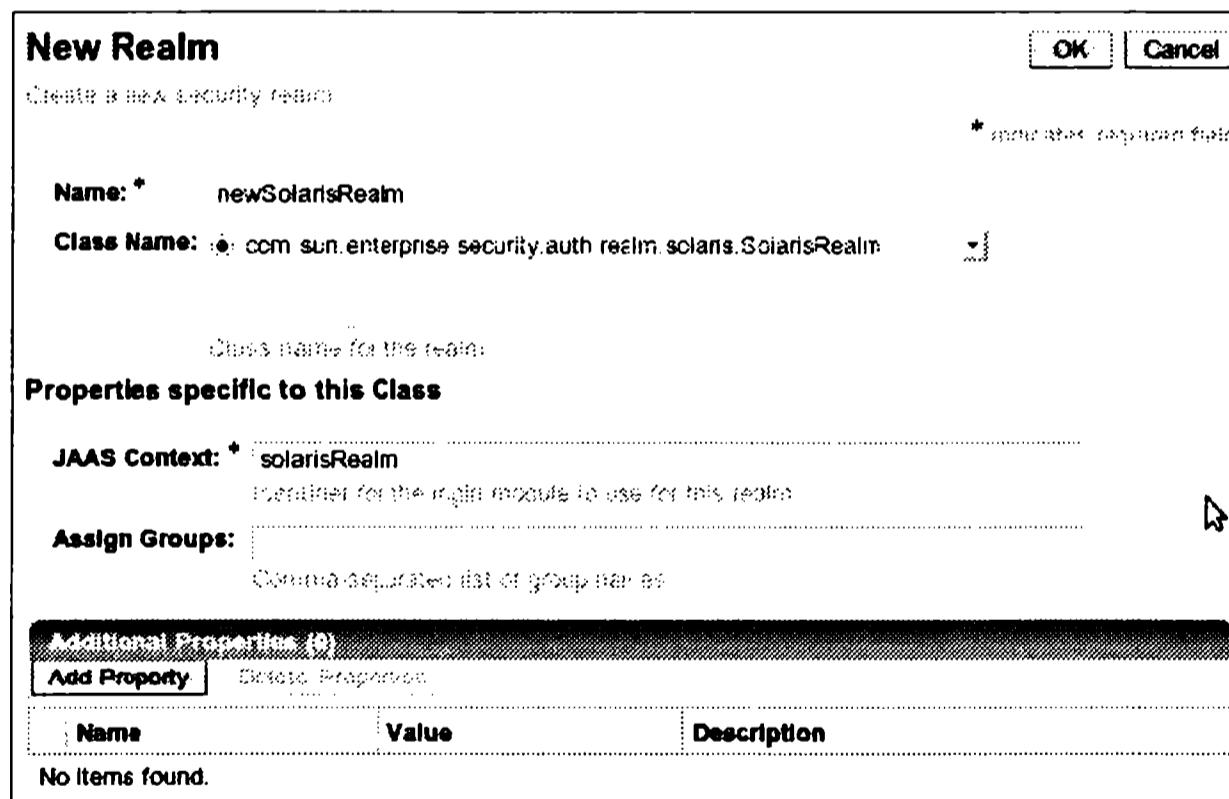
Чтобы создать область LDAP из командной строки, мы должны использовать следующий синтаксис¹:

```
asadmin create-auth-realm --classname com.sun.enterprise.security.auth.realm.ldap.LDAPRealm --property "jaascontext=ldapRealm:directory=ldap\://127.0.0.1\:1389:basedn=dc\=enode,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, например такие, как двоеточия и знаки равенства. Во избежание подобной ошибки мы просто снабжаем их префиксом наклонной черты влево (\).

Определение области Solaris

Когда GlassFish устанавливается на сервере Solaris, он может использовать дополнительный механизм аутентификации операционной системы через область Solaris. Для этого типа области нет никаких специфических свойств. Все, что мы должны сделать для ее создания, – указать ее имя и выбрать **com.sun.enterprise.security.auth.realm.SolarisRealm** в качестве значения поля **Имя класса** (Class Name):



Для поля **Контекст JAAS** (JAAS Context) должно быть установлено значение **SolarisRealm**. После добавления области приложения смогут аутентифицироваться через нее с использованием стандартной аутентификации либо аутентификации на основе формы. Группы и пользователи операционной системы могут быть отображены на роли приложения, определенные в дескрипторе развертывания приложения web.xml с помощью элементов <principal-name>, <role-name> и <group-name> в дескрипторе развертывания sun-web.xml.

¹ Вся команда записывается в одной строке. – Прим. перев.

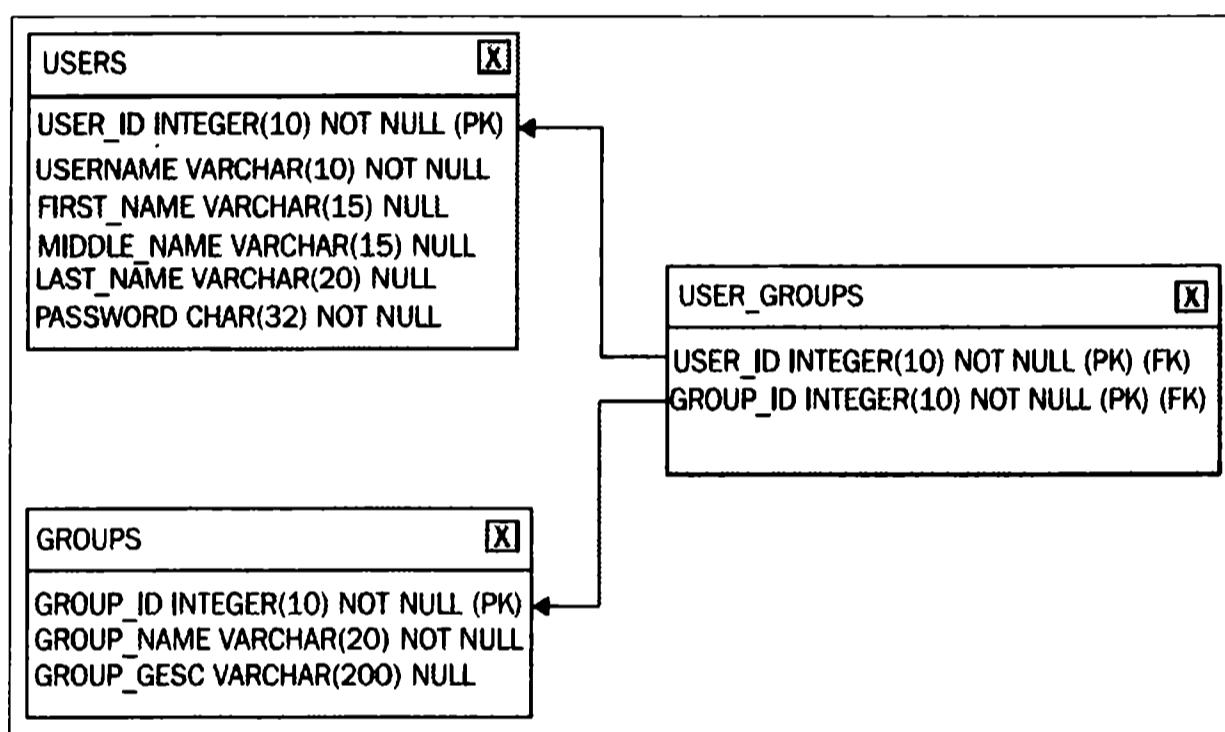
Область Solaris может быть создана из командной строки следующим образом¹:

```
asadmin create-auth-realm --classname com.sun.enterprise.security.
auth.realm.solaris.SolarisRealm --property jaas-context=solarisRealm
newSolarisRealm
```

Определение области JDBC

Другой тип области, которую мы можем создать, – область JDBC. Этот тип области использует пользовательскую информацию, хранящуюся в таблицах базы данных для аутентификации пользователей.

Чтобы пояснить, как мы можем аутентифицироваться через области JDBC, мы должны создать базу данных, содержащую пользовательскую информацию:



Наша база данных содержит три таблицы. Таблица **USERS** содержит информацию о пользователях, а таблица **GROUPS** – информацию о группах. Для реализации отношения «многие ко многим» между таблицами **USERS** и **GROUPS** мы должны добавить объединяющую таблицу, чтобы сохранить нормализацию данных. Имя этой таблицы – **USER_GROUPS**.

Обратите внимание, что столбец **PASSWORD** таблицы **USERS** имеет тип **CHAR(32)**. Причина, по которой мы выбрали этот тип вместо **VARCHAR**, заключается в том, что область JDBC ожидает, что по умолчанию пароли будут зашифрованы в виде хэша MD5, а эти хэши всегда имеют длину 32 символа.

Пароли легко могут быть зашифрованы в формат, ожидаемый по умолчанию при использовании класса **java.security.MessageDigest**, включенного в JDK. Следующий пример кода принимает пароль в виде открытого текста и создает из него зашифрованный хэш MD5:

```
package net.ensode.glassfishbook;
import java.security.MessageDigest;
```

¹ Вся команда записывается в одной строке. – Прим. перев.

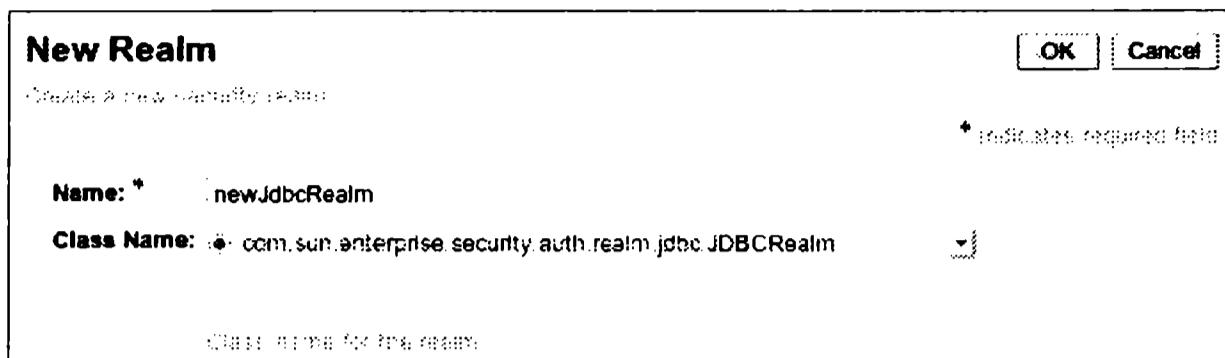
```
import java.security.NoSuchAlgorithmException;
public class EncryptPassword
{
    public static String encryptPassword(String password) throws
        NoSuchAlgorithmException
    {
        MessageDigest messageDigest = MessageDigest.getInstance("MD5");
        byte[] bs;
        messageDigest.reset();
        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("Используйте: java " +
                    "net.ensode.glassfishbook.EncryptPassword" + " cleartext");
            }
            else
            {
                encryptedPassword = encryptPassword(args[0]);
                System.out.println(encryptedPassword);
            }
        }
        catch (NoSuchAlgorithmException e)
        {
            e.printStackTrace();
        }
    }
}
```

Изюминкой этого класса является его метод `encryptPassword()`. Он в основном занимается тем, что принимает строку открытого текста и делает из нее дайджест (отпечаток), используя алгоритм MD5 с помощью метода `digest()` экземпляра `java.security.MessageDigest`. Затем он кодирует дайджест серией шестнадцатиричных чисел. 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 к области через консоль GlassFish. Вместо этого они добавляются путем вставки данных в соответствующие таблицы.

Как только мы получим базу данных, которая будет содержать в себе учетные данные пользователей, мы будем готовы создать новую область JDBC.

Мы можем создать область JDBC, указав имя в поле **Имя (Name)** формы **Новая область (New Realm)** в веб-консоли GlassFish, а затем выбрав **com.sun.enterprise.security.auth.realm.jdbc.JDBCRealm** в качестве значения поля **Имя класса (Class Name)**:



Имеется много других свойств, которые мы должны установить для нашей новой области JDBC:

Properties specific to this Class	
JAAS context: *	<code>jdbcrealm</code>
JNDI: *	<code>jdbc/_UserAuthPool</code>
User Table: *	<code>V_USER_ROLE</code>
User Name: *	<code>USERNAME</code>
Password: *	<code>PASSWORD</code>
Group Table: *	<code>V_USER_ROLE</code>
Group Name: *	<code>GROUP_NAME</code>
Assign Group:	
Database User:	
Database Password:	
Digest:	
Encoding:	
Charset:	

Поле **Контекст JAAS (JAAS context)** должно быть установлено в значение **jdbcrealm** для областей JDBC. Значение свойства **JNDI** должно быть JNDI именем источника данных, соответствующего базе данных, которая содержит данные области для пользователей и групп. Значение свойства **Таблица пользователей (User Table)** должно быть именем таблицы, которая содержит информацию об имени пользователя и пароле.



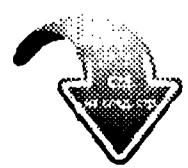
Обратите внимание, что на предыдущем снимке экрана мы использовали **V_USER_ROLE** в качестве значения для этого свойства. **V_USER_ROLE** является *представлением базы данных* (database view), которое содержит и информацию о пользователе, и информацию о группе. Мы не использовали таблицу **USERS** напрямую, поскольку GlassFish предполагает, что и таблица пользователей, и таблица групп содержат столбец с именем пользователя. Вследствие этого возникает дублирование данных. Во избежание подобной ситуации мы создали представление для того, чтобы можно было использовать в качестве значения свойства обоих: **Таблицы пользователей (User Table)** и **Таблицы групп (Group Table)**, которые мы вскоре обсудим.

Свойство **Имя пользователя (User Name)** должно содержать наименование столбца в **Таблице пользователей (User Table)**, которая содержит имена пользователей. Значение свойства **Пароль (Password)** должно быть именем столбца в **Таблице пользователей**, который содержит пароли пользователей. Значение свойства **Таблица групп (Group Table)** должно быть именем таблицы, содержащей группы пользователей. Свойство **Имя группы (Group Name)** должно содержать имя столбца в **Таблице групп (Group Table)**, содержащей имена групп пользователей.

Все другие свойства являются дополнительными и в большинстве случаев остаются незаполненными. Особенно интересно свойство **Дайджест (Digest)**. Оно позволяет нам указывать алгоритм используемого дайджеста сообщения для шифрования пароля пользователя. Допустимые значения этого свойства включают все алгоритмы, поддерживаемые JDK, а именно: MD2, MD5, SHA 1, SHA 256, SHA 384 и SHA 512. Если мы хотим сохранить пользовательские пароли в открытом тексте, можно указать для данного свойства значение **none** (никакой).

После определения нашей области JDBC мы должны сконфигурировать наше приложение с помощью его дескрипторов развертывания **web.xml** и **sun-web.xml**. Приложение, использующее область на основе JDBC для аутентификации и авторизации, конфигурируется точно так же, как и при использовании любого другого типа области.

```
<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>
    </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>FORM</auth-method>
<realm-name>newJdbcRealm</realm-name>
<form-login-config>
    <form-login-page>/login.jsp</form-login-page>
    <form-error-page>/loginerror.jsp</form-error-page>
</form-login-config>
</login-config>
</web-app>

```

В данном примере мы устанавливаем для элемента `<realm-name>` в дескрипторе развертывания `web.xml` значение `newJdbcRealm`. Это имя, которое мы решили дать нашей области, когда сконфигурировали ее через консоль GlassFish.

Здесь мы выбрали аутентификацию на основе формы, но с тем же успехом могли бы прибегнуть и к стандартной аутентификации.

В дополнение к заявлению о том, что мы будем использовать область на основе JDBC для аутентификации и авторизации, так же как и при использовании других типов областей, мы должны отобразить роли, определенные в дескрипторе развертывания `web.xml`, на названия групп безопасности. Это выполняется в дескрипторе развертывания `sun-web.xml`:

```

<? xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD
    Application Server 9.0 Servlet 2.5//EN"
    "http://www.sun.com/software/appserver/dtds/
    sun-web-app_2_5-0.dtd">
<sun-web-app>
    <security-role-mapping>
        <role-name>admin</role-name>
        <group-name>Admin</group-name>
    </security-role-mapping>
    <security-role-mapping>
        <role-name>user</role-name>
        <group-name>Users</group-name>
    </security-role-mapping>
</sun-web-app>

```

Значение элемента `<realm-name>` должно совпадать со значением соответствующего элемента `<role-name>` в файле `web.xml`. Значение `<group-name>` должно быть значением наименования столбца, указанном в свойстве **Столбец имени группы** (Group Name Column) области JDBC, которое было задано при конфигурировании в веб-консоли GlassFish.

Область JDBC может быть создана из командной строки следующим образом¹:

```

asadmin create-auth-realm --classname com.sun.enterprise.security.
auth.realms.jdbc.JDBCRealm --property jaas-context=jdbcRealm:
datasourcejndi=jdbc/_UserAuthPool:user-table=V_USER_ROLE:user-
namecolumn=USERNAME:password-column=PASSWORD:group-table=V_USER_
ROLE:groupname-column=GROUP_NAME fooJdbcRealm

```

¹ Вся команда записывается в одной строке. – Прим. перев.

Определение пользовательских областей

Хотя предопределенные типы областей могут использоваться в подавляющем большинстве случаев, нам предоставлена возможность создавать новые типы областей, если имеющиеся типы не удовлетворяют нашим потребностям. Выполнение этой задачи подразумевает кодирование пользовательских классов Realm и LoginModule.

Давайте сначала обсудим пользовательский класс области (Realm):

```
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("Users");
        vector.add("Admin");
        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;
    }
}
```

Наш пользовательский класс области должен расширять com.sun.enterprise.security.auth.realm.IASRealm. Этот класс можно найти в файле security.jar; следовательно, этот JAR-файл должен быть добавлен к CLASSPATH прежде, чем наша область сможет быть успешно скомпилирована.



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 репозитарии: <http://download.java.net/maven/glassfish>. Идентификатором группы является `org.glassfish.security`, а идентификатором артефакта – `security`. Во время написания этой книги самой последней версией была 3.0.

Наш класс должен переопределить метод, называемый `getGroupNames()`. Этот метод принимает единственный строковый параметр и возвращает `Enumeration`. Строковый параметр содержит имя пользователя, который пытается зарегистрироваться (войти) в область. `Enumeration` (перечисление) будет содержать набор строк, указывающих, к каким группам принадлежит пользователь. В нашем простом примере мы просто жестко закодировали группы. В реальном приложении эти группы были бы получены из некоторого постоянного хранилища (база данных, файл и т. д.).

Следующий метод, который должен быть переопределен в нашем классе области, – `getAuthType()`. Этот метод должен возвратить строку, содержащую описание типа аутентификации, используемого этой областью.

Два вышеназванных метода объявляются как абстрактные в родительском классе `IASRealm`. Хотя метод `getJAASContext()` не является абстрактным, мы тем не менее должны переопределить его, поскольку значение, которое он возвращает, используется для определения типа аутентификации, который в свою очередь используется сервером приложений из файла `login.conf`. Возвращаемое значение этого метода будет использовано для отображения области на соответствующий модуль входа в систему.

Наконец, наш класс области должен содержать метод аутентификации пользователя. Мы вольны называть его как угодно. Кроме того, мы можем использовать столько параметров любого типа, сколько нам нужно. В нашем примере просто есть значения для единственного имени пользователя и жестко закодированного пароля. Опять же реальное приложение получило бы допустимые учетные данные из некоторого постоянного хранилища. Этот метод предназначается для вызова из соответствующего класса модуля входа в систему:

```
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);
}
```

Наш класс модуля входа в систему должен расширять класс com.sun.appserv.security.AppservPasswordLoginModule. Этот класс также находится внутри файла security.jar; необходимо переопределить только один метод authenticateUser(). Этот метод не принимает параметров и вызывает исключение LoginException, если аутентификация пользователя завершилась неудачей. Переменная _currentRealm определяется в родительском классе; она имеет тип com.sun.enterprise.security.auth.realm.Realm родительского класса всех классов области. Эта переменная инициализируется перед выполнением метода authenticateUser(). Класс модуля входа в систему должен проверить, что данный класс имеет ожидаемый тип (в нашем примере – SimpleRealm). В противном случае нужно вызвать исключение LoginException.

Другие две переменные, которые определяются в родительском классе и инициализируются перед выполнением метода authenticateUser(), – это _username и _password. Они содержат учетные данные пользователя, вводимые в форме входа в систему (при аутентификации на основе формы) или во всплывающем окне (при стандартной аутентификации). В нашем примере эти значения просто передаются классу области так, чтобы он мог проверить учетные данные пользователя.

Метод authenticateUser() должен вызвать метод commitUserAuthentication() родительского класса после успешной аутентификации. Этот метод принимает мас-

сив строковых объектов, содержащих группы, которым принадлежит пользователь. Наш пример просто вызывает метод `getGroupNames()`, определенный в классе области, и добавляет элементы `Enumeration`, которые этот метод возвращает, в массив, а затем передает массив методу `commitUserAuthentication()`.

Очевидно, что GlassFish «не осознает» существование нашей пользовательской области и классов модуля входа в систему. Мы должны добавить эти классы в CLASS-PATH GlassFish. Самый легкий способ сделать это – копирование JAR-файла, содержащего нашу пользовательскую область и модуль входа в систему в следующий каталог:

[Каталог установки GlassFish]/GlassFish/domains/domain1/lib

Последний шаг, которой нам нужно сделать прежде, чем мы сможем аутентифицировать пользователей приложения через нашу пользовательскую область, заключается в добавлении нашей новой пользовательской 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;
};

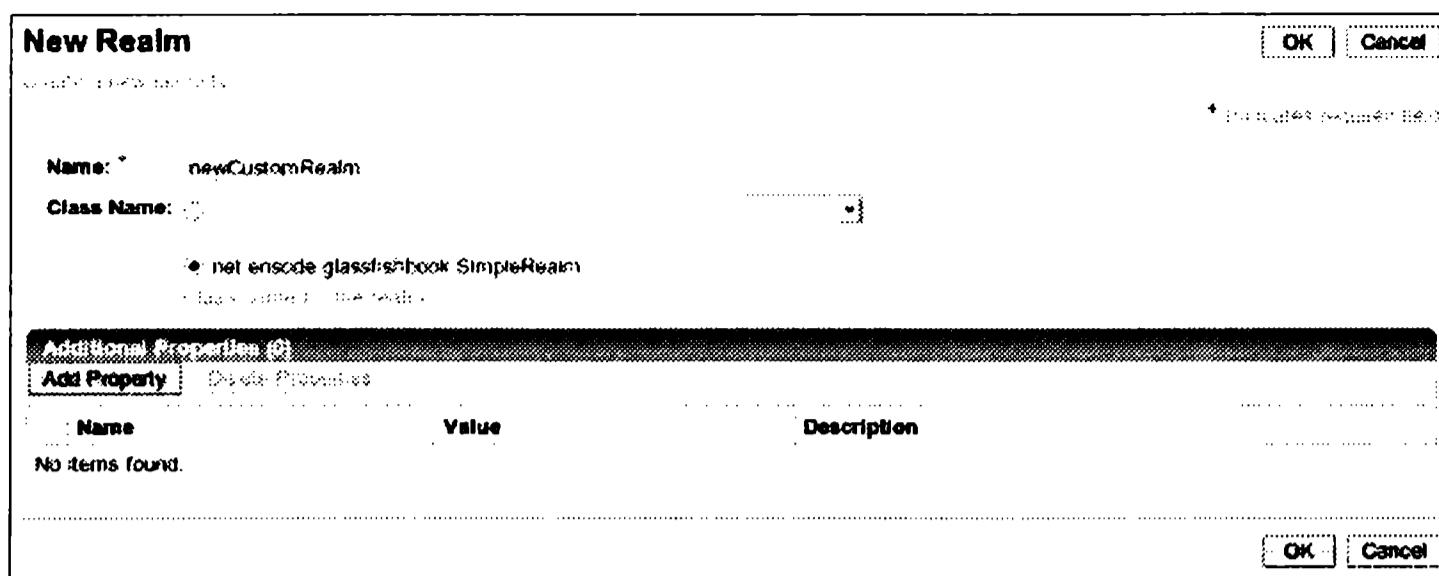
jdbcRealm
{
    com.sun.enterprise.security.auth.login.JDBCLoginModule required;
};

jdbcDigestRealm
{
    com.sun.enterprise.security.auth.login.JDBCDigestLoginModule required;
};

simpleRealm
{
    net.ensode.glassfishbook.SimpleLoginModule required;
};
```

Значение перед открывающейся фигурной скобкой должно соответствовать значению, возвращаемому методом `getJAASContext()` и определенному в классе области. Именно в этом файле классы области и модули входа в систему соединяются друг с другом. Домен GlassFish должен быть перезапущен для того, чтобы изменения вступили в силу.

Теперь мы готовы к использованию нашей пользовательской области для аутентификации пользователей в наших приложениях. Нам нужно добавить новый тип области, который мы создали, через консоль администрирования GlassFish:



На момент написания этой книги в GlassFish 3 существует проблема, которая препятствует успешному созданию пользовательских областей через веб-консоль. В качестве обходного решения пользовательские области могут быть созданы через утилиту командной строки `asadmin`, как будет показано ниже.

Чтобы создать нашу область, мы, как обычно, должны дать ей имя. Вместо того чтобы выбирать имя класса из раскрывающегося меню, необходимо ввести его в текстовое поле. У нашей пользовательской области не было никаких свойств, поэтому в данном примере нам нечего добавлять. Но если бы это требовалось, мы должны были бы щелкнуть по кнопке **Добавить свойства** (*Add Property*), а затем ввести имена свойств и соответствующие им значения. Наша область затем получила бы свойства путем переопределения метода `init()` из его родительского класса. Данный метод имеет следующую сигнатуру:

```
protected void init(Properties arg0) throws BadRealmException,
    NoSuchRealmException
```

Экземпляры `java.util.Properties`, которые он принимает в качестве параметра, были бы предварительно заполнены значениями свойств, вводимыми в страницу, которая показана на предыдущем снимке экрана (напомним: у нашей пользовательской области нет никаких свойств, но для тех, которые добавляются, значения свойств вводятся в названную страницу).

После того как мы добавили соответствующую информацию для нашей новой области, мы можем использовать ее точно так же, как любую из предопределенных областей. Приложения должны указывать ее имя в качестве значения элемента `<realm-name>` дескриптора развертывания приложения `web.xml`. Ничего необычного на уровне приложения делать не нужно.

Точно так же, как и стандартные области, пользовательские области могут быть добавлены через утилиту командной строки `asadmin`¹:

```
asadmin create-auth-realm --classname net.ensode.glassfishbook.
SimpleRealm newCustomRealm
```

¹ Вся команда записывается в одной строке. – Прим. перев.

Резюме

В этой главе мы обсудили использование областей GlassFish по умолчанию для аутентификации наших веб-приложений. Мы рассмотрели область файла, которая хранит пользовательскую информацию в плоском файле, и область сертификата, которая требует клиентских сертификатов для аутентификации пользователей.

Мы выяснили, как создать дополнительные области, которые ведут себя точно так же, как области по умолчанию при использовании классов области, включенных в GlassFish.

Также речь шла о том, как использовать дополнительные классы областей, включенные в GlassFish для создания областей, которые аутентифицируют через базу данных LDAP и через реляционную базу данных, и о том, как создать области, реализующие дополнительные механизмы аутентификации сервера, развернутого в операционной системе Solaris.

Наконец, мы показали, как создать классы пользовательских областей на тот случай, если имеющиеся области не соответствуют нашим потребностям.

9

Enterprise JavaBeans

Enterprise JavaBeans являются серверными компонентами, которые инкапсулируют бизнес-логику приложения. Enterprise JavaBeans упрощают разработку приложений, автоматически заботясь об управлении транзакциями и безопасности. Существуют два типа Enterprise JavaBeans: *сессионные бины*, которые выполняют бизнес-логику, и *управляемые сообщением бины*, которые действуют как приемники сообщений.

Читатели, знакомые с предыдущими версиями J2EE, заметят, что мы сейчас не упомянули сущностные бины. В Java EE 5 они были признаны устаревшими, и приоритет был отдан API Персистентности Java (Java Persistence API (JPA)). Сущностные бины все еще поддерживаются для обратной совместимости, однако предпочтительным способом выполнения объектно-реляционного отображения для Java EE 5 и Java EE 6 является JPA. Обратитесь к главе 5, «*API Персистентности Java*», для детального изучения JPA.

В этой главе мы затронем следующие темы:

- **сессионные бины:**
 - простой сессионный бин;
 - более реалистический пример;
 - использование сессионного бина для реализации шаблона проектирования DAO;
 - сессионный бин Одиночка (Singleton);
 - асинхронные вызовы метода;
- **управляемые сообщением бины;**
- **транзакции в Enterprise JavaBeans:**
 - транзакции, управляемые контейнером;
 - транзакции, управляемые бином;
- **жизненный цикл Enterprise JavaBeans:**
 - жизненный цикл сессионного бина, сохраняющего состояние;
 - жизненный цикл сессионного бина, не сохраняющего состояние;
 - жизненный цикл управляемого сообщением бина;
- **служба таймера EJB;**
- **безопасность EJB.**

Сеансовые бины

Как упоминалось выше, сеансовые бины обычно инкапсулируют бизнес-логику. Для создания сеансового бина в Java EE 5 должны быть созданы только два артефакта: собственно бин и бизнес-интерфейс. Эти артефакты должны быть декорированы соответствующими аннотациями, сообщающими контейнеру EJB, что они являются сеансовыми бинами.

Java EE 6 еще больше упрощает разработку сеансового бина. Локальные интерфейсы (будут обсуждаться позже в этой главе) теперь являются необязательными. Поэтому для разработки сеансового бина, который требует только локального доступа, мы должны разработать только один артефакт – класс сеансового бина.



Предыдущие версии J2EE требовали, чтобы разработчики приложений создавали несколько артефактов при создании сеансового бина. Эти артефакты включали собственно бин, локальный или удаленный интерфейс (или оба интерфейса), локальный домашний или удаленный домашний интерфейс (или оба интерфейса) и XML-дескриптор развертывания. Как мы увидим в этой главе, разработка EJB была значительно упрощена в Java EE 5 и еще более упрощена в Java EE 6.

Простой сеансовый бин

Следующий пример поясняет очень простой сеансовый бин:

```
package net.ensode.glassfishbook;

import javax.ejb.Stateless;
@Stateless
public class SimpleSessionBean
{
    private String message = "Если вы не видите этого, он не работает!";
    public String getMessage()
    {
        return message;
    }
}
```

Аннотация `@Stateless` сообщает контейнеру EJB, что этот класс является *сеансовым бином, не сохраняющим состояние* (stateless session bean). Существует два типа сеансовых бинов: не сохраняющие состояние и сохраняющие состояние. Прежде чем мы объясним различие между этими двумя типами, следует разъяснить, как экземпляр EJB предоставляется клиентским EJB-приложением.

Когда EJB (и сеансовые бины, не сохраняющие состояние, и управляемые сообщением бины) развертываются, контейнер EJB создает серию экземпляров каждого EJB, обычно называемую *пулом EJB* (EJB pool). Когда клиентское приложение EJB получает экземпляр EJB, один из экземпляров пула предоставляется этому клиентскому приложению.

Различие между сеансовыми бинами, сохраняющими состояние и не сохраняющими состояние, состоит 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 бина, сохраняющего состояние, нам гарантируется, что значение любых переменных экземпляра бина

будет непротиворечивым после вызова метода. Поэтому модифицировать любые переменные экземпляра на сеансовом бине, сохраняющем состояние, безопасно: они сохранят свое значение до следующего вызова метода. Контейнер EJB сохраняет состояние диалога при пассивации сохраняющих состояние сеансовых бинов и извлекает назначенное состояние, когда бин активируется. Из-за состояния диалога у сохраняющих состояние сеансовых бинов несколько более сложный жизненный цикл, чем у сеансовых бинов, не сохраняющих состояние, или управляемых сообщением бинов (жизненный цикл EJB будет обсуждаться ниже).

Контейнер EJB может предоставить любой экземпляр EJB из пула, когда клиентское EJB-приложение запрашивает экземпляр сеансового бина, не сохраняющего состояние. Для сеансовых бинов, не сохраняющих состояние, значение переменных экземпляра не устанавливается таким, каким оно было при последнем вызове метода (что, собственно, делает пассивация/активация для сохраняющих состояние сеансовых бинов). Поскольку нам не гарантирован тот же самый экземпляр для каждого вызова метода, значения, установленные в любые переменные экземпляра в сеансовом бине, не сохраняющем состояние, могут быть «потеряны» (в действительности они не теряются, а попросту могут находиться в другом экземпляре EJB из пула).

Кроме того, в декорировании аннотацией `@Stateless` нет ничего особенного по сравнению с предыдущим классом. Обратите внимание, что он реализует интерфейс под названием `SimpleSession`, являющийся бизнес-интерфейсом бина. Этот интерфейс показан в следующем коде:

```
package net.ensode.glassfishbook;

import javax.ejb.Remote;
@Remote
public interface SimpleSession
{
    public String getMessage();
}
```

Единственная особенность этого интерфейса состоит в том, что он декорируется аннотацией `@Remote`. Она указывает, что перед нами *удаленный бизнес-интерфейс* (*remote business interface*). Это означает, что интерфейс может находиться в другой JVM, отличной от JVM вызывающего его клиентского приложения. Удаленные бизнес-интерфейсы могут быть вызваны даже по сети.

Также бизнес-интерфейсы могут быть декорированы 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 быть вызван из той же JVM, где выполняется клиентское приложение, или из другой JVM. На первый взгляд кажется, что было бы здорово, если бы все наши бизнес-интерфейсы были удаленными. Однако следует учитывать, что гибкость, предоставляемая удаленными бизнес-интерфейсами, приводит к потере производительности, поскольку вызовы методов производятся при условии, что они будут сделаны по сети. В действительности самое типичное приложение Java EE состоит из веб-приложений, выступающих в качестве клиентских

приложений для EJB. В этом случае и клиентское приложение, и EJB запущены на одной и той же JVM. Поэтому локальные бизнес-интерфейсы используются намного чаще, чем удаленные бизнес-интерфейсы.

Скомпилировав сеансовые бины и соответствующие им бизнес-интерфейсы, мы должны поместить их в JAR-файл и затем развернуть. Как и в случае с WAR-файлами, самый легкий способ развертывания EJB JAR-файла – его копирование в [*Каталог установки glassfish*]/glassfish/domains/domain1/autodeploy.

Теперь, когда мы рассмотрели сеансовый бин и соответствующий ему бизнес-интерфейс, давайте рассмотрим пример 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 имеет тип: "
            + simpleSession.getClass().getName());
    }
    public static void main(String[] args)
    {
        new SessionBeanClient().invokeSessionBeanMethods();
    }
}
```

Данный код просто объявляет переменную экземпляра типа `net.ensode.SimpleSession`. Этот тип является бизнес-интерфейсом для нашего сеансового бина. Переменная экземпляра декорируется аннотацией `@EJB`. Она сообщает контейнеру EJB, что переменная является бизнес-интерфейсом для сеансового бина. Затем контейнер EJB инжектирует реализацию бизнес-интерфейса для того, чтобы использовать клиентский код.

Поскольку наш клиент является автономным приложением (в противоположность артефакту EJB, такому как WAR-файл, или другому EJB JAR-файлу), то, чтобы иметь возможность получить доступ к коду, развернутому на сервере, он должен быть помещен в JAR-файл и выполнен с помощью утилиты `appclient`. Ее можно найти в [*Каталог установки glassfish*]/glassfish/bin/. Предполагается, что этот путь находится в переменной окружения `PATH` и что мы поместили наш клиентский код в JAR-файл `simplesessionbeanclient.jar`. Мы будем выполнять предыдущий клиентский код, введя следующую команду в командной строке:

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

Результатом выполнения этой команды является следующий консольный вывод:

Если Вы не видите этого, он не работает!

SimpleSession имеет тип: net.ensode.glassfishbook._SimpleSession_Wrapper

Это вывод класса `SessionBeanClient`.



Мы используем Maven 2 для построения нашего кода. В данном примере мы использовали плагин сборки Maven (<http://maven.apache.org/plugins/maven-assembly-plugin/>) для создания клиентского JAR-файла, который включает все зависимости. Это освобождает нас от необходимости указывать все зависимые JAR-файлы в параметре командной строки `-classpath` утилиты `appclient`. Чтобы создать этот JAR-файл, достаточно вызвать из командной строки `mvn assembly:assembly`.

Первая строка вывода является просто возвращаемым значением метода `getMessage()`, который мы реализовывали в сеансовом бине. Вторая строка выводит на экран полностью определенное (квалифицированное) имя класса, реализующего бизнес-интерфейс. Обратите внимание, что имя класса не является полностью определенным именем сеансового бина, который мы написали. То, что фактически отображается, является реализацией бизнес-интерфейса, созданного контейнером EJB «за кулисами».

Более реалистичный пример

В предыдущем разделе мы рассмотрели очень простой пример в духе «Привет, мир». В этом разделе мы покажем более реалистический пример. Сеансовые бины часто используются в качестве *Объектов доступа к данным* (Data Access Objects (DAO)). Иногда они используются в качестве обертки для вызовов JDBC, в других случаях – в качестве обертки вызовов для получения или модификации JPA-сущностей. В этом разделе мы будем придерживаться последнего подхода.

Следующий пример поясняет, как реализовать шаблон проектирования DAO в сеансовом бине. Прежде чем переходить к реализации бина, давайте рассмотрим соответствующий ему бизнес-интерфейс:

```
package net.ensode.glassfishbook;

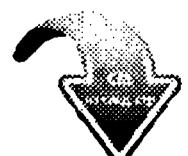
import javax.ejb.Remote;
@Remote
public interface CustomerDao
{
    public void saveCustomer(Customer customer);
    public Customer getCustomer(Long customerId);
    public void deleteCustomer(Customer customer);
}
```

Как мы видим, этот фрагмент кода является удаленным интерфейсом, реализующим три метода. Метод `saveCustomer()` сохраняет данные о заказчиках в базе данных, метод `getCustomer()` получает данные о заказчиках из базы данных, а метод `deleteCustomer()` удаляет данные о заказчиках из базы данных. Все эти методы принимают в качестве параметра экземпляр сущности `Customer`, который мы разработали в главе 4.

Теперь давайте рассмотрим сеансовый бин, реализующий предыдущий бизнес-интерфейс. Как мы скоро увидим, имеются некоторые различия между способом, которым код JPA реализуется в сеансовом бине, и тем, как он реализуется в «старом добром» (простом) объекте Java (POJO).

