THE EXPERT'S VOISE® IN JAVA

Изучаем Java EE 7

Энтони Гонсалвес

Apress®

ПИТЕР®

Beginning Java EE 7

Antonio Goncalves

Apress®

Изучаем Java EE 7

Энтони Гонсалвес

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2014

Э. Гонсалвес

Изучаем Java EE 7

Перевели с английского Е. Зазноба, О. Сивченко, О. Мясникова, В. Черник

Заведующий редакцией	*Д. Виницкий*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*В. Конаш*
Художник	*Л. Адуевская*
Корректор	*Е. Павлович*
Верстка	*А. Барцевич*

ББК 32.973.2-018.1 УДК 004.43

Гонсалвес Э.

Г65 Изучаем Java EE 7. — СПб.: Питер, 2014. — 640 с.: ил.

ISBN 978-5-496-00942-3

Java Enterprise Edition (Java EE) остается одной из ведущих технологий и платформ на основе Java.

Данная книга представляет собой логичное пошаговое руководство, в котором подробно описаны многие спецификации и эталонные реализации Java EE 7. Работа с ними продемонстрирована на практических примерах. В этом фундаментальном издании также используется новейшая версия инструмента GlassFish, предназначенного для развертывания и администрирования примеров кода.

Книга написана ведущим специалистом по обработке запросов на спецификацию Java EE, членом наблюдательного совета организации Java Community Process (JCP). В ней вы найдете максимально ценную информацию, изложенную с точки зрения эксперта по технологиям Java для предприятий.

Благодаря этой книге вы познакомитесь с новейшей версией платформы Java EE; исследуете и научитесь использовать API EJB и JPA — от компонентов-сущностей, компонентов-сеансов до компонентов, управляемых сообщениями, и многого другого; откроете для себя API для разработки на веб-уровне, в частности JSF, Facelet и Expression Language; научитесь обращаться с веб-службами SOAP и RESTful, а также с другими службами, доступными в новейшей версии Java EE; узнаете, как создавать динамические пользовательские интерфейсы для корпоративных и транзакционных Java-приложений.

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1430246268 англ.
ISBN 978-5-496-00942-3

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Подписано в печать 23.05.14. Формат 70×100/16. Усл. п. л. 51,600. Тираж 1000. Заказ 0000.

Отпечатано в полном соответствии с качеством предоставленных издательством материалов в ГППО «Псковская областная типография». 180004, Псков, ул. Ротная, 34.

Оглавление

Предисловие

Java EE 7 основана на предыдущих успешных версиях этой платформы. Упрощенный API сервиса сообщений Java значительно повышает продуктивность разработчика, широкое использование контекстов и внедрения зависимостей (CDI) и сокращение шаблонного кода обеспечивают более связанную интегрированную платформу. Java EE 7 также включает такие технологии, как WebSocket, JSON, Batch и Concurrency, весьма существенные для современной разработки веб-приложений. Все это сокращает необходимость в сторонних фреймворках, значительно облегчая приложения.

Энтони сыграл очень важную роль в формировании платформы Java EE 7. Благодаря своим глубоким техническим знаниям он принял активное участие в составлении спецификаций во время двух ключевых запросов на спецификацию Java (платформа и Enterprise JavaBeans 3.2). В документах он выделил несколько пунктов, что сделало их более простыми для понимания. Он также участвовал в разъяснительных совещаниях при поддержке комитета Java Community Process (JCP), а также входил в состав данного комитета.

Энтони возглавляет парижскую группу пользователей Java; работая консультантом, он использует Java EE для решения повседневных проблем. С энтузиазмом и преданностью своему делу он проводит конференцию Devoxx France. Кроме того, несколько лет назад он написал книгу о Java EE 5 на французском языке, а после — получившую высокое признание книгу «Введение в платформу Java EE 6 с GlassFish 3» *(Beginning Java EE 6 Platform with GlassFish 3)*. Благодаря этому Энтони стал наилучшим кандидатом на авторство данной книги.

Книга содержит несколько практических примеров кода, с которых можно начать изучение. Для развертывания примеров используется GlassFish, но также подойдет любой сервер приложений, совместимый с Java EE 7. Все представленные образцы кода также доступны на GitHub. Если вы искали практический учебник, написанный одним из экспертов по этой платформе, названный источник вам подойдет.

Арун Гупта,
специалист по Java EE и GlassFish

Об авторе

Энтони Гонсалвес — старший системный архитектор из Парижа. Выбрав Java-разработки основным направлением своей карьеры в конце 1990-х, он с тех пор посетил множество стран и компаний, где теперь работает консультантом по Java EE. В прошлом консультант компании BEA, Энтони является экспертом по таким серверам приложений, как WebLogic, JBoss и, конечно же, GlassFish. Он приверженец решений с открытым исходным кодом и даже принадлежит к парижскому объединению разработчиков таких решений (OSSGTP). Энтони также является одним из создателей и руководителей парижской группы пользователей Java, а с недавних пор еще и конференции Devoxx France.

Свою первую книгу по Java EE 5 на французском языке Энтони написал в 2007 году. Тогда же он присоединился к комитету JCP, чтобы стать экспертом в различных запросах на спецификацию Java (Java EE 6, JPA 2.0 и EJB 3.1), и выпустил книгу «Изучаем Java EE 6» *(Beginning Java EE 6)* в издательстве Apress. Оставаясь членом комитета JCP, в 2010 году Энтони присоединился к экспертным группам по Java EE 7 и EJB 3.2.

За последние несколько лет Энтони выступал с докладами преимущественно по Java EE на международных конференциях, включая JavaOne, Devoxx, GeeCon, The Server Side Symposium, Jazoon, а также во многих группах пользователей Java. Он также написал множество технических документов и статей для IT-сайтов (DevX) и журналов (Programmez, Linux Magazine). С 2009 года он входил в состав французского Java-подкаста под названием Les Cast Codeurs. За все свои заслуги перед Java-сообществом Энтони был выбран Java-чемпионом.

Энтони окончил Парижскую консерваторию искусств и ремесел (по специальности «Инженер информационных технологий»), Брайтонский университет (со степенью магистра точных наук по объектно-ориентированному проектированию) и Федеральный университет Сан-Карлоса в Бразилии (магистр философии по распределенным системам).

Вы можете подписаться на страницу Энтони в «Твиттере» (@agoncal) и в его блоге (www.antoniogoncalves.org).

О техническом редакторе

Массимо Нардоне получил степень магистра точных наук в области информатики в Университете Салерно, Италия. В настоящее время он является сертифицированным аудитором PCIQSA и работает старшим ведущим архитектором по IT-безопасности и облачным решениям в компании IBM в Финляндии, где в его основные обязанности входит облачная и IT-инфраструктура, а также контроль и оценка надежности системы защиты. В IBM Массимо также руководит финской командой исследовательских разработок (FIDTL). Имеет следующие IT-сертификаты: ITIL (Information Technology Infrastructure Library), Open Group Master Certified IT Architect и Payment Card Industry (PCI) Qualified Security Assessor (QSA). Он является экспертом в области закрытых, открытых и локальных облачных архитектур.

У Массимо более 19 лет опыта в области облачных решений, IT-инфраструктуры, мобильных и веб-технологий и безопасности как на национальных, так и на международных проектах на должностях руководителя проекта, инженера-программиста, инженера исследовательского отдела, главного архитектора по безопасности и специалиста по программному обеспечению. Он также был приглашенным лектором и методистом по практическим занятиям в Лаборатории сетевых технологий Хельсинкского политехнического института (впоследствии вошедшего в состав Университета Аалто).

Массимо хорошо владеет методологиями и приложениями для тестирования протоколов безопасной передачи данных, а также занимается разработками мобильных и интернет-приложений с использованием новейших технологий и многих языков программирования.

Массимо был техническим редактором множества книг, издающихся в сфере информационных технологий, например по безопасности, веб-технологиям, базам данных и т. д. Он обладает несколькими международными патентами (PKI, SIP, SAML и Proxy areas).

Он посвящает эту книгу своей любимой жене Пии и детям Луне, Лео и Неве.

Благодарности

Вы держите в руках мою третью книгу о платформе Java EE. Должен вам сказать, для того чтобы написать третью по счету книгу, надо быть немного ненормальным... а еще находиться в окружении людей, которые помогают вам чем могут (чтобы вы не сошли с ума окончательно). Самое время сказать им спасибо.

Прежде всего, я хочу поблагодарить Стива Англина из издательства Apress за очередную предоставленную мне возможность написать книгу для этой замечательной компании. В процессе ее написания мне постоянно приходилось сотрудничать с Джилл Балзано, Катлин Саливан и Джеймсом Маркхемом, которые редактировали книгу и давали мне ценные советы. Спасибо Массимо Нардоне за тщательную техническую проверку, позволившую улучшить книгу.

Отдельное спасибо ребятам из моей технической команды, которые помогали мне и давали ценные комментарии. Алексис Хасслер живет в Божоле во Франции; он программист-фрилансер, тренер и руководитель группы пользователей Java в Лионе. Брис Лепорини — опытный инженер, в последние десять лет специализируется на Java-разработках. Он обожает запускать новые проекты, повышать производительность приложений и обучать начинающих программистов. Мэтью Анселин — разработчик, который любит Java, виртуальную машину Java, свой Mac-бук и свою гитару, а также входит в состав экспертной группы по инструментарию CDI 1.1 и работает с CDI по технологии OSGi. Антуан Сабо-Дюран, старший инженер-программист в компании Red Hat и технический руководитель на фреймворке Agorava, внес существенный вклад в развитие проектов с использованием CDI. Я с большим удовольствием работал с такими компетентными и веселыми старшими разработчиками.

Я также хочу поблагодарить Юниса Теймури, совместно с которым была написана глава 12 о XML и JSON.

Для меня большая честь, что предисловие к этой книге написал Арун Гупта. Его вклад в Java EE бесценен, как и его технические статьи.

Спасибо моему корректору Трессану О'Донахью, благодаря усилиям которого книга приобрела литературный язык.

Схемы, приведенные в этой книге, были составлены в приложении Visual Paradigm. Я хочу поблагодарить сотрудников Visual Paradigm и JetBrains за бесплатную лицензию на их замечательные продукты.

Я крепко целую любимую дочь Элоизу. Она — мой самый большой подарок в жизни.

Я не написал бы эту книгу без помощи и поддержки всего сообщества Java-разработчиков: блогов, статей, рассылок, форумов, твитов... и, в частности, без тех, кто занимается платформой Java EE.

Отдельное спасибо тебе, Бетти, за то, что ты дарила мне свет в темные времена и силу, когда я ослабевал.

Я также часто вспоминаю друга Бруно Реау, который покинул нас так рано.

Введение

В сегодняшнем мире бизнеса приложения должны осуществлять доступ к данным, реализовывать бизнес-логику, добавлять уровни представления данных, быть мобильными, использовать геолокацию и взаимодействовать с внешними системами и онлайн-сервисами. Именно этого пытаются достичь компании путем минимизации затрат и применения стандартных и надежных технологий, которые могут справляться с большими нагрузками. Если это ваш случай, данная книга — для вас.

Среда Java Enterprise Edition появилась в конце 1990-х годов и привнесла в язык Java надежную программную платформу для разработок корпоративного уровня. J2 EE, хоть и составляла конкуренцию фреймворкам с открытым кодом, считалась достаточно тяжеловесной технологией, была технически перенасыщенной, и каждая ее новая версия подвергалась критике, неправильно понималась или использовалась. Благодаря этой критике Java EE была усовершенствована и упрощена.