```
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);
    }
}
```

Первое различие, которое мы должны заметить, состоит в том, что экземпляр `javax.persistence.EntityManager` инжектируется непосредственно в сеансовый бин. В предыдущих примерах JPA мы должны были инжектировать экземпляр `javax.persistence.EntityManagerFactory`, затем использовать инжектированный экземпляр `EntityManagerFactory` для получения экземпляра `EntityManager` – по той причине, что наши предыдущие примеры не были ориентированы на многопоточное выполнение. Это означает, что потенциально тот же самый код мог быть выполнен несколькими пользователями сразу. Поскольку `EntityManager` не разрабатывался для использования более чем одним потоком одновременно, мы задействовали экземпляр `EntityManagerFactory`, чтобы предоставить каждому потоку его собственный экземпляр `EntityManager`. Контейнер EJB в один момент времени присваивает сеансовый бин одному клиенту, и, следовательно, сеансовые компоненты по сути ориентированы на многопоточное исполнение. Поэтому мы можем инжектировать (ввести) экземпляр `EntityManager` непосредственно в сеансовый бин.

Следующее отличие данного сеансового бина от предыдущих примеров JPA заключается в том, что в предыдущих примерах вызовы JPA были обернуты в конструкцию из вызовов `UserTransaction.begin()` и `UserTransaction.commit()`.

Это требовалось нам, поскольку вызовы JPA должны быть обернуты в транзакцию. Если они не будут находиться в транзакции, то большинство вызовов JPA вызовет исключение `TransactionRequiredException`. Причина, по которой мы не должны явно оберачивать вызовы JPA в транзакцию в предыдущих примерах, состоит в том, что методы сеансового бина – неявно транзакционные. Поэтому нам нет смысла поступать иначе. Данное поведение является их поведением по умолчанию; оно также известно как *транзакции, управляемые контейнером* (*container-managed transactions*). Транзакции, управляемые контейнером, будут подробно обсуждаться ниже.



Как упомянуто в главе 5, когда сущность JPA получается в одной транзакции и обновляется в другой, должен быть вызван метод `EntityManager.merge()`, чтобы обновить данные в базе данных. В этом случае результатом вызова метода `EntityManager.persist()` будет исключение: «Невозможно сохранить, отсоединеный объект».

Вызов сеансовых бинов из веб-приложений

Часто приложения Java EE состоят из веб-приложений, играющих роль клиентов для EJB. До выпуска Java EE 6 наиболее распространенный способ развертывания приложений Java EE, которые состоят из веб-приложения и одного или нескольких сеансовых бинов, заключался в упаковке веб-приложения в WAR-файл, а JAR-файлов EJB – в файл EAR (архив предприятия).

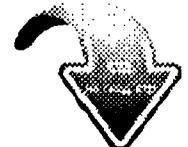
Java EE 6 упрощает упаковку и развертывание приложений, включающих EJB, и веб-компоненты.

В этом разделе мы модифицируем пример, рассмотренный в разделе «*Интеграция JSF и JPA*» главы 6 (см. стр. 210), таким образом, чтобы веб-приложение действовало как клиент для сеансового бина DAO, который мы видели в предыдущем разделе.

Чтобы это приложение могло выступать в качестве клиента EJB, мы изменим управляемый бин `CustomerController`, который должен делегировать логику сохранения нового заказчика в базе данных сеансовому бину `CustomerDaoBean`, разработанному нами в предыдущем разделе.

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

import javax.ejb.EJB;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.ManagedProperty;
import net.ensode.glassfishbook.Customer;
import net.ensode.glassfishbook.CustomerDaoBean;
@ManagedBean
public class CustomerController
{
    @EJB
    CustomerDaoBean customerDaoBean;
    @ManagedProperty(value = "#{customer}")
    private Customer customer;
    public String saveCustomer()
    {
        String returnValue = "customer_saved";
        try
        {
            customerDaoBean.saveCustomer(customer);
        }
        catch (Exception e)
        {
            returnValue = "customer_error";
        }
    }
}
```



```
        catch (Exception e)
        {
            e.printStackTrace();
            returnValue = "error_saving_customer";
        }
        return returnValue;
    }
    public Customer getCustomer()
    {
        return customer;
    }
    public void setCustomer(Customer customer)
    {
        this.customer = customer;
    }
}
```

Как видно из приведенного кода, нам нужно всего лишь объявить экземпляр сеансового бина `CustomerDaoBean`, декорировав его аннотацией `@EJB` так, чтобы экземпляр соответствующего EJB был инжектирован и заменил собой код для сохранения данных в базе данных с помощью вызова метода `saveCustomer()`, определенного в бизнес-интерфейсе `CustomerDao`.

Обратите внимание, что мы инжектировали экземпляр сеансового бина прямо в наш клиентский код. Нам позволяет это сделать новая функциональность Java EE 6. При использовании Java EE 6 мы можем покончить с локальными интерфейсами и использовать экземпляры сеансового бина прямо в нашем клиентском коде.

Теперь, когда мы модифицировали наше веб-приложение для того, чтобы оно выступало в роли клиента для нашего сеансового бина, мы должны упаковать его в WAR-файл (веб-архив), после чего развернуть его для использования.

Одноэлементный сеансовый бин (Singleton)

Новый тип сеансового бина, введенного в Java EE 6, – *одноэлементный сеансовый бин* (*singleton session bean*). На сервере приложений может существовать только единственный экземпляр каждого одноэлементного сеансового бина.

Сеансовые бины `Singleton` полезны для кэширования данных базы данных. Кэширование часто используемых данных в сеансовом бине-одиночке увеличивает производительность, поскольку это значительно снижает количество обращений в базу данных. Общий подход состоит в том, что наш бин должен иметь метод, декорированный аннотацией `@PostConstruct`. В этом методе мы получаем данные, которые хотим кэшировать. Затем мы предоставляем сеттер-метод для вызова бина клиентами. Следующий пример поясняет 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");
        stateList = query.getResultList();
    }
    @Override
    public List<UsStates> getStateList()
    {
        return stateList;
    }
}
```

Поскольку наш бин является одиночным элементом, все его клиенты получают доступ к одному и тому же экземпляру, избегая таким образом дублирования данных в базе данных. По этой же причине безопасно иметь переменную экземпляра, поскольку все клиенты получают доступ к одному и тому же экземпляру бина и, соответственно, к одной и той же переменной.

Асинхронные вызовы метода

Иногда полезно иметь некоторую обработку, выполняемую асинхронно, т. е. когда происходит вызов метода и управление сразу же возвращается клиенту, без необходимости для клиента ожидать завершения метода.

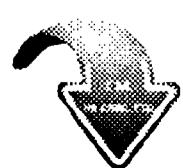
В более ранних версиях Java EE единственным способом асинхронного вызова метода EJB было использование управляемых сообщением бинов (они будут обсуждаться в следующем разделе). Хотя управляемые сообщением бины довольно легко написать, в действительности они требуют некоторого конфигурирования – такого как создание и настройка очереди или темы сообщений JMS, прежде чем их можно будет использовать.

EJB 3.1 вводит аннотацию `@Asynchronous`, которая может использоваться для того, чтобы пометить метод сеансового бина как асинхронный. Когда клиент EJB вызывает асинхронный метод, управление сразу возвращается клиенту, не ожидая завершения выполнения метода.

Асинхронные методы могут возвращать только `void` или реализацию интерфейса `java.util.concurrent.Future`. Следующий пример поясняет 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("входим " + this.getClass().getCanonicalName() +
            ".slowMethod()");
        try
        {
            Thread.sleep(10000); // Имитации обработки в течение 10 секунд
        }
        catch (InterruptedException ex)
        {
            Logger.getLogger(AsynchronousSessionBean.class.getName()).log(Level.SEVERE, null, ex);
        }
        logger.info("покидаем " + this.getClass().getCanonicalName() +
            ".slowMethod()");
        long endTime = System.currentTimeMillis();
        logger.info("выполнение заняло " + (endTime - startTime) +
            " миллисекунд");
    }
    @Asynchronous
    @Override
    public Future<Long> slowMethodWithValue()
    {
        try
        {
            Thread.sleep(15000); // Имитации обработки в течение 15 секунд
        }
        catch (InterruptedException ex)
        {
            Logger.getLogger(AsynchronousSessionBean.class.getName()).log(Level.SEVERE, null, ex);
        }
        return new AsyncResult<Long>(42L);
    }
}

```

Поскольку наш асинхронный метод возвращает `void`, единственное, что нам нужно сделать, – это декорировать метод аннотацией `@Asynchronous`, а затем вызвать его, как обычно, из клиентского кода.

Если нам нужно возвращаемое значение, оно должно быть обернуто в реализацию интерфейса `java.util.concurrent.Future`. Для удобства API Java EE 6 предоставляет реализацию в виде класса `javax.ejb.AsyncResult`. Оба они – и интерфейс `Future`, и класс `AsyncResult` – используют обобщения. Мы должны указать тип нашего возвращаемого значения в качестве параметра типа этих артефактов.

У интерфейса `Future` имеется несколько методов, которые можно использовать для отмены выполнения асинхронного метода. При этом необходимо придерживаться следующего порядка действий: выяснить, выполнился ли метод, затем получить возвращаемое методом значение либо выяснить, не был ли метод отменен.

В следующей таблице перечислены эти методы:

Метод	Описание
cancel(boolean mayInterruptIfRunning)	Отменяет исполнение метода. Если логический параметр имеет значение <code>true</code> , этот метод будет пытаться отменить исполнение метода, даже если он уже запущен
get()	Возвращает «оболочку», которая, в свою очередь, возвращает значение метода. Возвращаемое значение будет иметь тип параметра реализации интерфейса <code>Future</code> , возвращенного методом
get(long timeout, TimeUnit unit)	Возвратит «распакованное» возвращаемое значение метода. Возвращаемое значение будет иметь тип параметра реализации интерфейса <code>Future</code> , возвращенного методом. Этот метод блокируется на интервал времени, указанный первым параметром. Единица измерения времени ожидания определяется вторым параметром. Перечисление <code>TimeUnit</code> имеет такие константы: <code>NANOSECONDS</code> (наносекунды), <code>MILLISECONDS</code> (миллисекунды), <code>SECONDS</code> (секунды), <code>MINUTES</code> (минуты) и т. д. Для ознакомления с полным списком обратитесь к JavaDoc-документации
isCancelled()	Возвращает значение <code>true</code> , если метод был отменен; в противном случае возвращает <code>false</code>
isDone()	Возвращает <code>true</code> , если метод закончил работу; в противном случае возвращает <code>false</code>

Как видно из таблицы, аннотация `@Asynchronous` облегчает выполнение асинхронных вызовов, устранив издержки, связанные с созданием очередей или тем сообщений. Безусловно, это долгожданное дополнение к спецификации EJB 3.1.

Управляемые сообщением бины

Назначением управляемого сообщением бина является использование сообщения из очереди JMS или темы JMS – в зависимости от используемого домена обмена сообщениями (за подробностями обратитесь к главе 7, «Служба обмена сообщениями Java»). Управляемый сообщением бин должен быть декорирован аннотацией `@MessageDriven`. Атрибут `mappedName` этой аннотации должен содержать JNDI-имя очереди или темы сообщений JMS, из которой бин будет использовать сообщения. Следующий пример иллюстрирует простой управляемый сообщением бин:

```
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("Принято следующее сообщение: ");
            System.out.println(textMessage.getText());
            System.out.println();
        }
        catch (JMSEException e)
        {
            e.printStackTrace();
        }
    }
}
```

Как видно из приведенного кода, этот класс практически идентичен классу ExampleMessageListener, который мы видели в главе 7; разница только в имени класса и в том, что данный пример декорируется аннотацией @MessageDriven. Рекомендуется, но не требуется, чтобы управляемые сообщением бины реализовывали интерфейс javax.jms.MessageListener. Кроме того, у управляемых сообщением бинов должен быть метод onMessage() с сигнатурой, идентичной предыдущему примеру.

Клиентские приложения никогда не вызывают напрямую методы управляемого сообщением бина. Вместо этого они помещают сообщения в очередь или тему сообщений; затем бин потребляет эти сообщения и действует соответствующим образом. Предыдущий пример просто печатает сообщение в стандартный вывод. Поскольку управляемые сообщением бины выполняются в контейнере EJB, стандартный вывод получает переадресацию в файл журнала сервера. Чтобы просмотреть сообщения в журнале сервера GlassFish, откройте файл [Каталог установки GlassFish]/GlassFish/domains/domain1/logs/server.log¹.

Транзакции в Enterprise JavaBeans

Как мы уже отмечали выше, любые методы EJB по умолчанию автоматически обертываются в транзакцию. Это поведение по умолчанию известно как *транзакции, управляемые контейнером* (container-managed transactions), поскольку транзакциями управляет контейнер EJB. У разработчиков приложений также может возникнуть потребность в управлении транзакциями напрямую. Это может быть реализовано за счет использования *транзакций, управляемых бином* (bean-managed transactions). Оба этих подхода мы рассмотрим в следующих разделах.

¹ Предполагается, что используется домен по умолчанию. Если вы используете другой домен, то в приведенном пути domain1 нужно заменить именем вашего домена. – Прим. перев.

Транзакции, управляемые контейнером

Поскольку методы EJB являются транзакционными по умолчанию, мы сталкиваемся с интересной дилеммой, когда сеансовый бин вызывается из клиентского кода, который уже находится в транзакции: 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? А разве он не должен создать новую транзакцию и выполнить ее метод как часть клиентской транзакции? Или, может быть, он должен выдать исключение?

По умолчанию, если метод EJB будет вызван клиентским кодом, который уже находится в транзакции, контейнер EJB просто выполнит метод сеансового бина как часть клиентской транзакции. Если это не то поведение, которое нам нужно, мы можем изменить его, декорируя метод аннотацией `@TransactionAttribute`. У этой аннотации имеется атрибут `value`, который определяет, как будет вести себя контейнер EJB, когда будет вызван метод сеансового бина в пределах существующей транзакции и когда он вызывается вне любых транзакций. Значение атрибута `value` обычно является константой, определенной в перечислении `javax.ejb.TransactionAttributeType`.

В следующей таблице приведены возможные значения аннотации `@TransactionAttribute`:

Значение <code>@TransactionAttribute</code>	Описание
<code>TransactionAttributeType.MANDATORY</code>	Приводит к принудительному вызову метода как части клиентской транзакции. Если метод будет вызван вне каких-либо транзакций, он вызовет исключение <code>TransactionRequiredException</code>
<code>TransactionAttributeType.NEVER</code>	Этот метод никогда не выполняется в рамках транзакции. Если метод будет вызван как часть клиентской транзакции, это вызовет исключение <code>RemoteException</code> . Транзакция не создается, если метод не вызывается внутри клиентской транзакции
<code>TransactionAttributeType.NOT_SUPPORTED</code>	Если метод вызывается как часть клиентской транзакции, клиентская транзакция приостанавливается и метод выполняется вне любой транзакции. После того как метод завершается, клиентская транзакция возобновляется. Транзакция не создается, если метод не вызывается внутри клиентской транзакции
<code>TransactionAttributeType.REQUIRED</code>	Если метод вызывается как часть клиентской транзакции, метод выполняется как часть упомянутой транзакции. Если метод вызывается вне какой-либо транзакции, для метода создается новая транзакция. Это поведение по умолчанию

Значение @TransactionAttribute	Описание
TransactionAttributeType. REQUIRES_NEW	Если метод вызывается как часть клиентской транзакции, упомянутая транзакция приостанавливается и для метода создается новая транзакция. Как только метод завершается, клиентская транзакция возобновляется. Если метод вызывается вне каких-либо транзакций, для метода создается новая транзакция
TransactionAttributeType. SUPPORTS	Если метод вызывается как часть клиентской транзакции, он выполняется как часть названной транзакции. Если метод вызывается вне транзакции, новая транзакция для метода не создается

Хотя значение по умолчанию атрибута транзакции приемлемо в большинстве случаев, желательно иметь возможность переопределить этот атрибут транзакции, установленный по умолчанию, в случае необходимости. Например, поскольку транзакции оказывают влияние на производительность, удобно было бы выключать транзакции для метода, который в них не нуждается. В таких случаях мы будем декорировать наш метод, как показано в следующем фрагменте кода:

```
@TransactionAttribute (value=TransactionAttributeType.NEVER)
public void doitAsFastAsPossible()
{
    // Критичный к производительности код помещается здесь.
}
```

Другие типы атрибута транзакции могут быть объявлены путем аннотирования метода соответствующей константой перечисления TransactionAttributeType.

Если мы хотим переопределить атрибут транзакции по умолчанию последовательно для всех методов в сеансовом бине, мы можем декорировать класс сеансового бина аннотацией @TransactionAttribute. Значение его атрибута value будет применено к каждому методу в сеансовом бине.

Транзакции, управляемые контейнером, автоматически откатываются всякий раз, когда в методе EJB возникает исключение. Кроме того, мы можем программно откатывать транзакцию, управляемую контейнером, вызывая метод setRollbackOnly() на экземпляре javax.ejb.EJBContext, соответствующем рассматриваемому сеансовому бину. Следующий пример представляет новую версию сеансового бина, который мы видели ранее в этой главе. Теперь он модифицирован, чтобы откатывать транзакции в случае необходимости:

```
package net.ensode.glassfishbook;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.annotation.Resource;
import javax.ejb.EJBContext;
```

```
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.sql.DataSource;
@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);
        }
    }
    // Дополнительные методы для краткости опущены.
}
```

В этой версии сеансового бина DAO мы удалили метод `saveCustomer()` и сделали методы `saveNewCustomer()` и `updateCustomer()` общедоступными (`public`). Каждый из этих методов теперь выясняет, установлено ли значение поля `customerId` корректно для операции, которую мы пытаемся выполнить (`null` для вставок и `not null` – для обновлений). Он также проверяет, что объект, который будет сохранен, не является нулем. Если результатом какой-либо проверки будут неправильные данные, метод просто откатывает транзакцию, вызывая метод `setRollBackOnly()` на инжектированном экземпляре `EJBContext`, и не обновляет базу данных.

Транзакции, управляемые бином

Как мы уже убедились, транзакции, управляемые контейнером, делают смехотворно легким написание кода, который обертывается в транзакцию. Попросту говоря, для подобной реализации транзакции нам не нужно делать ничего особенного. Некоторые разработчики иногда даже не знают, что они пишут код, который в реальности будет транзакционным, когда они разработают сеансовый бин. Транзакции, управляемые контейнером, охватывают большинство типичных случаев, с которыми мы

встречаемся на практике. Однако в действительности у них имеется ограничение: каждый метод может быть обернут в единственную транзакцию либо должен находиться вне транзакции. С помощью транзакций, управляемых контейнером, невозможно реализовать метод, который генерирует больше чем одну транзакцию. Но это может быть осуществлено при использовании *транзакций, управляемых бином* (*bean-managed transactions*).

```
package net.ensode.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;
import javax.persistence.PersistenceContext;
import javax.sql.DataSource;
import javax.transaction.UserTransaction;
@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 мы реализовали метод `saveMultipleNewCustomers()`, принимающий `ArrayList` заказчиков в качестве единственного параметра. Целевая установка этого метода состоит в том, чтобы сохранить столько элементов в `ArrayList`, сколько окажется возможным. Исключение, возникающее при сохранении одной из сущностей, не должно мешать попыткам метода сохранить оставшиеся элементы. Такое поведение недопустимо при использовании транзакций, управляемых контейнером, поскольку исключение, возникшее при сохранении одной из сущностей, откатывает транзакцию целиком. Единственный способ достигнуть требуемого поведения заключается в использовании транзакций, управляемых бином.

Как видно из примера, мы объявляем, что сеансовый бин использует транзакции, управляемые бином, путем декорирования класса аннотацией `@TransactionManagement` и используя `TransactionManagementType.BEAN` в качестве значения его атрибута `value` (другое допустимое значение этого атрибута – `TransactionManagementType.CONTAINER`, но поскольку оно является значением по умолчанию, в его указании нет необходимости).

Чтобы иметь возможность программно управлять транзакциями, мы инжектируем экземпляр `javax.transaction.UserTransaction`, который затем используется в цикле `for` в методе `saveMultipleNewCustomers()` для запуска и фиксации транзакций на каждой итерации цикла.

Если нам нужно откатить транзакцию, управляемую бином, мы можем это сделать путем простого вызова метода `rollback()` на соответствующем экземпляре `javax.transaction.UserTransaction`.

Перед тем как двигаться дальше, отметим следующее: несмотря на то, что все приведенные в этом разделе примеры были сеансовыми бинами, объясненные здесь понятия также применяются и к управляемым сообщением бинам.

Жизненный цикл Enterprise JavaBeans

Бины Enterprise JavaBeans в их жизненном цикле проходят через различные состояния. Каждый тип EJB имеет различные состояния. Состояния, определенные для каждого типа EJB, обсуждаются в следующих разделах.

Жизненный цикл сеансового бина с сохранением состояния

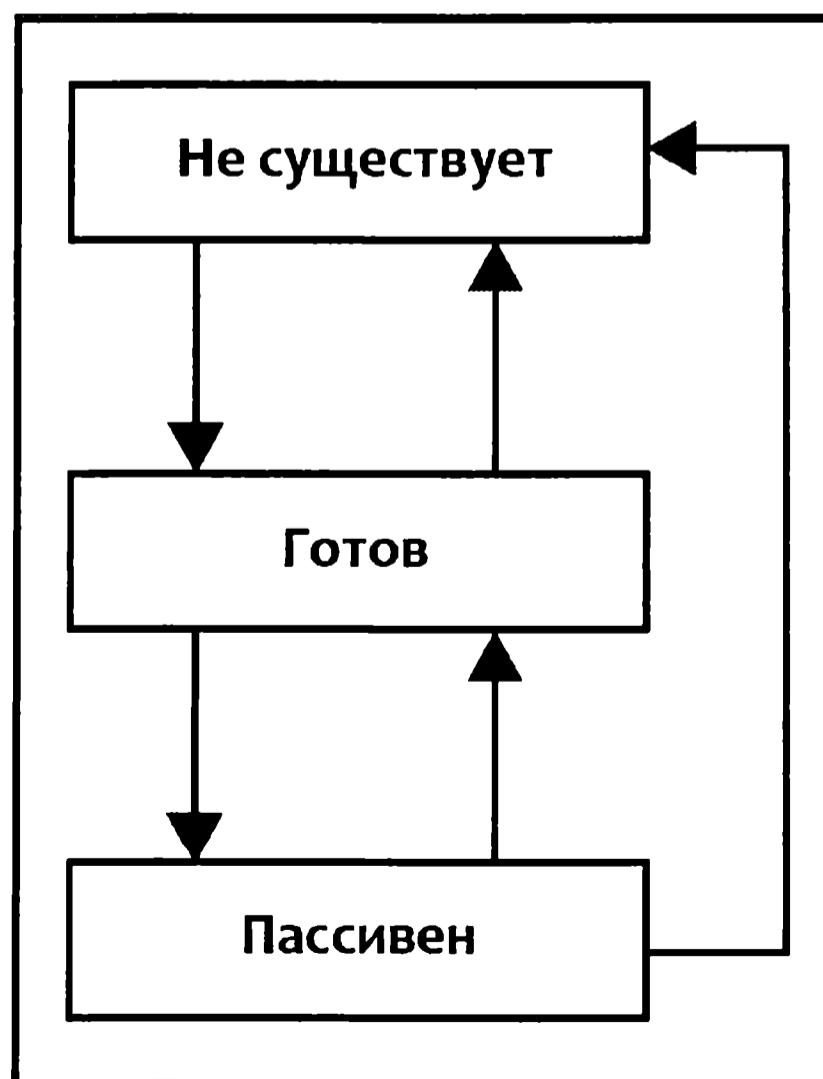
Читатели, имеющие опыт работы с предыдущими версиями J2EE, помнят, что в предыдущих версиях спецификации сеансовые бины были обязаны реализовывать интерфейс `javax.ejb.SessionBean`. Он предоставляет методы, которые будут выполняться в определенные моменты жизненного цикла сеансового бина. Методы, предоставляемые интерфейсом `SessionBean`, включают:

- `ejbActivate()`;
- `ejbPassivate()`;
- `ejbRemove()`;
- `setSessionContext (SessionContext ctx)`.

Первые три метода предназначены для выполнения в определенные моменты жизненного цикла бина. В большинстве случаев в реализации этих методов не нужно делать ничего (отсюда огромное количество сеансовых бинов, реализующих пустые версии этих методов). К счастью, начиная с Java EE 5, больше нет необходимости реализовывать интерфейс `SessionBean`. Однако при необходимости мы все же можем написать эти методы, которые будут выполняться в определенные моменты жизненного цикла бина. Этого можно достичь, декорируя методы определенными аннотациями.

Прежде чем объяснить аннотации, доступные для реализации методов жизненного цикла, вкратце поясним жизненный цикл сеансового бина. Жизненный цикл сеансового бина с сохранением состояния отличается от жизненного цикла сеансового бина, не сохраняющего состояние.

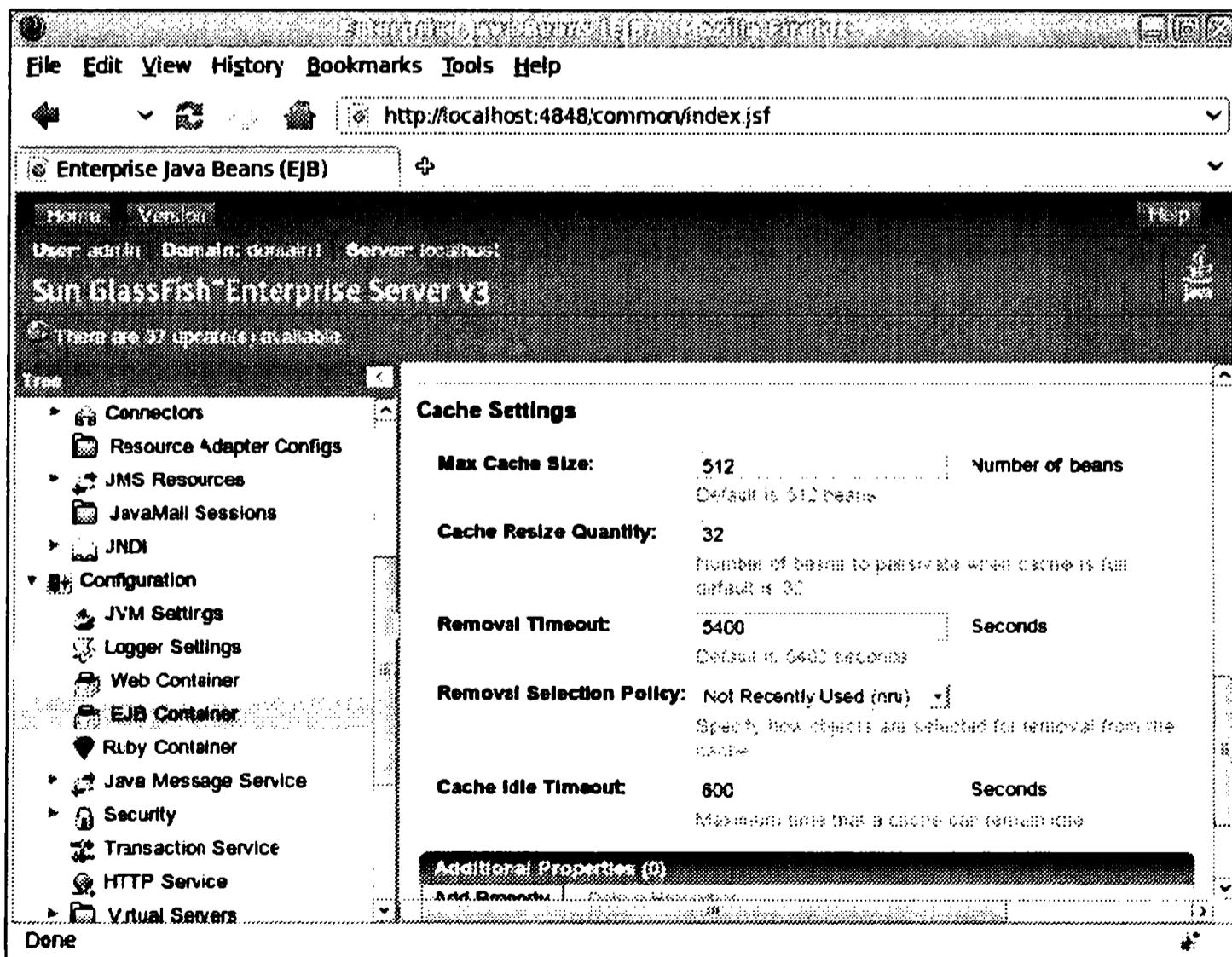
Жизненный цикл сеансового бина с сохранением состояния (stateful session bean) содержит три состояния: **Не существует** (Does Not Exist), **Готов** (Ready) и **Пассивен** (Passive).



Прежде чем сеансовый бин с сохранением состояния будет развернут, он попадает в состояние **Не существует** (Does Not Exist). После успешного развертывания контейнер EJB выполняет любую заданную инжекцию зависимости на бине, и он входит в **Состояние готовности** (Ready). На данном этапе бин готов предоставить свои методы для их вызова клиентским приложением.

Когда сеансовый бин с сохранением состояния находится в **Состоянии готовности**, контейнер EJB может решить, что нужно его пассивировать, т. е. переместить из оперативной памяти во внешнюю. Когда это происходит, бин входит в **Пассивное состояние** (Passive).

Если к экземпляру сеансового бина с сохранением состояния не получали доступ в течение определенного интервала времени, контейнер EJB установит бин в состояние **Не существует** (Does Not Exist). По умолчанию GlassFish переводит сеансовый бин с сохранением состояния в состояние **Не существует** после 90 минут отсутствия активности. Это значение по умолчанию можно изменить, перейдя в консоль администрирования GlassFish, развернув узел **Конфигурация** (Configuration) в дереве панели навигации слева, а затем щелкнув по узлу **Контейнер EJB** (EJB Container), прокрутив вниз, к концу страницы, панель основного содержания и изменив значение текстового поля **Тайм-аут удаления** (Removal Timeout). Для подтверждения сделанных изменений остается лишь щелкнуть по кнопке **Сохранить** (Save) в верхней правой части панели основного содержания страницы.



Однако этот метод устанавливает значение тайм-аута для всех сеансовых бинов с сохранением состояния. Если нам нужно модифицировать значение тайм-аута для конкретного сеансового бина, мы должны включить дескриптор развертывания

sun-ejb-jar.xml в файл JAR, содержащий сеансовый бин. В этом дескрипторе развертывания мы можем установить значение тайм-аута как значение элемента <removal-timeout-in-seconds>:

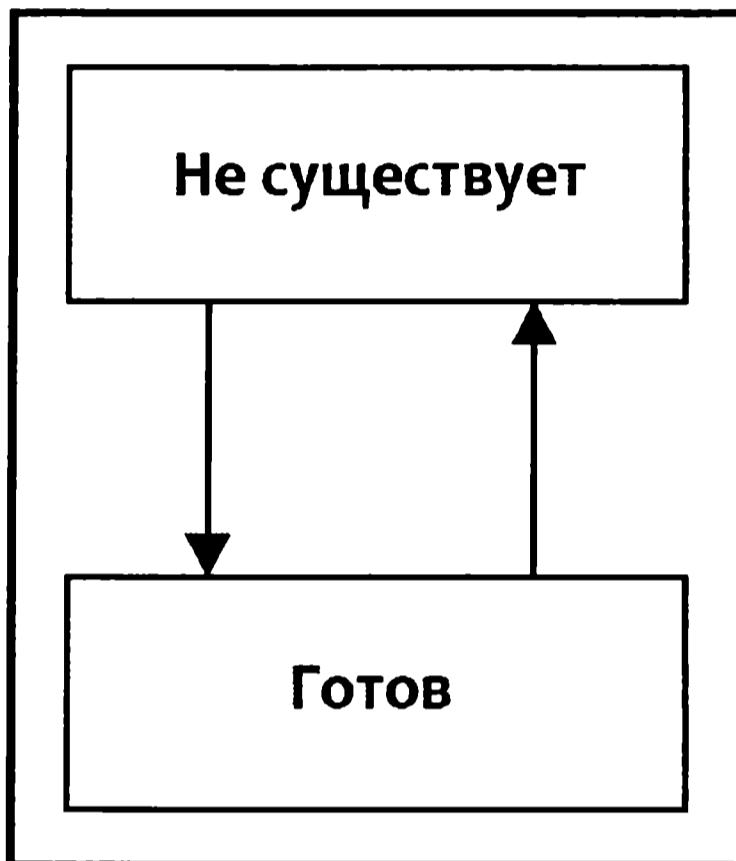
```
<? xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE sun-EJB-jar PUBLIC "-//Sun Microsystems,
    Inc./DTD Application Server 9.0 EJB 3.0//EN"
    "http://www.sun.com/software/appserver/dtds/sun-EJB-jar_3_0-0.dtd">
<sun-ejb-jar>
    <enterprise-beans>
        <ejb>
            <ejb-name>MyStatefulSessionBean</ejb-name>
            <bean-cache>
                <removal-timeout-in-seconds>
                    600
                </removal-timeout-in-seconds>
            </bean-cache>
        </ejb>
    </enterprise-beans>
</sun-ejb-jar>
```

Даже при том, что мы больше не обязаны создавать дескриптор развертывания ejb-jar.xml для наших сеансовых бинов (это требовалось раньше, в предыдущих версиях спецификации J2EE), мы все еще можем его записывать, если нам это понадобится. Элемент <ejb-name> в дескрипторе развертывания sun-ejb-jar.xml должен соответствовать значению одноименного элемента в файле ejb-jar.xml. Если мы не хотим создавать дескриптор развертывания ejb-jar.xml, это значение должно соответствовать имени класса EJB. Значение тайм-аута для сеансового бина с сохранением состояния должно быть значением элемента <removal-timeout-in-seconds>. Как видно из имени элемента, предполагается, что единицей изменения времени является секунда. В предыдущем примере мы установили значение тайм-аута равным 600 секундам (или 10 минутам). Любые методы в сеансовом бине с сохранением состояния, декорированный аннотацией @PostActivate, могут быть вызваны сразу после того, как сеансовый бин будет активирован. Это эквивалентно реализации метода ejbActivate() в предыдущих версиях J2EE. Точно так же любой метод, декорированный аннотацией @PrePassivate, будет вызван непосредственно перед тем, как сеансовый бин с сохранением состояния будет пассивирован. Это эквивалентно реализации метода ejbPassivate() в предыдущих версиях J2EE. Когда сеансовый бин с сохранением состояния, находящийся в состоянии **Готов**, переводится в состояние **Не существует**, выполняется любой метод, декорированный аннотацией @PreDestroy. Если сеансовый бин из состояния **Пассивен** переводится в состояние **Не существует**, методы, декорированные аннотацией @PreDestroy, не выполняются. Кроме того, если клиент сеансового бина с сохранением состояния выполняет какой-либо метод, декорированный аннотацией @Remove, то выполняются любые методы, декорированные аннотацией @PreDestroy, и бин помечается для сборщика «мусора». Декорирование метода аннотаций @Remove эквивалентно реализации метода ejbRemove() в предыдущих версиях спецификации J2EE. Аннотации @PostActivate, @PrePassivate и @Remove

допустимы только для сеансовых бинов с сохранением состояния. Аннотации `@PreDestroy` и `@PostConstruct` допустимы для сеансовых бинов с сохранением состояния, сеансовых бинов, не сохраняющих состояние, и управляемых сообщением бинов.

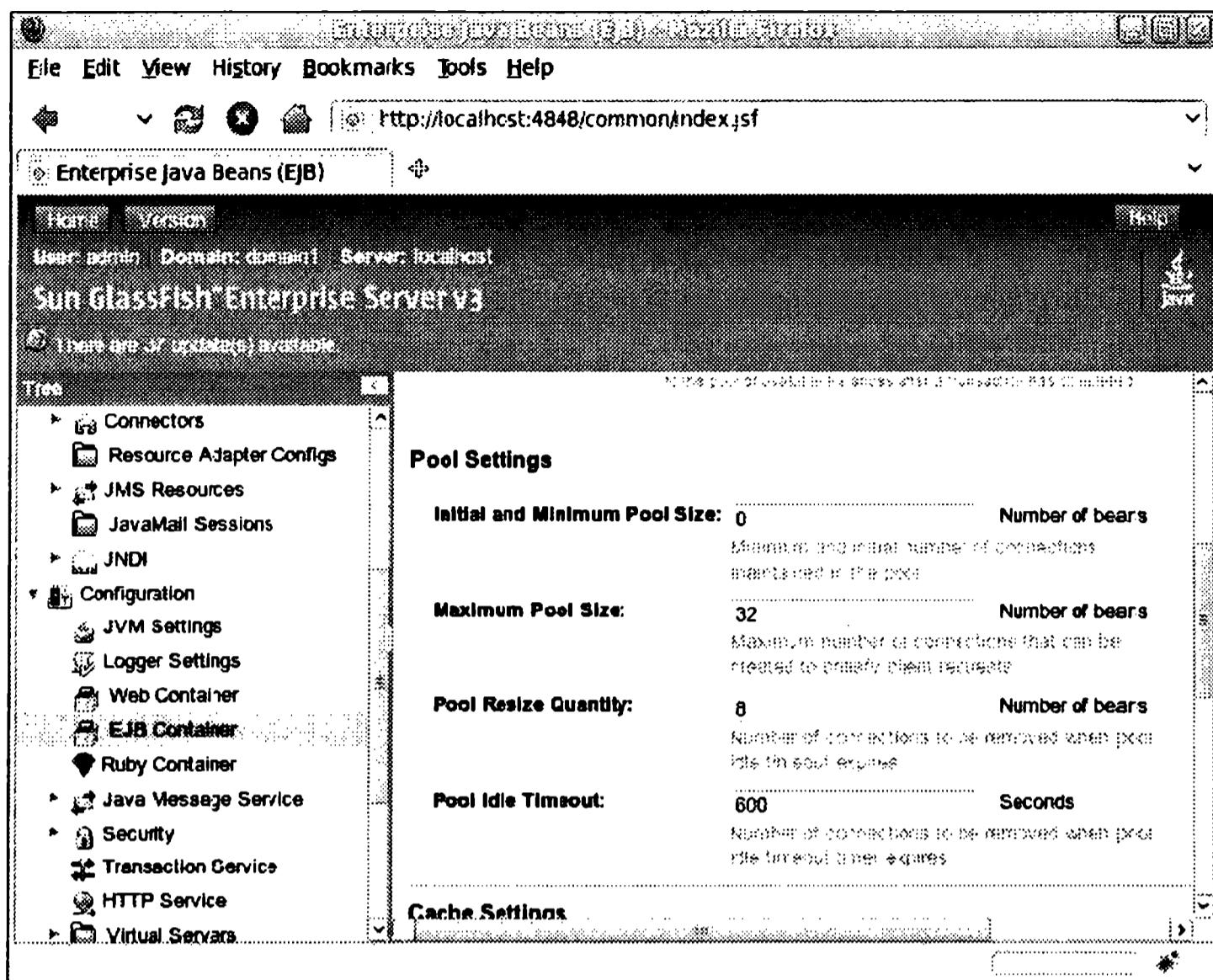
Жизненный цикл сеансового бина, не сохраняющего состояние

Жизненный цикл сеансового бина, не сохраняющего состояние, содержит только два состояния: **Не существует** (Does Not Exist) и **Готов** (Ready):



Сеансовые бины, не сохраняющие состояние, никогда не пассивируются. Методы сеансового бина, не сохраняющего состояние, могут быть декорированы аннотациями `@PostConstruct` и `@PreDestroy`. Так же, как в сеансовых бинах с сохранением состояния, любые методы, декорированные аннотацией `@PostConstruct`, будут выполняться, когда сеансовый бин, не сохраняющий состояние, переходит из состояния **Не существует** в состояние **Готов**; любые методы, декорированные аннотацией `@PreDestroy`, будут выполняться, когда сеансовый бин, не сохраняющий состояние, переходит из состояния **Готов** в состояние **Не существует**. Сеансовые бины, не сохраняющие состояние, никогда не пассивируются, поэтому любые аннотации `@PrePassivate` и `@PostActivate` в сеансовом бине, не сохраняющем состояние, просто игнорируются контейнером EJB.

Как и в случае с сеансовыми бинами, сохраняющими состояние, мы можем определять, как GlassFish управляет жизненным циклом сеансовых бинов, не сохраняющих состояние (а равно и управляемых сообщением бинов, которые будут обсуждаться в следующем разделе), через веб-консоль администрирования:



- **Начальный и минимальный размер пула (Initial and Minimum Pool Size)** – минимальное количество бинов в пуле;
- **Максимальный размер пула (Maximum Pool Size)** – максимальное количество бинов в пуле;
- **Величина изменения размера пула (Pool Resize Quantity)** – количество бинов, которые будут удалены из пула, когда истечет тайм-аут простоя пула;
- **Величина тайм-аута простоя пула (Pool Idle Timeout)** – время бездействия (в секундах), по истечении которого бины будут удалены из пула.

Эти настройки влияют на все EJB, помещаемые в пул (и сеансовые бины, не сохраняющие состояние, и управляемые сообщением бины). Как и в случае с сеансовыми бинами с сохранением состояния, эти настройки могут быть переопределены в каждом конкретном случае на индивидуальной основе – путем добавления специфического для GlassFish дескриптора развертывания `sun-ejb-jar.xml`:

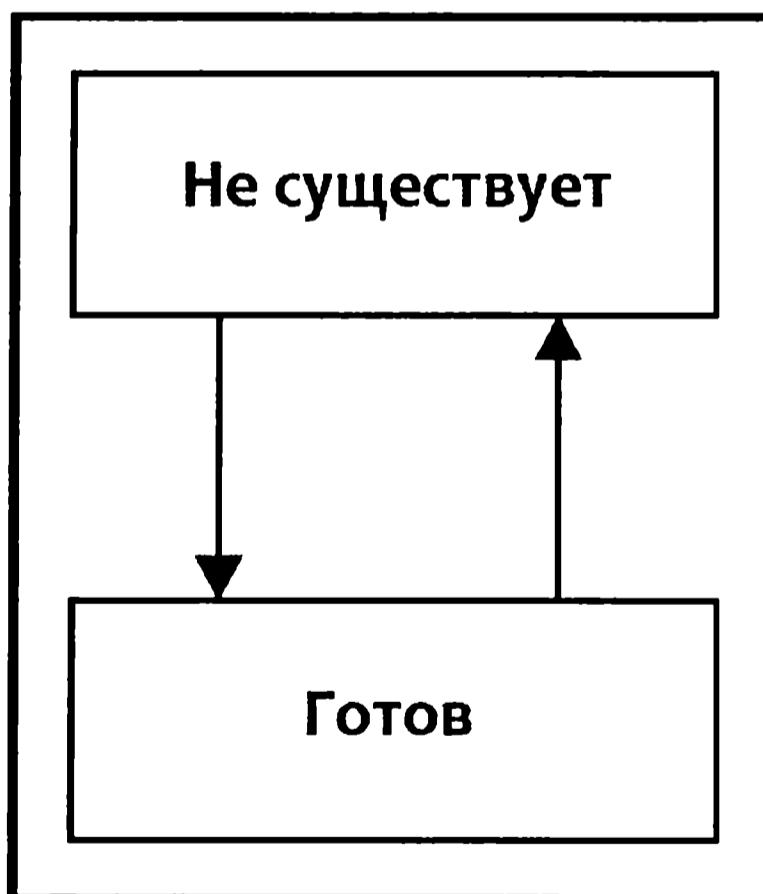
```
<? xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sun-ejb-jar PUBLIC "-//Sun Microsystems,
    Inc.//DTD Application Server 9.0 EJB 3.0//EN"
    "http://www.sun.com/software/appserver/dtds/sun-ejb-jar_3_0-0.dtd">
<sun-ejb-jar>
    <enterprise-beans>
        <ejb>
            <ejb-name>MyStatelessSessionBean</ejb-name>
            <bean-pool>
                <steady-pool-size>10</steady-pool-size>
                <max-pool-size>60</max-pool-size>
            </bean-pool>
        </ejb>
    </enterprise-beans>
</sun-ejb-jar>
```