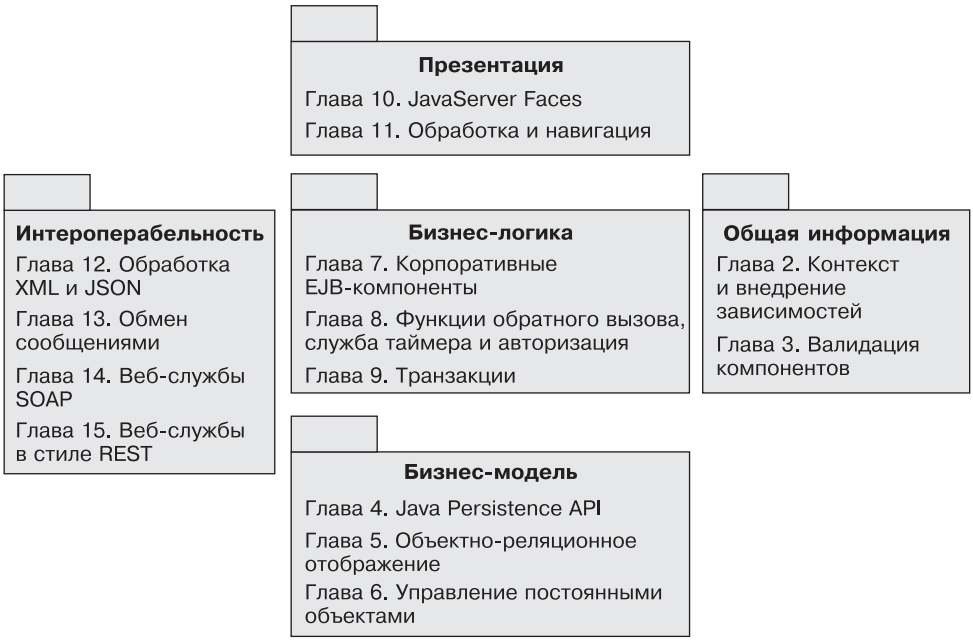
Если вы относитесь к числу людей, которые по-прежнему считают, что «архитектуры EJB — это зло», прочитайте эту книгу, и вы измените свое мнение. Архитектуры Enterprise Java Beans просто прекрасны, как и весь стек технологий Java EE 7. Если же вы, наоборот, адепт Java EE, то в книге вы увидите, как эта платформа обрела равновесие благодаря простоте разработки и несложной компонентной модели. Если вы начинающий пользователь Java EE, эта книга вам также подойдет: в ней очень понятно описываются наиболее важные спецификации, а также для наглядности приводится много примеров кода и схем.

Открытые стандарты являются в совокупности одной из наиболее сильных сторон Java EE. Теперь не составляет труда портировать прикладные программы между серверами приложений. Диапазон технологий, применяемых при написании этих программ, включает JPA, CDI, валидацию компонентов (Bean Validation), EJB, JSF, JMS, SOAP веб-службы либо веб-службы с передачей состояния представления (RESTful). Открытый исходный код — еще одна сильная сторона Java EE. Как вы увидите далее в книге, большинство базовых реализаций Java EE 7 (GlassFish, EclipseLink, Weld, Hibernate Validator, Mojarra, OpenMQ, Metro и Jersey) лицензируются как свободно распространяемое ПО.

Книга посвящена инновациям последней версии, различным спецификациям и способам их сочетания при разработке приложений. Java EE 7 включает около 30 спецификаций и является важным этапом разработок для корпоративного уровня (CDI 1.1, Bean Validation 1.1, EJB 3.2, JPA 2.1), для веб-уровня (Servlet 3.1, JSF 2.2, Expression Language 3.0) и для интероперабельности (JAX-WS 2.3 и JAX-RS 2.0). Издание охватывает большую часть спецификаций по Java EE 7 и использует инструментарий JDK 1.7 и некоторые известные шаблоны проектирования, а также сервер приложений GlassFish, базу данных Derby, библиотеку JUnit и фреймворк Maven. Издание изобилует UML-схемами, примерами Java-кода и скриншотами.

Структура книги

Книга посвящена наиболее важным спецификациям по Java EE 7 и новому функционалу этого релиза. Структура издания повторяет архитектурное выделение уровней в приложении.



В главе 1 приводятся основные понятия Java EE 7, которые далее обсуждаются в книге. Глава 2 посвящена контексту и внедрению зависимости (Context и Dependency Injection 1.1), а глава 3 — валидации компонентов (Bean Validation 1.1).

Главы 4–6 описывают уровень сохраняемости и подробно рассматривают интерфейс JPA 2.1. После краткого обзора и нескольких практических примеров в главе 4 глава 5 полностью посвящена объектно-реляционному отображению (атрибутам отображения, отношениям и наследованию), а глава 6 — тому, как вызывать сущности и управлять ими, их жизненному циклу, методам обратного вызова и слушателям.

Для разработки транзакций на уровне бизнес-логики на Java EE 7 может использоваться архитектура EJB. Главы 7–9 описывают этот процесс. После обзора спецификации и истории ее создания в главе 7 идет речь о компонент-сеансах EJB и модели их разработки. Глава 8 посвящена жизненному циклу архитектуры EJB, службе времени и способам обращения с авторизацией. Глава 9 объясняет понятие транзакций и то, каким образом интерфейс JTA переводит транзакции в EJB, а также в CDI-компоненты.

В главах 10 и 11 описано, как ведется разработка уровня представления данных с помощью фреймворка JSF 2.2. После обзора спецификации в главе 10 речь идет

о разработке веб-страницы с помощью компонентов фреймворков JSF и Facelets. В главе 11 рассказывается о способах взаимодействия с серверной частью архитектуры EJB и поддерживающими CDI-компонентами, а также о навигации по страницам.

Наконец, последние главы предлагают различные способы взаимодействия с другими системами. В главе 12 объясняется, как обрабатывать XML (используя интерфейсы JAXB и JAXP) и JSON (JSON-P 1.0). Глава 13 показывает, как обмениваться асинхронными сообщениями с новым интерфейсом JMS 2.0 и компонентами, управляемыми сообщениями. Глава 14 посвящена веб-службам SOAP, а глава 15 — веб-службам RESTful с новым интерфейсом JAX-RS 2.0.

Скачивание и запуск кода

Для создания примеров, приводимых в этой книге, использовался фреймворк Maven 3 и комплект Java-разработчика JDK 1.7 в качестве компилятора. Развертывание осуществлялось на сервере приложений GlassFish версии 4, а сохранение данных — в базе Derby. В каждой из глав объясняется, как построить, развернуть, запустить и протестировать компоненты в зависимости от используемой технологии. Код тестировался на платформе Mac OS X (но также должен работать в Windows или Linux). Исходный код примеров к этой книге размещен на странице Source Code на сайте Apress (www.apress.com). Загрузить его можно прямо с GitHub по адресу https://github.com/agoncal/agoncal-book-Java EE7.

Связь с автором

Вопросы по содержанию этой книги, коду либо другим темам отправляйте автору на электронную почту antonio.goncalves@gmail.com. Вы также можете посетить его сайт www.antoniogoncalves.org и подписаться на его страницу в «Твиттере»: @agoncal.

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты vinitski@minsk.piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

Глава 1
Краткий обзор Java EE 7

Сегодняшние предприятия существуют в условиях мировой конкуренции. Им нужны приложения, отвечающие их производственным нуждам, которые, в свою очередь, с каждым днем усложняются. В эпоху глобализации компании могут действовать по всему миру, имея представительства на разных континентах, работать в различных странах круглосуточно, без выходных, иметь по несколько центров обработки данных и международные системы, работающие с разными валютами и временными зонами. При этом им необходимо сокращать расходы, увеличивать быстродействие своих сервисов, хранить бизнес-данные в надежных и безопасных хранилищах, а также иметь несколько мобильных и веб-интерфейсов для клиентов, сотрудников и поставщиков.

Большинству компаний необходимо совмещать эти противоречивые требования с существующими корпоративными информационными системами (EIS), одновременно разрабатывая приложения типа «бизнес — бизнес» для работы с партнерами или системы типа «бизнес — потребитель» с использованием мобильных приложений, в том числе с возможностью геолокации. Довольно часто компании требуется координировать корпоративные данные, которые хранятся в разных местах, обрабатываются несколькими языками программирования и передаются с помощью разных протоколов. И конечно же, во избежание серьезных убытков необходимо предотвращать системные сбои, сохраняя при этом высокую доступность, масштабируемость и безопасность. Изменяясь и усложняясь, корпоративные приложения должны оставаться надежными. Именно для этого была создана платформа Java Enterprise Edition (Java EE).

Первая версия Java EE (изначально известная как J2EE) предназначалась для решения задач, с которыми сталкивались компании в 1999 году, а именно для работы с распределенными компонентами. С тех пор программным приложениям пришлось адаптироваться к новым техническим решениям, таким как веб-службы SOAP и RESTful. На сегодняшний день платформа Java EE отвечает этим техническим требованиям, регламентируя различные способы работы в стандартных спецификациях. Спустя годы Java EE изменилась, стала насыщеннее и проще в использовании, а также более мобильной и интегрированной.

В этой главе будут представлены общие сведения о Java EE. После описания внутренней архитектуры, компонентов и сервисов я расскажу о том, что нового в Java EE 7.

Понимание Java EE

Если вам нужно произвести какие-то операции с наборами объектов, вы не разрабатываете для этого свою собственную хеш-таблицу; вы используете API кол-

лекций (интерфейс программирования приложений). Точно так же, если вам требуется простое веб-приложение или безопасная, интероперабельная распределенная прикладная система, оперирующая транзакциями, вы не захотите разрабатывать низкоуровневые API-интерфейсы, а будете использовать платформу Java EE. Так же как платформа Java Standard Edition (Java SE) предоставляет API для работы с коллекциями, Java EE предоставляет стандартный способ работы с транзакциями через Java API для транзакций (JTA), с сообщениями через службу сообщений Java (JMS) и с сохраняемостью через интерфейс JPA. Java EE располагает рядом спецификаций, предназначенных для корпоративных приложений. Она может рассматриваться как продолжение платформы Java SE для более удобной разработки распределенных, надежных, мощных и высокодоступных приложений.

Версия Java EE 7 стала важной вехой в развитии платформы. Она не только продолжает традиции Java EE 6, предлагая более простую модель разработки, но и добавляет свежие спецификации, обогащая наработанный функционал новыми возможностями. Кроме того, контекст и внедрение зависимостей (CDI) становится точкой интеграции между всеми новыми спецификациями. Релиз Java EE 7 почти совпадает с 13-й годовщиной выпуска корпоративной платформы. Она объединяет преимущества языка Java и опыт последних 13 лет. Java EE выигрывает как за счет динамизма сообществ свободных разработчиков, так и за счет строгой стандартизации группы Java Community Process (JCP). На сегодняшний день Java EE — это хорошо документированная платформа с опытными разработчиками, большим сообществом пользователей и множеством развертываемых приложений, работающих на серверах компаний. Java EE объединяет несколько интерфейсов API, которые могут использоваться для построения стандартных компонентно-ориентированных многозвенных приложений. Эти компоненты развертываются в различных контейнерах, предлагая серию служб.

Архитектура

Java EE состоит из набора спецификаций, реализуемых различными контейнерами. Контейнерами называются средства среды времени выполнения Java EE, предоставляющие размещенным на них компонентам определенные службы, например управление жизненным циклом разработки, внедрение зависимости, параллельный доступ и т. д. Такие компоненты используют точно определенные контракты для сообщения с инфраструктурой Java EE и с другими компонентами. Перед развертыванием они должны упаковываться стандартным способом (повторяя структуру определенного каталога, который может быть сжат в архивный файл). Java EE представляет собой расширенный набор функций платформы Java SE, что означает, что API-интерфейсы Java SE могут использоваться любыми компонентами Java EE.

Рисунок 1.1 иллюстрирует логические взаимосвязи между контейнерами. Стрелками представлены протоколы, используемые одним контейнером для доступа к другому. Например, веб-контейнер размещает сервлеты, которые могут обращаться к компонентам EJB по протоколу RMI-IIOP.

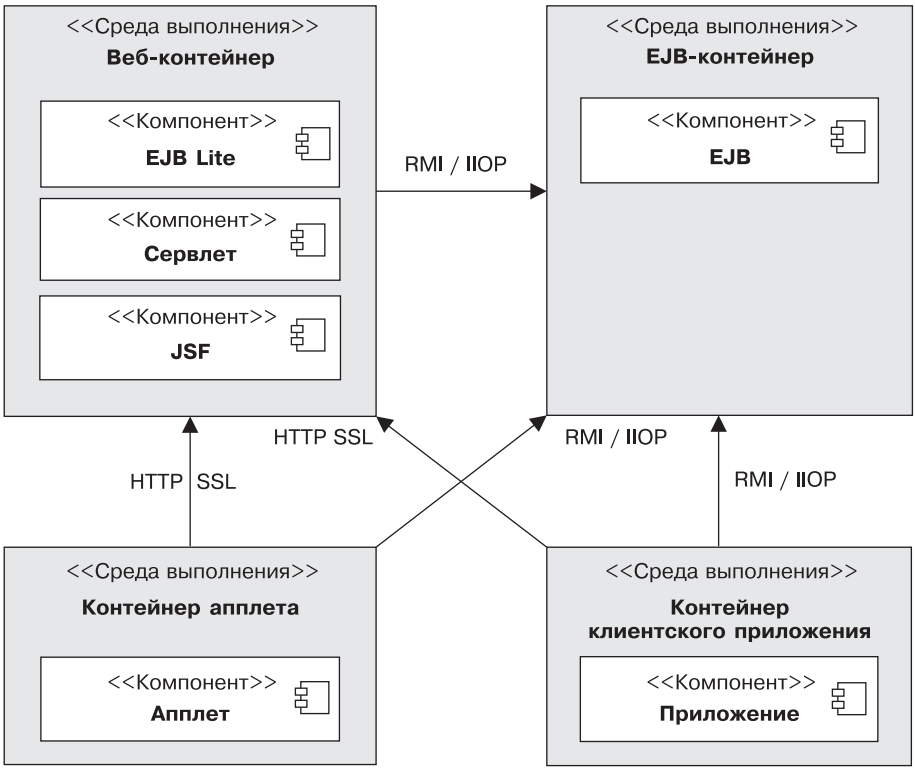


Рис. 1.1. Стандартные контейнеры Java EE

Компоненты

В среде времени выполнения Java EE выделяют четыре типа компонентов, которые должна поддерживать реализация.