```
<resize-quantity>5</resize-quantity>
<pool-idle-timeout-in-seconds>
    900
</pool-idle-timeout-in-seconds>
</bean-pool>
</ejb>
</enterprise-beans>
</sun-ejb-jar>
```

- Элемент `<steady-pool-size>` соответствует **Начальному и минимальному размеру пула** (Initial and Minimum Pool Size) в веб-консоли GlassFish;
- Элемент `<max-pool-size>` соответствует **Максимальному размеру пула** (Maximum Pool Size) в веб-консоли сервера GlassFis№
- Элемент `<resize-quantity>` соответствует **Величине изменения размера пула** (Pool Resize Quantity) в веб-консоли сервера GlassFish;
- Элемент `<pool-idle-timeout-in-seconds>` соответствует **Величина тайм-аута простоя пула** (Pool Idle Timeout) в веб-консоли сервера GlassFish.

Жизненный цикл управляемых сообщением бинов

Как и сеансовые бины, не сохраняющие состояние, управляемые сообщением бины содержат только два состояния – **Не существует** (Does Not Exist) и **Готов** (Ready):



Управляемый сообщением бин может иметь методы, декорированные аннотациями `@PostConstruct` и `@PreDestroy`. Методы, декорированные аннотацией `@PostConstruct`, выполняются непосредственно перед тем, как бин перейдет в состояние **Готов**. Методы, декорированные аннотацией `@PreDestroy`, выполняются непосредственно перед тем, как бин перейдет в состояние **Не существует**.

Служба таймера EJB

У сеансовых бинов, не сохраняющих состояние, и у управляемых сообщением бинов может быть метод, который должен выполняться через регулярные промежутки времени. Это можно реализовать при помощи службы таймера EJB (EJB timer service). Следующий пример поясняет, как использовать преимущества этой службы:

```
package net.ensode.glassfishbook;

import java.io.Serializable;
import java.util.Collection;
import java.util.Date;
import java.util.logging.Logger;
import javax.annotation.Resource;
import javax.ejb.EJBContext;
import javax.ejb.Stateless;
import javax.ejb.Timeout;
import javax.ejb.Timer;
import javax.ejb.TimerService;
@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("Это сообщение было инициировано: " + timer.getInfo()
            + " в " + System.currentTimeMillis());
    }
}
```

Здесь мы инжектируем реализацию интерфейса `javax.ejb.TimerService`, декорируя переменную экземпляра этого типа аннотацией `@Resource`. Затем мы можем создать таймер, вызывая метод `createTimer()` этого экземпляра `TimerService`.

Имеется несколько перегруженных версий метода `createTimer()`. Та версия метода, которую мы хотим использовать, принимает экземпляр `java.util.Date` в качестве его первого параметра. Этот параметр используется, чтобы указать первый

раз продолжительность действия таймера (до его отключения). В примере мы хотим использовать совершенно новый экземпляр класса Date, который в действительности заставляет таймер остановиться сразу. Вторым параметром метода `createTimer()` является продолжительность времени ожидания (в миллисекундах), прежде чем таймер остановится снова. В предыдущем примере время таймера истекает каждые пять секунд. Третий параметр метода `createTimer()` может быть экземпляром любого класса, реализующего интерфейс `java.io.Serializable`. Поскольку у одного EJB может быть несколько таймеров, выполняемых одновременно, этот третий параметр используется, чтобы однозначно определить каждый из таймеров. Если нам не нужно идентифицировать таймеры, можно передать `null` в качестве значения этого параметра.



Вызов EJB-метода `TimerService.createTimer()` должен производиться из клиента EJB. Размещение вызова этого метода в EJB, декорированном аннотацией `@PostConstruct` для автоматического запуска таймера, при переходе бина в состояние **Готов** приведет к возникновению исключения `IllegalStateException`.

Мы можем остановить таймер, вызывая его метод `cancel()`. Не существует способа прямого получения одного конкретного таймера, связанного с EJB. Для этого мы должны вызвать метод `getTimers()` на экземпляре `TimerService`, который связан с EJB. Этот метод вернет коллекцию, содержащую все таймеры, связанные с EJB. Затем можно выполнить итерацию по этой коллекции и корректную отмену конкретного таймера, вызывая его метод `getInfo()`. Он вернет сериализуемый объект, который мы передаем в качестве параметра в метод `createTimer()`.

Наконец, любой метод EJB, декорированный аннотацией `@Timeout`, будет выполняться, когда время таймера истечет. Методы, декорированные этой аннотацией, должны возвращать `void` и принимать единственный параметр типа `javax.ejb.Timer`. В нашем примере метод просто пишет сообщение в журнал сервера.

Следующий класс является автономным клиентом 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("Запущен timer 1...");
            ejbTimerExample.startTimer("Timer 1");
            System.out.println("Приостановлен на 2 секунды ...");
            Thread.sleep(2000);
            System.out.println("Запущен timer 2...");
            ejbTimerExample.startTimer("Timer 2");
            System.out.println("Приостановлен на 30 секунд ...");
            Thread.sleep(30000);
            System.out.println("Остановлен timer 1 ...");
            ejbTimerExample.stopTimer("Timer 1");
            System.out.println("Остановлен timer 2 ...");
            ejbTimerExample.stopTimer("Timer 2");
        }
    }
}
```



```
        System.out.println("Выполнено.");
    }
    catch (InterruptedException e)
    {
        e.printStackTrace();
    }
}
```

Пример просто запускает таймер, после чего ожидает в течение нескольких секунд, а затем запускает второй таймер. Затем он бездействует в течение 30 секунд и останавливает оба таймера. После развертывания EJB и выполнения клиента мы должны будем увидеть в журнале сервера примерно такой ряд записей:

```
[#|2007-05-05T20:41:39.518-0400|INFO|sunappserver9.1|
net.ensode.glassfishbook.EjbTimerExampleBean|_ThreadID=22;_
ThreadName=p: thread-pool-1; w: 16;|This message was triggered by
:Timer 1 at 1178412099518|#]

[#|2007-05-05T20:41:41.536-0400|INFO|sunappserver9.1|
net.ensode.glassfishbook.EjbTimerExampleBean|_ThreadID=22;_
ThreadName=p: thread-pool-1; w: 16;|This message was triggered by
:Timer 2 at 1178412101536|#]

[#|2007-05-05T20:41:46.537-0400|INFO|sunappserver9.1|
net.ensode.glassfishbook.EjbTimerExampleBean|_ThreadID=22;_
ThreadName=p: thread-pool-1; w: 16;|This message was triggered by
:Timer 1 at 1178412106537|#]

[#|2007-05-05T20:41:48.556-0400|INFO|sunappserver9.1|
net.ensode.glassfishbook.EjbTimerExampleBean|_ThreadID=22;_
ThreadName=p: thread-pool-1; w: 16;|This message was triggered by
:Timer 2 at 1178412108556|#]
```

Эти записи создаются каждый раз, когда время одного из таймеров истекает.

Выражения таймера EJB на основе календаря

У примера из предыдущего раздела имеется один недостаток. Метод `startTimer()` в сеансовом бине для запуска таймера должен быть вызван из клиента. Это ограничение мешает сделать так, чтобы таймер запустился, как только бин будет развернут.

Java EE 6 вводит выражения таймера EJB на основе календаря. Выражение на основе календаря позволяет одному или более методам наших сеансовых бинов выполнятьсь в определенные дни и в определенное время. Например, мы можем сконфигурировать один из наших методов, который будет выполняться каждую ночь 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("Это сообщение сгенерировано в: " + System.
currentTimeMillis());
    }
}

```

Как видно из этого примера, мы задаем значение времени, в которое метод будет выполняться, с помощью аннотации `javax.ejb.Schedule`. Мы настроили наш метод на выполнение в 20:10, установив атрибут `hour` аннотации `@Schedule` в значение "20" и ее атрибут `minute` – в значение "10" (значение атрибута `hour` выставляется в 24-часовом формате).

У аннотации `@Schedule` имеется несколько других атрибутов, которые предоставляют большую гибкость в указании времени выполнения метода, например: в третью пятницу каждого месяца, в последний день месяца и т. д.

Следующая таблица перечисляет все атрибуты аннотации `@Schedule`, которые позволяют нам управлять тем, когда аннотируемый метод будет выполняться:

Атрибут	Описание	Примеры значений	Значение по умолчанию
dayOfMonth	День месяца	"3": третий день месяца. "Last": последний день месяца. "-2": за два дня до конца месяца. "1st Tue": первый вторник месяца.	"*"
dayOfWeek	День недели	"3": каждая среда. "Thu": каждый четверг.	"*"
hour	Час дня (24-часовое основание)	"14": 2:00 pm.	"0"
minute	Минута часа	"10": через десять минут после начала часа.	"0"
month	Месяц года	"2": февраль. "March": март.	"*"
second	Секунда минуты	"5": через пять секунд после начала минуты.	"0"
timezone	Идентификатор часового пояса	Америка/Нью-Йорк.	""
year	Четыре цифры года	"2010"	"*"

В дополнение к одиночным значениям большинство атрибутов принимает звездочку ("*") как подстановочный символ, означающий, что аннотируемый метод будет выполняться каждую единицу времени (каждый день, час и т. д.).

У нас есть возможность указать несколько значений, разделяя их запятыми. Например, если нам нужен метод, который будет выполняться каждый вторник и четверг, мы можем аннотировать метод следующим образом:

```
@Schedule(dayOfWeek="Tue, Thu").
```

Допускается и указание диапазона значений, в котором первое и последнее значения разделяются знаком дефиса (-). Так, для того чтобы метод выполнялся с понедельника по пятницу, мы можем использовать запись `@Schedule(dayOfWeek="Mon-Fri")`.

Кроме того, мы можем указать, что нуждаемся в методе, который будет выполняться каждую «п»-ную единицу времени (например, каждый день, каждые два часа, каждые десять минут и т. д.). Для подобных целей мы можем использовать запись в формате `@Schedule(hour="*/12")`: это значит, что метод будет выполняться каждые 12 часов.

Как мы видим, аннотация `@Schedule` предоставляет большую гибкость для указания времени выполнения наших методов. Еще один ее плюс заключается в том, что отпадает необходимость в клиентском вызове для активации выполнения. Наконец, данная аннотация имеет преимущество использования хроноподобного синтаксиса. Разработчики, знакомые с этим инструментом из Unix, не будут чувствовать ни малейших затруднений при использовании `@Schedule`.

Безопасность EJB

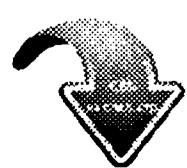
Enterprise JavaBeans позволяют нам декларативно решать, какие пользователи могут получить доступ к их методам. Например, некоторые методы могут быть доступны только пользователям в определенных ролях. Согласно самому распространенному сценарию, только пользователи с ролью администратора могут добавлять, удалять или редактировать информацию о других пользователях в системе.

Следующий пример представляет немного модифицированную версию сеансового бина DAO, который мы рассматривали выше в этой главе. В этой версии некоторые методы, ранее являвшиеся частными (`private`), были сделаны открытыми (`public`). Кроме того, сеансовый бин модифицирован таким образом, чтобы только пользователи с определенными ролями могли получать доступ к его методам.

```
package net.ensode.glassfishbook;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.annotation.Resource;
import javax.annotation.security.RolesAllowed;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
```

```
import javax.persistence.PersistenceContext;
import javax.sql.DataSource;
@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)
    {
        customer.setCustomerId(getNewCustomerId());
        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);
    }
    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;
}
}
```

Как видно из приведенного кода, мы объявляем, у каких ролей имеется доступ к методам, при помощи аннотации @RolesAllowed. Она может принимать в качестве параметра или одну строку, или массив строк. Когда в качестве параметра для этой аннотации используется одна строка, доступ к методу могут получить только пользователи с ролью, указанной параметром. Если же в качестве параметра используется массив строк, то доступ к методу разрешается для пользователей с любой из ролей, указанных элементами массива.

Аннотация @RolesAllowed может использоваться для декорирования класса EJB, когда ее значения применяются ко всем методам в EJB или к одному либо нескольким методам. В последнем случае ее значения применяются только к методу, который декорируется аннотацией. Если, как в нашем примере, и класс EJB, и один или несколько его методов декорируются аннотацией @RolesAllowed, аннотация на уровне метода имеет приоритет.

Роли приложения должны быть отображены на название группы области безопасности. Само это отображение, а также то, какую область при этом использовать, устанавливаются в дескрипторе развертывания sun-ejb-jar.xml:

```
<? xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sun-ejb-jar PUBLIC "-//Sun Microsystems,
  Inc./DTD Application Server 9.0 EJB 3.0//EN"
  "http://www.sun.com/software/appserver/dtds/sun-ejb-jar_3_0-0.dtd">
<sun-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>
</sun-ejb-jar>
```

Элемент <security-role-mapping> дескриптора развертывания sun-ejb-jar.xml выполняет отображение междуолями 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 мы отображаем две роли приложения на соответствующие группы в области безопасности. Хотя в данном конкретном примере имя роли приложения и группы безопасности совпадают, на практике этого не должно быть.



Автоматическое соответствие ролей группам безопасности

Можно установить автоматическое соответствие любых ролей приложения, именам групп безопасности в области безопасности. Это можно выполнить, войдя в веб-консоль GlassFish, последовательно щелкнув по узлу **Конфигурация** (Configuration), по узлу **Безопасность** (Security) и по флагу **Участник по умолчанию для ролевого отображения** (Default Principal to Role Mapping) и сохранив это изменение конфигурации.

Как видно из приведенного примера, область безопасности для использования аутентификации определяется в подэлементе <realm> элемента <as-context>. Значение этого подэлемента должно соответствовать имени допустимой области безопасности на сервере приложений. Другими подэлементами элемента <as-context> являются: <auth-method>, допустимым значением которого может быть только username_password, и <required>, допустимым значением которого в свою очередь могут быть только true и false.

Аутентификация клиента

Если клиентский код, получающий доступ к защищенному EJB, является частью веб-приложения, в котором пользователь уже прошел аутентификацию, то учетные данные пользователя будут использоваться для определения того, нужно ли разрешать пользователю доступ к методу, который он (или она) пытается выполнить.

Автономные клиенты должны быть выполнены через утилиту appclient. Следующий код показывает типичного клиента для предыдущего защищенного сеансового бина:

```
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("Сохранение нового заказчика...");
        newCustomerId = customerDao.saveNewCustomer(customer);
        System.out.println("Извлечение заказчика...");
        customer = customerDao.getCustomer(newCustomerId);
        System.out.println(customer);
    }
}
```

Как видно из приведенного кода, он ничего не делает для аутентификации пользователя. Сеансовый бин просто инжектируется в код через аннотацию @EJB и используется как обычно. А вот утилита appclient заботится об аутентификации пользователя, передавая аргументы -user и -password с предусмотренными значениями для аутентификации:

```
appclient -client ejbsecurityclient.jar -user peter -password secret
```

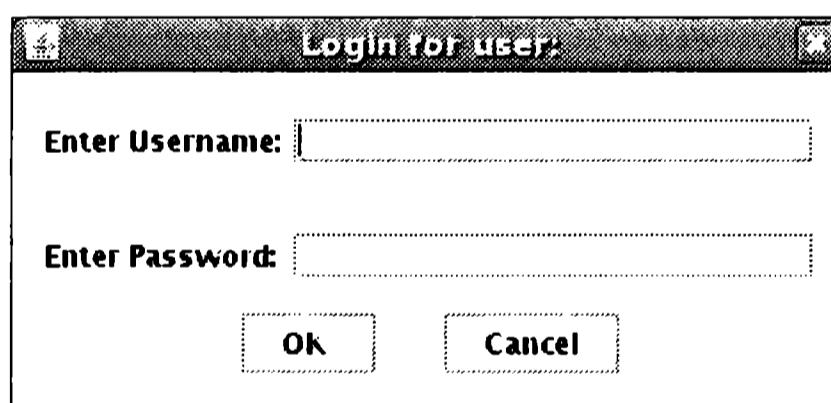
Эта команда будет аутентифицировать пользователя с именем peter и паролем secret. Если предположить, что учетные данные верны и что пользователь имеет соответствующие полномочия, код EJB выполнится, и мы должны будем увидеть ожидаемый вывод из предыдущего класса Client:

```
Сохранение нового заказчика...
Извлечение заказчика...
customerId = 29
firstName = Mark
lastName = Butcher
email = butcher@phony.org
```

Если мы не введем имя пользователя и пароль в командной строке, то appclient запросит у нас имя пользователя и пароль через диалоговое окно. В нашем примере – при вводе следующей команды:

```
appclient -client ejbsecurityclient.jar
```

Появится всплывающее диалоговое окно наподобие показанного ниже:



Мы можем просто ввести наше имя пользователя и пароль в соответствующих полях. После проверки допустимости учетных данных приложение выполнится, как ожидалось.

Резюме

В этой главе мы рассмотрели, как реализовать бизнес-логику с помощью сеансовых бинов, сохраняющих и не сохраняющих состояние. Мы объяснили, как использовать преимущества транзакционной природы EJB для упрощения реализации шаблона проектирования – Объект доступа к данным (DAO).

Дополнительно мы обсудили понятие транзакций, управляемых контейнером, и показали, как ими управлять при помощи соответствующих аннотаций. Мы также объяснили, как реализовать транзакции, управляемые бином, на случай, если транз-

акций, управляемых контейнером, оказывается недостаточно для удовлетворения наших требований.

Были рассмотрены жизненные циклы для различных типов Enterprise JavaBeans; в частности, было показано, как автоматически вызвать методы EJB контейнером EJB в определенные моменты жизненного цикла.

Также мы рассмотрели, как периодически вызывать методы EJB с помощью контейнера EJB, используя возможности службы таймера EJB.

Наконец, мы узнали, как убедиться в том, что методы EJB будут вызваны только авторизованными пользователями, аннотируя классы EJB и/или методы и добавляя соответствующие записи в файл дескриптора развертывания `sun-ejb-jar.xml`.

10

Контексты и инжекция зависимости

Контексты и инжекция зависимости (Contexts and Dependency Injection (CDI)) – новое дополнение к спецификации Java EE начиная с Java EE 6. Оно предоставляет несколько возможностей, которых ранее не имелось в распоряжении разработчиков Java EE – например, возможность использовать любой JavaBean в качестве управляемого JSF-бина, включая и сеансовые бины с сохранением состояния, и сеансовые бины без сохранения состояния. Как следует из наименования, CDI упрощает инжекцию зависимости в приложениях Java EE.

В этой главе мы затронем следующие темы:

- именованные бины;
- инжекция зависимости;
- контексты;
- квалификаторы.

Именованные бины

CDI предоставляет нам возможность именовать бины через аннотацию `@Named`. Именованные бины позволяют нам легко инжектировать наши бины в другие классы, которые от них зависят (см. следующий раздел) и легко 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 видно из приведенного кода, для именования наших классов нужно всего лишь декорировать их аннотацией @Named. По умолчанию имя бина идентично имени класса (не считая того, что первая буква набрана в нижнем регистре). В нашем примере имя бина было бы customer. Если мы хотим использовать другое имя, с этой целью можно установить атрибут value аннотации @Named. Например, если мы решили назначить нашему предыдущему бину имя customerBean, потребуется изменить аннотацию @Named следующим образом:

```
@Named(value="customerBean")
```

или просто:

```
@Named ("customerBean")
```

Поскольку в названии не обязательно использовать атрибут value, то, если мы просто указываем название, – value подразумевается.

Данное имя может применяться для получения доступа к нашему бину из JSF-страницы с использованием Унифицированного языка выражений.

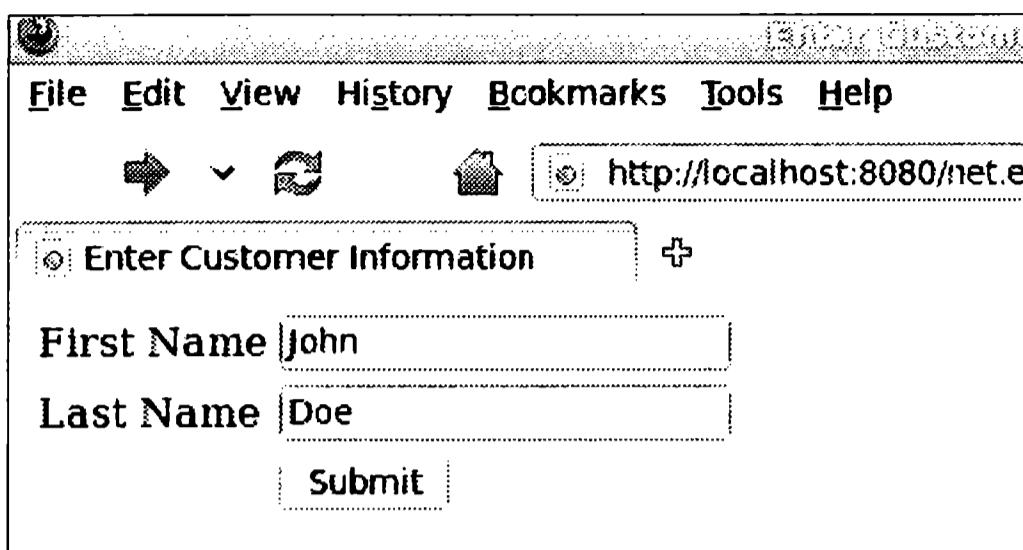
```

<? 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>Ввод сведений о заказчике</title>
</h:head>
<h:body>
    <h:form>
        <h:panelGrid columns="2">
            <h:outputLabel for="firstName" value="Имя"/>
            <h:inputText id="firstName" value="#{customer.firstName}" />
            <h:outputLabel for="lastName" value="Фамилия"/>
            <h:inputText id="lastName" value="#{customer.lastName}" />
        <h:panelGroup/>
    </h:panelGrid>
</h:form>
</h:body>
</html>

```

Как мы видим из этого кода, к именованным бинам можно получить доступ из JSF-страницы так же, как к управляемым бинам стандартного JSF. Это позволяет JSF получать доступ к любому именованному бину, отвязывая код Java от API JSF.

Когда наше простое приложение развернуто и выполняется, оно выглядит следующим образом:



Приложения CDI должны включать конфигурационный файл `beans.xml`: его наличие сообщает серверу приложений о необходимости активировать CDI для приложения. Файл может быть пустым – тем не менее он должен существовать. Как правило, он выглядит следующим образом:

```
<? xml version="1.0" encoding="UTF-8" ?>
<beans 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/beans_1_0.xsd">
</beans>
```

В веб-приложениях этот файл должен быть помещен в каталог `WEB-INF` в WAR-файле приложения. В файлах JAR EJB файл `beans.xml` должен быть помещен в каталог `META-INF`.

Инжекция зависимости

Инжекция зависимости (*dependency injection*) является технологией поддержки внешних зависимостей для класса Java. Java EE 5 вводит понятие инжекции зависимости через аннотацию `@Resource`. Однако эта аннотация ограничивается инжекцией ресурсов, таких как соединения с базой данных, ресурсы JMS и т. д. Java EE 6 вводит аннотацию `@Inject`, которая может использоваться для инжектирования экземпляров классов Java в любые зависящие от них объекты.

Приложения JSF обычно следуют шаблону проектирования «Модель-Представление-Контроллер» (Model-View-Controller (MVC)). В связи с этим некоторые управляемые бины JSF зачастую берут на себя роль контроллеров в шаблоне, между тем как другие берут на себя роль модели. Такой подход обычно необходим, когда управляемый бин, выступающий в роли контроллера, должен иметь доступ к одному или более управляемым бинам, выступающим в роли модели.

Из-за шаблона, описанного в предыдущем абзаце, один из наиболее часто задаваемых вопросов по JSF звучит следующим образом: «Как получить доступ к управляемому бину из другого управляемого бина?». Существует несколько способов сделать это, но до CDI ни один способ нельзя было назвать простым. Самый легкий

способ состоял в том, чтобы объявить управляемое свойство в управляемом бине, играющем роль контроллера. Это требовало изменения конфигурационного файла приложения faces-config.xml. Другой подход предусматривал использование кода наподобие следующего:

```
ELContext elc = FacesContext.getCurrentInstance().  
    getELContext();  
SomeBean someBean = (SomeBean) FacesContext.  
    getInstance().getApplication().getELResolver().  
    getValue(elc, null, "someBean");
```

Здесь someBean – имя бина, которое указано в конфигурационном файле приложения faces-config.xml. Как мы видим, ни один из описанных подходов не является простым или, по крайней мере, запоминающимся. К счастью, в Java EE 6 уже нет необходимости в коде, подобном вышеописанному, благодаря возможностям инъекции зависимости CDI.

```
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("Сохраняет следующую информацию \n" +  
            customer.toString());  
        // Если бы это было реальное приложение, здесь был бы код  
        // выполняющий сохранение данных customer в базе данных.  
        return "confirmation";  
    }  
}
```

Обратите внимание, что для инициализации нашего экземпляра customer нам нужно всего лишь декорировать его аннотацией @Inject. Когда бин создается сервером приложений, экземпляр бина Customer автоматически инжектируется в это поле. Заметьте, что инжектированный бин используется в методе saveCustomer(). Как мы видим, CDI ускоряет доступ к одному бину из другого бина, что существенно отличает его применение от вариантов кода, который мы должны были использовать в предыдущих версиях спецификации Java EE.

Квалификаторы

В некоторых случаях тип бина, который мы хотим инжектировать в наш код, может быть интерфейсом или суперклассом Java, но нас может интересовать инъектирование подкласса этого суперкласса либо класса, реализующего интерфейс. Для таких случаев CDI предоставляет квалификаторы, которые мы можем использовать

для указания конкретного типа, который мы хотим инжектировать в наш код.

Квалификатор CDI является аннотацией, которая в свою очередь должна быть декорирована аннотацией `@Qualifier`. Эта аннотация затем может использоваться для декорирования конкретного подкласса или реализации интерфейса, которую мы хотим квалифицировать. Кроме того, поле инжекции в коде клиента также должно быть декорировано квалификатором.

Предположим, что наше приложение может иметь особую разновидность типа заказчика, когда постоянным клиентам может быть присвоен статус «премиум-заказчиков» (клиентов). Чтобы обработать этих премиум-заказчиков, мы можем расширить наш бин `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 мы упоминали ранее, квалификаторы являются стандартными аннотациями. Обычно они сохраняются исполняющей средой и могут предназначаться для методов, полей, параметров или типов, как было показано в предыдущем примере. Единственная разница между квалификатором и стандартной аннотацией состоит в том, что квалификаторы декорируются аннотацией `@Qualifier`.

После того как мы получим наш квалификатор, нам следует использовать его для декорирования конкретного подкласса или реализации интерфейса:

```
package net.ensode.cdidependencyinjection.beans;

import javax.inject.Named;
import net.ensode.cdidependencyinjection.qualifiers.Premium;
@Named
@Premium
public class PremiumCustomer extends Customer
{
    private Integer discountCode;
    public Integer getDiscountCode()
    {
        return discountCode;
    }
    public void setDiscountCode(Integer discountCode)
    {
        this.discountCode = discountCode;
    }
}
```

После того как мы декорировали квалификатором конкретный экземпляр, можно использовать наш квалификатор в клиентском коде, чтобы указать точный тип нужной нам 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("Сохраняет следующую информацию \n"
            + premiumCustomer.getFirstName() + " "
            + premiumCustomer.getLastName() + ", Код скидки = "
            + premiumCustomer.getDiscountCode());
        // Если бы это было реальное приложение, здесь нам понадобился бы код
        // для сохранения информации о заказчике в базу данных.
        return "confirmation";
    }
    public Integer generateDiscountCode()
    {
        return new Random().nextInt(100000);
    }
}
```

Мы использовали квалификатор `@Premium`, чтобы декорировать поле заказчика, в которое инжектируется экземпляр `PremiumCustomer`, поскольку этот класс также декорируется квалификатором `@Premium`.

Что касается перехода к JSF-страницам, мы просто получаем доступ к нашему именованному бину, 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>Ввод сведений о заказчике</title>
    </h:head>
    <h:body>
        <h:form>
            <h:panelGrid columns="2">
                <h:outputLabel for="firstName" value="Имя"/>
                <h:inputText id="firstName"
                            value="#{premiumCustomer.firstName}"/>
```

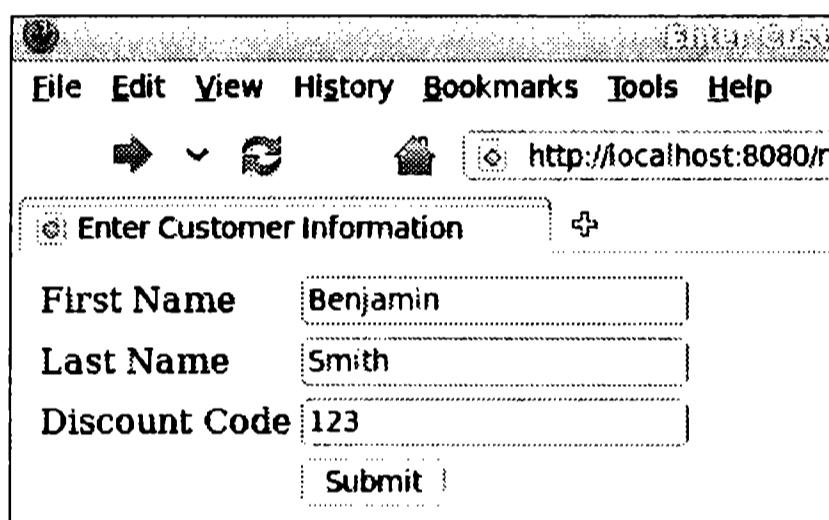


```

<h:outputLabel for="lastName" value="Фамилия"/>
<h:inputText id="lastName"
              value="#{premiumCustomer.lastName}"/>
<h:outputLabel for="discountCode" value="Код скидки"/>
<h:inputText id="discountCode"
              value="#{premiumCustomer.discountCode}"/>
<h:panelGroup/>
<h:commandButton value="Отправить"
                  action="#{customerController.saveCustomer}"/>
</h:panelGrid>
</h:form>
</h:body>
</html>

```

В этом примере мы используем имя по умолчанию для нашего бина, которое идентично имени класса (не считая того, что первая буква записана в нижнем регистре). Наше простое приложение по отношению к пользователю отображается и функционирует точно так же, как и «обычное» приложение JSF.



Контексты именованных бинов

Как и управляемые бины JSF, именованные бины CDI определяются (создаются) в некотором контексте. Это означает, что они являются контекстно зависимыми объектами. Когда необходим именованный бин – или из-за инжекции, или потому, что на него имеется ссылка из JSF-страницы, – CDI ищет экземпляр бина в контексте, которому он принадлежит, и инжектирует его в зависимый код. Если ни один экземпляр не будет найден, он создается и сохраняется в соответствующем контексте для последующего использования. Различные контексты являются окружением, в котором существует бин.

В следующей таблице перечислены различные допустимые контексты CDI:

Контекст	Аннотация	Описание
Запрос (Request)	@RequestScoped	Бины, определенные в контексте запроса, подразделяются по продолжительности единичного запроса. Единичный запрос может представлять собой HTTP-запрос, вызов метода в EJB, вызов веб-сервиса или отправку сообщения JMS управляемому сообщением бину

Контекст	Аннотация	Описание
Переговоры (Conversation)	@ConversationScoped	Контекст переговоров может охватить несколько запросов, но он обычно короче, чем контекст сеанса
Сеанс (Session)	@SessionScoped	Бины, определенные в контексте сеанса, действительны для всех запросов в пределах HTTP-сеанса. Каждый пользователь приложения получает свой собственный экземпляр бина, определенного в контексте сеанса
Приложение (Application)	@ApplicationScoped	Бины, определенные в контексте приложения, действительны на протяжении всего времени жизни приложения. Бины в этом контексте совместно используются всеми сеансами пользователей
Зависимый (Dependent)	@Dependent	Бины, определенные в контексте зависимости, не используются совместно. Каждый раз, когда инжектируется бин зависимости, создается новый экземпляр

Как мы видим, CDI включает все контексты, поддерживаемые JSF, и помимо этого добавляет ряд своих собственных.

Контекст запроса (request scope) CDI отличается от контекста запроса JSF; в нем запрос не обязательно является HTTP-запросом. Он просто может быть вызовом метода на EJB, вызовом веб-сервиса или отправкой сообщения JMS управляемому сообщением бину.

Контекст переговоров (conversation scope) не существует в JSF. Этот контекст более продолжителен по времени, чем контекст запроса, но короче, чем сеанс. Обычно он охватывает три страницы и более. В классы, желающие получить доступ к бину, определенному в контексте переговоров, должен быть инжектирован экземпляр javax.enterprise.context.Conversation. В той точке кода, где мы хотим начать переговоры, нужно вызвать метод `begin()` на данном объекте. В точке кода, где мы хотим завершить переговоры, нужно вызвать метод `end()` на данном объекте.

Контекст сеанса (session scope) CDI ведет себя точно так же, как его «коллега» из JSF. Жизненный цикл бинов, определенных в этом контексте, связан со сроком жизни HTTP-сеанса.

Контекст приложения (application scope) CDI ведет себя так же, как эквивалентный контекст в JSF. Бины, определенные в контексте приложения, привязываются к сроку жизни приложения. Для приложения существует единственный экземпляр

каждого бина, определенного в контексте приложения. Это значит, что тот же самый экземпляр доступен для всех HTTP-сеансов.

Подобно контексту переговоров, *Контекст зависимости* (dependent scope) CDI не существует в JSF. Новый бин, определенный в контексте зависимости, создает экземпляр каждый раз, когда это необходимо, – обычно когда он инжектируется в класс, который от него зависит. Предположим, мы хотим, чтобы пользователь вводил некоторые данные, которые накапливались бы в единственном именованном бине. Однако у этого бина имеется несколько полей, поэтому мы хотели бы разделить ввод данных на несколько страниц. Это довольно распространенная ситуация, и с ней не так легко справиться, используя JSF или API Сервлета. Причина кроется в нетривиальности управления использованием этих технологий. Нетривиальность заключается в том, что мы можем поместить класс или в контекст запроса, в котором классы уничтожаются после каждого отдельного запроса и их данные теряются в процессе, или в контекст сеанса, в котором класс находится на одном месте в памяти еще долгое время после того, как необходимость в нем отпала. Для подобных случаев контекст переговоров CDI идеален:

```
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;
    public String getAddrCity()
    {
        return addrCity;
    }
    public void setAddrCity(String addrCity)
    {
        this.addrCity = addrCity;
    }
    public String getAddrLine1()
    {
        return addrLine1;
    }
    public void setAddrLine1(String addrLine1)
    {
        this.addrLine1 = addrLine1;
    }
    public String getAddrLine2()
    {
```

```
        return addrLine2;
    }
    public void setAddrLine2(String addrLine2)
    {
        this.addrLine2 = addrLine2;
    }
    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 String getMiddleName()
    {
        return middleName;
    }
    public void setMiddleName(String middleName)
    {
        this.middleName = middleName;
    }
    public String getPhoneHome()
    {
        return phoneHome;
    }
    public void setPhoneHome(String phoneHome)
    {
        this.phoneHome = phoneHome;
    }
    public String getPhoneMobile()
    {
        return phoneMobile;
    }
    public void setPhoneMobile(String phoneMobile)
    {
        this.phoneMobile = phoneMobile;
    }
    public String getPhoneWork()
    {
        return phoneWork;
    }
    public void setPhoneWork(String phoneWork)
    {
        this.phoneWork = phoneWork;
    }
    public String getState()
    {
        return state;
    }
    public void setState(String state)
    {
        this.state = state;
    }
```



```

public String getZip()
{
    return zip;
}
public void setZip(String zip)
{
    this.zip = zip;
}
@Override
public String toString()
{
    return ReflectionToStringBuilder.reflectionToString(this);
}
}

```

Здесь мы объявляем, что наш бин находится в контексте переговоров, декорируя его аннотацией `@ConversationScoped`. Бины, определяемые в контексте переговоров, также должны реализовывать интерфейс `java.io.Serializable`. Кроме реализации этих двух требований, в нашем коде нет ничего особенного. Он является простым JavaBean с закрытыми (`private`) свойствами и соответствующими им методами геттеров и сеттеров.



В коде мы пользуемся библиотекой `commons-lang` проекта Apache для того, чтобы облегчить реализацию метода `toString()` для нашего бина. Библиотека `commons-lang` имеет несколько служебных методов вроде этого, которые реализуют часто востребованную, но утомительную для кодирования функциональность. Библиотека `commons-lang` доступна в центральных репозитариях инструмента Maven и на веб-странице <http://commons.apache.org/lang>.

В дополнение к инжекции нашего бина, определенного в контексте переговоров, в наш клиентский код должен быть инжектирован экземпляр `javax.enterprise.context.Conversation`, как поясняется в следующем примере:

```

package net.ensode.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.ensode.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";
    }
}
```

Переговоры могут быть *длительными* (*long running*) или *скоротечными* (*transient*). Скоротечные переговоры заканчиваются в конце запроса, длительные же охватывают множество запросов. В большинстве случаев мы будем использовать длительные переговоры для хранения ссылки на бин, определенный в контексте переговоров, дляящихся в течение множества HTTP-запросов в веб-приложении.

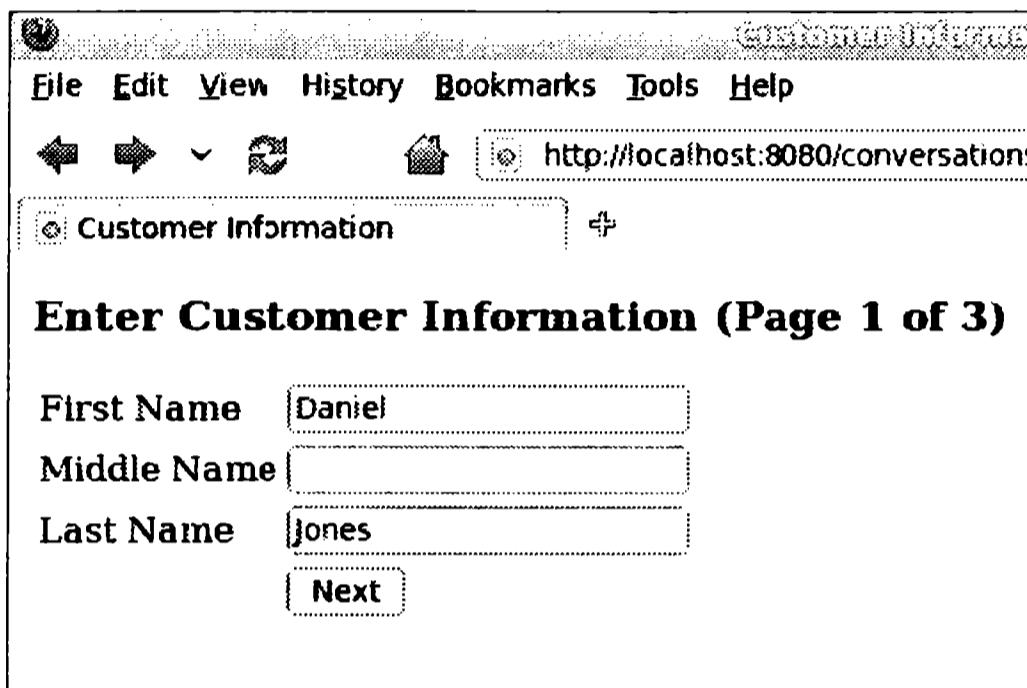
Длительные переговоры начинаются, когда вызывается метод `begin()` на инжектированном экземпляре `Conversation`, и заканчиваются, когда мы вызываем метод `end()` на том же самом объекте.

JSF-страницы получают доступ к нашим CDI-бинам обычным способом.

```
<?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>Информация о заказчике</title>
    </h:head>
    <h:body>
        <h3>Ввод сведений о заказчике (страница 1 из 3)</h3>
        <h:form>
            <h:panelGrid columns="2">
                <h:outputLabel for="firstName" value="Имя"/>
                <h:inputText id="firstName" value="#{customer.firstName}"/>
                <h:outputLabel for="middleName" value="Отчество"/>
                <h:inputText id="middleName" value="#{customer.middleName}"/>
                <h:outputLabel for="lastName" value="Фамилия"/>
                <h:inputText id="lastName" value="#{customer.lastName}"/>
                <h:panelGroup/>
                <h:commandButton value="Далее"
                                 action="#{customerInfoController.navigateToPage2}"/>
            </h:panelGrid>
        </h:form>
    </h:body>
</html>
```

Поскольку мы перемещаемся от одной страницы к другой, мы сохраняем один и тот же экземпляр нашего бина, определенного в контексте переговоров. Поэтому все вводимые пользователем данные остаются. Когда будет вызван метод `end()` на нашем бине переговоров, переговоры прекращаются и бин, определенный в контексте переговоров, уничтожается.

Сохранение наших данных в бине контекста переговоров значительно упрощает задачу реализации пользовательских интерфейсов в стиле «Мастер», где данные могут быть введены через несколько последовательных страниц.

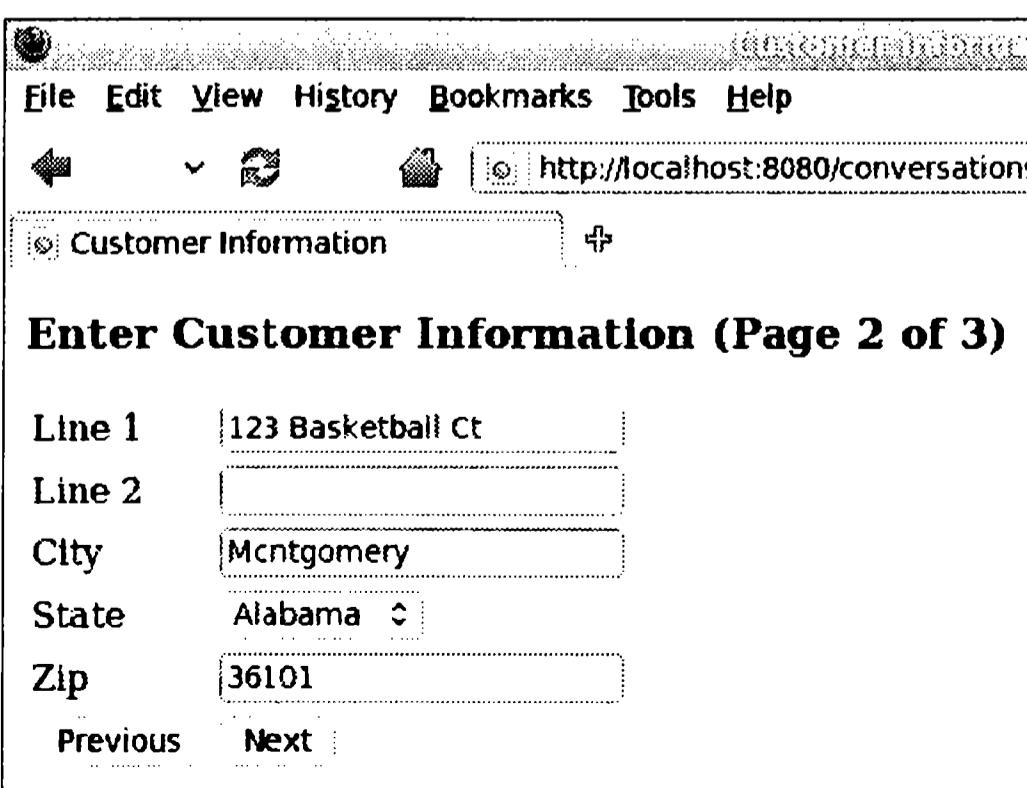


В нашем примере после щелчка по кнопке **Далее** (Next) на первой странице мы сможем увидеть частично заполненный бин в журнале сервера GlassFish:

INFO:

```
net.enode.conversationscope.model.Customer@6e1c51b4 [firstName=Daniel,
middleName=,lastName=Jones,addrLine1=,addrLine2=,addrCity=,state=AL,
zip=<null>,phoneHome=<null>,phoneWork=<null>,phoneMobile=<null>]
```

В этой точке выполнения кода на экран выводится вторая страница нашего простого мастера:



Щелкая по кнопке **Далее** (Next), мы сможем увидеть, что заполняются дополнительные свойства нашего бина, определенного в контексте переговоров:

INFO:

```
net.enseode.conversationscope.model.Customer@6e1c51b4 [firstName=Daniel,
middleName=,lastName=Jones,addrLine1=123 Basketball Ct,
addrLine2=,addrCity=Montgomery,state=AL,zip=36101,phoneHome=<null>,
phoneWork=<null>,phoneMobile=<null>]
```

При отправке третьей страницы нашего мастера (она здесь не показана) заполняются дополнительные свойства бина, соответствующие полям на этой странице.

Когда мы окажемся в точке кода, где больше не нужно сохранять в памяти информацию о клиентах, мы должны вызвать метод `end()` на бине переговоров, инжектированном в наш код. Именно это мы делаем в коде, перед тем как отобразить страницу подтверждения:

```
public String navigateToConfirmationPage()
{
    System.out.println(customer);
    conversation.end();
    return "confirmation";
}
```

После того как запрос на показ страницы подтверждения завершается, бин, определенный в контексте переговоров, уничтожается, когда мы вызываем метод `end()` на инжектированном классе `Conversation`.

Мы должны отметить, что поскольку контекст переговоров требует экземпляра `javax.enterprise.context.Conversation`, который будет инжектирован, этот контекст требует, чтобы действия кнопки или ссылки, используемые для навигации между страницами, были выражением, разрешаемым методом управляемого бина. Использование статической навигации со свойством по умолчанию, введенным в JSF 2.0 (при котором значение действия по умолчанию "foo" переместит нас к странице с названием `foo.xml`), не будет работать, поскольку экземпляр `Conversation` нигде не будет инжектирован.

Резюме

В этой главе мы представили введение в Контексты и Инжекцию зависимости (CDI). Мы рассмотрели, каким образом JSF-страницы могут получить доступ к именованному бину CDI, при котором для них это обращение ничем не будет отличаться от обращения к управляемому бину JSF. Было показано, как CDI облегчает инжекцию зависимости в наш код с помощью аннотации `@Inject`. Также мы объяснили, как можно использовать квалификаторы для определения того, какая конкретно реализация зависимости инжектируется в код. Наконец, мы обсудили все контексты, в которые может быть помещен бин CDI. Они включают эквиваленты всех контекстов JSF плюс еще два, которые не включены в JSF, а именно контекст переговоров и контекст зависимости.

11

Веб-сервисы JAX-WS

Спецификация Java EE 6 включает в себя API JAX-WS в качестве одной из своих технологий. JAX-WS – стандартный способ разработки веб-сервисов в нотации *Простого протокола доступа к объектам* (Simple Object Access Protocol (SOAP)) на платформе Java. Аббревиатура JAX-WS расшифровывается как «Java API for XML-Based Web Services» (API Java для веб-сервисов XML). JAX-WS является высокоуровневым API; вызов веб-сервисов с помощью JAX-WS выполняется через вызовы удаленных процедур. JAX-WS является очень естественным API для разработчиков Java.

Веб-сервисы (web services) – это прикладные программные интерфейсы (application programming interfaces (API)), которые могут быть вызваны удаленно. Веб-сервисы можно вызывать из клиентов, написанных на любом языке программирования.

В этой главе мы рассмотрим следующие темы:

- разработка веб-сервисов с помощью API JAX-WS;
- разработка клиентов веб-сервиса с помощью API JAX-WS;
- добавление вложений в вызовы веб-сервиса;
- представление EJB как веб-сервисов;
- обеспечение безопасности веб-сервисов.

Разработка веб-сервисов JAX-WS

JAX-WS является высокоуровневым API, который упрощает разработку веб-сервисов. JAX-WS представляет собой API Java для веб-сервисов, основанных на XML. Разработка веб-сервиса JAX-WS заключается в написании класса с открытыми методами, которые будут представлены как веб-сервисы. Класс должен быть декорирован аннотацией @WebService. Все открытые методы в классе автоматически представляются как веб-сервисы; они могут быть дополнительно декорированы аннотацией @WebService. Следующий пример поясняет 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 представляет два своих метода как веб-сервисы. Метод `add()` просто суммирует два числа типа `int`, которые он получает в качестве параметров, и возвращает результат; метод `subtract()` вычитает два своих параметра и возвращает результат.

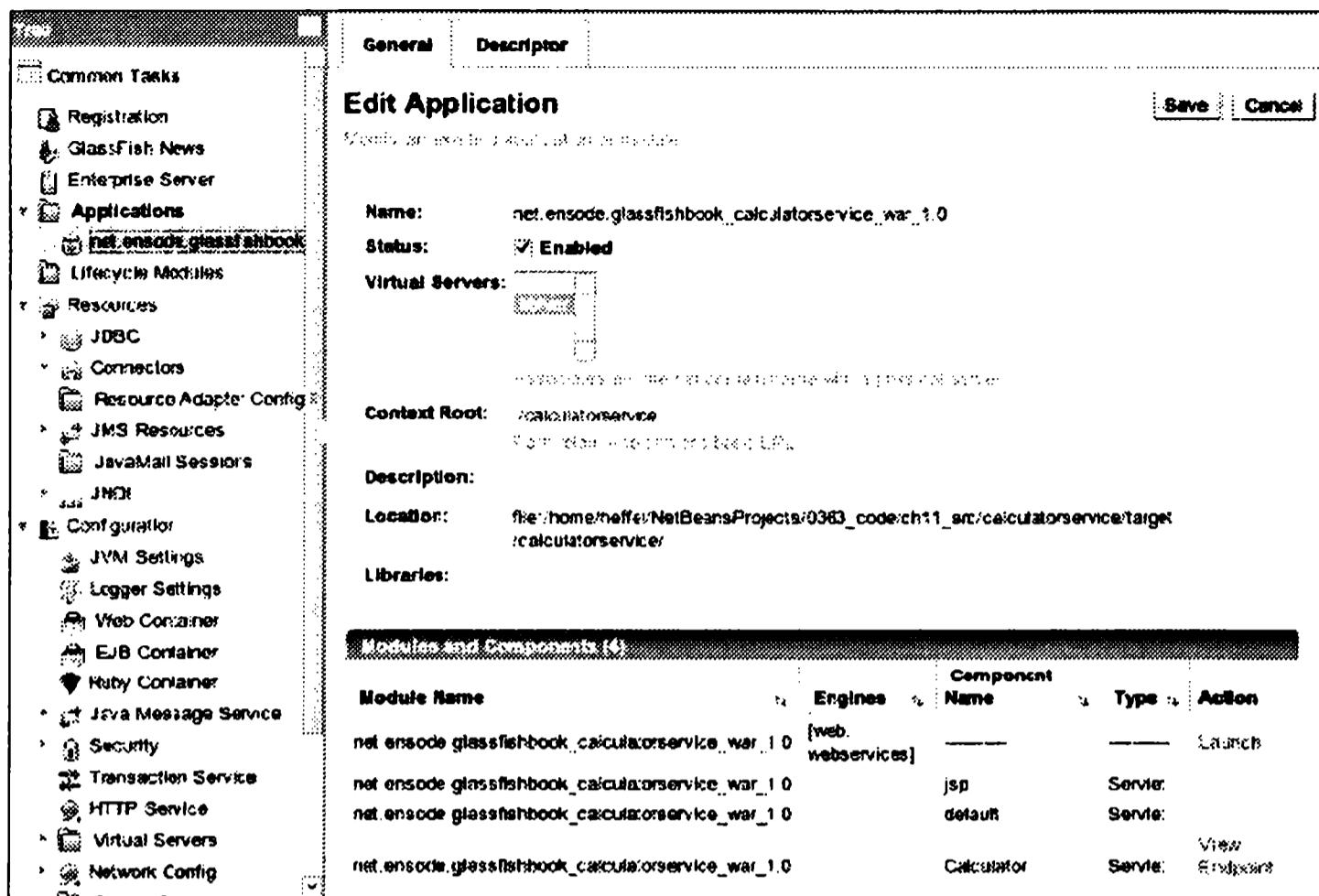
Мы указываем, что класс реализует веб-сервис, декорируя его аннотацией `@WebService`. Любые методы, которые мы хотели бы представить как веб-сервисы, могут быть декорированы аннотацией `@WebMethod`, но это не обязательно; все открытые методы автоматически представляются как веб-сервисы.

Чтобы развернуть наш веб-сервис, мы должны упаковать его в WAR-файл. До выпуска Java EE 6 все правильно построенные WAR-файлы обязаны были содержать дескриптор развертывания `web.xml` в каталоге `WEB-INF`. Как мы уже объясняли в предыдущих главах, этот дескриптор развертывания не обязателен при работе с Java EE 6 и не требуется для развертывания веб-сервисов в этой среде.

Если мы хотим добавить дескриптор развертывания `web.xml`, не нужно ничего добавлять в файл дескриптора развертывания `web.xml` WAR-файла. Чтобы успешно развернуть наш веб-сервис, достаточно просто иметь пустой элемент `<web-app>` в дескрипторе развертывания для успешного развертывания нашего WAR-файла.

```
<? 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 и упаковки этого кода и дескриптора развертывания в WAR-файле и последующего его развертывания следует убедиться, что он был развернут успешно. Для этого нужно войти в веб-консоль администрирования сервера GlassFish и развернуть узел **Приложения** (Applications) в панели навигации в левой части страницы. Только что развернутый веб-сервис должен быть представлен в списке этого узла:



Обратите внимание на ссылку **Представление Конечной точки (View Endpoint)** в правой нижней части экрана на основной панели страницы. Щелчок по этой ссылке приведет нас к странице **Информация о Конечной точке веб-сервиса (Web Service Endpoint Information)**, которая содержит некоторую информацию о нашем веб-сервисе.

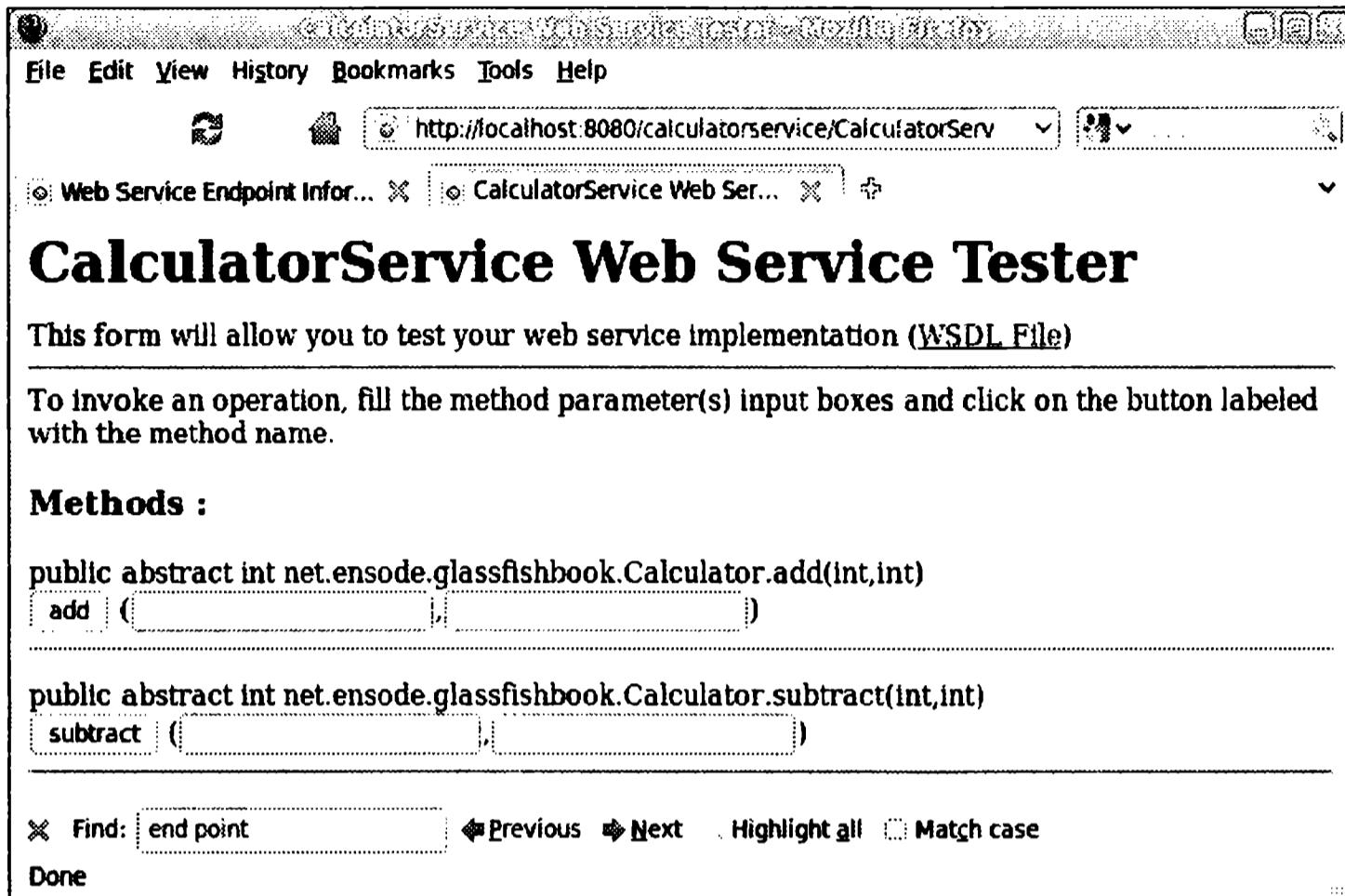
Web Service Endpoint Information

View details about a web service endpoint

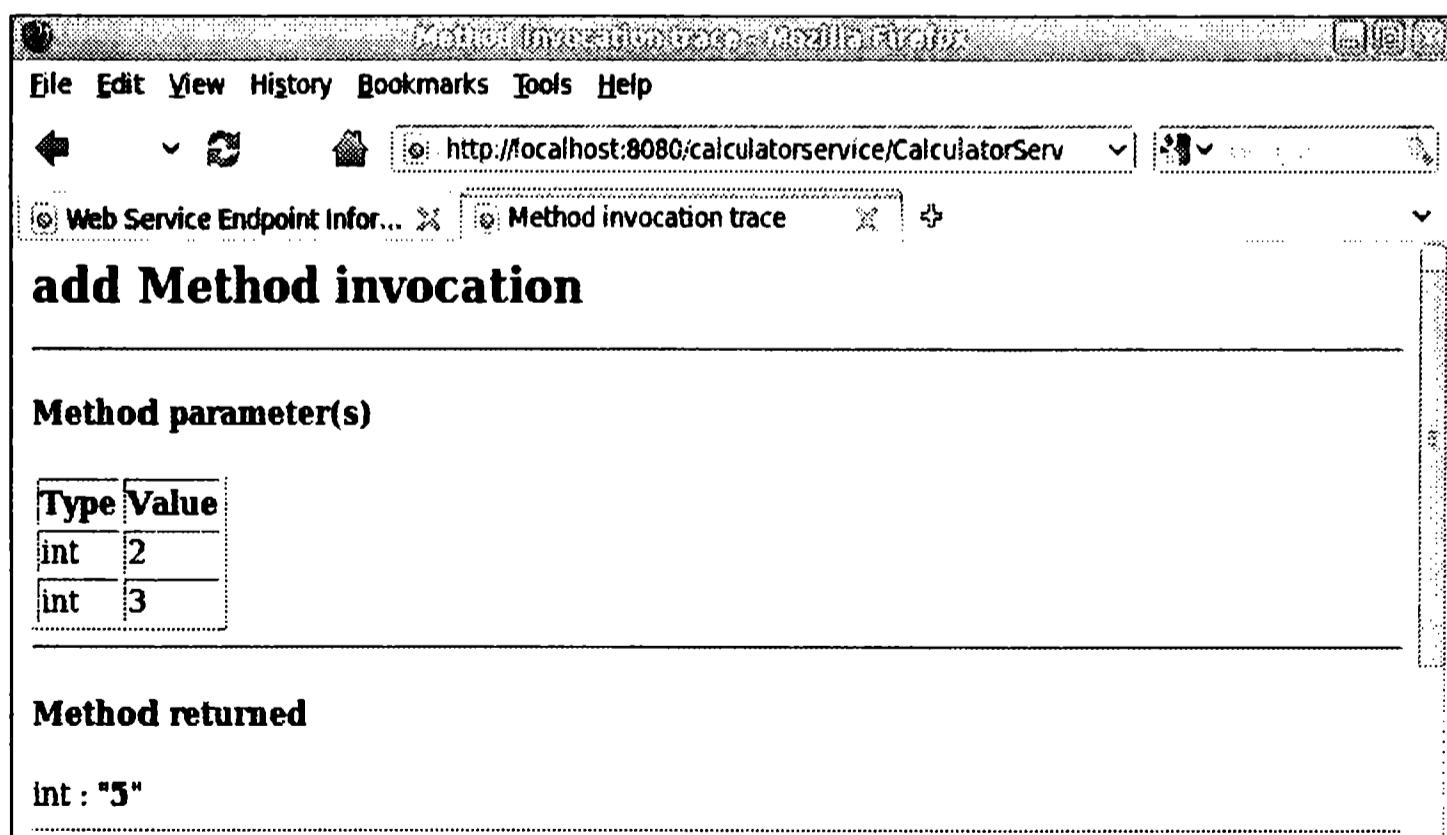
Back

Application Name:	net.enseode.glassfishbook_calculatorservice_war_1.0
Tester:	/calculatorservice/CalculatorService?Tester
WSDL:	/calculatorservice/CalculatorService?wsdl
Endpoint Name:	Calculator
Service Name:	http://glassfishbook.enseode.net/
Port Name:	CalculatorPort
Deployment Type:	109
Implementation Type:	SERVLET
Implementation Class Name:	net.enseode.glassfishbook.Calculator
Endpoint Address URI:	/calculatorservice/CalculatorService
Namespace:	net.enseode.glassfishbook.Calculator
Description:	

В окне, представленном на предыдущем рисунке, имеется ссылка **Тестер** (Tester); щелчок по этой ссылке приведет нас к автоматически сгенерированной странице, позволяющей тестировать наш веб-сервис:



Чтобы протестировать методы, мы должны просто ввести ряд параметров в текстовые поля и щелкнуть по соответствующей кнопке. Например, ввод значений 2 и 3 в текстовых полях рядом с названием метода add и щелчок по кнопке **суммировать** (add) привели бы к следующему результату:



«За кулисами» JAX-WS использует протокол SOAP для обмена информацией между клиентами и серверами веб-сервисов. Прокручивая предыдущую страницу вниз, мы сможем увидеть запрос и отклик SOAP, сгенерированные нашим тестом:

SOAP Request

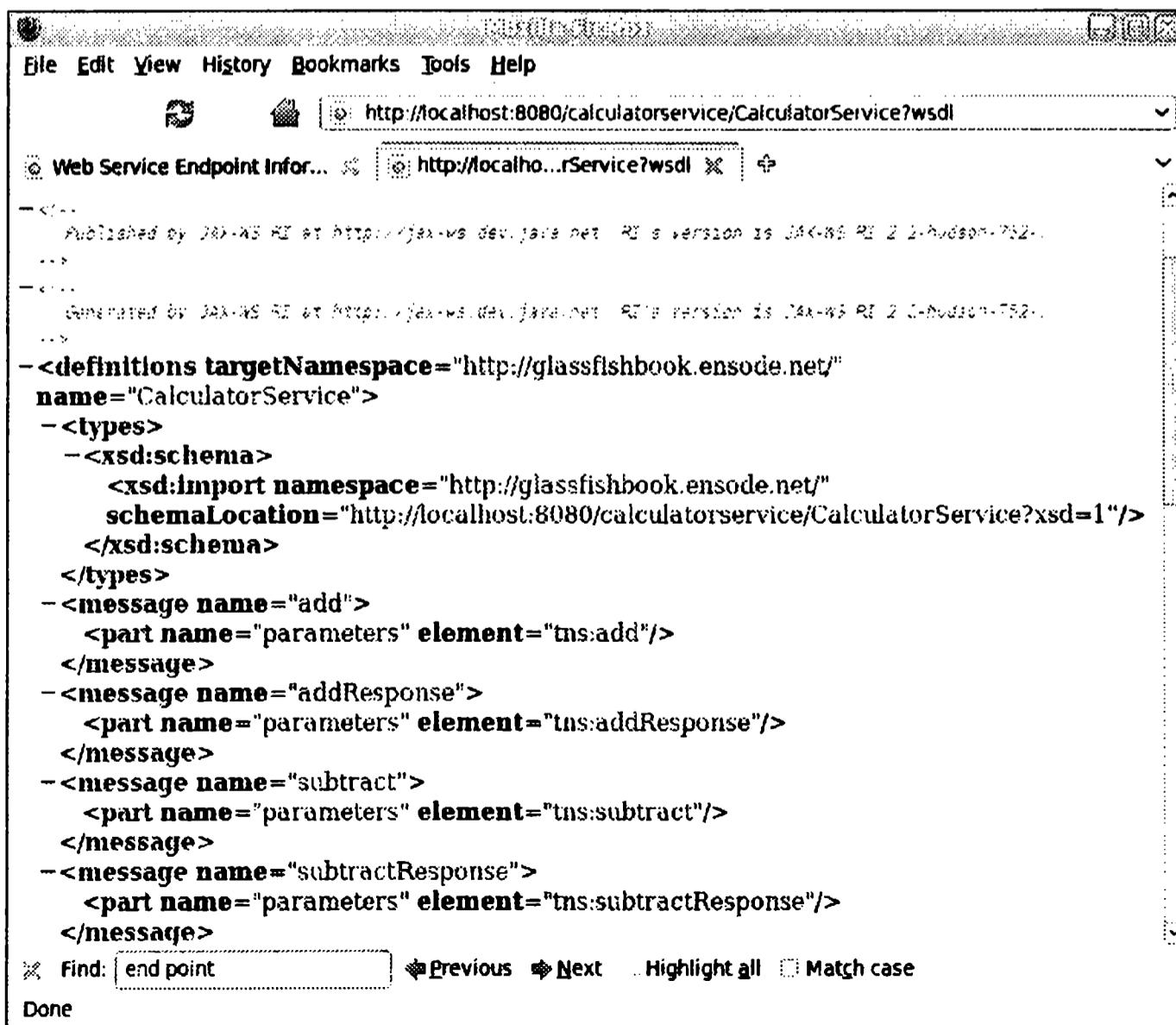
```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header/>
  <S:Body>
    <ns2:add xmlns:ns2="http://glassfishbook.ensode.net/">
      <arg0>2</arg0>
      <arg1>3</arg1>
    </ns2:add>
  </S:Body>
</S:Envelope>
```

SOAP Response

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:addResponse xmlns:ns2="http://glassfishbook.ensode.net/">
      <return>5</return>
    </ns2:addResponse>
  </S:Body>
</S:Envelope>
```

Как разработчики приложений, мы не должны проявлять повышенный интерес к этим запросам SOAP, поскольку о них автоматически заботится API JAX-WS.

Клиенты веб-сервиса нуждаются в файле на языке *определения веб-сервисов* (Web Services Definition Language (WSDL)) для генерации исполняемого кода, который они могут использовать для вызова веб-сервиса. Файлы WSDL обычно помещаются на веб-сервер, и клиенты получают к ним доступ через их URL. Когда развертывается веб-сервис, разработанный с использованием JAX-WS, то WSDL генерируется автоматически. Мы можем просмотреть его вместе с его URL, щелкнув по ссылке **Представление WSDL (View WSDL)** на странице **Информация о Конечной точке веб-сервиса** (Web Service Endpoint Information).



Обратите внимание на URL WSDL в адресной строке обозревателя; этот URL понадобится нам при разработке клиента для нашего веб-сервиса.

Разработка клиента веб-сервиса

Как мы уже отметили, исполняемый код должен быть сгенерирован из WSDL веб-сервиса. Затем клиент веб-сервиса вызовет этот исполняемый код, чтобы получить доступ к веб-сервису.

GlassFish включает утилиту `wsimport` для генерации кода Java из WSDL. Эту утилиту можно найти, зайдя в [Каталог установки `glassfish`]/`glassfish/bin/`. Единственным обязательным аргументом `wsimport` является URL WSDL, соответствующий веб-сервису¹:

```
wsimport http://localhost:8080/calculatorservice/
CalculatorService?WSDL
```

Данная команда генерирует много скомпилированных классов Java, которые позволяют клиентским приложениям получать доступ к нашему веб-сервису. 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 исходного кода для сгенерированных классов автоматически удаляются; они могут быть сохранены путем передачи параметра `-keer` утилите `wsimport`.

Эти классы должны быть добавлены в CLASSPATH клиента, чтобы они были доступны для кода клиента.

В дополнение к инструменту командной строки GlassFish включает пользовательскую задачу 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/sges-v3/glassfish/modules/webservices-osgi.jar"/>
      <classpath path="/opt/sges-v3/glassfish/modules/jaxb-osgi.jar"/>
      <classpath path="/opt/sges-v3/glassfish/lib/javaee.jar"/>
    </taskdef>
```

¹ Вся команда записывается в одной строке. – Прим. перев.

```

<wsimport wsdl="http://localhost:8080/calculatorservice/
           CalculatorService?wsdl"
           xendorsed="true"/>
</target>
</project>

```

Этот пример демонстрирует самый минимальный сценарий сборки Ant, который только поясняет, как настроить пользовательскую цель `<wsimport>` Ant. В действительности у сценария сборки проекта Ant было бы несколько других целей для компиляции, создания WAR-файла и т. д.

Поскольку `<wsimport>` является пользовательской целью Ant, а не стандартной целью, мы должны добавить элемент `<taskdef>` в наш сценарий сборки Ant. Нам следует определить атрибуты `name` и `classname`, как показано в приведенном примере. Кроме того, необходимо добавить нижеперечисленные JAR-файлы в `CLASSPATH` задачи с помощью вложенного элемента `<classpath>`:

- `webservices-osgi.jar`;
- `jaxb-osgi.jar`;
- `javaee.jar`.

Файлы `webservices-osgi.jar` и `jaxb-osgi.jar` могут быть найдены в каталоге [Каталог установки *glassfish*]/*glassfish/modules*. Файл `javaee.jar` содержит весь API Java EE 6 и может быть найден в [Каталог установки *glassfish*]/*glassfish/lib*.

После создания задачи `<wsimport>` через элемент `<taskdef>` мы готовы ее использовать. Мы должны указать расположение WSDL через его атрибут `wsdl`. Сразу по выполнении этой задачи будет сгенерирован код Java, требуемый для получения доступа к веб-сервису, определенному в WSDL.