- *Апплеты* представляют собой приложения из графического пользовательского интерфейса (GUI), выполняемые в браузере. Они задействуют насыщенный интерфейс Swing API для производства мощных пользовательских интерфейсов.
- *Приложениями* называются программы, выполняемые на клиентской стороне. Как правило, они относятся к графическому пользовательскому интерфейсу (GUI) и применяются для пакетной обработки. Приложения имеют доступ ко всем средствам среднего звена.
- *Веб-приложения* (состоят из сервлетов и их фильтров, слушателей веб-событий, страниц JSP и JSF) выполняются в веб-контейнере и отвечают на запросы HTTP от веб-клиентов. Сервлеты также поддерживают конечные точки веб-служб SOAP и RESTful. Веб-приложения также могут содержать компоненты EJBLite (подробнее об этом читайте в гл. 7).
- *Корпоративные приложения* (созданные с помощью технологии Enterprise Java Beans, службы сообщений Java Message Service, интерфейса Java API для транзакций, асинхронных вызовов, службы времени, протоколов RMI-IIOP) вы-

полняются в контейнере EJB. Управляемые контейнером компоненты EJB служат для обработки транзакционной бизнес-логики. Доступ к ним может быть как локальным, так и удаленным по протоколу RMI (или HTTP для веб-служб SOAP и RESTful).

Контейнеры

Инфраструктура Java EE подразделяется на логические домены, называемые контейнерами (см. рис. 1.1). Каждый из контейнеров играет свою специфическую роль, поддерживает набор интерфейсов API и предлагает компонентам сервисы (безопасность, доступ к базе данных, обработку транзакций, присваивание имен каталогам, внедрение ресурсов). Контейнеры скрывают техническую сложность и повышают мобильность. При разработке приложений каждого типа необходимо учитывать возможности и ограничения каждого контейнера, чтобы знать, использовать один или несколько. Например, для разработки веб-приложения необходимо сначала разработать уровень фреймворка JSF и уровень EJB Lite, а затем развернуть их в веб-контейнер. Но если вы хотите, чтобы веб-приложение удаленно вызывало бизнес-уровень, а также использовало передачу сообщений и асинхронные вызовы, вам потребуется как веб-, так и EJB-контейнер.

Java EE использует четыре различных контейнера.

- *Контейнеры апплетов* выполняются большинством браузеров. При разработке апплетов можно сконцентрироваться на визуальной стороне приложения, в то время как контейнер обеспечивает безопасную среду. Контейнер апплета использует модель безопасности изолированной программной среды («песочницы»), где коду, выполняемому в «песочнице», не разрешается «играть» за ее пределами. Это означает, что контейнер препятствует любому коду, загруженному на ваш локальный компьютер, получать доступ к локальным ресурсам системы (процессам либо файлам).
- *Контейнер клиентского приложения (ACC)* включает набор Java-классов, библиотек и других файлов, необходимых для реализации в приложениях Java SE таких возможностей, как внедрение, управление безопасностью и служба именования (в частности, Swing, пакетная обработка либо просто класс с методом `main()`). Контейнер ACC обращается к EJB-контейнеру, используя протокол RMI-IIOP, а к веб-контейнеру — по протоколу HTTP (например, для веб-служб).
- *Веб-контейнер* предоставляет базовые службы для управления и исполнения веб-компонентов (сервлетов, компонентов EJB Lite, страниц JSP, фильтров, слушателей, страниц JSF и веб-служб). Он отвечает за создание экземпляров, инициализацию и вызов сервлетов, а также поддержку протоколов HTTP и HTTPS. Этот контейнер используется для подачи веб-страниц к клиент-браузерам.
- *EJB-контейнер* отвечает за управление и исполнение компонентов модели EJB (компонент-сеансы EJB и компоненты, управляемые сообщениями), содержащих уровень бизнес-логики вашего приложения на Java EE. Он создает новые сущности компонентов EJB, управляет их жизненным циклом и обеспечивает реализацию таких сервисов, как транзакция, безопасность, параллельный доступ, распределение, служба именования либо возможность асинхронного вызова.

Сервисы

Контейнеры развертывают свои компоненты, предоставляя им соответствующие базовые сервисы. Контейнеры позволяют разработчику сконцентрироваться на реализации бизнес-логики, а не решать технические проблемы, присутствующие в корпоративных приложениях. На рис. 1.2 изображены сервисы, предлагаемые каждым контейнером. Например, веб- и EJB-контейнеры предоставляют коннекторы для доступа к информационной системе предприятия, но не к контейнеру апплетов или контейнеру клиентского приложения. Java EE предлагает следующие сервисы.

- *Java API для транзакций* — этот сервис предлагает интерфейс разграничения транзакций, используемый контейнером и приложением. Он также предоставляет интерфейс между диспетчером транзакций и диспетчером ресурсов на уровне интерфейса драйвера службы.
- *Интерфейс сохраняемости Java* — стандартный интерфейс для объектно-реляционного отображения (ORM). С помощью встроенного языка запросов JPQL вы можете обращаться к объектам, хранящимся в основной базе данных.
- *Валидация* — благодаря валидации компонентов объявляется ограничение целостности на уровне класса и метода.
- *Интерфейс Java для доступа к службам сообщений* — позволяет компонентам асинхронно обмениваться данными через сообщения. Он поддерживает надежный обмен сообщениями по принципу «от точки к точке» (P2P), а также модель публикации-подписки (pub-sub).
- *Java-интерфейс каталогов и служб именования (JNDI)* — этот интерфейс, появившийся в Java SE, используется как раз для доступа к системам служб именования и каталогов. Ваше приложение применяет его, чтобы ассоциировать (связывать) имена с объектами и затем находить их в каталогах. Вы можете задать поиск источников данных, фабрик классов JMS, компонентов EJB и других ресурсов. Интерфейс JNDI, повсеместно присутствовавший в коде до версии 1.4 J2EE, в настоящее время используется более прозрачным способом — через внедрение.
- *Интерфейс JavaMail* — многим приложениям требуется функция отправки сообщений электронной почты, которая может быть реализована благодаря этому интерфейсу.
- *Фреймворк активизации компонентов JavaBeans (JAF)* — интерфейс JAF, являющийся составной частью платформы Java SE, предоставляет фреймворк для обработки данных различных MIME-типов. Используется сервисом JavaMail.
- *Обработка XML* — большинство компонентов Java EE могут развертываться с помощью опциональных дескрипторов развертывания XML, а приложениям часто приходится манипулировать XML-документами. Интерфейс Java для обработки XML (JAXP) поддерживает синтаксический анализ документов с применением интерфейсов SAX и DOM, а также на языке XSLT.
- *Обработка JSON (объектной нотации JavaScript)* — появившийся только в Java EE 7 Java-интерфейс для обработки JSON (JSON-P) позволяет приложениям

синтаксически анализировать, генерировать, трансформировать и запрашивать JSON.

- *Архитектура коннектора Java EE* — коннекторы позволяют получить доступ к корпоративным информационным системам (EIS) с компонента Java EE. К таким компонентам относятся базы данных, мейнфреймы либо программы для планирования и управления ресурсами предприятия (ERP).
- *Службы безопасности* — служба аутентификации и авторизации для платформы Java (JAAS) позволяет сервисам аутентифицироваться и устанавливать права доступа, обязательные для пользователей. Контракт поставщика сервиса авторизации Java для контейнеров (JACC) определяет соглашение о взаимодействии между сервером приложений Java EE и поставщиком сервиса авторизации, позволяя, таким образом, сторонним поставщикам такого сервиса подключаться к любому продукту Java EE. Интерфейс поставщика сервисов аутентификации Java для контейнеров (JASPIC) определяет стандартный интерфейс, с помощью которого модули аутентификации могут быть интегрированы с контейнерами. В результате модули могут установить идентификаторы подлинности, используемые контейнерами.
- *Веб-службы* — Java EE поддерживает веб-службы SOAP и RESTful. Интерфейс Java для веб-служб на XML (JAX-WS), сменивший интерфейс Java с поддержкой вызовов удаленных процедур на основе XML (JAX-RPC), обеспечивает работу веб-служб, работающих по протоколу SOAP/HTTP. Интерфейс Java для веб-служб RESTful (JAX-RS) поддерживает веб-службы, использующие стиль REST.
- *Внедрение зависимостей* — начиная с Java EE 5, некоторые ресурсы могут внедряться в управляемые компоненты. К таким ресурсам относятся источники данных, фабрики классов JMS, единицы сохраняемости, компоненты EJB и т. д. Кроме того, для этих целей Java EE 7 использует спецификации по контексту и внедрению зависимости (CDI), а также внедрение зависимости для Java (DI).
- *Управление* — Java EE с помощью специального управляющего компонента определяет API для операций с контейнерами и серверами. Интерфейс управляющих расширений Java (JMXAPI) также используется для поддержки управления.
- *Развертывание* — спецификация Java EE по развертыванию определяет соглашение о взаимодействии между средствами развертывания и продуктами Java EE для стандартизации развертывания приложения.

Сетевые протоколы

Как показано на рис. 1.2, компоненты, развертываемые в контейнерах, могут вызываться с помощью различных протоколов. Например, сервлет, развертываемый в веб-контейнере, может вызываться по протоколу HTTP, как и веб-служба с конечной точкой EJB, развертываемый в EJB-контейнере. Ниже приводится список протоколов, поддерживаемых Java EE.

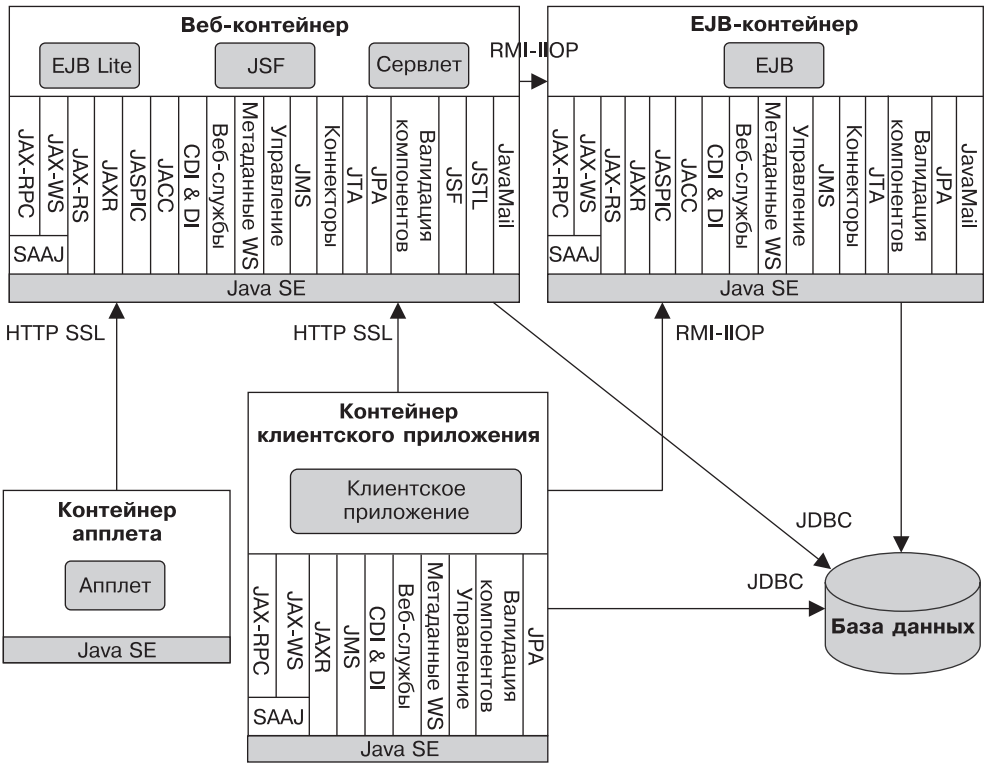


Рис. 1.2. Сервисы, предоставляемые контейнерами

- *HTTP* — веб-протокол, повсеместно используемый в современных приложениях. В Java SE клиентский API определяется пакетом `java.net`. Серверный API для работы с HTTP определяется сервлетами, JSP-страницами, интерфейсами JSF, а также веб-службами SOAP и RESTful.
- *HTTPS* — представляет собой комбинацию HTTP и протокола безопасных соединений SSL.
- *RMI-IIOP* — удаленный вызов методов (RMI) позволяет вызывать удаленные объекты независимо от основного протокола. В Java SE нативным RMI-протоколом является протокол Java для удаленного вызова методов (JMRP). RMI-IIOP представляет собой расширение технологии RMI, которое используется для интеграции с архитектурой CORBA. Язык описания Java-интерфейсов (IDL) позволяет компонентам приложений Java EE вызывать внешние объекты CORBA с помощью протокола IIOP. Объекты CORBA могут быть написаны на разных языках (Ada, C, C++, Cobol и т. д.), включая Java.