JDK 1.6 поставляется совместно с JAX-WS 2.1. Если мы используем эту версию JDK, мы должны сообщить Ant, что необходимо использовать API JAX-WS 2.2, поставляемый с сервером GlassFish. Это легко можно сделать, установив атрибут `xendorsed` пользовательской задачи `wsimport` Ant в значение `true` (истина).

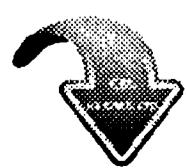
Читатели, использующие инструмент Maven для разработки своих проектов, могут воспользоваться преимуществами плагина `AntRun` инструмента Maven, чтобы выполнить цель `wsimport` Ant для создания их кода. Этот подход показан в следующем файле `pom.xml`:

```

<? 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.ensode.glassfishbook</groupId>
    <artifactId>calculatorserviceclient</artifactId>
    <packaging>jar</packaging>
    <name>Клиент простого веб-сервиса</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"/>
              <taskdef name="wsimport"
                classname="com.sun.tools.ws.ant.
                WsImport">
                <classpath path="/home/heffel/
                  ges-v3/glassfish/
                  modules/webservices-osgi.
                  jar"/>
                <classpath path="/home/heffel/
                  ges-v3/glassfish/
                  modules/jaxb-osgi.jar"/>
                <classpath path="/home/heffel/
                  ges-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"
                endorsed="true"/>
            </tasks>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```



```

<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>

```

 Имеется плагин wsimport для Maven, однако, во время написания книги, он не был обновлен для работы с JAX-WS 2.2, которая является версией, поставляемой с сервером GlassFish.

Внутри тега `<configuration>` файла `pom.xml`, соответствующего плагину `AntRun`, мы помещаем любые задачи `Ant`, которые нужно выполнить. Неудивительно, что тело этого тега в нашем примере почти идентично файлу типа «`build`» `Ant`, который мы только что обсудили.

Теперь, когда мы знаем, как создать наш код с помощью `Ant` или `Maven`, можно разработать простого клиента для получения доступа к нашему веб-сервису:

```

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` инжектирует экземпляр веб-сервиса в наше клиентское приложение. Ее атрибут `wsdlLocation` содержит URL WSDL, соответствующий веб-сервису, который мы вызываем.

Обратите внимание, что класс веб-сервиса является экземпляром класса под названием `CalculatorService`; этот класс был создан, когда мы вызвали утилиту `wsimport`. Утилита `wsimport` всегда генерирует класс с именем, идентичным имени реализованного нами класса плюс суффикс `Service`. Этот класс службы мы используем для получения экземпляра класса веб-сервиса, разработанного нами. В данном примере это осуществляется вызовом метода `getCalculatorPort()` на экземпляре `CalculatorService`. В общем, вызываемый метод (геттер) для получения экземпляра нашего класса веб-сервиса следует шаблону именования `getNamePort()`, где `Name` – имя класса, написанного нами для реализации веб-сервиса. После того как мы получим экземпляр класса нашего веб-сервиса, мы сможем просто вызвать его методы, как и в случае с любым регулярным объектом Java.



Строго говоря, метод `getNamePort()` класса сервиса возвращает экземпляр класса, реализующего интерфейс, сгенерированный утилитой `wsimport`. Этому интерфейсу присваивается имя нашего класса веб-сервиса, и он объявляет все методы, которые мы объявили как веб-сервисы. Для всех практических целей использования (задач) возвращенный объект эквивалентен классу нашего веб-сервиса.

Напомним из главы 9: чтобы предоставить ресурсы для работы с автономным клиентом (не развернутым на сервере GlassFish), мы должны выполнить его через утилиту `appclient`. Предположим, мы упаковали наших клиентов в JAR-файл `calculatorserviceclient.jar`; тогда команда для их выполнения будет выглядеть так:

```
appclient -client calculatorserviceclient.jar
```

После ввода этой команды в командной строке мы должны увидеть вывод нашего клиента в консоли:

```
1 + 2 = 3
1 - 2 = -1
```

Здесь мы передали примитивные типы и в качестве параметров, и в качестве получаемых значений. Конечно, таким же образом можно передать объекты – и в качестве параметров, и в качестве возвращаемых значений. К сожалению, не все стандартные классы Java и примитивные типы могут использоваться в качестве параметров метода или возвращаемых значений при вызове веб-сервисов. Причина в том, что «за кулисами» параметры метода и возвращаемые типы получают отображения на определения XML, и не каждый тип может быть отображен надлежащим способом.

Ниже перечислены типы, которые могут использоваться в вызовах веб-сервиса 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 примитивные типы:

- `boolean;`
- `byte;`
- `byte[];`
- `double;`
- `float;`
- `int;`
- `long;`
- `short.`

Мы можем также использовать наши собственные классы в качестве параметров и/или возвращаемых значений для методов веб-сервиса, но переменные, задействованные в наших классах, должны быть одного из перечисленных типов.

Кроме того, допустимо использование массивов – как в качестве параметров метода, так и в качестве возвращаемых значений. Однако при выполнении `wsimport` эти массивы преобразуются в списки, создавая тем самым несоответствие между сигнатурой метода веб-службы и сигнатурой метода ее вызова на стороне клиента. По этой причине предпочтительнее использовать списки в качестве параметров метода и/или возвращаемых значений: данный подход также допустим и не создает несоответствий между клиентом и сервером.



JAX-WS использует внутри *Архитектуру Java для связывания с XML* (Java Architecture for XML Binding (JAXB)), чтобы создать сообщения SOAP из вызовов метода. Типы, которые нам разрешается использовать для вызовов метода и возвращаемых значений, должны поддерживаться JAXB. Для получения дополнительной информации по JAXB см. <http://jaxb.java.net/>.

Отправка вложений веб-сервисам

В дополнение к отправке и принятию данных в виде типов, обсуждаемых в предыдущих разделах, методы веб-сервиса могут отправлять и принимать вложенные файлы. Следующий пример поясняет, 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 требуемый 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-сервиса, который получает одно или более вложений, нам 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.

Как и в случае с любым стандартным веб-сервисом, предыдущий код должен быть упакован в WAR-файл и развернут. После его развертывания будет автоматически сгенерирован WSDL. Затем мы должны выполнить утилиту `wsimport`, чтобы сгенерировать код, который может использовать наш клиент веб-сервиса для получения доступа к веб-сервису. Как упоминалось ранее, утилита `wsimport` может быть вызвана прямо из командной строки или через пользовательскую цель Ant. Выполнив `wsimport` с целью генерации кода для получения доступа к веб-сервису, мы сможем написать и скомпилировать наш 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("Вложение успешно отправлено.");
    }
    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-сервиса, который должен отправить веб-сервису одно или более вложений, сначала получает экземпляр веб-сервиса, как обычно. Затем он создает экземпляр `java.io.File`, передавая расположение файла для вложения в качестве параметра его конструктора. После того как у нас появится экземпляр `java.io.File`, содержащий файл, который мы хотим присоединить к вложению, нам нужно будет преобразовать этот файл в байтовый массив и передать его методу веб-сервиса, который ожидает вложения.

Обратите внимание, что, в отличие от передачи стандартных параметров, тип параметра, используемый для вызова клиентом метода, ожидающего вложение, отличается от типа параметра метода в коде веб-сервиса. Метод в коде веб-сервиса ожидает экземпляр `javax.activation.DataHandler` для каждого вложения. Однако код, сгенерированный утилитой `wsimport`, ожидает массив байтов для каждого вложения. Эти массивы байтов преобразуются в правильный тип (`javax.activation.DataHandler`) «за кулисами» утилитой `wsimport`, сгенерировавшей код.

Как разработчиков приложений нас не должно интересовать подробное объяснение того, почему это происходит. Следует только иметь в виду, что при отправке вложения методу веб-сервиса типы параметров будут отличаться в коде веб-сервиса и в клиентском вызове.

Представление EJB как веб-сервисов

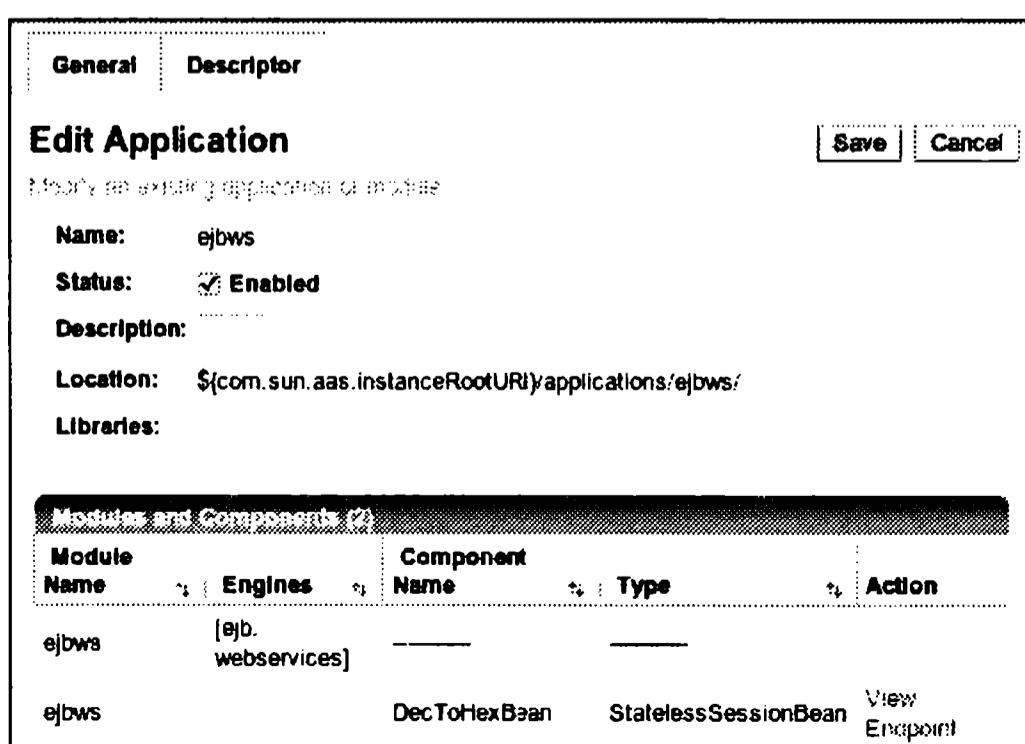
Помимо того что можно создавать веб-сервисы, как показано в предыдущем разделе, открытые методы сеансовых бинов, не сохраняющих состояние, могут быть легко представлены как веб-сервисы. Следующий пример поясняет, как это сделать:

```
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 видно из приведенного кода, для представления открытых методов сеансового бина, не сохраняющего состояние, в качестве веб-сервисов нам достаточно декорировать объявление его класса аннотацией `@WebService`. Само собой разумеется, класс сеансового бина также должен быть декорирован аннотацией `@Stateless`.

Точно так же, как и регулярные сеансовые бины, не сохраняющие состояние, методы которых представляются как веб-сервисы, они должны быть упакованы в JAR-файле. После развертывания этого файла мы сможем увидеть новый веб-сервис в узле **Приложения** (Applications) в веб-консоли администрирования сервера GlassFish. Щелкнув по узлу приложения, мы сможем увидеть некоторые подробные данные в панели основного содержания консоли GlassFish:



Обратите внимание, что значение в столбце **Тип** (Type) для нашего нового веб-сервиса установлено в **StatelessSessionBean**. Это позволяет нам сразу увидеть, что веб-сервис реализован как Enterprise JavaBean.

Точно так же как стандартные веб-сервисы, веб-сервисы EJB автоматически генерируют WSDL для его использования клиентами. После развертывания нашего EJB доступ к нему можно получить тем же самым путем – щелкая по ссылке **Представление Конечной точки** (View End Point).

Клиенты веб-сервиса EJB

Следующий класс демонстрирует процедуру получения доступа к методу веб-сервиса 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("Десятичное число 4013 в шестнадцатеричном
            представлении: " + decToHexBean.convertDecToHex(4013));
    }
    public static void main(String[] args)
    {
        new DecToHexClient().convert();
    }
}
```

Как видно из приведенного кода, нет ничего особенного, что нам нужно было бы сделать для получения доступа к веб-сервису EJB со стороны клиента. Процедура та же, что и для стандартных веб-сервисов.

Поскольку предыдущий код представляет собой автономное приложение, он должен быть выполнен через приложение appclient:

```
appclient -client ejbwsclient.jar
```

Эта команда дает следующий результат:

```
Десятичное число: 4013 в шестнадцатеричном представлении: fad
```

Безопасность веб-сервисов

Как и регулярные веб-приложения, веб-сервисы могут быть защищены таким образом, чтобы только авторизованные пользователи могли получить к ним доступ. Этого можно достичь, изменив дескриптор развертывания веб-сервиса 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 модифицируем нашу службу калькулятора таким образом, чтобы только авторизованные пользователи могли получить к ней доступ. Обратите внимание, что изменения для обеспечения безопасности веб-сервиса не отличаются от изменений для обеспечения безопасности любого регулярного веб-приложения. Шаблон URL для использования элемента `<url-pattern>` может быть получен щелчком по ссылке **Представление WSDL** (View WSDL), соответствующей нашей службе. В нашем примере URL для ссылки выглядит так: `http://localhost:8080/calculatorservice/CalculatorService?WSDL`.

Значение, используемое в элементе `<url-pattern>`, является значением URL сразу после корня контекста (в нашем примере это `/CalculatorService`) и перед вопросительным знаком, сопровождаемым 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 для получения WSDL и генерации соответствующего кода. Если GET-запросы будут защищены, утилита `wsimport` перестанет работать, поскольку она будет лишена доступа к WSDL. Будущие версии `wsimport` позволят нам указывать имя пользователя и пароль для аутентификации. В то же время обходным решением является защита только POST-запросов.

Следующий код поясняет, как автономный клиент может получить доступ к защищенному веб-сервису:

```
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()
    {
        // Добавление пользователя с именем "joe" и паролем "password"
        // в область файла для успешного выполнения веб-сервиса.
        // Пользователь "joe" должен принадлежать группе "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();
}
}

```

Код представляет собой модифицированную версию автономного клиента веб-сервиса калькулятора, который мы видели ранее. Эта версия была изменена для получения доступа к защищенной версии веб-сервиса. Как можно видеть из кода, для выполнения данной задачи мы должны поместить имя пользователя и пароль в контекст запроса. Имя пользователя и пароль должны быть допустимыми для области, используемой при аутентификации веб-сервиса.

Мы можем добавить имя пользователя и пароль к контексту запроса путем приведения класса конечной точки нашего веб-сервиса к типу javax.xml.ws.BindingProvider и вызова его метода getRequestContext(). Этот метод возвращает экземпляр java.util.Map. Затем можно просто добавить имя пользователя и пароль, вызывая метод put в карте (Map) и используя константы USERNAME_PROPERTY и PASSWORD_PROPERTY, определенные в BindingProvider, в качестве ключей, а соответствующие строковые объекты – в качестве значений.

Безопасность веб-сервисов EJB

Аналогично стандартным веб-сервисам, EJB, представленные как веб-сервисы, могут быть защищены таким образом, чтобы доступ к ним могли получить только авторизованные клиенты. Это может быть достигнуто с помощью конфигурирования EJB через файл sun-ejb-jar.xml:

```

<? xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sun-ejb-jar PUBLIC "-//Sun Microsystems,
Inc./DTD Application Server 9.0 EJB 3.0//EN"
"http://www.sun.com/software/appserver/dtds/sun-ejb-jar_3_0-0.dtd">
<sun-ejb-jar>
    <enterprise-beans>
        <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>
    </enterprise-beans>
</sun-ejb-jar>

```

Как видно из этого дескриптора развертывания, безопасность для EJB, представленных как веб-сервисы, настраивается иначе, чем для стандартных EJB. Для EJB, представленных как веб-сервисы, конфигурация безопасности помещается в элемент `<webservice-endpoint>` дескриптора развертывания `sun-ejb-jar.xml`.

Элемент `<port-component-name>` должен устанавливать имя EJB, который мы представляем как веб-сервис. Это имя EJB определяется в элементе `<ejb-name>`.

Элемент `<login-config>` очень похож на соответствующий элемент в дескрипторе развертывания веб-приложения `web.xml`. Элемент `<login-config>` должен содержать метод авторизации, определяемый подэлементом `<auth-method>`, и область для использования аутентификации. Область определяется подэлементом `<realm>`.



Не используйте аннотацию `@RolesAllowed` для EJB, который будет представлен, как веб-сервисы. Эта аннотация применима, когда к методам EJB получают доступ через его удаленный или локальный интерфейс. Если EJB или один и более его методов будет декорирован этой аннотацией, то вызов метода перестанет работать и будет выдано исключение безопасности.

После того как мы сконфигурируем веб-сервис EJB для аутентификации, его можно упаковывать в JAR-файл и развертывать как обычно. Теперь веб-сервис EJB готов для получения к нему доступа со стороны клиентов.

Следующий пример кода поясняет, как клиент веб-сервиса EJB может получить доступ к защищенному веб-сервису 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("Десятичное число 4013 в шестнадцатеричном
            представлении: " + secureDecToHexBean.convertDecToHex(4013));
    }
    public static void main(String[] args)
    {
        new DecToHexClient().convert();
    }
}
```

Приведенный пример показывает, что процедура получения доступа к EJB, представленному как веб-сервис, идентична обеспечению доступа к стандартному веб-сервису. Способ реализации веб-сервиса клиенту не важен.

Резюме

В этой главе мы рассмотрели разработку веб-сервисов и клиентов веб-сервисов через API JAX-WS. Мы объяснили, как включить генерацию кода веб-сервиса для клиентов веб-сервиса путем использования Ant или Maven 2 в качестве инструмента сборки. Также мы обсудили допустимые типы, которые могут использоваться для удаленных вызовов метода через JAX-WS. Была рассмотрена отправка вложений веб-сервису. Мы также выяснили, как представить методы EJB в качестве веб-сервисов. Наконец, было показано, как обезопасить веб-сервисы от не разрешенных для них клиентов.

12

Веб-сервисы RESTful в Jersey и JAX-RS

Передача состояния представления (Representational State Transfer (REST)) является архитектурным стилем, в котором веб-сервисы рассматриваются как ресурсы и могут быть идентифицированы *Унифицированными идентификаторами ресурсов* (Uniform Resource Identifiers (URI)).

Веб-сервисы, разработанные в стиле REST, известны как *RESTful веб-сервисы*.

Java EE 6 добавляет поддержку веб-сервисов RESTful посредством добавления *API Java для веб-сервисов RESTful* (Java API for RESTful Web Services (JAX-RS)). JAX-RS некоторое время был доступен как автономный API; он стал частью спецификации Java EE в версии 6. В этой главе мы покажем, как разработать веб-сервисы RESTful через API JAX-RS, используя Jersey – реализацию спецификации JAX-RS, включенную в GlassFish.

В главе будут затронуты следующие темы:

- введение в веб-сервисы RESTful и JAX-RS;
- разработка простого веб-сервиса RESTful;
- параметры пути;
- параметры запроса.

Введение в веб-сервисы RESTful и JAX-RS

Веб-сервисы RESTful отличаются большой гибкостью. Они могут использовать несколько различных MIME-типов, хотя их обычно пишут для потребления и/или производства данных в формате XML или *Объектной нотации JavaScript* (JavaScript Object Notation (JSON)).

Веб-сервисы должны поддерживать один или несколько методов HTTP из следующих четырех:

- **GET** – в соответствии с соглашением запрос этого типа используется для получения существующих ресурсов;

- **POST** – в соответствии с соглашением такой запрос используется для обновления существующего ресурса;
- **PUT** – в соответствии с соглашением такой запрос используется для создания нового ресурса;
- **DELETE** – в соответствии с соглашением подобный запрос используется для удаления существующего ресурса.

Мы разрабатываем веб-сервис RESTful в спецификации JAX-RS, создавая класс с аннотированными методами, которые вызываются, когда наш веб-сервис получает один из вышеперечисленных методов HTTP-запроса. После разработки и развертывания нашего веб-сервиса RESTful необходимо разработать клиента, который будет отправлять запросы нашей службе. Jersey, являющийся реализацией JAX-RS, поставляемой в комплекте с GlassFish, – включает в себя API, который можно использовать для упрощения разработки клиентов веб-сервиса RESTful. Клиентский API Jersey представляет собой дополнительную функциональность и не является частью спецификации JAX-RS.

Разработка простого веб-сервиса RESTful

В этом разделе мы разработаем простой веб-сервис, чтобы пояснить, как можно заставить методы нашего веб-сервиса отвечать на различные методы HTTP-запросов.

Разработка веб-сервиса RESTful с использованием JAX-RS проста и прямолинейна. Каждый из наших веб-сервисов RESTful должен быть вызван через его Унифицированный идентификатор ресурса (URI). Этот URI указывается аннотацией `@Path`, которую мы должны использовать для декорирования нашего класса RESTful-ресурса веб-сервиса.

Разрабатывая веб-сервисы RESTful, мы должны разработать методы, которые будут вызваны, когда наш веб-сервис получит HTTP-запрос. Мы должны реализовать методы для обработки одного или нескольких из четырех типов запросов, которые обрабатывают веб-сервисы RESTful, а именно: GET, POST, PUT и/или DELETE.

API JAX-RS предоставляет четыре аннотации, которые мы можем использовать для декорирования методов в нашем веб-сервисе. Аннотации соответственно называют `@GET`, `@POST`, `@PUT` и `@DELETE`. Декорирование метода нашего веб-сервиса одной из этих аннотаций заставит его отвечать на HTTP-запрос, использующий соответствующий метод.

Кроме того, каждый метод в нашей службе должен производить и/или потреблять определенный MIME-тип. MIME-тип, который будет производиться, должен быть указан аннотацией `@Produces`. В свою очередь MIME-тип, который будет потребляться, должен быть указан аннотацией `@Consumes`.

Следующий пример поясняет понятия, которые мы только что объяснили.

 Отметим, что этот пример ничего «существенного» не делает — его цель заключается в том, чтобы показать, как заставить различные методы нашего класса RESTful-ресурса веб-сервиса отвечать на различные методы HTTP-запросов.

```
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 реальном RESTful-сервисе мы будем извлекать данные
        // из базы данных, а затем возвращать XML-представление данных.
        System.out.println("--- " + this.getClass().getCanonicalName()
            + " вызван метод .getCustomer()");
        return "<customer>\n"
            + "<id>123</id>\n"
            + "<firstName>Joseph</firstName>\n"
            + "<middleName>William</middleName>\n"
            + "<lastName>Graystone</lastName>\n"
            + "</customer>\n";
    }
    /**
     * Создает нового заказчика - customer.
     * @param customer - данные в XML-представлении для создания customer.
     */
    @PUT
    @Consumes("text/xml")
    public void createCustomer(String customerXML)
    {
        // В реальном RESTful-сервисе мы вначале разберем XML-данные,
        // полученные в XML-параметре customer, а затем вставим
        // новую строку в базу данных.
        System.out.println("--- " + this.getClass().getCanonicalName()
            + " вызван метод .createCustomer()");
        System.out.println("customerXML = " + customerXML);
    }
    @POST
    @Consumes("text/xml")
    public void updateCustomer(String customerXML)
    {
        // В реальном RESTful-сервисе мы вначале разберем XML-данные,
        // полученные в XML-параметре customer, а затем обновим
        // строку в базе данных.
        System.out.println("--- " + this.getClass().getCanonicalName()
            + " вызван метод .updateCustomer()");
        System.out.println("customerXML = " + customerXML);
    }
    @DELETE
    @Consumes("text/xml")
    public void deleteCustomer(String customerXML)
    {
        // В реальном RESTful-сервисе мы вначале разберем XML-данные,
        // полученные в XML-параметре customer, а затем удалим
        // строку из базы данных.
        System.out.println("--- " + this.getClass().getCanonicalName())
    }
}
```



```
        + " вызван метод .deleteCustomer()");  
        System.out.println("customerXML = " + customerXML);  
    }  
}
```

Обратите внимание, что этот класс декорирован аннотацией `@Path`. Она определяет Унифицированный идентификатор ресурса (URI) для нашего веб-сервиса RESTful. Полный URI для нашей службы будет включать протокол, имя сервера, порт, корневой контекст, путь ресурсов REST (см. следующий подраздел) и значения, переданные в эту аннотацию.

Предположим, что наш веб-сервис был развернут на сервере `example.com` с использованием протокола HTTP по порту 8080, имеет корневой контекст `jaxrsintro` и путь к ресурсам REST `resources`. В таком случае полный URI для нашей веб-службы будет выглядеть так: `http://example.com:8080/jaxrsintro/resources/customer`.



Поскольку веб-обозреватели генерируют GET-запросы, то, когда указывается URL, мы можем протестировать метод GET нашей службы, просто задав в адресной строке обозревателя URI нашей службы.

Обратите внимание, что каждый из методов в нашем классе аннотирован одной из аннотаций `@GET`, `@POST`, `@PUT` или `@DELETE`. Эти аннотации заставляют наши методы отвечать на HTTP-запросы, которые используют соответствующие методы.

В дополнение ко всему, если наш метод возвращает данные клиенту, мы объявляем, какой MIME-тип данных будет возвращен в аннотации `@Produces`. В нашем примере только метод `getCustomer()` возвращает данные клиенту. Мы хотим возвратить данные в формате XML, поэтому устанавливаем для аннотации `@Produces` значение `"text/xml"`. Аналогичным образом, если наш метод должен потреблять данные от клиента, мы должны указать MIME-тип данных, которые будут потреблены. Это делается через аннотацию `@Consumes`. Все методы в нашей службе, кроме `getCustomer()`, используют данные. Во всех случаях мы ожидаем, что данные будут в формате XML, поэтому снова указываем `"text/xml"` в качестве MIME-типа, который будет использован.

Конфигурирование пути к ресурсам REST для нашего приложения

В предыдущем разделе мы уже коротко упомянули о том, что, прежде чем успешно развернуть веб-сервис RESTful, разработанный с использованием JAX-RS, мы должны сконфигурировать путь ресурсов REST для нашего приложения. Есть два способа это сделать: использовать дескриптор развертывания `web.xml` или разработать класс, который расширяет `javax.ws.rs.core.Application`, и декорировать его аннотацией `@ApplicationPath`.

Конфигурирование через `web.xml`

Мы можем сконфигурировать путь ресурсов REST для нашего JAX-RS веб-сервиса RESTful с помощью дескриптора развертывания `web.xml`.

Библиотеки Jersey включают сервлет, который мы можем сконфигурировать обычным способом в нашем дескрипторе развертывания 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">
    <servlet>
        <servlet-name>JerseyServlet</servlet-name>
        <servlet-class>
            com.sun.jersey.spi.container.servlet.ServletContainer
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>JerseyServlet</servlet-name>
        <url-pattern>/resources/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Как мы видим из приведенной разметки, квалифицированное (полностью определенное) имя класса сервлета Jersey com.sun.jersey.spi.container.servlet.ServletContainer является значением, которое мы добавляем в элемент <servlet-class> файла web.xml. Затем мы даем этому сервлету логическое имя (в нашем примере выбрано JerseyServlet) и объявляем, что шаблон URL обрабатывается сервлетом как обычно. Любые URL, соответствующие шаблону, будут перенаправлены к соответствующим методам наших веб-сервисов RESTful.

Итак, мы имеем возможность убедиться, что конфигурирование сервлета Jersey несколько не отличается от конфигурирования любого другого сервлета.

Конфигурирование через аннотацию @ApplicationPath

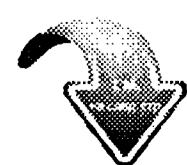
Мы уже неоднократно говорили о том, что Java EE 6 добавляет ряд новых возможностей к спецификации Java EE – с той целью, чтобы свести к минимуму необходимость в создании дескриптора развертывания web.xml. JAX-RS не является исключением; мы можем сконфигурировать путь REST-ресурсов в коде Java через аннотацию.

Чтобы сконфигурировать путь к REST-ресурсам без необходимости обращения к дескриптору развертывания web.xml, мы должны написать класс, который расширяет javax.ws.ApplicationPath, и декорировать его аннотацией @ApplicationPath. Значение, которое будет передано этой аннотации, является путем REST-ресурсов для наших служб.

Следующий пример кода поясняет этот процесс:

```
package com.ensode.jaxrsintro.service.config;

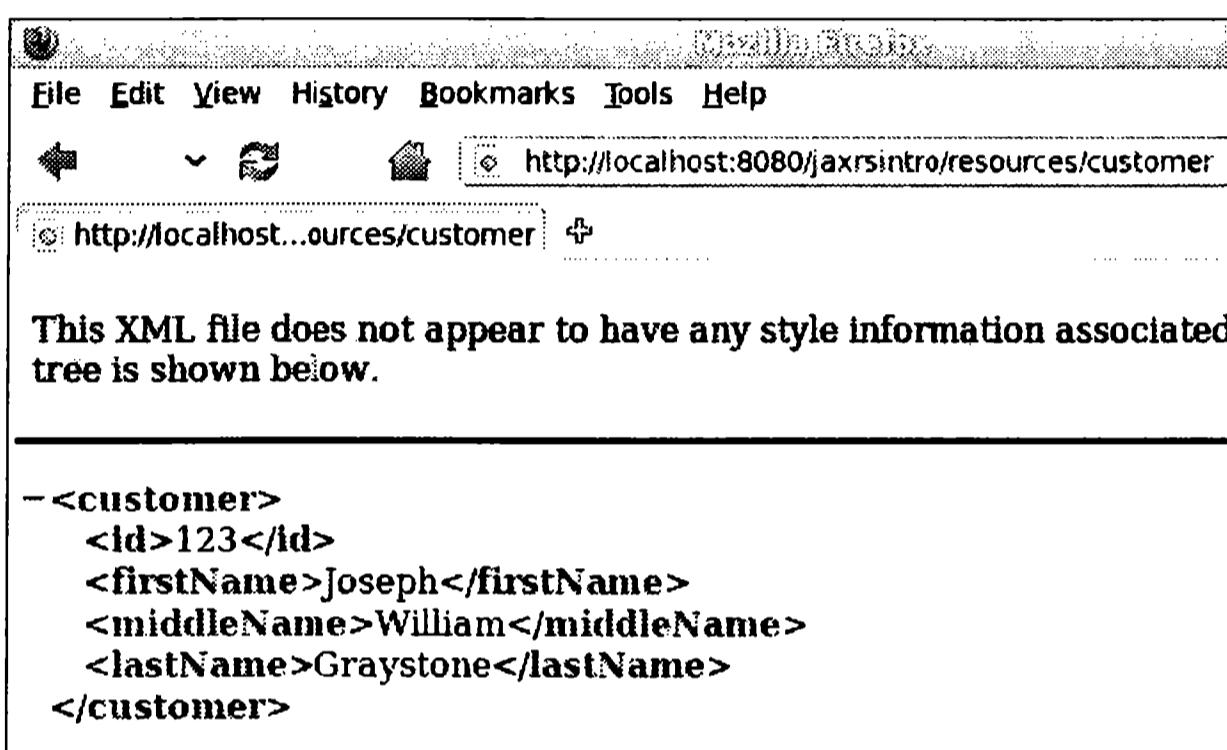
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;
@ApplicationPath("resources")
public class JaxRsConfig extends Application
{}
```



Обратите внимание, что класс не должен реализовывать методы. Он просто должен расширять javax.ws.rs.Application и декорироваться аннотацией @ApplicationPath. Класс должен быть общедоступным (public), может иметь любое имя и быть помещен в любой пакет.

Тестирование нашего веб-сервиса

Как мы упоминали ранее, веб-обозреватели отправляют GET-запросы любым URL, которые мы указываем в их адресной строке. Поэтому самый простой способ протестировать GET-запросы к нашей службе – простое указание в адресной строке обозревателя URI нашего веб-сервиса.



Веб-обозреватели поддерживают только GET- и POST-запросы. Чтобы протестировать POST-запрос через веб-обозреватель, нам нужно написать приложение, содержащее HTML-форму, имеющую атрибут action со значением URI нашего веб-сервиса. Хотя это приложение будет тривиальным для одного веб-сервиса, оно может стать громоздким и неудобным, если распространить его действие на все веб-сервисы RESTful, которые мы разрабатываем.

К счастью, есть утилита командной строки с открытым исходным кодом, называемая curl, которую мы можем использовать для тестирования наших веб-сервисов. Утилита curl включена в большинство дистрибутивов Linux и может быть легко загружена для Windows, Mac OSx и нескольких других платформ (адрес для загрузки – <http://curl.haxx.se/>).

Утилита curl может передать все четыре типа метода запроса (GET, POST, PUT и DELETE) нашему веб-сервису. Реакция сервера будет выведена на экран в консоли командной строки. Утилита curl принимает параметр командной строки -X, который позволяет нам указывать, какой метод запроса нужно передать. Чтобы отправить GET-запрос, мы просто должны ввести в командную строку такую команду:

```
curl -XGET http://localhost:8080/jaxrsintro/resources/customer
```

Результатом выполнения этой команды будет следующий вывод:

```
<customer>
  <id>123</id>
  <firstName>Joseph</firstName>
  <middleName>William</middleName>
  <lastName>Graystone</lastName>
</customer>
```

Неудивительно, что вывод является тем же самым выводом, который мы видели, когда указывали в адресной строке обозревателя URI нашей службы.

Методом запроса по умолчанию для curl является GET, поэтому параметр -X в нашем предыдущем примере избыточен. Мы смогли бы достичь того же самого результата, выполняя следующую команду из командной строки:

```
curl http://localhost:8080/jaxrsintro/resources/customer
```

После выполнения любой из предыдущих двух команд и исследования журнала GlassFish мы должны увидеть вывод операторов System.out.println(), которые мы добавили в метод getCustomer():

```
INFO: --- com.ensode.jaxrsintro.service.CustomerResource.getCustomer()
 вызван метод .getCustomer()
```

Для всех других типов метода запроса мы должны отправить некоторые данные нашей службе. Это может быть выполнено с помощью аргумента командной строки --data для curl¹:

```
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 видно из этого примера, мы должны указать MIME-тип через параметр командной строки -H для утилиты curl, используя формат, который мы видели в примере.

Мы можем убедиться, что предыдущая команда отработала как ожидалось, просмотрев журнал GlassFish:

```
INFO: ---
com.ensode.jaxrsintro.service.CustomerResource.createCustomer() вызван
метод .createCustomer()
INFO: customerXML =<customer><id>321</id><firstName>Amanda</
firstName><middleName>Zoe</middleName><lastName>Adams</lastName></
customer>
```

Так же просто можно протестировать другие типы методов запроса¹:

¹ Все должно быть записано в одну строку через пробел. – Прим. перев.

```
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
```

Результатом будет следующий вывод в журнале GlassFish:

```
INFO: ---
com.ensode.jaxrsintro.service.CustomerResource.updateCustomer() вызван
метод .updateCustomer()
INFO: customerXML = <customer><id>321</id><firstName>Amanda</
firstName><middleName>Tamara</middleName><lastName>Adams</lastName></
customer>
```

Мы можем протестировать метод DELETE, выполняя команду¹:

```
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
```

Результатом будет следующий вывод в журнале GlassFish:

```
INFO: ---
com.ensode.jaxrsintro.service.CustomerResource.deleteCustomer() вызван
метод .deleteCustomer()
INFO: customerXML = <customer><id>321</id><firstName>Amanda</
firstName><middleName>Tamara</middleName><lastName>Adams</lastName></
customer>
```

Преобразование данных между Java и XML с помощью JAXB

В предыдущем примере мы обрабатывали «сырой» XML, полученный в качестве параметра, так же, как возвращали «сырой» XML нашему клиенту. В реальном приложении мы более чем вероятно разобрали бы (проанализировали) XML, полученный от клиента, и использовали его для заполнения объекта Java. Кроме того, любой XML, который мы должны возвратить клиенту, должен был бы создаваться из объекта Java.

Преобразование данных из Java в XML и обратно – настолько частый случай его практического использования, что спецификация Java EE предоставляет для этих целей API, называемый *API Java для связывания с XML* (Java API for XML Binding (JAXB)).

JAXB делает преобразование данных из Java в XML прозрачным и тривиальным. Для этого нам нужно декорировать класс, который мы хотим преобразовать в XML, аннотацией @XmlRootElement. Следующий пример кода поясняет, как это сделать:

```
package com.ensode.jaxrtest.entity;

import java.io.Serializable;
import javax.xml.bind.annotation.XmlRootElement;
```

¹ Все должно быть записано в одну строку через пробел. – Прим. перев.

```
@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 мы видим из приведенного кода, кроме аннотации `@XmlRootElement` на уровне класса нет ничего необычного в этом классе Java.

Как только у нас будет класс, который мы декорировали аннотацией `@XmlRootElement`, мы должны изменить тип параметра нашего веб-сервиса со `String` на наш пользовательский 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 реальных приложениях данные
        // будут поступать из базы данных.
        customer = new Customer(1L, "David", "Raymond", "Heffelfinger");
    }
    @GET
    @Produces("text/xml")
    public Customer getCustomer()
    {
        // В реальном RESTful-сервисе мы будем извлекать данные
        // из базы данных, а затем возвращать их в XML-представлении.
        System.out.println("--- " + this.getClass().getCanonicalName()
            + " вызван метод .getCustomer()");
        return customer;
    }
    @POST
    @Consumes("text/xml")
    public void updateCustomer(Customer customer)
    {
        // В реальном RESTful-сервисе JAXB будет разбирать параметр customer
        // с XML-представлением данных,
        // а затем будет изменять данные в базе данных.
        System.out.println("--- " + this.getClass().getCanonicalName()
            + " вызван метод .updateCustomer()");
        System.out.println("---- получили следующего заказчика: " + customer);
    }
    @PUT
    @Consumes("text/xml")
    public void createCustomer(Customer customer)
    {
        // В реальном RESTful-сервисе мы будем вставлять строку
        // с новыми данными в базу данных, полученную из параметра
        // customer
        System.out.println("--- " + this.getClass().getCanonicalName()
            + " вызван метод .createCustomer()");
        System.out.println("заказчик = " + customer);
    }
    @DELETE
    @Consumes("text/xml")
    public void deleteCustomer(Customer customer)
    {
```

```
// В реальном RESTful-сервисе мы будем удалять строку
// из базы данных, соответствующую параметру customer
System.out.println("--- " + this.getClass().getCanonicalName()
    + " вызван метод .deleteCustomer()");
System.out.println("заказчик = " + customer);
}
}
```

Как видно из приведенного кода, разница между этой и предыдущей версией нашего веб-сервиса RESTful заключается в том, что все типы параметров и возвращаемые значения были изменены со `String` на `Customer`. JAXB заботится о преобразовании наших типов параметров и возвращаемых значений к XML и из XML по мере необходимости. При использовании JAXB объекты наших пользовательских классов автоматически заполняются данными из XML, передаваемого от клиента. Точно так же возвращаемые значения прозрачно преобразуются в XML.

Разработка клиента веб-сервиса RESTful

Хотя утилита `curl` позволяет нам быстро тестировать наши веб-сервисы RESTful и является дружественным для разработчика инструментом, она все же недостаточно удобна для пользователя. Мы не хотим ожидать, когда наши пользователи введут команды `curl` в ее командную строку для использования нашего веб-сервиса. Поэтому мы должны разработать клиента для наших служб. Jersey – реализация JAX-RS, поставляемая с сервером GlassFish, – включает в себя клиентский API, который мы можем использовать для упрощения разработки клиентских приложений.

Следующий пример поясняет, как использовать клиентский API Jersey:

```
package com.ensode.jaxrsintroclient;
import com.ensode.jaxbxmlconversion.entity.Customer;
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.UniformInterface;
import com.sun.jersey.api.client.WebResource;
import javax.ws.rs.core.MediaType;
public class App
{
    private WebResource baseUriWebResource;
    private WebResource webResource;
    private Client client;
    private static final String BASE_URI =
        "http://localhost:8080/jaxbxmlconversion/resources";
    public static void main(String[] args)
    {
        App app = new App();
        app.initWebResource();
        app.getCustomer();
        app.insertCustomer();
    }
    private void initWebResource()
    {
        com.sun.jersey.api.client.config.ClientConfig config =
            new com.sun.jersey.api.client.config.DefaultClientConfig();
        client = Client.create(config);
        baseUriWebResource = client.resource(BASE_URI);
        webResource = baseUriWebResource.path("customer");
    }
}
```



```
public void getCustomer()
{
    UniformInterface uniformInterface = webResource.type(
        MediaType.TEXT_XML);
    Customer customer = uniformInterface.get(Customer.class);
    System.out.println("заказчик = " + customer);
}
public void insertCustomer()
{
    Customer customer = new Customer(234L, "Tamara", "A", "Graystone");
    UniformInterface uniformInterface = webResource.type(
        MediaType.TEXT_XML);
    uniformInterface.put(customer);
}
}
```

Первое, что мы должны сделать, – создать экземпляр com.sun.jersey.api.client.config.DefaultClientConfig, а затем передать его статическому методу create() класса com.sun.jersey.api.client.Client. В этой точке кода мы создали экземпляр com.sun.jersey.api.client.Client. Затем мы должны создать экземпляр com.sun.jersey.api.client.WebResource, вызывая метод resource() нашего нового созданного экземпляра Client, передавая основной URI нашего веб-сервиса, определенного в его конфигурации, как было показано выше в этой главе.

После того как у нас появился экземпляр WebResource, указывающий на основной URI нашего веб-сервиса, мы должны создать новый экземпляр, указывающий на URI конкретного веб-сервиса в качестве цели, как определено в его аннотации @Path. Мы можем сделать это, просто вызывая метод path() на WebResource и передавая значение, соответствующее содержанию аннотации @Path нашего веб-сервиса RESTful.

У класса WebResource есть метод type(), который возвращает экземпляр реализации класса com.sun.jersey.api.client.UniformInterface. Метод type() принимает в качестве параметра строку (String), которая может использоваться для указания MIME-типа, обрабатываемого веб-сервисом. У класса javax.ws.rs.core.MediaType имеется несколько предопределенных строковых констант, соответствующих наиболее распространенным поддерживаемым MIME-типам. В нашем примере мы использовали XML, поэтому в качестве значения этого метода была указана соответствующая константа MediaType.TEXT_XML.

У UniformInterface есть методы, которые мы можем вызвать, чтобы сгенерировать HTTP-запросы GET, POST, PUT и DELETE. Эти методы, соответственно, названы так: get(), post(), put() и delete().

В методе getCustomer() в вышеприведенном примере мы вызываем метод get(), который генерирует GET-запрос к нашему веб-сервису. Обратите внимание, что мы передаем класс Java такого типа данных, который ожидаем получать. JAXB автоматически заполняет экземпляр этого класса данными, возвращаемыми из веб-сервиса.

В методе `insertCustomer()` мы вызываем метод `put()` на реализации `UniformInterface`, возвращенной методом `WebResource.type()`. Мы передаем экземпляр нашего класса `Customer`, который JAXB автоматически преобразует в XML, прежде чем отправить его серверу. Тот же самый метод может использоваться при вызове методов `post()` и `delete()` на реализации `UniformInterface`.

Параметры запроса и пути

В нашем предыдущем примере мы работали с веб-сервисом RESTful для управления единственным объектом заказчика. Очевидно, что на практике это не имеет особого смысла. В общем случае веб-сервис RESTful разрабатывается для обработки коллекции объектов (в нашем примере – заказчиков). Чтобы определить, с каким конкретно объектом в наборе мы работаем, можно передавать нашим веб-сервисам RESTful параметры. Существуют два типа параметров, которые мы можем использовать: параметры запроса и параметры пути.

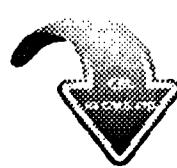
Параметры запроса

Мы можем добавить параметры в методы, которые обрабатывают HTTP-запросы в нашем веб-сервисе. Параметры, декорированные аннотацией `@QueryParam`, будут получены из URL запроса.

Следующий пример поясняет, как использовать параметры запроса в нашем JAX-RS веб-сервисе 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 реальном RESTful-сервисе мы будем извлекать данные
        // из базы данных, используя предоставленный идентификатор (id).
        System.out.println("--- " + this.getClass().getCanonicalName()
            + " вызван метод .getCustomer(), id = " + id);
        return customer;
    }
}
```



```

/**
 * Создание нового заказчика - customer.
 * @param customer - данные создаваемого customer в XML-представлении.
 */
@PUT
@Consumes("text/xml")
public void createCustomer(Customer customer)
{
    // В реальном RESTful-сервисе мы будем вначале разбирать XML-данные,
    // полученные в XML-параметре customer, а затем на их основе
    // вставим новую строку в базу данных.
    System.out.println("--- " + this.getClass().getCanonicalName()
        + " вызван метод .createCustomer()");
    System.out.println("заказчик = " + customer);
}
@POST
@Consumes("text/xml")
public void updateCustomer(Customer customer)
{
    // В реальном RESTful-сервисе мы будем вначале разбирать XML-данные,
    // полученные в XML-параметре customer, а затем обновим этими
    // данными строку в базе данных.
    System.out.println("--- " + this.getClass().getCanonicalName()
        + " вызван метод .updateCustomer()");
    System.out.println("заказчик = " + customer);
    System.out.println("заказчик = " + customer);
}
@DELETE
@Consumes("text/xml")
public void deleteCustomer(@QueryParam("id") Long id)
{
    // В реальном RESTful-сервисе мы будем вызывать DAO
    // и удалять строку в базе данных, передавая первичный ключ
    // в качестве параметра "id".
    System.out.println("--- " + this.getClass().getCanonicalName()
        + " вызван метод .deleteCustomer(), id = " + id);
    System.out.println("заказчик = " + customer);
}
}

```

Обратите внимание, что нам всего лишь нужно декорировать параметры аннотацией `@QueryParam`. Она позволяет JAX-RS получать любые параметры запроса, соответствующие значению аннотации, и присваивать ее значение переменной параметра.

Можно добавить параметр в URL веб-сервиса точно так же, как мы передаем любые параметры в URL¹:

```

curl -XGET -HContent-type:text/xml
http://localhost:8080/queryparams/resources/customer?id=1

```

Отправка параметров запроса через клиентский API Jersey

Клиентский API Jersey предоставляет легкий и прямой способ отправки параметров запроса веб-сервисам RESTful. Следующий пример поясняет, как это сделать:

¹ Все должно быть записано в одну строку через пробел. – Прим. перев.

```
package com.ensode.queryparamsclient;

import com.ensode.queryparamsclient.entity.Customer;
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.UniformInterface;
import com.sun.jersey.api.client.WebResource;
import javax.ws.rs.core.MediaType;
public class App
{
    private WebResource baseUriWebResource;
    private WebResource webResource;
    private Client client;
    private static final String BASE_URI =
        "http://localhost:8080/queryparams/resources";
    public static void main(String[] args)
    {
        App app = new App();
        app.initWebResource();
        app.getCustomer();
    }
    private void initWebResource()
    {
        com.sun.jersey.api.client.config.ClientConfig config =
            new com.sun.jersey.api.client.config.DefaultClientConfig();
        client = Client.create(config);
        baseUriWebResource = client.resource(BASE_URI);
        webResource = baseUriWebResource.path("customer");
    }
    public void getCustomer()
    {
        UniformInterface uniformInterface =
            webResource.type(MediaType.TEXT_XML);
        Customer customer = (Customer) webResource.queryParam("id", "1").
            get( Customer.class );
        System.out.println("заказчик = " + customer);
    }
}
```

Как видно из приведенного кода, для передачи параметра нам нужно вызвать метод `queryParam()` на `com.sun.jersey.api.client.WebResource`. Первый параметр этого метода является названием параметра и должен соответствовать значению аннотации `@QueryParam` веб-сервиса. Второй параметр представляет собой значение, которое мы должны передать веб-сервису.

Если необходимо передать несколько параметров одному из методов нашего веб-сервиса, то мы должны использовать экземпляр класса, реализующего интерфейс `javax.ws.rs.core.MultivaluedMap`. Jersey предоставляет реализацию по умолчанию в форме `com.sun.jersey.core.util.MultivaluedMapImpl`, которая должна удовлетворять большинству случаев.

Следующий фрагмент кода поясняет, как передать несколько параметров методу веб-сервиса:

```
MultivaluedMap multivaluedMap = new MultivaluedMapImpl();
multivaluedMap.add("paramName1", "value1");
multivaluedMap.add("paramName2", "value2");
String s =
    webResource.queryParams(multivaluedMap).get(String.class);
```

Мы должны добавить все параметры, которые требуется отправить нашему веб-сервису, вызывая метод `add()` на реализации `MultivaluedMap`. Этот метод принимает название параметра в качестве своего первого параметра и значение параметра – в качестве второго. Необходимо вызвать данный метод для каждого параметра, который мы планируем отправить.

Как мы видим, у `com.sun.jersey.api.client.WebResource` имеется метод `queryParams()`, который принимает экземпляр класса, реализующего интерфейс `MultivaluedMap` в качестве параметра. Чтобы отправить несколько параметров нашему веб-сервису, мы просто должны передать экземпляр `MultivaluedMap`, содержащий все обязательные параметры, этому методу.

Параметры пути

Другой способ передачи параметров нашим веб-сервисам RESTful – использование параметров пути. Следующий пример поясняет, как разработать JAX-RS веб-сервис RESTful, который принимает параметры пути:

```
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.rs.PathParam;
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 реальном RESTful-сервисе мы будем извлекать данные
        // из базы данных с помощью ключа передаваемого в параметре "id".
        System.out.println("--- " + this.getClass().getCanonicalName()
            + " вызван метод .getCustomer(), id = " + id);
        return customer;
    }
    @PUT
    @Consumes("text/xml")
    public void createCustomer(Customer customer)
    {
        // В реальном RESTful-сервисе мы будем разбирать XML-данные
        // полученные в XML-параметре customer, а затем вставим
        // новую строку из полученных данных - в базу данных.
        System.out.println("--- " + this.getClass().getCanonicalName()
            + " вызван метод .createCustomer()");
```

```
        System.out.println("заказчик = " + customer);
    }
@POST
@Consumes("text/xml")
public void updateCustomer(Customer customer)
{
    // В реальном RESTful-сервисе мы будем разбирать XML-данные,
    // полученные в XML-параметре customer, а затем обновим этими
    // данными строку в базе данных.
    System.out.println("--- " + this.getClass().getCanonicalName()
        + " вызван метод .updateCustomer()");
    System.out.println("заказчик = " + customer);
    System.out.println("заказчик = " + customer);
}
@DELETE
@Consumes("text/xml")
@Path("{id}/")
public void deleteCustomer(@PathParam("id") Long id)
{
    // В реальном RESTful-сервисе мы будем вызывать
    // DAO и удалять строку в базе данных с помощью
    // первичного ключа, передаваемого в качестве параметра "id".
    System.out.println("--- " + this.getClass().getCanonicalName()
        + " вызван метод .deleteCustomer(), id = " + id);
    System.out.println("заказчик = " + customer);
}
}
```

Любой метод, который принимает параметр пути, должен быть декорирован аннотацией `@Path`. Атрибут значения этой аннотации должен записываться в формате "`{paramName}/`", где `paramName` – параметр, который метод ожидает получать. Кроме того, параметры метода должны быть декорированы аннотацией `@PathParam`. Значение этой аннотации должно соответствовать названию параметра, объявленному в аннотации `@Path` для метода.

Мы можем передать параметры пути из командной строки путем изменения URI нашего веб-сервиса в соответствующих случаях. Например, мы можем передать 1 в качестве параметра `"id"` предыдущему методу `getCustomer()` (который обрабатывает HTTP-запросы GET) из командной строки следующим образом¹:

```
curl -XGET -HContent-type:text/xml
http://localhost:8080/pathparams/resources/customer/1
```

Эта команда возвращает XML-представление объекта `Customer`, возвращаемого методом `getCustomer()`. При этом ожидаемый вывод будет таким:

```
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?><customer><firstName>William</firstName><id>1</id>
<lastName>Graystone</lastName><middleName>Daniel</middleName></
customer>
```

¹ Все должно быть записано в одну строку через пробел. Прим. перев.

Отправка параметров пути через клиентский API Jersey

Отправка параметров пути веб-сервису через клиентский API Jersey производится просто: мы должны добавить любые параметры пути к тому пути, который используется для создания нашего экземпляра WebResource.

Следующий пример поясняет, как это сделать:

```
package com.ensode.queryparamsclient;

import com.ensode.queryparamsclient.entity.Customer;
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.WebResource;
public class App
{
    private WebResource baseUriWebResource;
    private WebResource webResource;
    private Client client;
    private static final String BASE_URI =
        "http://localhost:8080/queryparams/resources";
    public static void main(String[] args)
    {
        App app = new App();
        app.initWebResource();
        app.getCustomer();
    }
    private void initWebResource()
    {
        com.sun.jersey.api.client.config.ClientConfig config =
            new com.sun.jersey.api.client.config.DefaultClientConfig();
        client = Client.create(config);
        baseUriWebResource = client.resource(BASE_URI);
        webResource = baseUriWebResource.path("customer/1");
    }
    public void getCustomer()
    {
        Customer customer = (Customer) webResource.get(Customer.class);
        System.out.println("заказчик = " + customer);
    }
}
```

В этом примере мы просто добавляли значение 1 в качестве параметра пути к строке, используемой для построения реализации WebResource, который используется для вызова нашего веб-сервиса. Этот параметр автоматически принимается API JAX-RS и присваивается соответствующему параметру метода, декорированному аннотацией `@PathParam`.

Если нам необходимо передать несколько параметров одному из наших веб-сервисов, мы просто должны использовать следующий формат параметра `@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 веб-сервис может быть вызван путем изменения URI веб-сервиса для передачи параметров в порядке, указанном в аннотации @Path. Например, следующий URI передал бы 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 через клиента веб-сервиса, которого мы разработали с помощью клиентского API Jersey.

Резюме

В этой главе мы обсудили, как быстро разработать веб-сервисы RESTful, используя JAX-RS – новое дополнение к спецификации Java EE.

Мы показали процесс разработки веб-сервиса с добавлением нескольких простых аннотаций в наш код. Также мы объяснили, как автоматически преобразовать данные из Java в XML и обратно, используя возможности API Java для связывания с XML (JAXB).

И наконец, мы рассмотрели передачу параметров нашим веб-сервисам RESTful через аннотации @PathParam и @QueryParam.

Приложение А

Отправка электронной почты из приложений Java EE

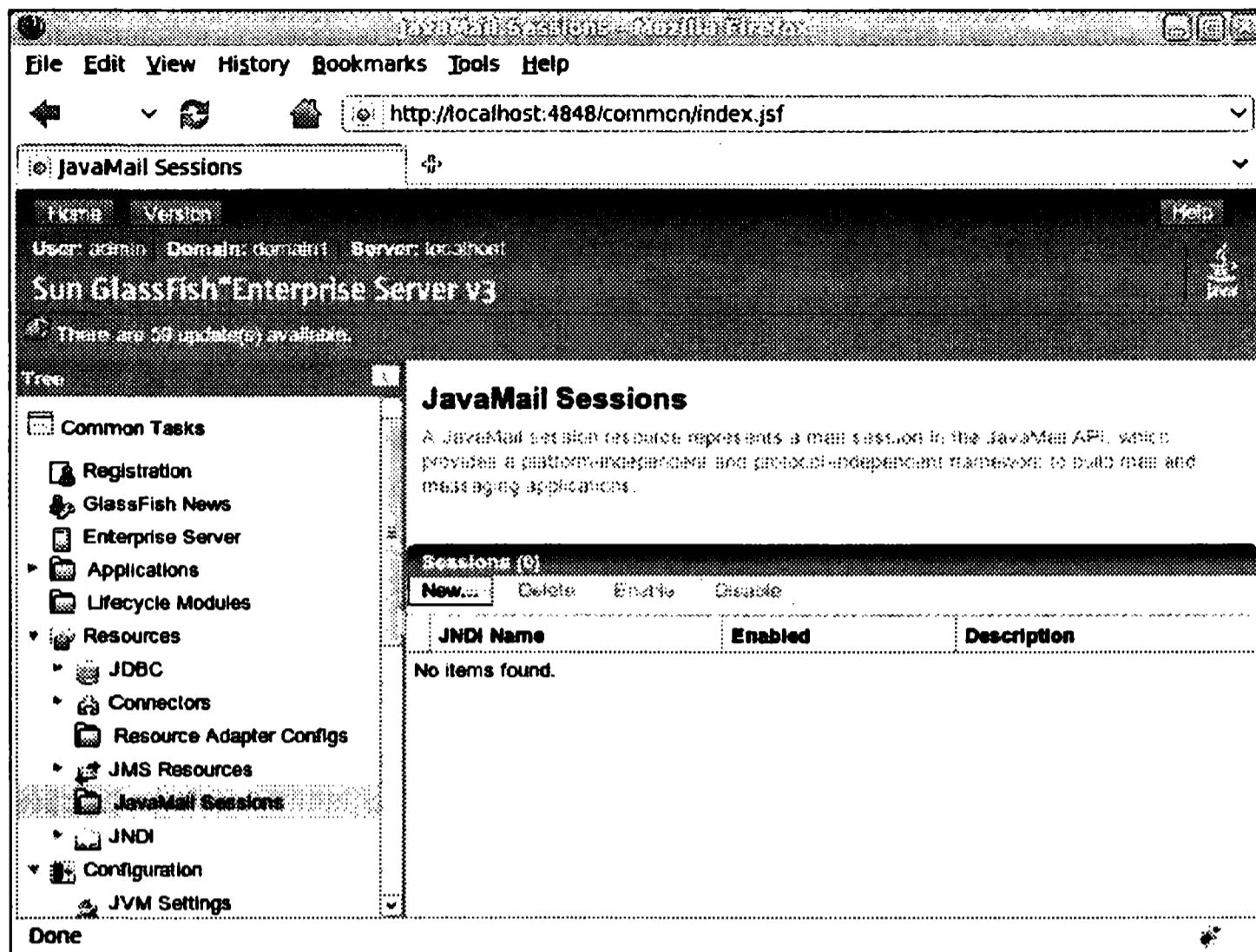
Приложения, развернутые на сервере GlassFish или любом другом Java EE-совместимом сервере приложений, часто должны предусматривать возможность отправки электронных писем. Благодаря API JavaMail, являющемуся частью спецификации Java EE, отправлять электронные письма из приложений Java EE довольно просто.

Чтобы реализовать возможность отправки электронных писем из приложения Java EE, мы должны иметь доступ к почтовому серверу. Обычно это реализуется за счет использования *Простого протокола передачи почты* (Simple Mail Transfer Protocol (SMTP)).

Конфигурирование сервера GlassFish

Прежде чем мы сможем отправлять электронные письма из наших приложений Java EE, нам понадобится произвести некоторое конфигурирование GlassFish. Вначале нужно добавить новый сеанс JavaMail, войдя в систему к веб-консоли администрирования GlassFish, развернув узел **Ресурсы** (Resources) в дереве панели навигации в левой части страницы и затем щелкнув по узлу **Сеансы JavaMail** (JavaMail Sessions).

Это можно увидеть на следующем снимке экрана:



Чтобы создать новый сеанс JavaMail, мы должны щелкнуть по кнопке **Новый...** (New...). При этом основная панель области экрана будет выглядеть, как показано на следующем снимке экрана:

The screenshot shows the "New JavaMail Session" dialog box. The title bar says "New JavaMail Session" with "OK" and "Cancel" buttons. The main area contains fields for configuration:

- JNDI Name:** `mail/myjavamail`
- Mail Host:** `mail.example.com`
ONC name of the default mail server
- Default User:** `mailadmin`
User name to provide when connecting to a mail server. Must contain only alphanumeric, underscore, dash, or dot characters
- Default Return Address:** `mailadmin@example.com`
E-mail address of the default user
- Description:**
Makes it easier to find this session later
- Status:** Enabled

Below this is an "Advanced" section with the following settings:

- Store Protocol:** `imap`
Same IMAP or POP3, default is IMAP
- Store Protocol Class:** `com.sun.mail.imap.IMAPStore`
Default is `com.sun.mail.imap.IMAPStore`
- Transport Protocol:** `smtp`
Default is SMTP
- Transport Protocol Class:** `com.sun.mail.smtp.SMTPTransport`
Default is `com.sun.mail.smtp.SMTPTransport`
- Debug:** Enabled

At the bottom are buttons for "Additional Properties (0)", "Add Property", and "Delete Properties".

В поле **Имя JNDI** (JNDI Name) необходимо ввести JNDI-имя для нашего сеанса JavaMail. Это имя, выбранное нами, должно быть допустимым и уникальным. Приложения будут использовать его для получения доступа к почтовому сеансу.

В поле **Почтовый сервер** (Mail Host) следует указать DNS-имя почтового сервера, который мы будем использовать для отправки электронных писем.

В поле **Пользователь по умолчанию** (Default User) мы должны указать имя пользователя по умолчанию, которое используется для соединения с почтовым сервером.

В поле **Обратный адрес по умолчанию** (Default Return Address) мы должны указать адрес электронной почты по умолчанию, который получатели электронной почты могут использовать для ответа на сообщения, отправленные нашими приложениями.



Указание поддельного обратного адреса

Обратный адрес по умолчанию не должен быть реальным адресом электронной почты; здесь мы можем указать недопустимый адрес электронной почты. Имейте в виду, что если мы это сделаем, то получатели будут не в состоянии ответить на электронные письма, посланные из наших приложений. В этой ситуации будет целесообразно включить в текст сообщения предупреждение о том, что получатели не смогут ответить на данное сообщение.

Кроме всего вышенназванного, мы можем добавить описание сеанса JavaMail в поле **Описание** (Description).

Флажок **Состояние** (Status) позволяет нам включать или отключать сеанс JavaMail. Отключенные сеансы недоступны для приложений.

Поле **Протокол хранилища** (Store Protocol) используется для указания значения протокола хранилища почтового сервера, который позволяет нашим приложениям получать электронные письма из него. Допустимые значения этого поля: **imap**, **imaps**, **pop3** и **pop3s**. Проконсультируйтесь со своим системным администратором для выбора правильного значения для Вашего сервера.



Протокол хранилища игнорируется, если приложения только посылают электронные письма

Как правило, наши приложения посыпают электронные письма гораздо чаще, чем получают их. Если все приложения, использующие наш почтовый сеанс, будут только посыпать электронные письма, то значение поля **Протокол хранилища** (Store Protocol) (равно как и поля **Класс протокола хранилища** (Store Protocol Class), о котором пойдет речь ниже), игнорируется.

В поле **Класс протокола хранилища** (Store Protocol Class) указывается класс реализации поставщика услуг, соответствующий указанному протоколу хранилища. Допустимые значения этого поля:

- **com.sun.mail.pop3.POP3Store** – для протокола хранилища **pop3**;
- **com.sun.mail.pop3.POP3SSLStore** – для протокола хранилища **pop3s**;
- **com.sun.mail.imap.IMAPStore** – для протокола хранилища **imap**;
- **com.sun.mail.imap.IMAPSSLStore** – для протокола хранилища **imaps**;

Поле **Транспортный протокол** (*Transport Protocol*) используется для указания значения транспортного протокола почтового сервера, по которому наши приложения смогут отправлять электронные письма. Допустимые значения этого поля – **smtp** и **smtpls**. Проконсультируйтесь со своим системным администратором, чтобы выбрать правильное значение для Вашего сервера.

В поле **Класс транспортного протокола** (*Transport Protocol Class*) указывается класс реализации поставщика услуг, соответствующий указанному транспортному протоколу. Допустимые значения этого поля:

- **com.sun.mail.smtp.SMTPTransport** – для транспортного протокола **smtp**;
- **com.sun.mail.smtp.SMTPSSLTransport** – для транспортного протокола **smtpls**.

Флажок **Отладка** (*Debug*) позволяет нам включать или отключать устранение неисправностей для сеанса JavaMail.

Если нам понадобится добавить дополнительные свойства в сеанс JavaMail, мы можем щелкнуть по кнопке **Добавить свойство** (*Add Property*) в конце страницы веб-консоли и ввести имя свойства и его значение в соответствующие поля.

После ввода всей необходимой информации для нашего сервера остается лишь щелкнуть по кнопке **OK** в верхней правой части основной панели страницы, чтобы создать сеанс JavaMail. Теперь созданный сеанс смогут использовать развернутые приложения.

Реализация функциональности доставки электронной почты

Получив настроенный сеанс JavaMail, как было показано в предыдущем разделе, мы можем довольно легко реализовать функциональность доставки электронной почты. Этот процесс поясняется следующим кодом:

```
package net.ensode.glassfishbook;

import javax.annotation.Resource;
import javax.faces.bean.ManagedBean;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.AddressException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
@ManagedBean
public class FeedbackBean
{
    private String subject;
    private String body;
    @Resource(name = "mymailserver")
    Session session;
    public String sendEmail()
    {
```



```

try
{
    Message msg = new MimeMessage(session);
    msg.setRecipient(Message.RecipientType.TO, new
        InternetAddress("customer@customerdomain.com"));
    msg.setSubject(subject);
    msg.setText(body);
    Transport.send(msg);
}
catch (AddressException e)
{
    e.printStackTrace();
    return "сбой";
}
catch (MessagingException e)
{
    e.printStackTrace();
    return "сбой";
}
return "успех";
}