Упаковка

Для последующего развертывания в контейнере компоненты сначала необходимо упаковать в стандартно отформатированный архив. Java SE определяет файлы

архива Java (формат JAR), используемые для агрегации множества файлов (Java-классов, дескрипторов развертывания, ресурсов или внешних библиотек) в один сжатый файл (на основе формата ZIP). Как видно на рис. 1.3, Java EE определяет различные типы модулей, имеющих собственный формат упаковки, основанный на общем формате JAR.

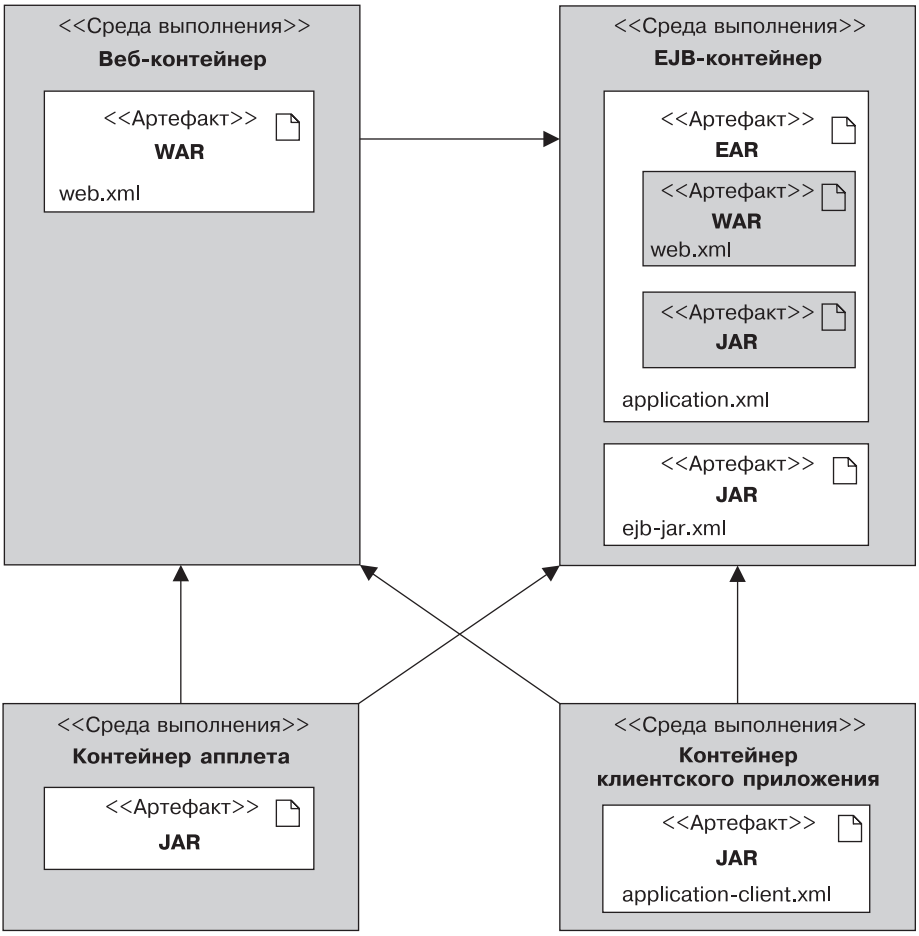


Рис. 1.3. Архивы в контейнерах

- Модуль клиентских приложений содержит Java-классы и другие ресурсные файлы, упакованные в архив JAR. Этот файл может выполняться в среде Java SE или в контейнере клиентского приложения. Как любой другой архивный формат, JAR-файл содержит опциональный каталог META-INF для мета-информации, описывающей архив. Файл `META-INF/MANIFEST.MF` используется для определения данных, относящихся к расширениям и упаковке. При развертывании в контейнере клиентских приложений соответствующий дескриптор развертывания может быть опционально размещен по адресу `META-INF/application-client.xml`.

- Модуль EJB содержит один или несколько компонент-сеансов и/или компонентов, управляемых сообщениями (MDB), упакованных в архив JAR (часто называемый JAR-файл EJB). Он содержит опциональный дескриптор развертывания `META-INF/ejb-jar.xml` и может развертываться только в контейнере EJB.
- Модуль веб-приложений содержит сервлеты, страницы JSP и JSF, веб-службы, а также любые другие файлы, имеющие отношение к Сети (страницы HTML и XHTML, каскадные таблицы стилей (CSS), Java-сценарии, изображения, видео и т. д.). Начиная с Java EE 6 модуль веб-приложения также может содержать компоненты EJB Lite (подмножество интерфейса EJBAPI, описанное в главе 7). Все эти артефакты упаковываются в архив JAR с расширением WAR (также называемый архивом WAR или веб-архивом). Опциональный веб-дескриптор развертывания определяется в файле `WEB-INF/web.xml`. Если архив WAR содержит компоненты EJB Lite, то файл `WEB-INF/ejb-jar.xml` может быть снабжен опциональным дескриптором развертывания. Java-файлы с расширением `.class` помещаются в каталог WEB-INF/classes, а зависимые архивные JAR-файлы — в каталог WEB-INF/lib.
- Корпоративный модуль может содержать нуль или более модулей веб-приложений, модулей EJB, а также других общих или внешних библиотек. Они упаковываются в корпоративный архив (файл JAR с расширением `.ear`) таким образом, чтобы развертывание всех этих модулей происходило одновременно и согласованно. Опциональный дескриптор развертывания корпоративного модуля определяется в файле `META-INF/application.xml`. Специальный каталог lib используется для разделения общих библиотек по модулям.

Аннотации и дескрипторы развертывания

В парадигме программирования существует два подхода: императивное и декларативное программирование. Первое устанавливает алгоритм для достижения цели (что должно быть сделано), тогда как второе определяет, как достичь цели (как это должно быть сделано). В Java EE декларативное программирование выполняется с помощью метаданных, а именно аннотаций и/или дескрипторов развертывания.

Как вы могли видеть на рис. 1.2, компоненты выполняются в контейнере, который, в свою очередь, дает компоненту набор сервисов. Метаданные используются для объявления и настройки этих сервисов, а также для ассоциирования с ними дополнительной информации, наряду с Java-классами, интерфейсами, конструкторами, методами, полями либо параметрами.

Начиная с Java EE 5, количество аннотаций в корпоративной платформе неуклонно растет. Они декорируют метаданными ваш код (Java-классы, интерфейсы, поля, методы). Листинг 1.1 показывает простой Java-объект в старом стиле (POJO), объявляющий определенное поведение с использованием аннотаций к классу и к атрибуту (подробнее о компонентах EJB, контексте хранения и аннотациях — в следующих главах).

Листинг 1.1. Компонент EJB с аннотациями

```
@Stateless
@Remote(ItemRemote.class)
@Local(ItemLocal.class)
```

```
@LocalBean
public class ItemEJB implements ItemLocal, ItemRemote {
  @PersistenceContext(unitName = "chapter01PU")
  private EntityManager em;
  public Book findBookById(Long id) {
    return em.find(Book.class, id);
  }
}
```

Второй способ объявления метаданных — использование дескрипторов развертывания. Дескриптор развертывания (DD) означает файл XML-конфигуратора, который развертывается в контейнере вместе с компонентом. Листинг 1.2 показывает дескриптор развертывания компонента EJB. Как и большинство дескрипторов развертывания в Java EE 7, он определяет пространство имен `http://xmlns.jcp.org/xml/ns/javaee` и содержит атрибут версии с указанием версии спецификации.