public StringgetBody()
{
    return body;
}

public voidsetBody(String body)
{
    this.body = body;
}

public StringgetSubject()
{
    return subject;
}

public voidsetSubject(String subject)
{
    this.subject = subject;
}
}

```

Этот класс используется в качестве управляемого бина для простого приложения JSF. Для краткости не показаны другие части приложения, поскольку они не имеют отношения к функциональности электронной почты. Полное приложение может быть загружено с веб-сайта www.dmk-press.ru.

Первое, что нам нужно сделать, – инжектировать (ввести) экземпляр создаваемого сеанса JavaMail, как описано в предыдущем разделе, добавляя на уровне класса переменную типа `javax.mail.Session` и декорируя ее аннотацией `@Resource`. Значение атрибута `name` этой аннотации должно соответствовать JNDI-имени, которое мы дали нашему сеансу JavaMail в процессе его создания.

Затем мы должны создать экземпляр `javax.mail.internet.MimeMessage`, передавая объект сеанса в качестве параметра его конструктору.

Как только экземпляр `javax.mail.internet.MimeMessage` будет создан, нам следует добавить получателя сообщения, вызывая его метод `setRecipient()`. Первый параметр этого метода указывает получателя сообщения, которому оно адресовано (TO), список рассылки, где перечислены получатели копии этого сообщения

(carbon copied (CC)), или список рассылки по скрытым адресам, также содержащий копии получателей (blind carbon copied (BCC)). Мы можем указать тип получателя с помощью предопределенных констант `Message.RecipientType.TO`, `Message.RecipientType.CC` или `Message.RecipientType.BCC`. Второй параметр метода `setRecipient()` указывает адрес электронной почты получателя; этот параметр имеет тип `javax.mail.Address`. Поскольку данный класс является абстрактным, мы должны использовать один из его подклассов, а именно `javax.mail.internet.InternetAddress`. Конструктор для этого класса принимает строковый параметр, содержащий адрес электронной почты получателя. Метод `setRecipient()` может быть вызван многократно для добавления получателей, для выполнения отправки, копирования или включения в список рассылки. Для каждого типа получателя может быть указан только один адрес.

Если мы должны отправить сообщение нескольким получателям, можно использовать метод `addRecipients()` класса `javax.mail.Message` (или один из его подклассов, таких как `javax.mail.internet.MimeMessage`). Этот метод принимает тип приемника в качестве своего первого параметра и массив `javax.mail.Address` – в качестве второго. Сообщение будет отправлено всем получателям в массиве. При использовании этого метода вместо `setRecipient()` мы не ограничиваемся единственным типом получателя для получателей.

После указания одного или нескольких получателей следует добавить тему сообщения и текст, вызывая соответственно методы `setSubject()` и `setText()` на экземпляре сообщения.

Теперь мы готовы отправить наше сообщение. Для этого достаточно вызвать статический метод `send()` на классе `javax.mail.Transport`. Данный метод принимает экземпляр сообщения в качестве параметра.

Приложение Б

Интеграция с IDE

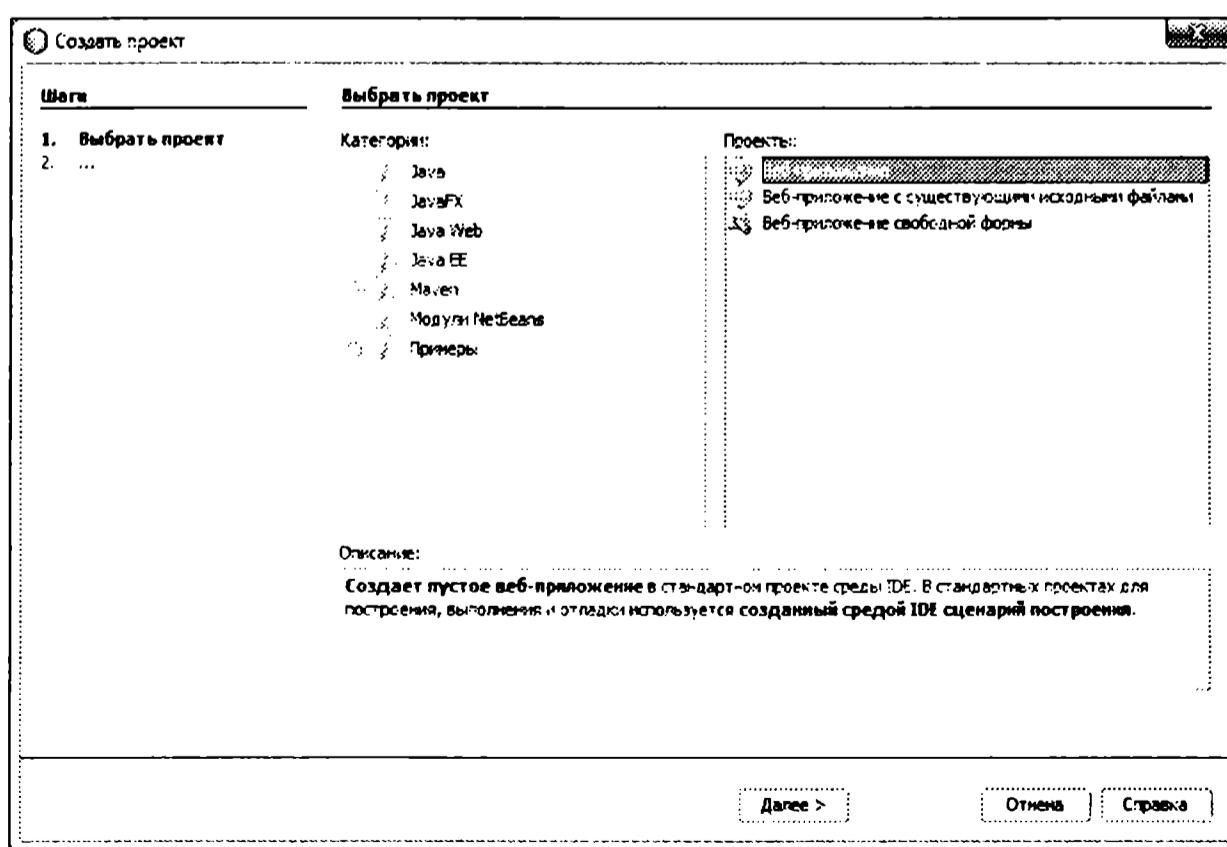
GlassFish предоставляет интеграцию с двумя самыми популярными IDE Java: NetBeans и Eclipse. NetBeans, будучи продуктом Oracle (ранее Sun Microsystems), точно так же как GlassFish, обеспечивает «коробочную» интеграцию с GlassFish. Oracle предоставляет для GlassFish плагин Eclipse – для интеграции с Eclipse, аналогично тому как Eclipse поставляется с плагином Eclipse Java IDE plus для интеграции с сервером GlassFish.

NetBeans

Все выпуски NetBeans Java содержат «коробочную» интеграцию с сервером GlassFish. При установке одного из этих выпусков NetBeans также устанавливается GlassFish.

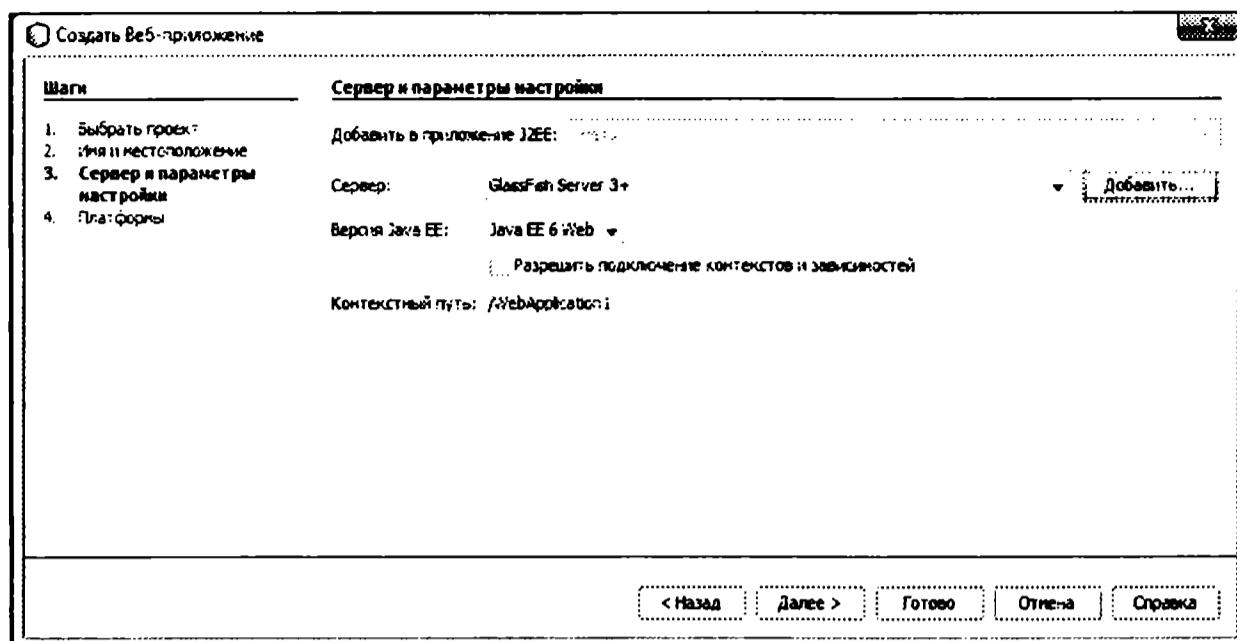
NetBeans может быть загружен с сайта http://netbeans.org/index_ru.html.

В NetBeans имеется несколько категорий создаваемых проектов; приложения Java EE могут быть созданы из категорий Java Web и Java EE.

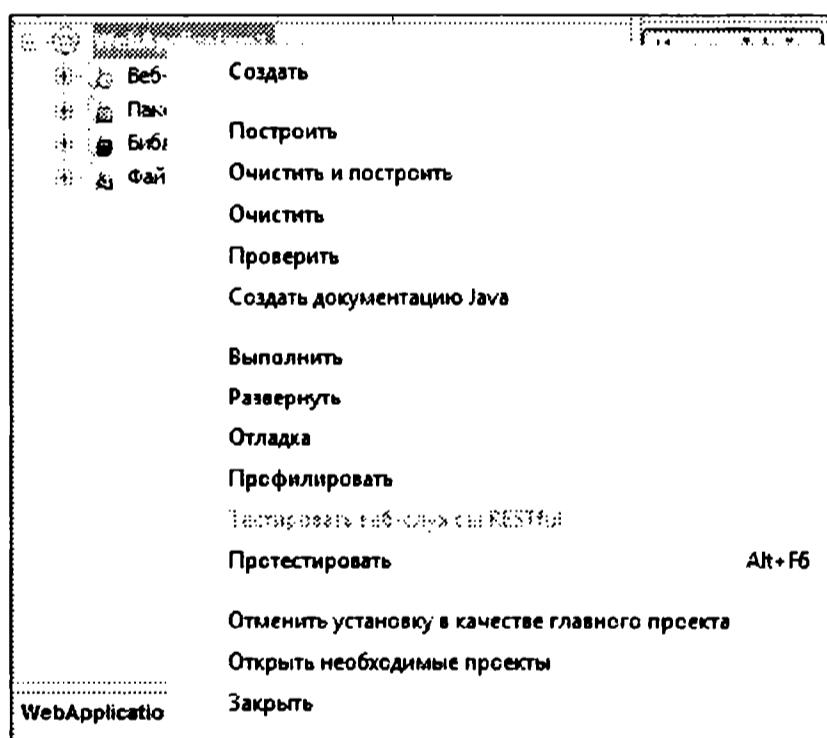


Для большинства типов проектов в категориях Java EE или Java Web среда NetBeans требует, чтобы мы выбрали сервер приложений, на котором будет развернут проект.

GlassFish помечается как **GlassFish Server 3+** в поле раскрывающегося списка выбора сервера:



После того как мы создадим проект и будем готовы развернуть его, достаточно всего лишь щелкнуть правой кнопкой по проекту и выбрать **Развернуть** (Deploy) из появившегося всплывающего меню:



Проект будет автоматически построен, упакован и развернут. Для веб-приложения у нас также имеются опции **Выполнить** (Run) и **Отладка** (Debug). Обе они, в дополнение к построению, упаковке и развертыванию проекта, автоматически открывают новое окно обозревателя и указывают на URL приложения. При выборе пункта **Отладка** (Debug) GlassFish будет запущен в режиме отладки, и мы сможем использовать отладчик NetBeans, чтобы устранить неисправности нашего проекта.

Кроме прочего, имеется функция NetBeans – автоматическое инкрементное развертывание. Она означает, что каждый раз, когда файл (управляемый бин, EJB, страница фреймворка и т. д.) сохраняется, он автоматически переразвертывается на сервере. Таким образом, наше развернутое приложение обновляется по мере его разработки в реальном масштабе времени. Тестирование наших изменений по большей части столь же просто, как перезагрузка текущей страницы в обозревателе, поскольку

сессия пользователя не теряется при выполнении повторного развертывания. Это очень удобная, экономящая время функция, сильно отличающаяся от стандартного цикла тестирования, подразумевающего сборку, упаковку, развертывание, которые мы должны осуществлять полностью на большинстве других серверов приложений Java EE.

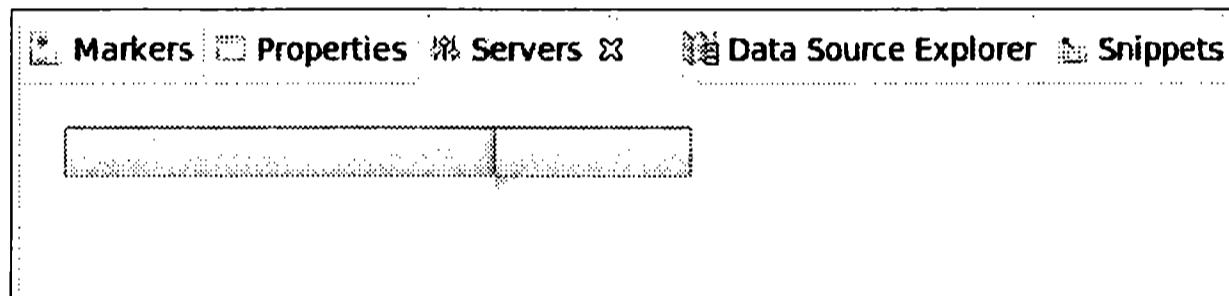
Eclipse

В отличие от NetBeans, Eclipse не поставляется с «коробочной» поддержкой сервера GlassFish. К счастью, в нем очень легко добавить поддержку GlassFish. Eclipse может быть загружен по адресу <http://www.eclipse.org/>.

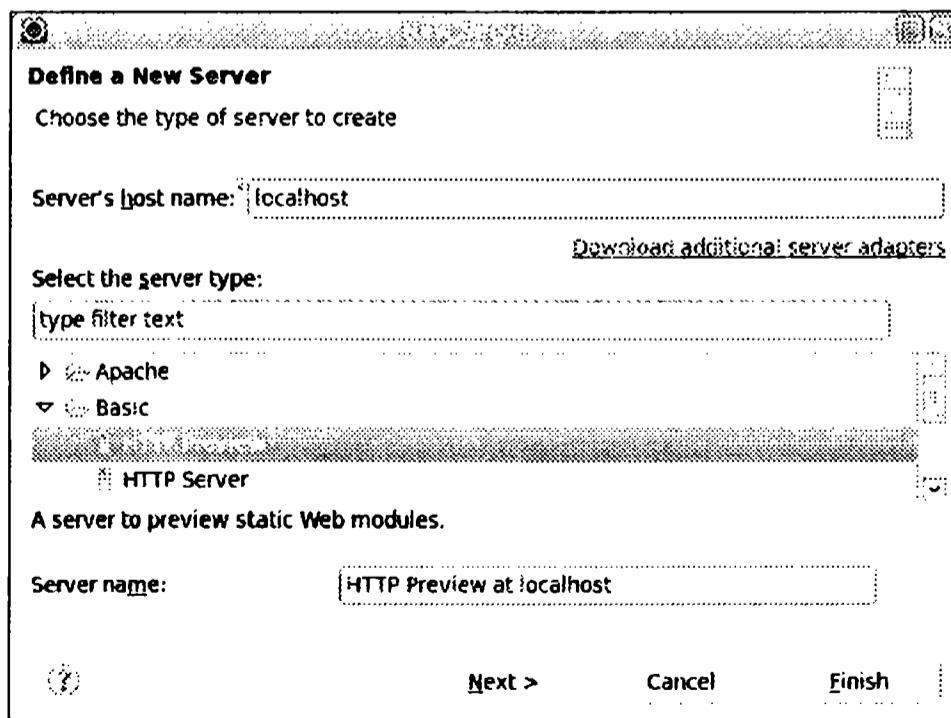


В этом разделе мы предполагаем, что устанавливается IDE Eclipse для разработчиков Java EE. Эта версия Eclipse включает инструменты для разработки элементов Java EE (JSF, JPA, EJB и т. д.).

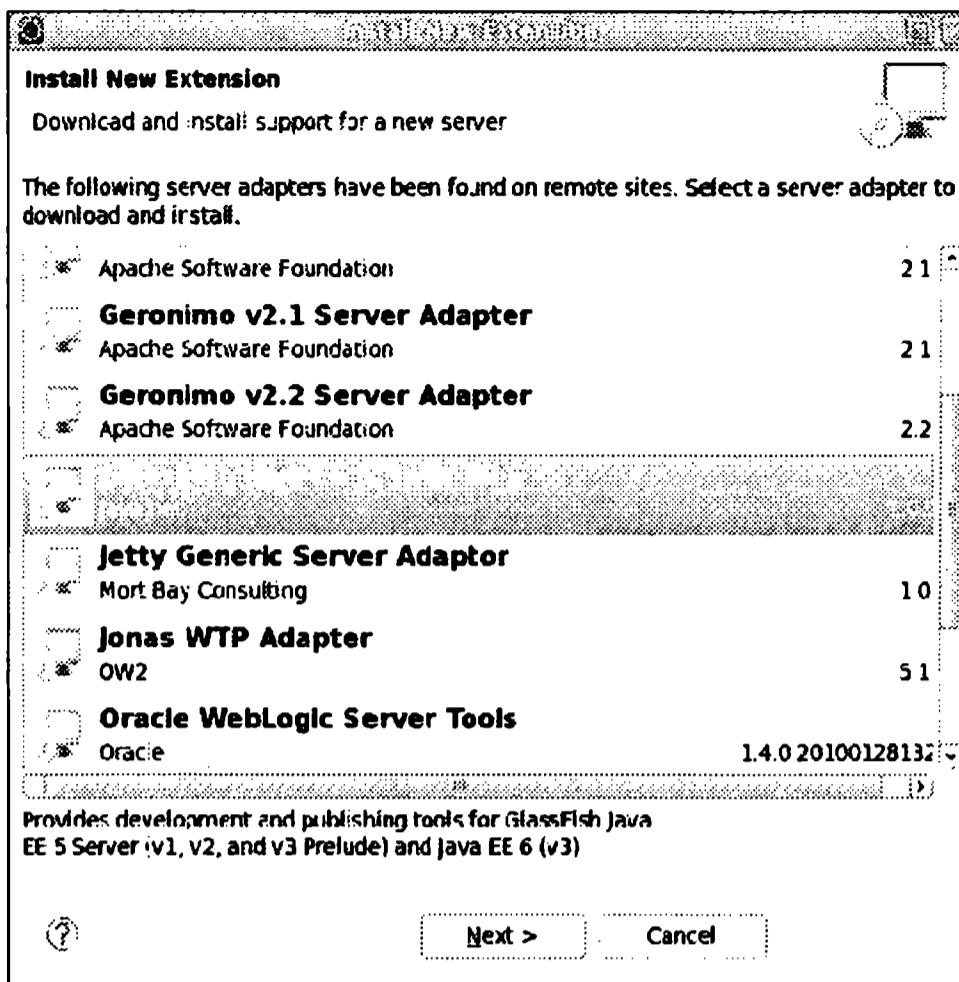
Чтобы интегрировать Eclipse и GlassFish, нам нужно загрузить адаптер сервера GlassFish для Eclipse. С этой целью необходимо щелкнуть правой кнопкой по вкладке **Серверы (Servers)** внизу перспективы Java EE и выбрать **Новый (New) | Сервер (Server)**:



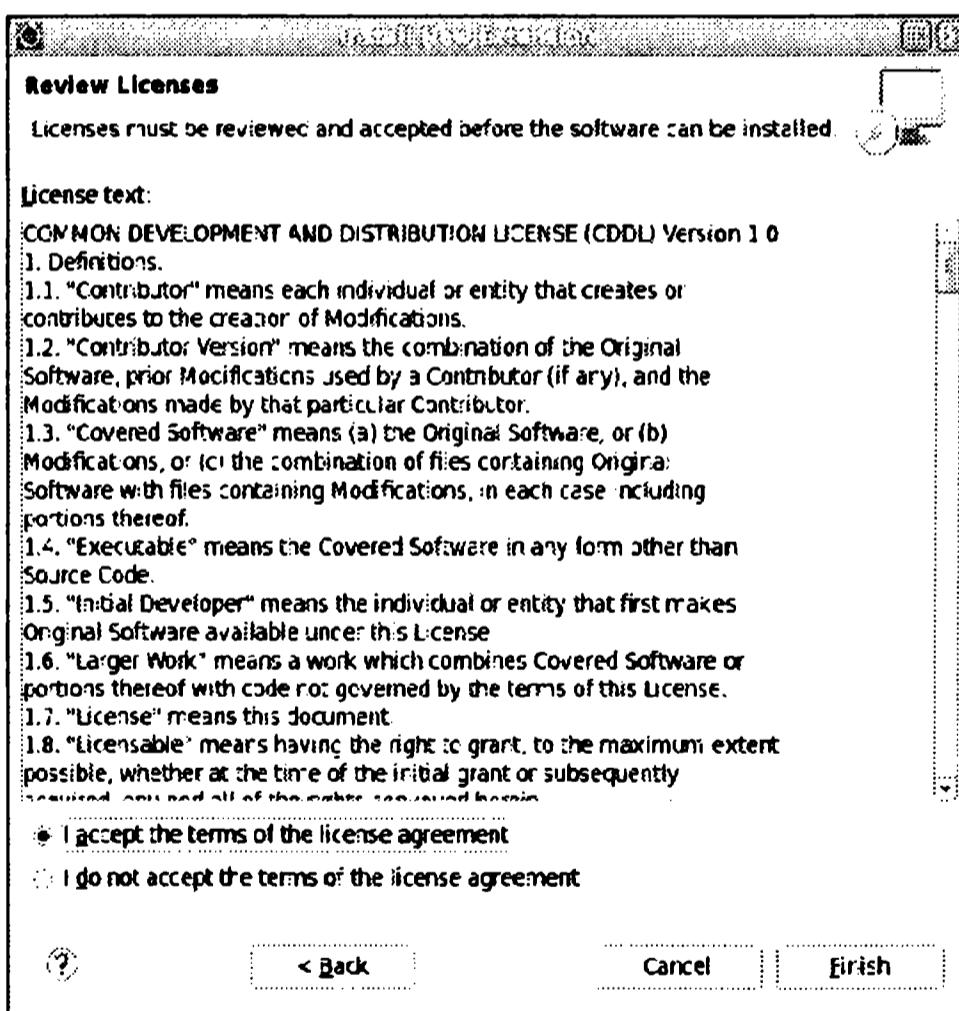
Затем мы должны щелкнуть по ссылке **Загрузка дополнительных адаптеров серверов (Download additional server adapters)** в открывшемся окне:



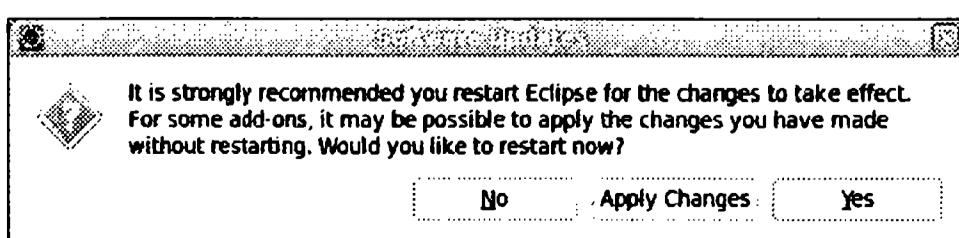
Как только это будет выполнено, мы увидим список всех доступных адаптеров серверов:



Необходимо выбрать в этом списке GlassFish и щелкнуть по кнопке Далее (Next).



Теперь мы должны принять лицензионное соглашение и щелкнуть по кнопке Завершить (Finish).

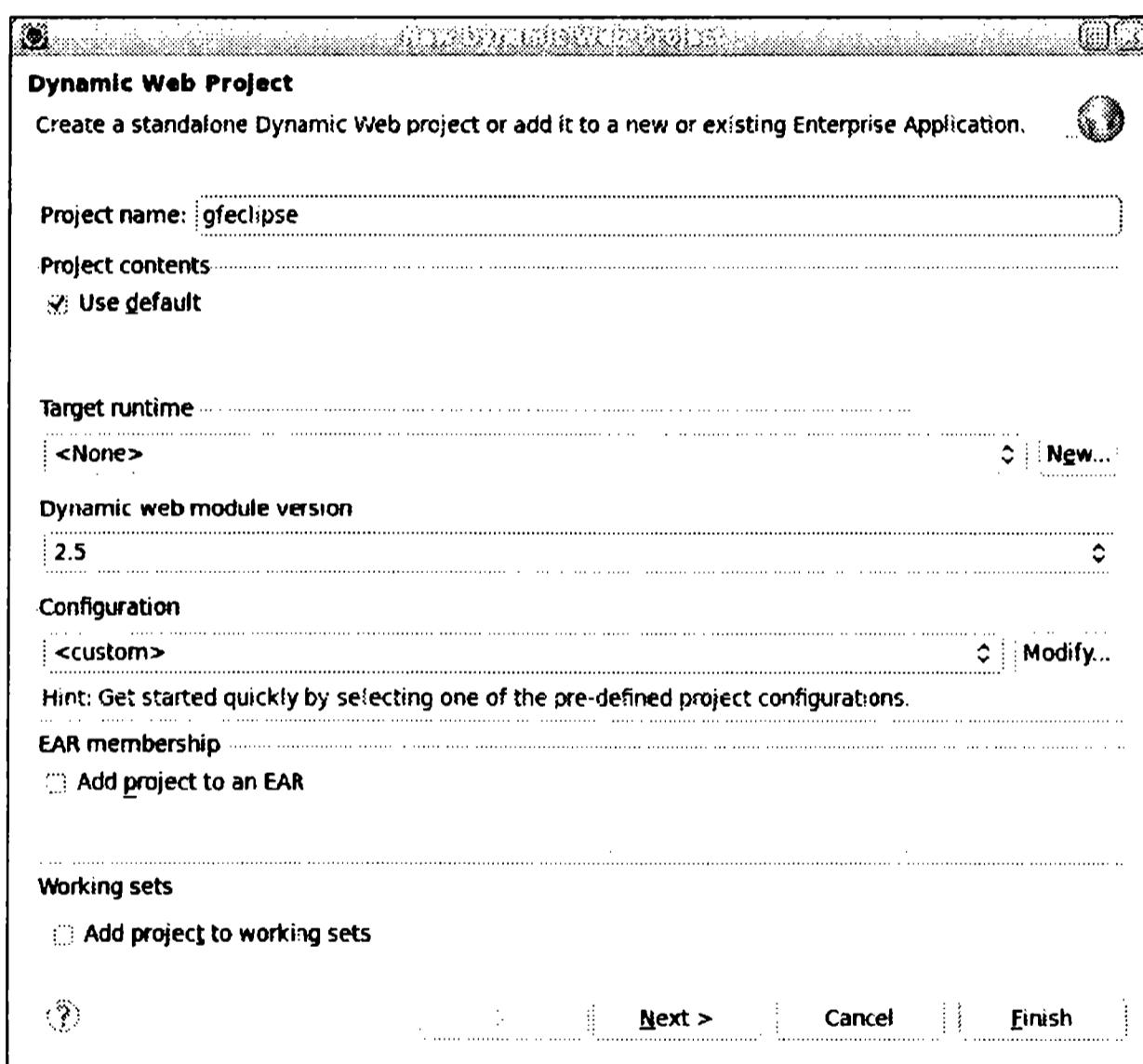


После того как адаптер сервера GlassFish для Eclipse будет загружен, мы увидим на экране сообщение о том, что настоятельно рекомендуется перезапустить Eclipse. Это разумный совет, и нужно ему последовать.

После перезапуска Eclipse адаптер сервера GlassFish будет полностью установлен, и мы будем готовы развернуть наши приложения на сервере GlassFish прямо из Eclipse.

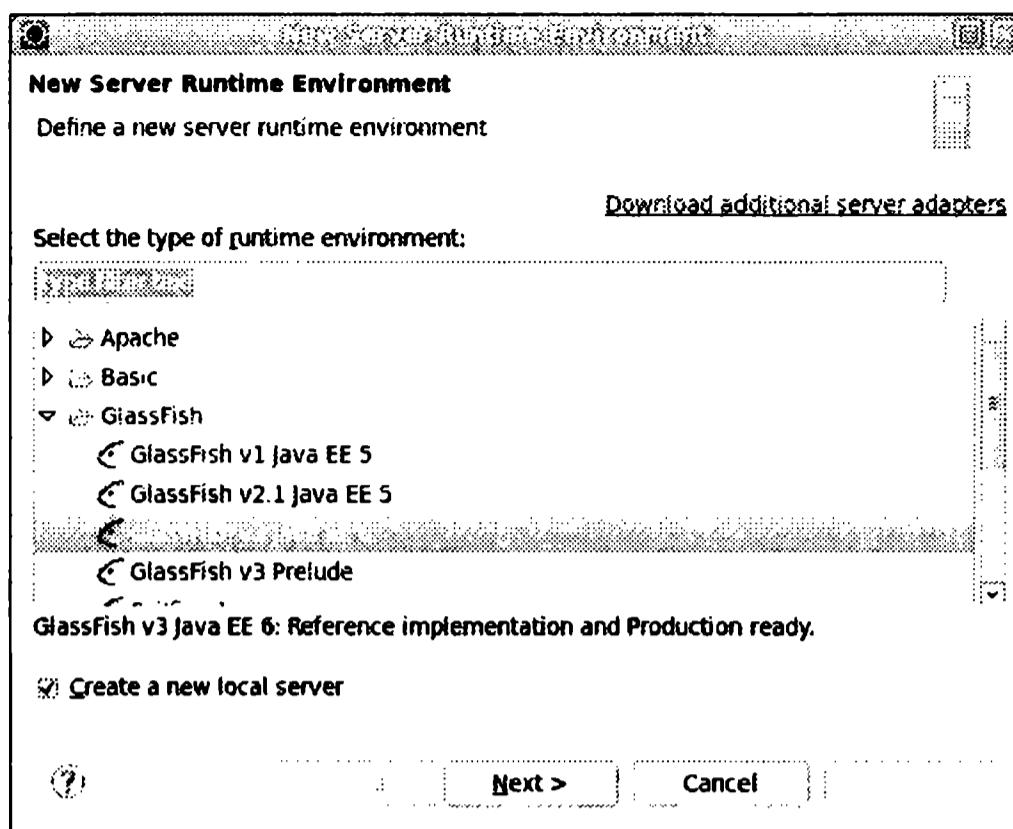
Чтобы продемонстрировать интеграцию между GlassFish и Eclipse, мы используем проект типа «динамический веб-проект» (Dynamic Web Project), однако стоит отметить, что эта процедура аналогична и для других типов проектов Java EE.

Для создания нового динамического веб-проекта следует выбрать пункты меню **Файл (File) | Новый (New) | Динамический веб-проект (Dynamic Web Project)**.

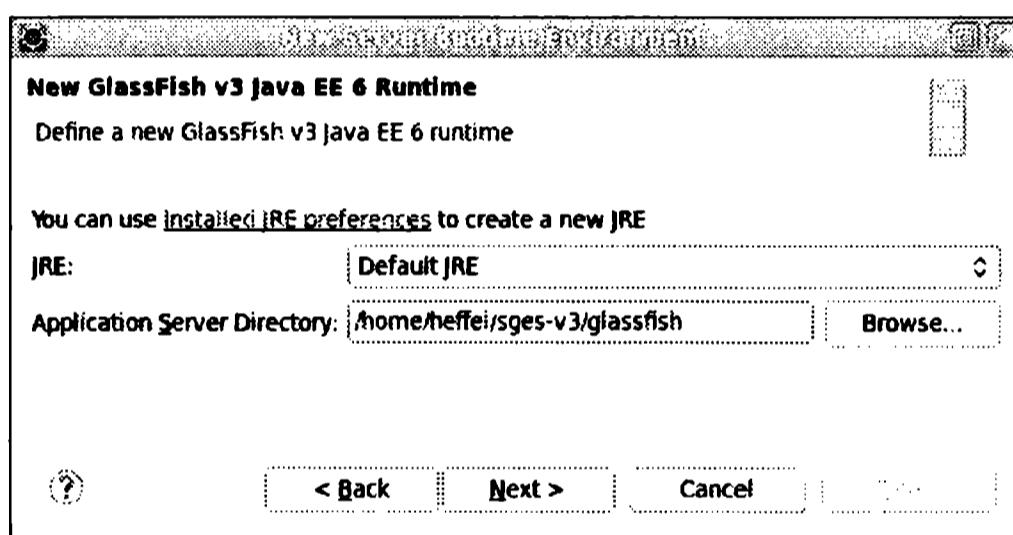


При создании динамического веб-проекта Eclipse среди прочего спросит нас о *целевой исполняющей среде* (target runtime) для проекта. Целевой исполняющей средой для сервера приложений Java EE будет Eclipse Speck.

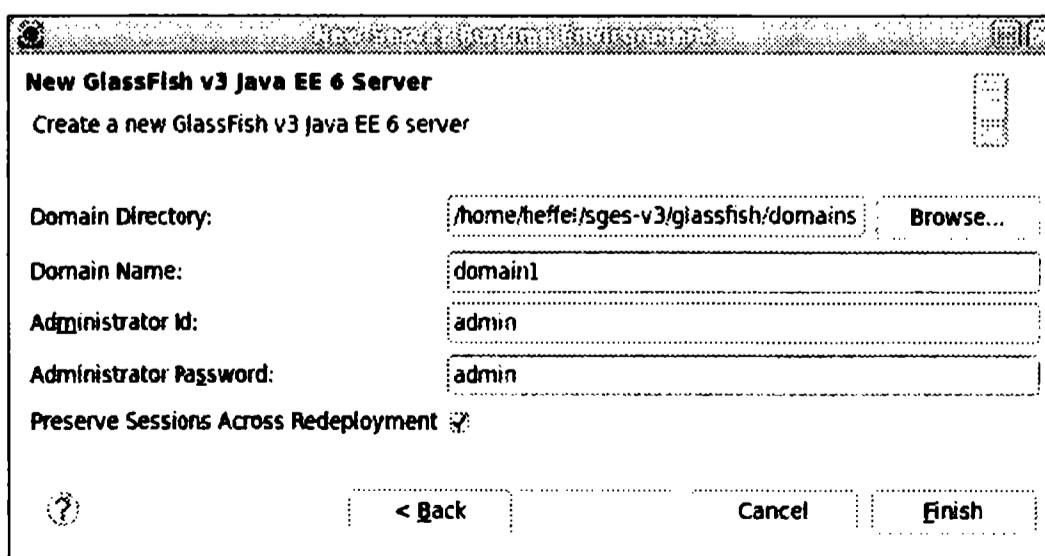
Чтобы выбрать GlassFish в качестве целевой исполняющей среды, щелкнем по кнопке **Новая... (New...)**, затем выберем **GlassFish v3 Java EE 6** из папки **GlassFish**.



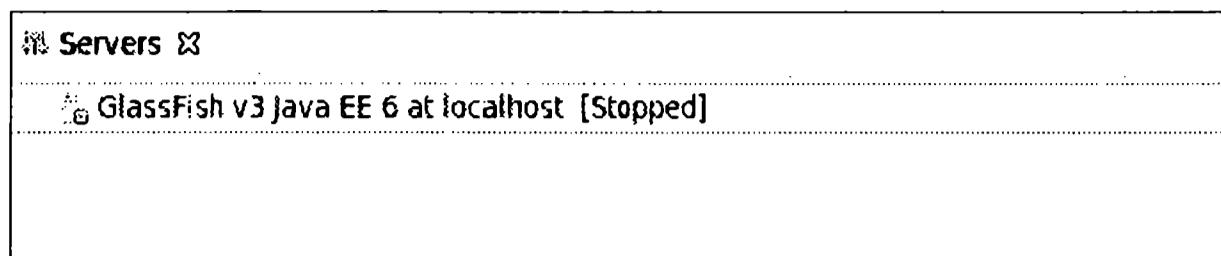
После щелчка по кнопке **Далее (Next)** нужно указать каталог, где установлен сервер GlassFish:



Затем мы должны указать каталог домена, наименование домена, идентификатор и пароль администратора для нашего домена. В большинстве этих полей установлены вполне разумные значения по умолчанию, которые можно не менять. Чаще всего единственное, что нам нужно ввести самостоятельно, – это пароль.

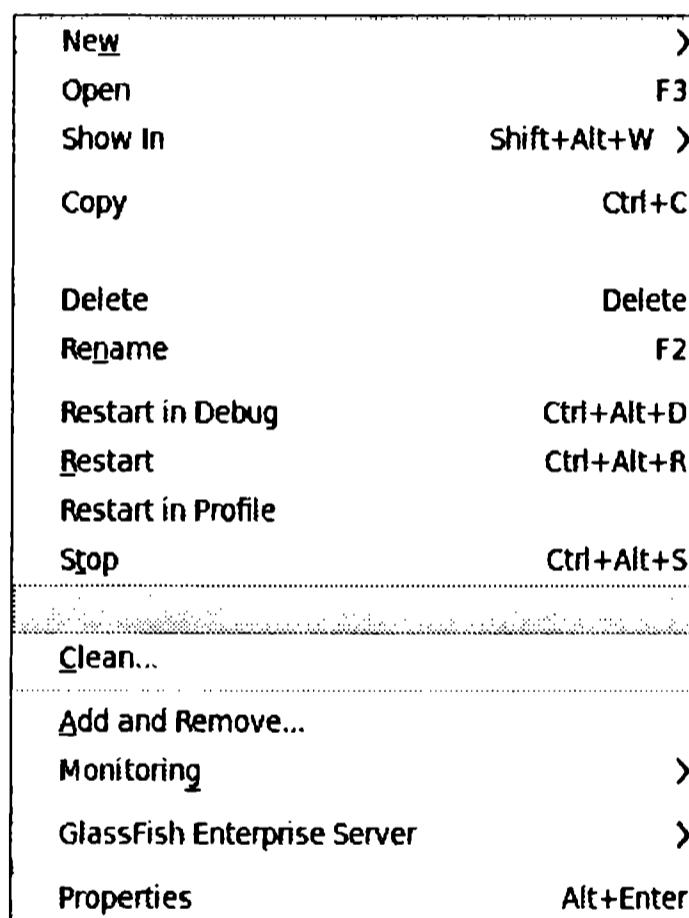


После того как мы завершим создание нашего проекта, GlassFish появится на панели **Серверы** (Servers), которая обычно находится внизу экрана:



Если панель **Серверы** не видна, можно открыть ее щелчком по элементам основного меню: **Окно** (Window) | **Показать панель** (Show View) | **Серверы** (Servers).

Теперь мы готовы приступить к разработке приложения. Когда мы перейдем к этапу, на котором требуется развернуть приложение, нам понадобится щелкнуть по значку сервера GlassFish на панели **Серверы** (Servers) и выбрать пункт **Опубликовать** (Publish):



В результате Eclipse создаст, упакует и развернет приложение.

Что касается веб-приложений, мы сможем выполнить приложение, как только оно будет развернуто, щелкнув правой кнопкой по проекту и выбирая пункты **Выполнить как** (Run As) | **Выполнить на сервере** (Run on Server).

По этой команде Eclipse создаст, упакует, развернет приложение и откроет его в окне обозревателя, встроенным в Eclipse.

Устранить неисправности приложения с помощью отладчика Eclipse можно, щелкнув правой кнопкой по проекту и выбрав пункты **Отладить как** (Debug As) | **Отладка на сервере** (Debug on Server). Это позволит Eclipse при необходимости запускать или перезапускать GlassFish в режиме отладки и развертывать приложение. Затем мы сможем устранить его неисправности, воспользовавшись встроенным отладчиком Eclipse.

Алфавитный указатель

!. См. Операторы: !.

@ См. Аннотация.

#{} 88, 111.

<....>. См. Тег.

\${} 110–111.

\${<имя бина>.<имя свойства>} 110.

{0}. См. Параметр: замена.

”{paramName}” 381. См. также Параметр: пути.

A

Авторизация 261, 280.

Аннотация

@ApplicationPath 368, 369–370.

@ApplicationScoped 199, 339.

@Asynchronous 307, 309.

@Column 155, 161.

@Consumes 366, 368.

@ConversationScoped 339, 342.

@CustomScoped 200.

@DELETE 366, 368.

@Dependent 339.

@EJB 302, 306, 330.

@Entity 154.

@FacesConverter 224.

@FacesValidator 202, 229.

@GeneratedValue 213–214.

@GET 366, 368.

@Id 155, 161, 176.

@IdClass 176.

@Inject 334, 335.

@JoinColumn 161, 170.

@JoinTable 170–171.

@Local 301.

@ManagedBean 188, 199, 205, 213.

@ManagedBeanProperty 216.

@ManagedProperty 212.

@ManyToMany 168, 170, 171.

@ManyToOne 167.

@MessageDriven 309, 310.

@Named 332–333.

@NamedQuery 180.

@NoneScoped 200.

@NotNull 185, 228.

@OneToMany 165.

@OneToOne 161.

@Path 366, 368, 376, 381, 383.

@PathParam 381–382.

@Pattern 228.

@PersistenceUnit 157, 159.

@POST 366, 368.

@PostActivate 318, 319.

@PostConstruct 306, 319.

@PreDestroy 318, 319.

@PrePassivate 318, 319.

@Produces 368.

@PUT 366, 368.

@Qualifier 336.

@QueryParam 377, 378, 379.

@Remote 301.

@Remove 318.

@RequestScoped 199, 338.

@Resource 152, 157, 247, 256, 322, 334, 388.

@RolesAllowed 328, 363.

@Schedule 325.

@SessionScoped 199, 339.

@Size 185.

@Stateless 300, 301, 359.

@Table 155.

@Timeout 323.

- @TransactionAttribute 311, 312.
значения
 TransactionAttributeType.MANDATORY 311.
 TransactionAttributeType.NEVER 311.
 TransactionAttributeType.NOT_SUPPORTED 311.
 TransactionAttributeType.REQUIRED 311.
 TransactionAttributeType.REQUIRES_NEW 312.
- @TransactionManagement 315.
значения
 TransactionManagementType.BEAN 315.
 TransactionManagementType.CONTAINER 315.
- @ViewScoped 200.
@WebFilter 74–76.
@WebInitParam 73, 76.
@WebListener 76.
@WebMethod 347.
@WebService 346, 347, 359.
@WebServiceRef 355.
@WebServlet 71–72.
@XmlRootElement 372–374.
- Архитектура Java для связывания с XML (JAXB) 2.2. См. Java Architecture for XML Binding (JAXB) 2.2.
- Асинхронный JavaScript и XML.
См. Ajax.
- Атрибут 87.
 action 188, 198, 217, 370.
 ошибка использования 200.
 actionListener 220.
 asyncSupported 83.
 autoFlush 87.
 basename 129, 225.
 begin 124.
 buffer 87.
 charset 87.
 class 99.
 columnClasses 191.
 columns 191, 236.
 contentType 87.
 converterId 224.
 datasource 133, 134.
- dateStyle 130.
 dayOfMonth 325.
 dayOfWeek 325.
 deferredSyntaxAllowedAsStringLiteral 88.
 delims 124.
 dir 236.
 doc 137.
 end 124.
 errorPage 88.
 escape 236.
 event 219.
 execute 220, 222.
 extends 88.
 file 102.
 filterName 75.
 flush 103.
 for 234.
 globalOnly 235.
 hour 325.
 href 236.
 id 99, 234.
 import 88, 110.
 info 88.
 initParams 76.
 inverseJoinColumns 170.
 isELIgnored 89.
 isErrorPage 89, 90.
 isThreadSafe 89.
 items 124.
 joinColumns 170.
 key 129.
 label 107, 192.
 lang 236.
 language 87, 89.
 library 188, 190.
 listener 224.
 mappedBy 162, 167.
 mappedName 247, 257, 309.
 maximum 196–197, 228.
 method 191.
 minimum 196–197, 228.
 minute 325.
 month 325.
 name 99, 170, 180, 190, 223, 226, 230.
 page 103.
 pageEncoding 87, 89.
 param 100.

- password 133.
 pattern 197, 229.
 prefix 107.
 property 99.
 query 180.
 referencedColumnName 170.
 render 219, 220.
 required 193, 197, 229.
 scope 99, 121.
 second 325.
 select 138.
 session 89.
 size 238, 239.
 step 124.
 strategy 214.
 style 207, 236.
 styleClass 207, 236.
 tagdir 111.
 target 227.
 test 122, 124.
 timezone 325.
 trimDirectiveWhitespaces 89.
 type 130, 222, 224, 226.
 URL 125, 133, 137.
 urlPatterns 75.
 user 133.
 validator 205, 206.
 value 99, 107, 121, 128, 192, 223, 226,
 227, 230, 238, 311, 333.
 var 121, 124, 137, 225.
 variable 121.
 varReader 125.
 WSDL 352.
 wsdlLocation 355.
 xendorsed 352.
 XML 138, 140.
 XSLT 138, 140.
 year 325.
 получение значения 110.
- Аутентификация** 261.
- Solaris 283.
 - на основе формы 267, 286.
 - по клиентскому сертификату 267.
 - по отпечатку 267.
 - стандартная 265–276, 286.
 - через базу данных JDBC 288–292.
- через базу данных LDAP 283, 286–287.
 через веб-сервис EJB 363.
 через область файла 265.
 через пользовательскую область
 293–297.
- Б**
- Безопасность.** См. также **Область безопасности.**
- EJB 326–330.
 - веб-сервисов 360–362.
 - EJB 362–363.
 - приложения Java EE 261–298.
 - элемент меню 263, 264, 283.
- Библиотека.** См. также **Apache:**
- библиотека commons-lang.
 - SQL-тегов JSTL 131–135.
 - XML-тегов JSTL 136–140.
 - базовых тегов JSTL 119–127.
 - дополнительных компонентов JSF 240.
 - ICEfaces 240.
 - MyFaces Trinidad 240.
 - Primefaces 240.
 - RichFaces 240.
 - тегов форматирования JSTL 127–131.
- Бины**
- именованные 332–334.
 - контекст 338–345.
 - Application (Приложение) 339.
 - Conversation (Переговоры) 339.
 - Dependent (Зависимый) 339.
 - Request (Запрос) 338.
 - Session (Сессия) 339.
 - определение 333.
 - компонентные 299, 300. См. Enterprise JavaBeans (EJB) 3.1.
 - определение 97.
 - получение свойства 110.
 - сессионные 300–306.
 - не сохраняющие состояния 300.
 - жизненный цикл 319–321.
 - одноэлементные 306–307.
 - с сохранением состояния 316–319.
 - жизненный цикл 316–319.
 - сущностные 24, 154.
 - управляемые

JSF 198–199.
контекст 199.
получение доступа из другого бина 334.
сообщением 309–310, 321.
жизненный цикл 321.

В

Веб-консоль 241, 320. *См.* Консоль администрирования.
Веб-приложение 305.
развертывание 56.
тестирование 56.
упаковка 55.
Веб-сервис
EJB
клиенты. *См.* Клиенты веб-сервиса: EJB.
JAX-WS 346–359.
RESTful. *См.* Java API for RESTful Web Services (JAX-RS).
допустимые типы 356–357.
определение 346.
отправка вложений 357–359.
тестирование 370–372.
Величина
изменения размера пула 320.
тайм-аута простоя пула 320.
Выполнить на сервере 396.
Выражение
JSP 94. *См. также* JavaServer Pages (JSP): свойство: выражение.
XPath 136, 138.
метода связывания 198.
отложенное, значение 226.
привязки 192.

Г

Генерация кода Java из WSDL 351.
Группа безопасности 261.

Д

Декларация
JSP 102. *См.* JavaServer Pages (JSP):
свойство: объявление.

Дескриптор
библиотеки тегов 105.
развертывания. *См.* Файл: XML: конфигурационный.
Динамический веб-проект 394.
Директива 87.
См. также JavaServer Pages (JSP):
директива.
Долговременный подписчик 257–260.
См. также Тема сообщений.
Домен. *См. также* GlassFish: домены.
обмена сообщениями
публикация/подписка 241.
точка-точка 241, 245.
службы 41.
номера портов 41.

З

Загрузка дополнительных адаптеров серверов 392.
Запрос
FROM 182.
HTTP 366, 376, 377. *См. также* HTTP:
запрос.
именованный 180.
метод
DELETE 366.
GET 365.
POST 361, 366.
PUT 366.
на спецификацию Java 197.

И

Идентификатор пользователя 264.
Инжекция
SQL-кода 146.
зависимости 152, 154, 317, 334–345.
интерфейса 159, 302, 322.
ресурса 250.
Интерфейс
java.util.Collection 115.
java.util.concurrent.Future 307.
java.util.List 179.
javax.ejb.TimerService 322.

-
- javax.faces.context.FacesContext** 191.
javax.faces.convert.Converter 224.
javax.faces.event.ActionListener 222.
javax.faces.event.PhaseListener 226.
javax.faces.event.ValueChangeListener 229.
javax.faces.validator.Validator 229.
javax.jms.Message 250.
javax.jms.MessageListener 251, 310.
javax.jms.QueueBrowser 254.
javax.jms.Session 247.
javax.jms.TextMessage 248.
javax.jms.Topic 255, 256.
javax.persistence.criteria.CriteriaBuilder 182.
 методы 183.
javax.persistence.criteria.CriteriaQuery 182.
javax.persistence.criteria.Predicate 183.
javax.persistence.criteria.Root 182.
javax.persistence.EntityManager 157, 180.
javax.persistence.EntityManagerFactory 157.
javax.persistence.metamodel.EntityType 183.
javax.persistence.metamodel.Metamodel 183.
javax.persistence.TypedQuery 183.
javax.servlet.http.HttpServletRequest 51.
javax.servlet.http.HttpServletResponse 51.
javax.servlet.jsp.tagext.SimpleTag 104.
javax.sql.DataSource 147.
javax.transaction.UserTransaction 157.
javax.ws.rs.core.MultiValuedMap 379.
PreparedStatement 146, 151.
 методы 146, 152.
 преимущества 146.
ResultSet 149.
 методы 149.
Runnable 83.
Serializable 97, 323, 342.
ServletContext 82.
SessionBean 316.
UniformInterface 377.
Validator 202, 205.
 именования и каталогов Java. См. Java
Naming and Directory Interface (JNDI).
 локальный бизнес-интерфейс 301.
 прикладной, программный.
 См. Application programming interfaces (API).
слушателя
HttpSessionAttributeListener 77.
HttpSessionListener 77.
ServletContextAttributeListener 77.
ServletContextListener 77.
ServletRequestAttributeListener 77.
ServletRequestListener 77.
 удаленный бизнес-интерфейс 301.
Иключение
ClassCastException 99.
IllegalStateException 323.
InvalidClientIdException 257.
java.sql.SQLException 134.
javax.faces.validator.ValidatorException 203.
javax.validation.
ConstraintViolationException 185, 205.
JMSEException 257.
LoginException 295.
RemoteException 311.
TransactionRequiredException 305, 311.
 безопасности 272.
Источник данных, создание 48.

K

Каркас
 объектно-реляционного отображения
 Hibernate 24.
 JDO 24.
 стандартных компонентов.
 См. JavaServer Faces (JSF).
 сторонний 78.
Каскадные таблицы стилей 207.
Каталог
 META-INF 80, 188, 334.
 resources 188.
 tags 109.
 WEB-INF 55, 334, 347.

- WEB-INF/classes 55, 129.
 WEB-INF/lib 55.
 WEB-INF/tags 111.
Квалификатор 335–338.
@Premium 337.
- Класс**
- AsyncResult 308.
 - com.sun.appserv.security.
 - AppservPasswordLoginModule 295.
 - com.sun.enterprise.security.auth.realm.
 - IASRealm 293.
 - com.sun.enterprise.security.auth.realm.
 - Realm 295.
 - com.sun.jersey.api.client.Client 376.
 - com.sun.jersey.api.client.UniformInterface 376.
 - com.sun.jersey.spi.container.servlet.
 - ServletContainer 369.
 - EntityManager 156, 304.
 - java.lang.Number 223.
 - java.lang.Throwable 125.
 - java.security.MessageDigest 288.
 - java.util.StringTokenizer 125.
 - javax.faces.event.ComponentSystemEvent 224.
 - javax.mail.Message 389.
 - javax.mail.Transport 389.
 - javax.servlet.GenericServlet 50.
 - javax.servlet.http.HttpServlet 50.
 - javax.servlet.jsp.JspWriter 87.
 - javax.ws.ApplicationPath 369.
 - javax.ws.rs.Application 370.
 - javax.ws.rs.core.Application 368.
 - LoginModule 293.
 - Realm 293.
 - ServletContext 80.
 - SimpleTagSupport 104–109.
 - первичного ключа**
 - определение 175.
 - особенность 177.
 - получение ссылки 176.
- Клиенты веб-сервиса**
- EJB 360.
 - JAX-WS 351, 354. *См. также Java API for XML-Based Web Services (JAX-WS).*
 - RESTful 375. *См. также Java API for RESTful Web Services (JAX-RS): клиент.*
- Ключ**
- внешний 160, 161, 174.
 - первичный**
 - заместитель 173.
 - искусственный 173.
 - составной 174.
 - стратегия генерации 214.
 - GenerationType.AUTO 214.
 - GenerationType.IDENTITY 214.
 - GenerationType.SEQUENCE 214.
 - GenerationType.TABLE 214.
 - файл 284.
 - хранилище cacerts.jks 279.
- Кодировка**
- Base64. *См. Base64.*
 - страницы 89.
- Команды insert, update и delete** 146.
- Компонент**
- inputText 219.
 - outputText 219.
- Конечная точка**
- веб-сервиса
 - URL 351.
 - информация 348, 350.
 - представление 348, 360.
- Консоль администрирования** 35–49, 44–49.
- Константа**
- MediaType.TEXT_XML 376.
 - Message.RecipientType.BCC 389.
 - Message.RecipientType.CC 389.
 - Message.RecipientType.TO 389.
 - PASSWORD_PROPERTY 362.
 - USERNAME_PROPERTY 362.
- Контейнер EJB.** *См. EJB: контейнер.*
- Контекст JAAS** 284, 287, 290.
- Контексты и инжекция зависимости.**
- См. Contexts and Dependency Injection (CDI).*
- Курсор** 149.

M

Максимальный размер пула 320.

Метод

- acknowledge() 247.
- add() 347, 380.
- addMapping() 81.
- addMessage() 191.
- addRecipients() 389.
- addServlet() 81.
- attachFile() 357.
- authenticateUser() 295.
- begin() 158, 339, 343.
- cancel() 323.
- cancel(boolean mayInterruptIfRunning) 309.
- close() 148, 248.
- commit() 158, 247.
- commitUserAuthentication() 295.
- complete() 84.
- create() 376.
- createBrowser() 254.
- createConnection() 247.
- createConsumer() 256.
- createDurableSubscriber() 259.
- createEntityManager() 157.
- createNamedQuery() 180.
- createProducer() 248, 255.
- createQuery() 178, 182.
- createServlet() 81.
- createSession() 247.
- createSubscriber() 259.
- createTextMessage() 248.
- createTimer() 322.
- delete() 376, 377.
- digest() 289.
- doDelete() 51.
- doGet() 51, 74, 83.
- doPost() 51, 59, 74, 83.
- doPut() 51.
- doTag() 104.
- ejbActivate() 316, 318.
- ejbPassivate() 316, 318.
- ejbRemove() 316.
- encryptPassword() 289.
- end() 339, 343, 344, 345.
- executeQuery() 146, 148.
- executeUpdate() 152.
- find() 158, 164.
- findByPrimaryKey() 158.
- forward() 66.
- get() 309, 376.
- getAttribute() 69, 70, 94.
- getAttributes() 223.
- getAuthType() 294.
- getConnection() 147, 154.
- getCriteriaBuilder() 182.
- getDeclaredSingularAttribute() 183.
- getGroupNames() 294, 296.
- getInfo() 323.
- getInitParameter() 74, 76.
- getJAASContext() 294, 296.
- getJSPContext() 105.
- getLabel() 203.
- get(long timeout, TimeUnit unit) 309.
- getMetamodel() 183.
- getNamePort() 355.
- getOut() 105.
- getParameter() 60–61.
- getParameterMap() 121.
- getParameterValues() 61.
- getRequestContext() 362.
- getResultList() 179, 184.
- getServletConfig() 74, 90.
- getServletContext() 70, 90.
- getServletInfo() 88.
- getServletRegistration() 81.
- getSession() 70, 91.
- getSingleResult() 179.
- getString() 149.
- getText() 254.
- getTimers() 323.
- getWriter() 87, 90.
- init() 297.
- InitialContext.lookup() 147.
- isCancelled() 309.
- isDone() 309.
- like() 183.
- main 246.
- merge() 159, 305.
- next() 149.
- onMessage() 251, 310.
- path() 376.
- persist() 158, 164, 212, 305.

post() 376, 377.
 prepareStatement() 147.
 print() 105.
 println() 105.
 processAction() 222.
 processValueChange() 229.