Листинг 1.2. Дескриптор развертывания компонента EJB

```
<ejb-jar xmlns="http://xmlns.jcp.org/xml/ns/javaee" ➥
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee ➥
                     http://xmlns.jcp.org/xml/ns/javaee/ejb-jar_3_2.xsd" ➥
         version="3.2" >

  <enterprise-beans>
    <session>
      <ejb-name>ItemEJB</ejb-name>
      <remote>org.agoncal.book.javaee7.ItemRemote</remote>
      <local>org.agoncal.book.javaee7.ItemLocal</local>
      <local-bean/>
      <ejb-class>org.agoncal.book.javaee7.ItemEJB</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Для того чтобы дескрипторы развертывания учитывались при работе, они должны упаковываться вместе с компонентами в специальный каталог META-INF или WEB-INF. В табл. 1.1 приведен список дескрипторов развертывания Java EE и соответствующая спецификация (подробнее об этом читайте в следующих главах).

Таблица 1.1. Дескрипторы развертывания в Java EE0

Файл	Спецификация	Местоположение
application.xml	Java EE	META-INF
application-client.xml	Java EE	META-INF
beans.xml	CDI	META-INF или WEB-INF

Продолжение ⇩

Таблица 1.1 *(продолжение)*

Файл	Спецификация	Местоположение
ra.xml	JCA	META-INF
ejb-jar.xml	EJB	META-INF или WEB-INF
faces-config.xml	JSF	WEB-INF
persistence.xml	JPA	META-INF
validation.xml	Валидация компонентов	META-INF или WEB-INF
web.xml	Сервлет	WEB-INF
web-fragment.xml	Сервлет	WEB-INF
webservices.xml	Веб-службы SOAP	META-INF или WEB-INF

Начиная с Java EE 5, большинство дескрипторов развертывания опциональны, и вместо них можно использовать аннотацию. Но вы также можете взять для вашего приложения наилучшее от обоих методов. Самое большое преимущество аннотаций в том, что они могут значительно сократить количество кода, который необходимо написать разработчику. Кроме того, используя аннотацию, вы можете избежать необходимости в дескрипторе развертывания. С другой стороны, дескрипторы развертывания — это внешние XML-файлы, для замены которых не требуется изменений исходного кода и рекомпиляции. Если вы используете сразу оба метода, то во время развертывания приложения или компонента метаданные переопределяются дескриптором развертывания (то есть XML приоритетнее аннотаций).

ПРИМЕЧАНИЕ

Сегодня в Java-программировании предпочтительнее использование аннотаций, а не дескрипторов развертывания. Это происходит в рамках тенденции перехода от двуязычного программирования (Java + XML) к одноязычному (Java). Точно так же приложение проще анализировать и создавать его прототип, когда все (данные, методы и метаданные с аннотациями) хранится в одном месте.

Java EE использует понятие программирования путем исключения (также известное как программирование по соглашениям), когда большая часть общего поведения не требует сопровождения метаданными («в программировании метаданные являются исключением, контейнер заботится о настройках по умолчанию»). Это означает, что даже при малом количестве аннотаций или XML контейнер может выдать стандартный набор настроек с заданным по умолчанию поведением.

Стандарты

Платформа Java EE основана на нескольких стандартах. Это означает, что Java EE проходит процесс стандартизации, принятый группой Java Community Process, и описывается в спецификациях. На самом деле Java EE объединяет несколько других спецификаций (или запросов на спецификацию Java), поэтому ее можно называть обобщающей. Вы можете спросить, почему стандарты так важны, если наиболее успешные Java-фреймворки не стандартизированы (Struts, Spring и т. д.).

На протяжении всей своей истории люди создавали стандарты, чтобы облегчить коммуникацию и обмен. В качестве наиболее выдающихся примеров можно привести язык, валюту, время, навигацию, системы измерений, инструменты, железные дороги, электричество, телеграф, телефонию, протоколы и языки программирования.

В самом начале развития Java, если вы занимались корпоративной или веб-разработкой, вы должны были подчиняться законам проприетарного мира, создавая свои собственные фреймворки, либо ограничивать себя проприетарным коммерческим фреймворком. Затем настало время свободных фреймворков, которые не всегда основываются на открытых стандартах. Вы можете использовать свободные фреймворки и ограничиваться одной-единственной реализацией либо применять такие открытые фреймворки, которые подчиняются стандартам, чтобы приложение было портируемым. Java EE предусматривает открытые стандарты, реализуемые несколькими коммерческими (WebLogic, Websphere, MQSeries и др.) или свободными (GlassFish, Jboss, Hibernate, Open JPA, Jersey и т. д.) фреймворками для работы с транзакциями, безопасностью, хранящими состояние компонентами, хранимостью объектов и т. д. Сегодня ваше приложение может быть развернуто на любом совместимом сервере приложений с очень малым количеством изменений.

JCP

Java Communication Process — открытая организация, созданная в 1998 году компанией Sun Microsystems. Одно из направлений работы JCP — определение будущих версий и функционала платформы Java. Когда определяется необходимость в стандартизации существующего компонента или интерфейса, ее инициатор (руководитель спецификации) создает запрос на спецификацию Java (JSR) и формирует группу экспертов. Такие группы, состоящие из представителей компаний, организаций, университетов или частных лиц, ответственны за разработку запроса на спецификацию (JSR) и по итогу представляют:

- одну или несколько спецификаций, объясняющих детали и определяющих основные понятия запроса на спецификацию Java (JSR);
- *базовую реализацию* (RI), которая является фактической реализацией спецификации;
- *пакет проверки совместимости* (известный также как *пакет проверки технологической совместимости*, или TCK), представляющий собой набор тестов, которые должна пройти каждая реализация для подтверждения соответствия спецификации.

После одобрения исполнительным комитетом (EC) спецификация выпускается в Java-сообщество для внедрения.

Портируемость

С самого создания целью Java EE было обеспечить разработку приложения и его развертывание на любом сервере приложений без изменения кода или конфигурационных файлов. Это никогда не было так просто, как казалось. Спецификации

не покрывают всех деталей, а реализации в итоге предоставляют непортируемые решения. Например, это случилось с именами JNDI. При развертывании компонента EJB на серверах GlassFish, Jboss или WebLogic на каждом из них было свое собственное имя для JNDI, так как в спецификации оно не было явно указано. В код приходилось вносить изменения в зависимости от используемого сервера приложений. Эта конкретная проблема была устранена в Java EE, когда установили синтаксис для имен JNDI.

Сегодня платформа представила наиболее портируемые параметры конфигурации, увеличив таким образом портируемость. Несмотря на то что некоторые API были признаны устаревшими (отсечены), приложения Java EE сохраняют полную совместимость с предыдущими версиями, позволяя вашему приложению перемещаться до более новых версий сервера приложений без особых проблем.

Модель программирования

Большинство спецификаций Java EE 7 используют одну и ту же модель программирования. Обычно это POJO с некоторыми метаданными (аннотациями или XML), которые развертываются в контейнере. Чаще всего POJO даже не реализует интерфейс и не расширяет суперкласс. Благодаря метаданным контейнер знает, какие сервисы необходимо применить к этому развертываемому компоненту.

В Java EE 7 сервлеты, управляемые компоненты JSF, компоненты EJB, сущности, веб-службы SOAP и REST являются аннотированными классами с опциональными дескрипторами развертывания XML. Листинг 1.3 показывает управляемый компонент JSF, представляющий собой Java-класс с одиночной CDI-аннотацией.

Листинг 1.3. Управляющий компонент JSF

```
@Named
public class BookController {
  @Inject
  private BookEJB bookEJB;
  private Book book = new Book();
  private List<Book> bookList = new ArrayList<Book>();
  public String doCreateBook() {
    book = bookEJB.createBook(book)
    bookList = bookEJB.findBooks();
    return "listBooks.xhtml";
  }
  // Геттеры, сеттеры
}
```

Компоненты EJB следуют той же модели. Как показано в листинге 1.4, если вам нужно обратиться к компоненту EJB локально, для этого достаточно простого аннотированного класса без интерфейса. Компоненты EJB могут также развертываться напрямую в файл WAR без предварительного упаковывания в файл JAR. Благодаря этому компоненты EJB являются простейшими транзакционными сущностями и могут быть использованы как в сложных корпоративных, так и в простых веб-приложениях.

Листинг 1.4. Компонент EJB без сохранения состояния

```
@Stateless
public class BookEJB {
  @Inject
  private EntityManager em;
    public Book findBookById(Long id) {
    return em.find(Book.class, id);
  }
  public Book createBook(Book book) {
    em.persist(book);
    return book;
  }
}
```

Веб-службы RESTful широко распространились в современных приложениях. Спецификация JAX-RS для Java EE 7 была усовершенствована с учетом нужд больших предприятий. Как показано в листинге 1.5, веб-служба RESTful является аннотированным Java-классом, соответствующим действиям HTTP (подробнее читайте в главе 15).

Листинг 1.5. Веб-служба RESTful (с сохранением состояния представления)

```
@Path("books")
public class BookResource {
  @Inject
  private EntityManager em;
  @GET
  @Produces({"application/xml", "application/json"})
  public List<Book> getAllBooks() {
    Query query = em.createNamedQuery("findAllBooks");
    List<Book> books = query.getResultList();
    return books;
  }
}
```

В следующих главах вы будете периодически встречаться с кодом такого типа, где компоненты только содержат бизнес-логику, а метаданные представлены аннотациями (или XML). Таким образом гарантируется, что контейнер использует именно необходимые сервисы.

Java Standard Edition 7

Важно подчеркнуть, что Java EE — это расширенная версия Java SE. Это означает, что в Java EE доступны все возможности языка Java, а также API.

Официальный релиз платформы Java SE 7 состоялся в июле 2011 года. Она была разработана в рамках запроса на спецификацию JSR 336 и предоставляла много новых возможностей, обеспечивая простоту разработки платформ предыдущих версий. Напомню, что функционал Java SE 5 включал автобоксинг, аннотирование, дженерики (обобщенные сущности), перечисление и т. д. Новинками Java SE 6 стали инструменты диагностирования, управления, мониторинга и интерфейс JMX API,

упрощенное выполнение сценарных языков на виртуальной машине Java. Java SE 7 объединяет запросы на спецификацию JSR 334 (чаще употребляется название Project Coin), JSR 292 (InvokeDynamic или поддержка динамических языков на виртуальной машине Java), JSR 203 (новый API ввода/вывода, обычно называемый NIO.2), а также несколько обновлений существующих спецификаций (JDBC 4.1 (JSR 221)). Даже если в этой книге не удастся подробно описать Java SE 7, некоторые из этих обновлений будут приведены в качестве примеров, поэтому я хочу предложить вам краткий обзор того, как они могут выглядеть.

Строковый оператор

До появления Java SE 7 в качестве оператора ветвления могли использоваться только числа (типов `byte`, `short`, `int`, `long`, `char`) или перечисления. Теперь стало возможным применять переключатель с цифро-буквенными значениями Strcompare. Это помогает избежать длинных списков, начинающихся с `if/then/else`, и сделать код более удобочитаемым. Теперь вы можете использовать в своих приложениях код, показанный в листинге 1.6.

Листинг 1.6. Строковый оператор

```
Stringaction = "update";
switch (action) {
  case "create":
    create();
    break;
  case "read":
    read();
    break;
  case "udpate":
    udpate();
    break;
  case "delete":
    delete();
    break;
  default:
    noCrudAction(action);
}
```

Ромбовидная нотация

Дженерики впервые появились в Java SE 5, но синтаксис их был достаточно пространным. В Java SE 7 появился более лаконичный синтаксис, называемый ромбовидным. В таком варианте записи объявление объекта не повторяется при его инстанцировании. Листинг 1.7 содержит пример объявления дженериков как с применением ромбовидного оператора, так и без него.

Листинг 1.7. Объявление дженериков

```
// Без ромбовидного оператора
List<String> list = new ArrayList<String>();
Map<Reference<Object>, Map<Integer, List<String>>> map =
```

```
        new HashMap<Reference<Object>, Map<Integer, List<String>>>();

    // С ромбовидным оператором
    List<String> list = new ArrayList<>();
    Map<Reference<Object>, Map<Integer, List<String>>> map = new HashMap<>();
```

Конструкция try-with-resources

В некоторых Java API закрытием ресурсов необходимо управлять вручную, обычно с помощью метода `close` в блоке `finally`. Это касается ресурсов, управляемых операционной системой, например файлов, сокетов или соединений интерфейса JDBC. Листинг 1.8 показывает, как необходимо ставить закрывающий код в блоке `finally` с обработкой исключений, но удобочитаемость кода из-за этого снижается.

Листинг 1.8. Закрытие входных/выходных потоков в блоках Finally

```
try {
  InputStream input = new FileInputStream(in.txt);
  try {
    OutputStream output = new FileOutputStream(out.txt);
    try {
      byte[] buf = new byte[1024];
      int len;
      while ((len = input.read(buf)) >= 0)
      output.write(buf, 0, len);
    } finally {
      output.close();
    }
  } finally {
    input.close();
  }
} catch (IOException e) {
  e.printStrackTrace();
}
```

Конструкция `try-with-resources` решает проблему читаемости с помощью нового, более простого синтаксиса. Это позволяет ресурсам в блоке `try` автоматически высвобождаться в его конце. Нотация, описанная в листинге 1.9, может использоваться для любого класса, реализующего новый интерфейс `java.lang.AutoCloseable`. Сейчас он реализуется множеством классов (`InputStream`, `OutputStream`, `JarFile`, `Reader`, `Writer`, `Socket`, `ZipFile`) и интерфейсов (`java.sql.ResultSet`).

Листинг 1.9. Закрытие входных/выходных потоков с помощью конструкции try-with-resources

```
try (InputStream input = new FileInputStream(in.txt);
     OutputStream output = new FileOutputStream(out.txt)) {
  byte[] buf = new byte[1024];
  int len;
  while ((len = input.read(buf)) >= 0)
    output.write(buf, 0, len);
```

```
} catch (IOException e) {
  e.printStrackTrace();
}
```

Multicatch-исключения

До появления Java SE 6 блок захвата мог обрабатывать только одно исключение в каждый момент времени. Поэтому приходилось накапливать несколько исключений, чтобы потом применить нужное действие для обработки исключений каждого типа. И как показано в листинге 1.10, для каждого исключения зачастую необходимо выполнять одно и то же действие.

Листинг 1.10. Использование нескольких конструкций для захвата исключений

```
try {
  // Какое-либо действие
} catch(SAXException e) {
  e.printStackTrace();
} catch(IOException e) {
  e.printStackTrace();
} catch(ParserConfigurationException e) {
  e.printStackTrace();
}
```

Если разные исключения требуют одинаковой обработки, в Java SE 7 вы можете добавить столько типов исключений, сколько нужно, разделив их прямым слешем, как показано в листинге 1.11.

Листинг 1.11. Использование Multicatch-исключения

```
try {
  // Какое-либо действие
} catch( SAXException | IOException | ParserConfigurationException e ) {
  e.printStackTrace();
}
```

NIO.2

Если вам, как и многим Java-разработчикам, с трудом удается читать или записывать некоторые файлы, вы оцените новую возможность Java SE 7: пакет ввода/вывода `java.nio`. Этот пакет, обладающий более выразительным синтаксисом, призван заменить существующий пакет `java.io`, чтобы обеспечить:

- более аккуратную обработку исключений;
- полный доступ к файловой системе с новыми возможностями (поддержка атрибутов конкретной операционной системы, символических ссылок и т. д.);
- добавление понятий `FileSystem` и `FileStore` (например, возможность разметки диска);
- вспомогательные методы (перемещение/копирование файлов, чтение/запись бинарных или текстовых файлов, путей, каталогов и т. д.).

В листинге 1.12 показан новый интерфейс `java.nio.file.Path` (используется для определения файла или каталога в файловой системе), а также утилитный класс

`java.nio.file.Files` (применяется для получения информации о файле или манипулирования им). Начиная с Java SE 7 рекомендуется использовать новый NIO.2, даже пока старый пакет `java.io` не вышел из употребления. В листинге 1.12 код получает информацию о файле `source.txt`, копирует его в файл `dest.txt`, отображает его содержимое и удаляет его.

Листинг 1.12. Использование нового пакета ввода/вывода

```
Path path = Paths.get("source.txt");
boolean exists = Files.exists(path);
boolean isDirectory = Files.isDirectory(path);
boolean isExecutable = Files.isExecutable(path);
boolean isHidden = Files.isHidden(path);
boolean isReadable = Files.isReadable(path);
boolean isRegularFile = Files.isRegularFile(path);
boolean isWritable = Files.isWritable(path);
long size = Files.size(path);
// Копирует файл
Files.copy(Paths.get("source.txt"), Paths.get("dest.txt"));
// Считывает текстовый файл
List<String> lines = Files.readAllLines(Paths.get("source.txt"), UTF_8);
for (String line : lines) {
  System.out.println(line);
}
// Удаляет файл
Files.delete(path);
```

Обзор спецификаций Java EE

Java EE — это обобщающая спецификация, которая объединяет и интегрирует остальные. На сегодняшний день для обеспечения совместимости с Java EE 7 сервер приложений должен реализовывать 31 спецификацию, а разработчику для оптимального использования контейнера необходимо знать тысячи API. Несмотря на то что требуется знать столько спецификаций и API, основная цель Java EE 7 — упростить платформу, предоставив несложную модель, основанную на POJO, веб-профиле и отсечении некоторых неактуальных технологий.

Краткая история Java EE

На рис. 1.4 кратко изложена 14-летняя эволюция Java EE. Раньше Java EE называлась J2EE. Платформа J2EE 1.2 была разработана компанией Sun и впервые выпущена в 1999 году в качестве обобщающей спецификации, содержащей десять запросов JSR. В то время всеобщий интерес вызывала архитектура CORBA, поэтому J2EE была изначально нацелена на работу с распределенными системами. В ней уже существовала архитектура Enterprise Java Beans (EJB) с поддержкой удаленных служебных объектов как с сохранением состояния, так и без него и с возможностью поддержки хранимых объектов (компонентов-сущностей EJB). Они были основаны на транзакционной и распределенной компонентной модели, а в качестве базового протокола использовали RMI-IIOP (протокол удаленного вызова методов

и обмена между ORB в Интернете). Веб-уровень содержал сервлеты и страницы JSP, а для сетевой коммуникации использовалась служба сообщений Java (JMS).

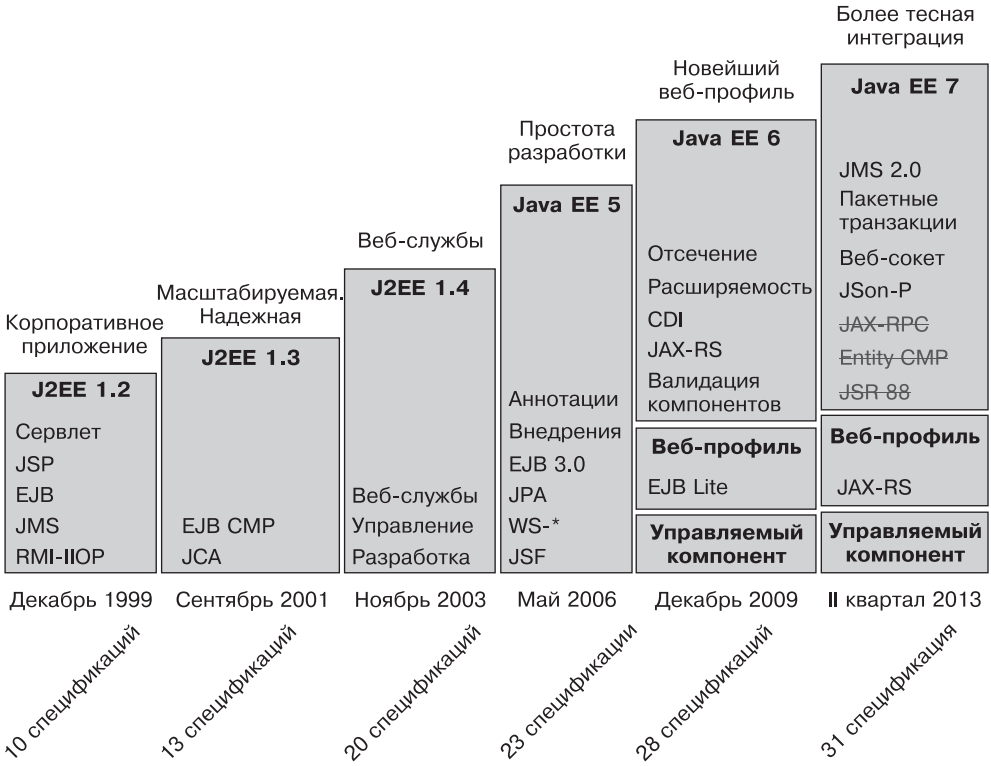


Рис. 1.4. История J2EE/Java EE

Начиная с J2EE 1.3, разработкой спецификаций занималась группа Java Community Process (JCP) в рамках запроса JSR 58. Поддержка компонентов-сущностей стала обязательной, а в компонентах EJB появились дескрипторы развертывания XML для хранения метаданных (которые были сериализованы в отдельном файле в EJB 1.0). В этой версии была решена проблема издержек, связанных с передачей аргументов по значению при использовании удаленных интерфейсов. Для ее устранения появились локальные интерфейсы и возможность передачи аргументов по ссылке. Была разработана архитектура JCA (J2EE Connector Architecture), позволившая связать Java EE с корпоративной информационной системой (EIS).

ПРИМЕЧАНИЕ

CORBA была разработана в 1988 году именно потому, что корпоративные системы становились распределенными (примеры таких систем: Tuxedo и CICS). Затем последовали EJB и J2EE, которые создавались на десять лет позже, но также предназначались для работы с распределенными системами. К моменту разработки J2EE CORBA имела основательную поддержку и являлась промышленным ПО, однако компании нашли более

простые и гибкие способы соединения распределенных систем, такие как веб-службы SOAP или REST. Поэтому для большинства корпоративных систем исчезла необходимость применения CORBA.

В 2003 году в J2EE 1.4 (запрос на спецификацию JSR 151) входило 20 спецификаций и добавилась поддержка веб-служб. Спецификация EJB 2.1 разрешала вызов компонент-сеансов EJB по протоколам SOAP/HTTP. Была создана служба времени для вызова компонентов в указанное время или с заданным интервалом. Эта версия оптимизировала поддержку сборки и развертывания приложений. И хотя сторонники J2EE предсказывали ей большое будущее, не весь обещанный функционал был реализован. Системы, созданные с ее применением, были очень сложными, а на их разработку тратилось слишком много времени даже при самых тривиальных требованиях пользователя. J2EE считалась тяжеловесной компонентной моделью, сложной для тестирования, развертывания и запуска. Как раз в то время появились фреймворки Struts, Spring и Hibernate, предложившие новый способ разработки корпоративного приложения.

К счастью, во втором квартале 2006 года вышла значительно улучшенная Java EE 5 (запрос на спецификацию JSR 244). Опираясь на примеры свободных фреймворков, она возродила модель программирования POJO. Метаданные могли определяться аннотациями, а дескрипторы развертывания XML стали опциональными. С точки зрения разработчика, создание EJB 3 и нового интерфейса JPA было настоящим качественным скачком, а не рядовым эволюционным изменением. В качестве стандартного фреймворка был представлен JavaServer Faces (JSF), а в качестве API для работы с SOAP-службами стал использоваться JAX-WS 2.0, заменивший JAX-RPC.

В 2009 году появилась Java EE 6 (запрос на спецификацию JSR 316), продолжившая традицию простоты разработки. Она включала в себя концепции аннотаций, программирования POJO, а также механизма конфигурации путем исключения в масштабах всей платформы, в том числе на веб-уровне. Ее отличал широкий спектр инноваций, например принципиально новый интерфейс JAX-RS 1.1, валидация компонентов 1.0, контекст и внедрение зависимости (CDI 1.0). Некоторые из ранее разработанных API (например, EJB 3.1) были упрощены, другие, наоборот, доработаны (JPA 2.0 или служба времени EJB). Однако основными новшествами Java EE 6 стали портируемость (например, с помощью стандартизации глобального именования JNDI), избавление от некоторых спецификаций (с помощью отсечения) и создание подсистем платформы с использованием профилей.

Сегодня Java EE 7 предлагает много новых спецификаций (пакетная обработка, веб-сокеты, обработка JSON), а также совершенствует существующие. Java EE 7 также улучшает интеграцию между технологиями, задействуя контексты и внедрения зависимостей (CDI) в большинстве спецификаций. В данной книге я хочу продемонстрировать эти улучшения, а также показать, насколько проще и полнее стала платформа Java Enterprise Edition.

Отсечение

Впервые версия Java EE была выпущена в 1999 году, и с тех пор в каждом релизе добавлялись новые спецификации (см. рис. 1.4). Постепенно это стало проблемой. Некоторые функции поддерживались не полностью или не очень широко применялись,

так как были технологически устаревшими, либо им находились достойные альтернативы. Поэтому экспертная группа внесла предложение об удалении некоторого функционала методом отсечения. Процесс отсечения заключается в составлении списка функций, подлежащих возможному удалению в следующем релизе Java EE. Обратите внимание, что ни одна из функций, предложенных к удалению, не убирается из текущей версии, но может быть удалена в следующей. Java EE 6 предложила удалить следующие спецификации и функции, и в Java EE 7 их уже не было.

- *Компоненты-сущности EJB 2.xCMP (входили в запрос на спецификацию JSR 318).* Сложная и тяжеловесная модель персистентности, состоящая из компонентов-сущностей EJB2.x, была заменена интерфейсом JPA.
- *Интерфейс JAX-RPC (запрос на спецификацию JSR 101).* Это была первая попытка моделирования веб-служб SOAP в качестве вызовов удаленных процедур (RPC). Ему на замену пришел гораздо более простой в использовании и надежный интерфейс JAX-WS.
- *Интерфейс JAXR (запрос на спецификацию JSR 93).* JAXR — интерфейс, посвященный обмену данными с реестрами стандарта UDDI. Поскольку этот стандарт недостаточно широко используется, интерфейс JAXR исключен из Java EE и развивается в рамках отдельного запроса JSR.
- *Развертывание приложений Java EE (запрос JSR 88).* JSR 88 — спецификация, которую разработчики инструментов могут использовать для развертывания на серверах приложений. Этот интерфейс не получил большой поддержки разработчиков, поэтому он также исключается из Java EE 7 и будет развиваться в рамках отдельной спецификации.

Спецификации Java EE 7

Спецификация Java EE 7 определяется запросом JSR 342 и объединяет в себе 31 спецификацию. Сервер приложений, призванный обеспечить совместимость с Java EE 7, должен реализовывать все эти спецификации. Они перечислены в табл. 1.2–1.6, где сгруппированы по технологическим предметным областям, с указанием версии и номера запроса JSR.

Таблица 1.2. Спецификация Java Enterprise Edition

Спецификация	Версия	JSR	URL
Java EE	7.0	342	http://jcp.org/en/jsr/detail?id=342
Web Profile (Веб-профиль)	7.0	342	http://jcp.org/en/jsr/detail?id=342
Managed Beans (Управляемые компоненты)	1.0	316	http://jcp.org/en/jsr/detail?id=316

В области веб-служб (см. табл. 1.3) службы SOAP не дорабатывались, так как никакие спецификации не обновлялись (см. главу 14).

Веб-службы REST в последнее время активно использовались в наиболее важных веб-приложениях. Интерфейс JAX-RS 2.0 также подвергся крупному

обновлению, в частности, в нем появился клиентский API (см. главу 15). Новая спецификация обработки объектных нотаций JavaScript (JSON-P) эквивалентна интерфейсу Java для обработки XML (JAXP), только вместо XML используется JSON (см. главу 12).

Таблица 1.3. Спецификация веб-служб

Спецификация	Версия	JSR	URL
JAX-WS	2.2a	224	http://jcp.org/en/jsr/detail?id=224
JAXB	2.2	222	http://jcp.org/en/jsr/detail?id=222
Web Services (Веб-службы)	1.3	109	http://jcp.org/en/jsr/detail?id=109
Web Services Metadata (Метаданные веб-служб)	2.1	181	http://jcp.org/en/jsr/detail?id=181
JAX-RS	2.0	339	http://jcp.org/en/jsr/detail?id=339
JSON-P	1.0	353	http://jcp.org/en/jsr/detail?id=353