 produceMessages() 247.
 put() 376.
 queryParam() 379.
 queryParams() 380.
 receive() 250.
 remove() 158.
 requestDestroyed() 78.
 requestInitialized() 78.
 resource() 376.
 rollback() 247, 315.
 run() 84.
 send() 248, 389.
 sendRedirect() 67, 125.
 setAttribute() 65, 70.
 setCustomerId() 157.
 setMessageListener() 252.
 setRecipient() 388.
 setRollbackOnly() 312.
 setSessionContext (SessionContext ctx)
 316.
 setString() 151.
 setSubject() 389.
 setText() 248, 389.
 start() 83–84, 250.
 startAsync() 83.
 startTimer() 324.
 type() 376.
 UIComponent.getFacet() 225.
 validate() 202, 205.
 асинхронный вызов 307–309.
 атрибут 59.
 действия слушателя 220, 222.
Модель-Представление-Контроллер.
 См. Шаблон: проектирования:
 Model-View-Controller (MVC).

H

Начальный и минимальный размер пула
 320.

O

Область
 безопасности 261–262.
 администратора 262–264.
 предопределенная 262–265.
 элемент меню 263.
Облегченный протокол доступа к
 каталогам. См. Аутентификация:
 через базу данных LDAP.
Обобщения 308.
Обратный адрес по умолчанию 386.
Объект
 java.lang.Object 71.
 доступа к данным 180. См. также
 Шаблон: проектирования:
 Объект доступа к данным.
неявный
 application 90, 94.
 config 90.
 exception 90.
 out 90.
 page 90.
 pageContext 90.
 request 91, 94.
 response 91.
 session 91, 94.
 определение 87.
Объектная нотация Java Script.
 См. JavaScript Object Notation
 (JSON).
Операторы
 ! 122.
 empty 122, 123.
 import 87.
 LIKE 179.
 new 157.
 select 146, 147.
 SQL 134, 154.
 арифметические 123.
 логические 123.
 объекты 146.
 отношения 122.
 подготовленные 134, 146.
 языка запросов. См. Язык: запросов:
 JPA (JPQL): операторы.
Основное DN 286.

-
- Отладка на сервере 396.
- Отношение
- владелец 161.
 - двунаправленное 159, 164.
 - многие ко многим 168–173.
 - один к одному 159–164.
 - один ко многим 164–168.
 - однонаправленное 159.
 - принадлежит 171.
- Очередь сообщений 241, 245–248.
- просмотр 253.
- П**
- Пакет
- javax.jms 248.
 - javax.servlet 76.
 - javax.validation.constraints 184, 197.
 - импорт 88, 110.
 - ресурса 128, 129.
 - Messages.properties 209.
 - пользовательский 211.
- Параметр
- alias 277, 279.
 - file 279.
 - keep 351.
 - keypass 277.
 - keystore 277, 279.
 - portbase 42.
 - property 285.
 - savemasterpassword 280.
 - storepass 277, 279.
 - замена 129–130.
 - именованный 178.
 - местозаполнители 134, 151.
 - пути 380–381.
- Переадресация запроса 64–66.
- Переговоры
- длительные 343.
 - скоротечные 343.
- Перенаправление отклика 67–69.
- Персистентность. См. Java Persistence API (JPA) 2.0.
- Подключаемость 78–80.
- Порт
- 3700 41.
 - 3820 42.
 - 3920 42.
 - 4848 41.
 - 7676 41.
 - 8080 59.
 - 8181 41.
 - 8686 42.
- Почтовый сервер 386.
- Представление
- EJB как веб-сервисов 359.
 - WSDL 350, 361.
 - XML 381.
- конечной точки. См. Конечная точка: представление.
- Префикс
- c 121.
 - f 190.
 - h 190.
- Приложение
- Java EE
 - отмена развертывания 37.
 - развертывание 35.
 - с помощью командной строки 38.
 - через веб-консоль 35.
 - через каталог autodeploy 39.
 - автономное 246.
- Проверки допустимости
- пользовательская 201–202.
 - со стороны бина 184–185, 197.
- Просмотр
- сертификатов 278.
 - сообщений в очереди 253–254.
 - сохраняемых атрибутов сущностей 183.
- Пространство
- дисковое 31.
 - имен 189.
 - xmlns:f="http://java.sun.com/jsf/core" 190.
 - xmlns:h="http://java.sun.com/jsf/html" 190.
- Протокол
- доступа к объектам, простой 346.
 - передачи почты, простой 384.
 - транспортный 387.
 - smtp 387.
 - smtpls 387.
 - класс 387.
 - хранилища 386.
 - imap 386.

imaps 386.
pop3 386.
pop3s 386.
класс 386.

Псевдоним сущности 178.

Пул соединений

закрытие 148.
имя, поле 243.
создание 44.

Пункт назначения 241.

ресурсы 243, 244.
физический, имя 244.

P

Развернуть, пункт меню 391.

Ресурсы

JMS, пункт меню 243, 244, 258.
определение 188.
стандартное расположение 188–189.
тип, поле 194, 242–243, 245.

C

Свойство

ClientId 257, 258.
дайджест 291.
добавление 194, 258, 297.
управляемое 200.

Сеанс

JavaMail 384–386, 387.
JMS 247.
недействительный 275.

Серверные страницы Java.

См. JavaServer Pages (JSP).

Сервлет 3.0. *См. Servlet 3.0; См. также Java Servlet API 3.0.*

регистрация в контейнере 81.

Сертификат цифровой 272.
клиентский 276.
самоподписанный 276.
серверный 272.

Синглетон. *См. Бины: сеансовые: однокомпонентные.*

Системный журнал. *См. Файл: журнала сервера.*

Скриптлет 87.

Служба

аутентификации и авторизации.

См. Java Authentication and Authorization Service (JAAS).

безопасности сервера. *См. Безопасность.*

Событие

Ajax 219.
JavaScript 219.
blur 220.
change 220.
click 220.
dblclick 220.
focus 220.
keydown 220.
keypress 220.
keyup 219, 220.
mousedown 220.
mousemove 220.
mouseout 220.
mouseover 220.
mouseup 220.
select 220.
valueChange 220.

Сообщения

домен. *См. Домен: обмена сообщениями.*

извлечение

из очереди 249–250.
асинхронно 250–252.
из темы 255–257.

отправка

очереди 245–248.
темы 254–255.

потребитель 241, 245.

продюсер 241, 245.

просмотр. *См. Просмотр: сообщений в очереди.*

тип

BytesMessage 248.
MapMessage 248.
ObjectMessage 248.
StreamMessage 248.
TextMessage 248.

Состояние диалога 300.

Состояния

сеансового бина

не сохраняющего состояния 319.

- См.* Бины: сеансовые: не сохраняющие состоянияс. 316. *См.* также Бины: сеансовые: с сохранением состояния. 316. Готов 316. Не существует 316. Пассивен 316. управляемого сообщением бина. *См.* Бины: управляемые: сообщением. Список рассылки 388. по скрытым адресам 389. Ссылка из JSF-страницы 338. Тестер 349. Страница JSP. *См.* JavaServer Pages (JSP). определение 86. фрейслета 189, 190, 198. СУРБД JavaDB 144, 214. обзор существующих 43. подключение 43.
- ## Т
- Тайм-аут удаления 317.
- Тег HTML input 100.
- JSF
- <f:actionListener> 222.
 - <f:ajax> 218, 220, 222.
 - <f:attribute> 223.
 - <f:convertDateTime> 223.
 - <f:converter> 224.
 - <f:convertNumber> 223.
 - <f:event> 224.
 - <f:facet> 225.
 - <f:loadBundle> 225.
 - <f:metadata> 225.
 - <f:param> 226.
 - <f:phaseListener> 226.
 - <f:selectItem> 226.
 - <f:selectItems> 226.
 - <f:setPropertyActionListener> 227.
 - <f:subview> 227.
 - <f:validateBean> 196, 227.
 - <f:validateDoubleRange> 196, 228.
 - <f:validateLength> 195, 196, 228.
 - <f:validateLongRange> 197, 228.
 - <f:validateRegex> 197, 229.
 - <f:validateRequired> 197, 229.
 - <f:validator> 203, 207, 229.
 - <f:valueChangeListener> 229.
 - <f:verbatim> 230.
 - <f:view> 230.
 - <f:viewParam> 225, 230.
 - <h:body> 190, 231.
 - <h:button> 231.
 - <h:column> 231.
 - <h:commandButton> 197, 217, 231.
 - <h:commandLink> 232.
 - <h:dataTable> 232.
 - <h:form> 191, 232.
 - <h:graphicImage> 233.
 - <h:head> 190, 233.
 - <h:inputField> 195.
 - <h:inputHidden> 233.
 - <h:inputSecret> 233.
 - <h:inputText> 192, 196, 234.
 - <h:inputTextarea> 234.
 - <h:inputTextField> 193.
 - <h:link> 234.
 - <h:message> 207, 234.
 - <h:messages> 191, 193, 195, 208, 235.
 - <h:outputFormat> 235.
 - <h:outputLabel> 235.
 - <h:outputLink> 236.
 - <h:outputScript> 236.
 - <h:outputStylesheet> 188, 190, 236.
 - <h:outputText> 192, 236.
 - <h:panelGrid> 191, 197, 236.
 - <h:panelGroup> 197, 237.
 - <h:selectBooleanCheckbox> 238.
 - <h:selectManyCheckbox> 238.
 - <h:selectManyListbox> 238.
 - <h:selectManyMenu> 239.
 - <h:selectOneListbox> 239.
 - <h:selectOneMenu> 239.
 - <h:selectOneRadio> 239.
 - <style> 207.
 - <td> 192.
- JSP
- <jsp:getProperty> 98–100.
 - <jsp:include> 101, 103, 125, 227.
 - <jsp:root> 118.
 - <jsp:setProperty> 98–100.

- <jsp:useBean> 98–100.
пользовательский 103–112.
- JSTL**
- <c:catch> 125, 126.
 - <c:choose> 121, 126.
 - <c:forEach> 124, 126.
 - <c:forTokens> 124.
 - <c:if> 124, 126.
 - <c:import> 125, 126, 137, 138, 227.
 - <c:otherwise> 122, 126.
 - <c:out> 121, 126.
 - <c:param> 125, 127.
 - <c:redirect> 125, 127.
 - <c:remove> 124, 127.
 - <c:set> 121, 127.
 - <c:url> 125, 127.
 - <c:when> 122, 127.
 - <fmt:bundle> 128, 130.
 - <fmt:formatDate> 130.
 - <fmt:formatNumber> 130.
 - <fmt:message> 129, 131.
 - <fmt:param> 129, 131.
 - <fmt:parseDate> 131.
 - <fmt:parseNumber> 131.
 - <fmt:requestEncoding> 131.
 - <fmt:setBundle> 131.
 - <fmt:setLocale> 128, 131.
 - <fmt:setTimeZone> 131.
 - <fmt:timeZone> 131.
 - <sql:dateParam> 135.
 - <sql:param> 134, 135.
 - <sql:query> 134, 135.
 - <sql:setDataSource> 133, 135.
 - <sql:transaction> 134, 135.
 - <sql:update> 134, 135.
 - <x:choose> 137, 138.
 - <x:forEach> 137, 138, 139.
 - <x:if> 137.
 - <x:otherwise> 137, 138, 139.
 - <x:out> 138, 139.
 - <x:param> 137, 139.
 - <x:parse> 137, 139.
 - <x:set> 137, 139.
 - <x:transform> 138, 140.
 - <x:when> 137, 138, 140.
- TLD**
- <attribute> 106.
 - <name> 106.
 - <required> 106.
 - <rtexprvalue> 106.
 - <tag> 106.
 - <tag-class> 106.
 - <tlib-version> 106.
- XML**
- <as-context> 329.
 - <auth-constraint> 266, 281.
 - <auth-method> 273.
 - <auth-method> 266, 285, 286, 329, 363.
 - <classpath> 352.
 - <configuration> 354.
 - <context-root> 57.
 - <converter> 224.
 - <ejb-name> 318, 363.
 - <filter-mapping> 74.
 - <form> 59.
 - <form-error-page> 274.
 - <form-login-config> 274.
 - <form-login-page> 274.
 - <group-name> 268, 286, 287, 292, 329.
 - <init-param> 73, 92.
 - <jta-data-source> 158.
 - <listener> 77.
 - <login-config> 266, 267, 363.
 - <max-pool-size> 321.
 - <message-bundle> 210.
 - <param-name> 92.
 - <param-value> 92.
 - <persistence-unit> 158.
 - <pool-idle-timeout-in-seconds> 321.
 - <port-component-name> 363.
 - <principal-name> 281, 282, 286, 287.
 - <realm> 329, 363.
 - <realm-name> 267, 285, 286, 292, 297.
 - <removal-timeout-in-seconds> 318.
 - <required> 329.
 - <res-auth> 134.
 - <resize-quantity> 321.
 - <resource-ref> 133.
 - <res-ref-name> 133.
 - <res-type> 133.
 - <role-name> 266, 268, 281, 286, 287, 328.
 - <security-constraint> 266, 270.
 - <security-role-mapping> 267, 281, 282, 328.
 - <servlet> 53.
 - <servlet-class> 369.
 - <servlet-mapping> 53, 112.
 - <steady-pool-size> 321.
 - <taskdef> 352.
 - <transport-guarantee> 270, 271, 281.

- <url-pattern> 54, 59, 93, 266, 361.
 <user-data-constraint> 270.
 <web-fragment> 79.
 <web-resource-collection> 266.
 <webservice-endpoint> 363.
 <welcome-file> 58.
 <wsimport> 352.
- Тема сообщений** 254–259.
- Транзакция**
 сеанса-JMS 247.
 управляемая бином 310, 313–315.
 управляемая контейнером 305, 310–313.
- Трансформации XSL** 140.
См. eXtensible Stylesheet Language Transformations (XSLT).
- У**
- Унифицированный идентификатор ресурса** 365, 366.
- Управление пользователями** 264.
- Управляемое свойство** 216. *См. также* Аннотация:
 @ManagedProperty.
- Утилита командной строки** 40. *См. также* asadmin.
- Участник по умолчанию для ролевого отображения** 329.
- Ф**
- Фабрика**
 JMS-соединений 241–243.
 класс 194.
- Файл**
 cacerts.jks 283.
 EAR 24, 305.
 HTML 58.
 JAR 247, 293, 302, 305, 334, 359.
 javaee.jar 352.
 jaxb-osgi.jar 352.
 security.jar 294.
 webservices-osgi.jar 352.
 login.conf 294, 296.
 security.jar 295.
 TLD 105, 111.
- implicit.tld 111.
WAR 24, 81, 86, 302, 305, 347, 352, 357.
 создание 55.
- WSDL** 350.
- XML**
 pom.xml 352.
 конфигурационный 53.
 beans.xml 334.
 ejb-jar.xml 318.
 faces-config.xml 188, 198, 210, 224, 335.
 persistence.xml 158.
 sun-ejb-jar.xml 318, 320, 328, 362.
 sun-web.xml 57, 265, 267, 281.
 web-fragment.xml 79.
 web.xml 53, 79, 194, 265, 267, 271, 281, 292, 347, 360, 368.
 с данными 381.
- ZIP**
 платформонезависимый 28.
 журнала сервера 39, 56, 78, 310, 324.
- расширение (суффикс)**
 .cer 279.
 .faces 201.
 .jsf 201.
 .jspf 102.
 .jspx 116.
 .p12 277.
 .tag 109.
 .xhtml 201.
- Фасет** 225.
- Функции**
 JSTL 140–143.
 fn:containsIgnoreCase (String, String) 141.
 fn:contains(String, String) 141.
 fn:endsWith(String, String) 141.
 fn:escapeXml(String) 142.
 fn:indexOf(String, String) 142.
 fn:join(String[], String) 142.
 fn:length() 140–141.
 fn:length(Object) 142.
 fn:replace(String, String, String) 142.
 fn:split() 141.
 fn:split(String, String) 142.
 fn:startsWith(String, String) 142.
 fn:substringAfter(String, String) 143.
 fn:substringBefore(String, String) 143.
 fn:substring(String, int, int) 142.
 fn:toLowerCase(String) 143.
 fn:toUpperCase() 141.

`fn:toUpperCase(String)` 143.
`fn:trim(String)` 143.
 утилиты keytool. *См.* keytool.
 Файлы 187–188.
 преимущества 23.

III

Шаблон

URL 71, 201, 266, 369.
 даты и времени 130.
 именования 355.
 проектирования 131.
 Model-View-Controller (MVC) 86, 180, 334.

Инжекция зависимости. *См.* Инжекция: зависимости.

Объект доступа к данным 180, 303.

Шифрование

Э

Этап

проекта 193–195.
 определение 194.
 слушатель, регистрация 226.
 установки 32.

Я

Язык

выражений
 JSF 216.
 унифицированный 106, 113–116.
 нотация 111.
 операторы. *См. здесь* Операторы.

запросов

JPA (JPQL) 177–180.
 ограничения 181.
 операторы 179.
 улучшения 24.
 SQL 146.

определения веб-сервисов 350.
 сценариев 89.

A

Action listener method. *См.* Метод: действия слушателя.

admin realm. *См.* Область: безопасности: администратора.

Ajax 82.

в приложениях JSF 218–221.
 поведение 222.

ANT. *См.* Apache: ANT.

Apache

ANT 53, 352.

Maven 53, 303, 352.

библиотека commons-lang 205, 342.
 блок проверки допустимости 203.

API Java

JMS. *См.* Java Messaging System API (JMS).

JSF. *См.* JavaServer Faces (JSF).

для веб-сервисов RESTful (JAX-RS).

См. Java API for RESTful Web Services (JAX-RS).

для веб-сервисов XML (JAX-WS).

См. Java API for XML-Based Web Services (JAX-WS).

для связывания с XML. *См.* Java Architecture for XML Binding (JAXB) 2.2.

Контекстов и инжекции зависимости.

См. Contexts and Dependency Injection (CDI).

Критериев 181–184.

Метамодели 182.

Персистентности (JPA) 2.0. *См.* Java Persistence API (JPA) 2.0.

Подключения к базе данных. *См.* Java Database Connectivity (JDBC).

Системы обмена сообщениями.

См. Java Messaging System API (JMS).

Службы аутентификации и авторизации. *См.* Java Authentication and Authorization Service (JAAS).

API JavaMail 384.

API Jersey 366, 375.
 отправка параметров 378–380.
 клиенту 382–383.
appclient 246–247, 250, 252, 302, 329,
 330, 355, 360.
команды
 -client 247, 250, 302, 330, 355.
Application programming interfaces (API)
 346.
Application scope. См. Бины: именованные: контекст: Application (Приложение).
asadmin 264, 284, 285, 287, 288, 292,
 297.
команда
 change-master-password 280.
 create-auth-realm 284–286, 288, 292.
 create-domain 41.
 delete-domain 43.
 restart-domain 44.
 start-domain 241.
 stop-domain 43.
 остановка домена 43.
параметр. См. также Параметр.
 --classname 284, 286.
 --help 41.
 --property 285, 287.
развертывание 40.
создание доменов 41.
создание пулов соединений 44.
удаление доменов 43.

B

Base64 266.
Base DN. См. Основное DN.
BASIC. См. Аутентификация: стандартная.
Bean validation. См. Проверки допустимости: со стороны бина.
Binding expression. См. Выражение: привязки.

C

Cascading Style Sheets (CSS). См. Каскадные таблицы стилей.

CLIENT-CERT 280. См. Аутентификация: по клиентскому сертификату.

CONFIDENTIAL 271.

Connection Pools. См. Пул соединений.

Container-managed transactions.

См. Транзакция: управляемая контейнером.

Contexts and Dependency Injection (CDI)
 332–345.

назначение 24.

Conversational state. См. Состояние диалога.

Conversation scope. См. Бины: именованные: контекст: Conversation (Переговоры).

Core JSTL tags. См. Библиотека: базовых тегов JSTL.

CRUD 144.

curl 370–372, 375, 378.

Cursor. См. Курсор.

D

Data Access Objects (DAO). См. Шаблон: проектирования: Объект доступа к данным.

Debug on Server. См. Отладка на сервере.

Default

Principal to Role Mapping. См. Участник по умолчанию для ролевого отображения.

Return Address. См. Обратный адрес по умолчанию.

Deferred value expression. См. Выражение: отложенное, значение.

Dependency injection. См. Инжекция: зависимости.

Dependent scope 340. См. также Бины: именованные: контекст: Dependent (Зависимый).

Deploy. См. Развернуть, пункт меню.

Destination. См. Пункт назначения.

DIGEST. См. Аутентификация: по отпечатку.

`doGet()`. См. Метод: `doGet()`.
`doPost()`. См. Метод: `doPost()`.
Download additional server adapters.
См. Загрузка дополнительных адаптеров серверов.
Dynamic Web Project. См. Динамический веб-проект.

E

EAR. См. Файл: EAR.
Eclipse 392–396.
EJB. См. Бины: компонентные, как веб-сервисы. См. Представление: EJB как веб-сервисов.
контейнер 300, 301.
пул 300.
служба таймера 322–326.
empty. См. Операторы: empty.
Encryption. См. Шифрование.
Enterprise JavaBeans (EJB) 3.1. См. также Бины: компонентные.
новые возможности 23.
Entity Beans. См. Бины: сущностные.
eXtensible Stylesheet Language Transformations (XSLT) 136.

F

Facelets. См. Фэйслеты.
FORM. См. Аутентификация: на основе формы.
Formatting JSTL tags. См. Библиотека: тегов форматирования JSTL.

G

GET. См. HTTP: запрос: GET.
getAttribute(). См. Метод: `getAttribute()`.
getServletContext(). См. Метод: `getServletContext()`.
GlassFish
домены 41.
остановка 43.
по умолчанию 41.
создание 41.

удаление 43.
зависимости 28.
новые возможности 25.
получение 27.
получение справки 35.
почтовая служба 384.
преимущества 26.
проверка установки 34.
установка 29.

H

HTML форма 58–84.
HTTP
запрос 77, 82.
DELETE 51.
GET 51.
POST 51.
PUT 51.
сервлет. См. Servlet 3.0.
HTTPS 270, 273.

I

Initial and Minimum Pool Size. См. Начальный и минимальный размер пула.
INTEGRAL 271.
InvalidClientIdException. См. Исключение: `InvalidClientIdException`.

J

JAAS context. См. Контекст JAAS.
Java API for RESTful Web Services (JAX-RS) 365–375.
клиент 375–383.
определение 25.
Java API for XML-Based Web Services (JAX-WS) 346.
определение 25.
Java Architecture for XML Binding (JAXB) 2.2 356, 372–375.
определение 25.
Java Authentication and Authorization Service (JAAS) 261.
JavaBeans. См. Бины: определение.

-
- J**
- Java Database Connectivity (JDBC) 144, 145–154.
 - JavaDB. *См.* СУРБД: JavaDB.
 - Java EE
 - новые возможности 23.
 - общий обзор 22.
 - JavaMail Sessions. *См.* Сеанс: JavaMail.
 - Java Messaging System API (JMS) 241–260.
 - ресурсы 243–245, 258.
 - Java Naming and Directory Interface (JNDI)
 - имя 309, 386.
 - в веб-консоли 47.
 - поиск 147.
 - Java Persistence API (JPA) 2.0 144, 154–180.
 - новые возможности 24.
 - Java Persistence Query Language (JPQL).
 - См.* Язык: запросов: JPA (JPQL).
 - JavaScript Object Notation (JSON) 365.
 - JavaServer Faces (JSF) 187–240.
 - интеграция с JPA 211.
 - новые возможности 23.
 - JavaServer Pages (JSP) 85–118.
 - XML-синтаксис 116–118.
 - директива
 - attribute 110, 116.
 - include 101–102, 116.
 - page 116.
 - tag 116.
 - taglib 107, 111, 116, 121.
 - variable 117.
 - документ 116.
 - допустимый 117, 122.
 - свойство
 - выражение 116.
 - комментарий 116.
 - объявление 116.
 - скриптлет 116.
 - Java Servlet API 3.0. *См.* также Сервер 3.0.
 - новые возможности 24.
 - определение 50–51.
 - Java Specification Requests (JSR).
 - См.* Запрос: на спецификацию Java.
 - Jersey 365.
 - JerseyServlet 369.
 - j_password 274.
 - j_security_check 274.
 - JSP
 - declaration. *См.* Декларация: JSP; *См.* также JavaServer Pages (JSP): свойство: объявление.
 - documents. *См.* JavaServer Pages (JSP): документ.
 - expressions. *См.* Выражение: JSP; *См.* также JavaServer Pages (JSP): свойство: выражение.
 - page directives. *См.* Директива: JSP-страницы; *См.* также JavaServer Pages (JSP): директива: page.
 - Standard Tag Library (JSTL) 119–143.
 - JSR 303 184, 227.
 - j_username 274.
- K**
- Key File. *См.* Ключ: файл.
 - keytool 276–277, 281.
 - команда
 - export 279.
 - genkey 277.
 - import 279.
- L**
- Lightweight Directory Access Protocol (LDAP). *См.* Аутентификация: через базу данных LDAP.
 - Local business interface. *См.* Интерфейс: локальный бизнес-интерфейс.
 - Long running. *См.* Переговоры: длительные.
- M**
- Mail Host. *См.* Почтовый сервер.
 - Manage Users. *См.* Управление пользователями.
 - master-password 280.
 - Maven. *См.* Apache: Maven.

Maximum Pool Size. См. Максимальный размер пула.

MD2 291.

MD5 267, 288, 291.

Message

consumer. См. Сообщения: потребитель.
producer. См. Сообщения: продюсер.

Method binding expression. См. Выражение: метода связывания.

MIME-тип 365, 366, 368, 376.

Model-View-Controller (MVC). См. Шаблон: проектирования: Model-View-Controller (MVC).

N

Named query. См. Запрос: именованный.

NetBeans 390–392.

P

Phase listener. См. Этап: слушатель, регистрация.

Plain old Java objects (POJO) 157, 303.

pluggability. См. Подключаемость.

Point-to-point (PTP). См. Домен: обмена сообщениями: точка-точка.

Pool

Idle Timeout. См. Величина тайм-аута простоя пула.

Resize Quantity. См. Величина изменения размера пула.

POST. См. HTTP: запрос: POST.

PreparedStatement. См. Интерфейс: PreparedStatement.

Prepared statements. См. Операторы: подготовленные.

Project stages. См. Этап: проекта.

Publish/subscribe (pub/sub). См. Домен: обмена сообщениями: публикация/подписка.

PUT. См. HTTP: запрос: PUT.

Q

queue. См. Очередь сообщений.

R

Remote business interface. См. Интерфейс: удаленный бизнес-интерфейс.

Removal Timeout. См. Тайм-аут удаления.

Request scope. См. Бины: именованные: контекст: Request (Запрос).

Resource Type. См. Ресурсы: тип, поле.

RESTful-ресурс 366.

Run on Server. См. Выполнить на сервере.

S

Scriptlet. См. Скриптлет.

Security. См. также Безопасность.

groups. См. Группа безопасности: .

realm. См. Область: безопасности.

Servlet 3.0

API Java. См. Java Servlet API 3.0. интерфейс

HttpSessionAttributeListener 77.

HttpSessionListener 77.

ServletContextAttributeListene 77.

ServletContextListener 77.

ServletRequestAttributeListener 77.

ServletRequestListener 77.

компиляция 52.

конфигурирование 53.

новые возможности 71–84.

определение 50–51.

ServletContext. См. Класс: ServletContext.

Session scope. См. Бины: именованные: контекст: Session (Сеанс).

setAttribute(). См. Метод: setAttribute().

SHA 1 291.

SHA 256 291.

SHA 384 291.

SHA 512 291.

Simple

Mail Transfer Protocol (SMTP). См. Протокол: передачи почты, простой.

Object Access Protocol (SOAP). См. Протокол: передачи данных, объектно-ориентированный.

токол: доступа к объектам, простой.

Singleton. См. Бины: сеансовые: одноделементные.

session bean. См. Бины: сеансовые: одноэлементные.

SQL JSTL tag. См. Библиотека: SQL-тегов JSTL.

SSL 272.

Stateful session bean. См. Бины: сеансовые: с сохранением состояния.

Stateless session bean. См. Бины: сеансовые: не сохраняющие состояние.

Statement objects. См. Операторы: объекты.

Store Protocol. См. Протокол: хранилища.

Store Protocol Class. См. Протокол: хранилища: класс.

sun-web.xml. См. Файл: XML: конфигурационный: sun-web.xml.

Т

Tag Library Descriptor (TLD). См. Дескриптор: библиотеки тегов.

Thawte 272, 276.

Transient. См. Переговоры: скоротечные.

Transport Protocol. См. Протокол: транспортный.

Class. См. Протокол: транспортный: класс.

У

Unified Expression Language. См. Язык: выражений: унифицированный.

Uniform Resource Identifiers (URI). См. Унифицированный идентификатор ресурса.

User ID. См. Идентификатор пользователя.

В

Valid XML. См. JavaServer Pages (JSP): документ: допустимый.

Verisign 272, 276.

View

Endpoint. См. Конечная точка: представление.

WSDL. См. Представление: WSDL.

W

WAR. См. Файл: WAR.

WEB-INF. См. Каталог: WEB-INF.

WEB-INF/classes. См. Каталог: WEB-INF/classes.

WEB-INF/lib. См. Каталог: WEB-INF/lib.

Web Service. См. Веб-сервис.

Definition Language (WSDL). См. Язык: определения веб-сервисов.

Endpoint Information. См. Конечная точка: веб-сервиса: информация.

web.xml. См. Файл: XML: конфигурационный.

wsimport 351, 355, 357, 361.

Х

XML JSTL tag. См. Библиотека: XML-тегов JSTL.

З

ZIP. См. Файл: ZIP.

Дэвид Хеффельфингер

Java EE 6 и сервер приложений GlassFish 3

Главный редактор *Мовчан Д.А.*
dm@dmk-press.ru

Переводчик *Карышев Е.Н.*

Литературный редактор Готлиб О.В.

Верстка Карышев Е.Н.

Дизайн обложки Карышев Е.Н.

Подписано в печать 06.02.2013. Формат 60×90 $\frac{1}{16}$.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 26. Тираж 200 экз.