В веб-спецификациях (см. табл. 1.4) не вносилось никаких изменений в страницы JSP и библиотеки тегов JSTL, поскольку эти спецификации не обновлялись. Из JSP-страниц был выделен язык выражений, который сейчас развивается в рамках отдельного запроса на спецификацию (JSR 341). Сервлет и фреймворк JSF (см. главы 10 и 11) были обновлены. Кроме того, в Java EE 7 был представлен новейший интерфейс WebSocket 1.0.

Таблица 1.4. Веб-спецификации

Спецификация	Версия	JSR	URL
JSF	2.2	344	http://jcp.org/en/jsr/detail?id=344
JSP	2.3	245	http://jcp.org/en/jsr/detail?id=245
Debugging Support for Other Languages (Поддержка отладки для других языков)	1.0	45	http://jcp.org/en/jsr/detail?id=45
JSTL[1]	1.2	52	http://jcp.org/en/jsr/detail?id=52
Servlet (Сервлет)	3.1	340	http://jcp.org/en/jsr/detail?id=340
WebSocket (Веб-сокет)	1.0	356	http://jcp.org/en/jsr/detail?id=356
Expression Language (Язык выражений)	3.0	341	http://jcp.org/en/jsr/detail?id=341

В области корпоративных приложений (см. табл. 1.5) выполнено два основных обновления: JMS 2.0 (см. главу 13) и интерфейс JTA 1.2 (см. главу 9), до этого не обновлявшиеся более десяти лет. В свою очередь, спецификации по компонентам EJB (см. главы 7 и 8), интерфейсу JPA (см. главы 4–6) и перехватчикам (см. главу 2) перешли в эту версию с минимальными обновлениями.

[1] JSTL (JavaServer Pages Standard Tag Library) — стандартная библиотека тегов для страниц JavaServer.

Таблица 1.5. Корпоративные спецификации

Спецификация	Версия	JSR	URL
EJB	3.2	345	http://jcp.org/en/jsr/detail?id=345
Interceptors (Перехватчики)	1.2	318	http://jcp.org/en/jsr/detail?id=318
JavaMail	1.5	919	http://jcp.org/en/jsr/detail?id=919
JCA	1.7	322	http://jcp.org/en/jsr/detail?id=322
JMS	2.0	343	http://jcp.org/en/jsr/detail?id=343
JPA	2.1	338	http://jcp.org/en/jsr/detail?id=338
JTA	1.2	907	http://jcp.org/en/jsr/detail?id=907

Java EE 7 включает несколько других спецификаций (см. табл. 1.6), например новый функционал пакетной обработки (запрос JSR 352) и утилиты параллельного доступа для Java EE (запрос JSR 236). Среди других обновлений стоит отметить валидацию компонентов версии 1.1 (см. главу 3), контекст и внедрение зависимостей CDI 1.1 (см. главу 2) и интерфейс JMS 2.0 (см. главу 13).

Таблица 1.6. Управление, безопасность и другие спецификации

Спецификация	Версия	JSR	URL
JACC	1.4	115	http://jcp.org/en/jsr/detail?id=115
Bean Validation (Валидация компонентов)	1.1	349	http://jcp.org/en/jsr/detail?id=349
Contexts and Dependency Injection (Контексты и внедрение зависимости)	1.1	346	http://jcp.org/en/jsr/detail?id=346
Dependency Injection for Java (Внедрение зависимости для Java)	1.0	330	http://jcp.org/en/jsr/detail?id=330
Batch (Пакетная обработка)	1.0	352	http://jcp.org/en/jsr/detail?id=352
Concurrency Utilities for Java EE (Утилиты параллельного доступа для Java EE)	1.0	236	http://jcp.org/en/jsr/detail?id=236
Java EE Management (Управление Java EE)	1.1	77	http://jcp.org/en/jsr/detail?id=77
Java Authentication Service Provider Interface for Containers (Интерфейс поставщика сервисов аутентификации Java для контейнеров)	1.0	196	http://jcp.org/en/jsr/detail?id=196

Java EE 7 не только состоит из 31 собственной спецификации, но и в большой степени опирается на Java SE 7. В табл. 1.7 перечислены спецификации, которые относятся к Java SE, но влияют на Java EE.

Таблица 1.7. Смежные корпоративные технологии в Java SE 7

Спецификация	Версия	JSR	URL
Common Annotations (Общие аннотации)	1.2	250	http://jcp.org/en/jsr/detail?id=250
JDBC	4.1	221	http://jcp.org/en/jsr/detail?id=221
JNDI	1.2	—	—
JAXP	1.3	206	http://jcp.org/en/jsr/detail?id=206
StAX	1.0	173	http://jcp.org/en/jsr/detail?id=173
JAAS	1.0	—	—
JMX	1.2	3	http://jcp.org/en/jsr/detail?id=3
JAXB	2.2	222	http://jcp.org/en/jsr/detail?id=222
JAF	1.1	925	http://jcp.org/en/jsr/detail?id=925
SAAJ	1.3	—	http://java.net/projects/saaj

Спецификации веб-профиля 7

Впервые профили были представлены в Java EE 6. Их основной целью было уменьшение платформы в соответствии с нуждами разработчиков. Сегодня размер и сложность приложения, разрабатываемого Java EE 7, не имеют значения, так как вы сможете развернуть его на сервере приложений, который предложит вам API и службы по 31 спецификации. Больше всего версию Java EE критиковали за то, что она получилась слишком громоздкой. Профили были разработаны как раз для устранения этой проблемы. Как показано на рис. 1.5, профили — это подсистемы либо настройки платформы, поэтому некоторые их функции могут пересекаться с функциями платформы или других профилей.

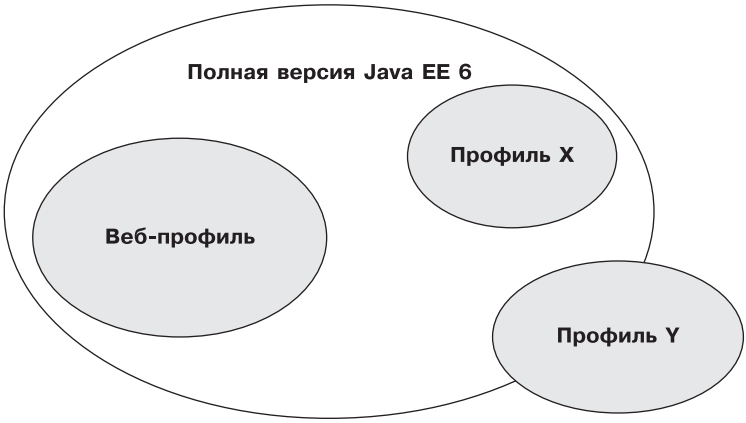


Рис. 1.5. Профили в платформе Java EE

Java EE 7 определяет один профиль, который называется веб-профилем. Его цель — позволить разработчикам создавать веб-приложения с соответствующим

набором технологий. Веб-профиль версии 7.0 указывается в отдельном JSR и на данный момент является единственным профилем платформы Java EE 7. В будущем могут быть созданы другие профили. В табл. 1.8 приведены спецификации, входящие в веб-профиль.

Таблица 1.8. Спецификации веб-профиля 7.0

Спецификация	Версия	JSR	URL
JSF	2.2	344	http://jcp.org/en/jsr/detail?id=344
JSP	2.3	245	http://jcp.org/en/jsr/detail?id=245
JSTL	1.2	52	http://jcp.org/en/jsr/detail?id=52
Servlet	3.1	340	http://jcp.org/en/jsr/detail?id=340
WebSocket	1.0	356	http://jcp.org/en/jsr/detail?id=356
Expression Language	3.0	341	http://jcp.org/en/jsr/detail?id=341
EJBLite	3.2	345	http://jcp.org/en/jsr/detail?id=345
JPA	2.1	338	http://jcp.org/en/jsr/detail?id=338
JTA	1.2	907	http://jcp.org/en/jsr/detail?id=907
Bean Validation	1.1	349	http://jcp.org/en/jsr/detail?id=349
Managed Beans	1.0	316	http://jcp.org/en/jsr/detail?id=316
Interceptors	1.2	318	http://jcp.org/en/jsr/detail?id=318
Contexts and Dependency Injection	1.1	346	http://jcp.org/en/jsr/detail?id=346
Dependency Injection for Java	1.0	330	http://jcp.org/en/jsr/detail?id=330
Debugging Support for Other Languages	1.0	45	http://jcp.org/en/jsr/detail?id=45
JAX-RS	2.0	339	http://jcp.org/en/jsr/detail?id=339
JSON-P	1.0	353	http://jcp.org/en/jsr/detail?id=353

Приложение CD-BookStore

На протяжении всей книги вы будете встречать фрагменты кода, содержащие сущности, ограничения валидации, компоненты EJB, страницы JSF, слушателей JMS, веб-службы SOAP и RESTful. Все они относятся к приложению CD-BookStore. Это приложение представляет собой коммерческий сайт, который позволяет пользователям просматривать каталог книг и компакт-дисков, имеющихся в продаже. С помощью карты покупателя посетители сайта могут выбирать товары в процессе просмотра каталога (а также удалять их из списка), а затем подсчитать общую стоимость покупки, оплатить товары и получить свой заказ. Приложение осуществляет внешние взаимодействия с банковской системой для валидации номеров кредитных карт. Схема такого примера на рис. 1.6 описывает участников и функции системы.

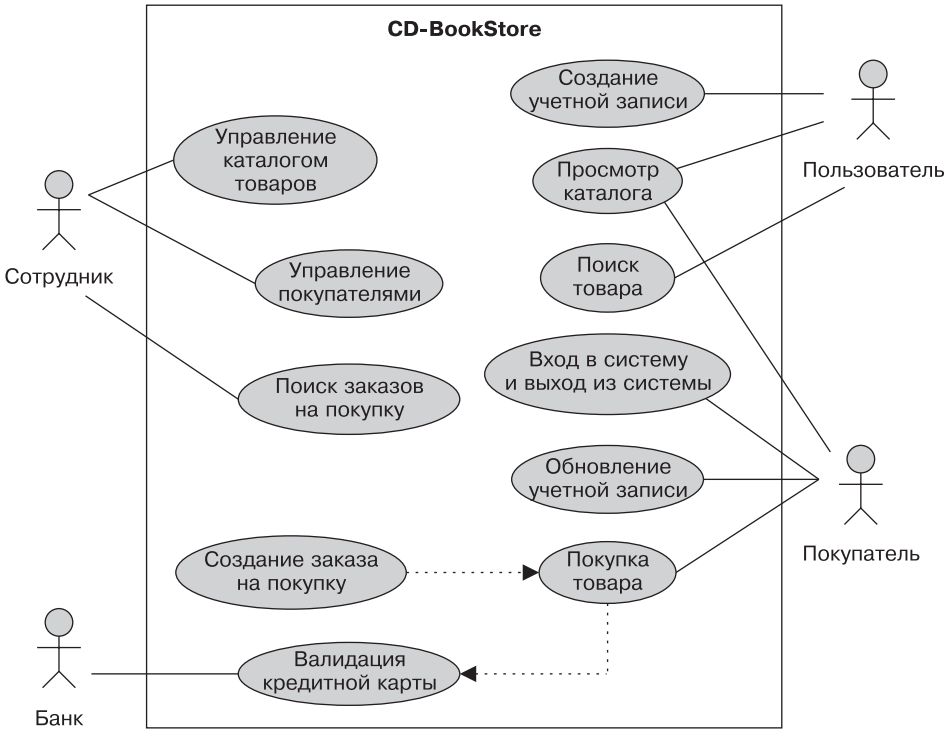


Рис. 1.6. Схема примера использования приложения CD-BookStore

Участниками, взаимодействующими с описанной системой, являются:

- сотрудники компании, которым необходимо управлять как каталогом товаров, так и пользовательской информацией. Они также могут просматривать заказы на покупку;
- пользователи — анонимные лица, посещающие сайт для просмотра каталога книг и компакт-дисков. Если они хотят купить какой-либо товар, им необходимо создать учетную запись, чтобы стать покупателями;
- покупатели, которые могут просматривать каталог, обновлять информацию в своей учетной записи и покупать товары в режиме онлайн;
- внешний банк, которому система делегирует валидацию кредитных карт.

ПРИМЕЧАНИЕ

Вы можете скачать примеры кода из этой книги прямо из репозитория Git по адресу https://github.com/agoncal/agoncal-book-javaee7.

Резюме

Если компания разрабатывает Java-приложения с добавлением таких корпоративных возможностей, как управление транзакциями, безопасность, параллельный

доступ или обмен сообщениями, то следует обратить внимание на платформу Java EE. Она хорошо стандартизирована, работает с различными протоколами, а компоненты развертываются в различные контейнеры, благодаря чему можно пользоваться многими сервисами. Java EE 7 идет по стопам предыдущей версии, упрощая использование веб-уровня. Эта версия платформы легче (благодаря технике отсечения, применению профилей и EJBLite), а также проще в использовании (нет необходимости в интерфейсах для компонентов EJB или в использовании аннотаций на веб-уровне). Благодаря новым спецификациям и функционалу, а также стандартизированному контейнеру свойств дескриптора развертывания и стандартным именам JNDI, платформа стала более насыщенной и удобной для портирования.

В этой главе я сделал очень краткий обзор Java EE 7. В следующих главах мы более подробно разберем спецификации Java EE 7. Каждая глава содержит несколько фрагментов кода и раздел «Все вместе». Вам понадобятся некоторые инструменты и фреймворки для компиляции, развертывания, запуска и тестирования кода: JDK 1.7, Maven 3, Junit 4, Derby 10.8 и Glassfish v4.

Глава 2
Контекст и внедрение зависимостей

В самой первой версии Java EE (в то время именуемой J2EE) была представлена *концепция инверсии управления (IoC)*, в рамках которой контейнер обеспечивал управление вашим бизнес-кодом и предоставлял технические сервисы (такие как управление транзакциями или безопасностью). Это подразумевало управление жизненным циклом компонентов, а также предоставление компонентам внедрения зависимостей и конфигурации. Данные сервисы были встроены в контейнер, и программистам пришлось дожидаться более поздних версий Java EE, чтобы получить к ним доступ. Конфигурация компонентов в ранних версиях стала возможной благодаря дескрипторам развертывания XML, однако простой и надежный API для управления жизненным циклом и внедрения зависимостей появился только в Java EE 5 и Java EE 6.

Версия Java EE 6 предоставила контекст и внедрение зависимостей (Context and Dependency Injection — CDI) для упрощения некоторых задач, фактически став центральной спецификацией, объединившей все эти концепции. Сегодня CDI дает управляемым объектам EJB одну из наиболее приоритетных моделей программирования. Она преобразует практически все сущности Java EE в управляемые компоненты, которые можно внедрять и перехватывать. Концепция CDI основана на принципе «слабой связанности и строгой типизации». Это означает, что компоненты связаны слабо, но при этом строго типизированы. Добавление в платформу перехватчиков, декораторов и событий придает ей дополнительную гибкость. И в то же время CDI соединяет веб-уровень и серверную часть путем гомогенизации областей видимости.

В этой главе рассказывается о внедрении зависимостей, ограничении и слабой связанности, то есть здесь охватывается большинство концепций, лежащих в основе CDI.

Понятие компонентов

В Java SE имеются компоненты JavaBeans, а в Java EE — Enterprise JavaBeans. Но Java EE также использует другие типы компонентов: сервлеты, веб-службы SOAP и RESTful, сущности и, конечно, управляемые компоненты. Не стоит забывать и об объектах POJO. POJO — это просто Java-классы, запускаемые в пределах виртуальной машины Java (JVM). JavaBeans — это те же объекты POJO, которые следуют определенным шаблонам (например, правилам именования для процессов доступа и модифицирующих методов (геттеров/сеттеров) для свойства, конструктора по умолчанию) и исполняются в пределах JVM. Все остальные компоненты Java EE также следуют

определенным шаблонам: например, компонент Enterprise JavaBean должен иметь метаданные, конструктор по умолчанию не может быть конечным, и т. д. Они также выполняются внутри контейнера (например, контейнера EJB), который предоставляет определенные сервисы: например, транзакции, организацию пула, безопасность и т. д. Теперь поговорим о простых и управляемых компонентах.

Управляемые компоненты — это объекты, которые управляются контейнером и поддерживают только небольшой набор базовых сервисов: внедрение ресурса, управление жизненным циклом и перехват. Они появились в Java EE 6, обеспечив более легковесную компонентную модель, приведенную в соответствие с остальной частью платформы Java EE. Они дают общее основание различным типам компонентов, существующих в платформе Java EE. Например, Enterprise JavaBean может рассматриваться как управляемый компонент с дополнительными сервисами. Сервлет также может считаться управляемым компонентом с дополнительными сервисами (отличным от EJB) и т. д.

Компоненты — это объекты CDI, основанные на базовой модели управляемых компонентов. Они имеют улучшенный жизненный цикл для объектов с сохранением состояния; привязаны к четко определенным контекстам; обеспечивают сохранение безопасности типов при внедрении зависимостей, перехвате и декорации; специализируются с помощью аннотаций квалификатора; могут использоваться в языке выражений (EL). По сути, с очень малым количеством исключений потенциально каждый класс Java, имеющий конструктор по умолчанию и исполняемый внутри контейнера, является компонентом. Поэтому компоненты JavaBeans и Enterprise JavaBeans также могут воспользоваться преимуществами этих сервисов CDI.

Внедрение зависимостей

Внедрение зависимостей (DI) — это шаблон разработки, в котором разделяются зависимые компоненты. Здесь мы имеем дело с инверсией управления, причем инверсии подвергается процесс получения необходимой зависимости. Этот термин был введен Мартином Фаулером. Внедрение зависимостей в такой управляемой среде, как Java EE, можно охарактеризовать как полную противоположность применения интерфейса JNDI. Объекту не приходится искать другие объекты, так как контейнер внедряет эти зависимые сущности без вашего участия. В этом состоит так называемый принцип Голливуда: «Не звоните нам (не ищите объекты), мы сами вам позвоним (внедрим объекты)».

Java EE была создана в конце 1990-х годов, и в самой первой версии уже присутствовали компоненты EJB, сервлеты и служба JMS. Эти компоненты могли использовать JNDI для поиска ресурсов, управляемых контейнером, таких как интерфейс JDBC DataSource, JMS-фабрики либо адреса назначения. Это сделало возможной зависимость компонентов и позволило EJB-контейнеру взять на себя сложности управления жизненным циклом ресурса (инстанцирование, инициализацию, упорядочение и предоставление клиентам ссылок на ресурсы по мере необходимости). Однако вернемся к теме внедрения ресурса, выполняемого контейнером.

В платформе Java EE 5 появилось внедрение ресурсов для разработчиков. Это позволило им внедрять такие ресурсы контейнера, как компоненты EJB, менеджер

сущностей, источники данных, фабрики JMS и адреса назначения, в набор определенных компонентов (сервлетов, связующих компонентов JSF и EJB). С этой целью Java EE 5 предоставила новый набор аннотаций (`@Resource`, `@PersistenceContext`, `@PersistenceUnit`, `@EJB` и `@WebServiceRef`).

Новшеств Java EE 5 оказалось недостаточно, и тогда Java EE 6 создала еще две спецификации для добавления в платформу настоящего внедрения зависимостей (DI): Dependency Injection (запрос JSR 330) и Contexts and Dependency Injection (запрос JSR 299). На сегодняшний день в Java EE 7 внедрение зависимостей используется еще шире: для связи спецификаций.

Управление жизненным циклом

Жизненный цикл POJO достаточно прост: вы, Java-разработчик, создаете экземпляр класса, используя ключевое слово `new`, и ждете, пока *сборщик мусора* (Garbage Collector) избавится от него и освободит некоторое количество памяти. Но если вы хотите запустить компонент CDI внутри контейнера, вам нельзя указывать ключевое слово `new`. Вместо этого вам необходимо внедрить компонент, а все остальное сделает контейнер. Тут подразумевается, что только контейнер отвечает за управление жизненным циклом компонента: сначала он создает экземпляр, затем избавляется от него. Итак, как же вам инициализировать компонент, если вы не можете вызвать конструктор? В этом случае контейнер дает вам указатель после конструкции экземпляра и перед его уничтожением.

Рисунок 2.1 показывает жизненный цикл управляемого компонента (следовательно, и компонента CDI). Когда вы внедряете компонент, только контейнер (EJB, CDI или веб-контейнер) отвечает за создание экземпляра (с использованием кодового слова `new`). Затем он разрешает зависимости и вызывает любой метод с аннотацией `@PostConstruct` до первого вызова бизнес-метода на компоненте. После этого оповещение с помощью обратного вызова `@PreDestroy` сигнализирует о том, что экземпляр удаляется контейнером.

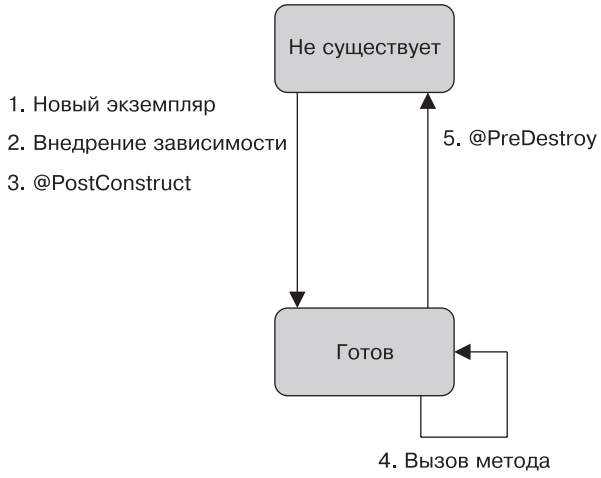


Рис. 2.1. Жизненный цикл управляемого компонента

В следующих главах вы увидите, что большинство компонентов Java EE следуют жизненному циклу, описанному на рис. 2.1.

Области видимости и контекст

Компоненты CDI могут сохранять свое состояние и являются контекстуальными. Это означает, что они живут в пределах четко определенной области видимости. В CDI такие области видимости предопределены в пределах запроса, сеанса, приложения и диалога. Например, контекст сеанса и его компоненты существуют в течение жизни сеанса HTTP. В течение этого времени внедренные ссылки на компоненты также оповещены о контексте. Таким образом, целая цепочка зависимостей компонентов является контекстуальной. Контейнер автоматически управляет всеми компонентами в пределах области видимости, а в конце сессии автоматически уничтожает их.

В отличие от компонентов, не сохраняющих состояние (например, сеансовых объектов без сохранения состояния), или синглтонов (сервлетов), различные клиенты компонента, сохраняющего состояние, видят этот компонент в разных состояниях. Когда компонент сохраняет свое состояние (ограничен сессией, приложением и диалогом), имеет значение, какой экземпляр компонента находится у клиента. Клиенты (например, другие компоненты), выполняющиеся в том же контексте, будут видеть тот же экземпляр компонента. Но клиенты в другом контексте могут видеть другой экземпляр (в зависимости от отношений между контекстами). Во всех случаях клиент не управляет жизненным циклом экземпляра исключительно путем его создания и уничтожения. Это делает контейнер в соответствии с областью видимости.

Перехват

Методы-перехватчики используются для вставки между вызовами бизнес-методов. Это похоже на аспектно-ориентированное программирование (АОП). АОП — это парадигма программирования, отделяющая задачи сквозной функциональности (влияющие на приложение) от вашего бизнес-кода. Большинство приложений имеют общий код, который повторяется среди компонентов. Это могут быть технические задачи (сделать запись в журнале и выйти из любого метода, сделать запись в журнале о длительности вызова метода, сохранить статистику использования метода и т. д.) или бизнес-логика. К последней относятся: выполнение дополнительных проверок, если покупатель приобретает товар более чем на $10 000, отправка запросов о повторном заполнении заказа, если товара недостаточно в наличии, и т. д. Эти задачи могут применяться автоматически посредством AOP ко всему вашему приложению либо к его подсистеме.

Управляемые компоненты поддерживают функциональность в стиле AOP, обеспечивая возможность перехвата вызова с помощью методов-перехватчиков. Перехватчики автоматически инициируются контейнером, когда вызывается метод управляемого компонента. Как показано на рис. 2.2, перехватчики можно объединять в цепочки и вызывать до и/или после исполнения метода.

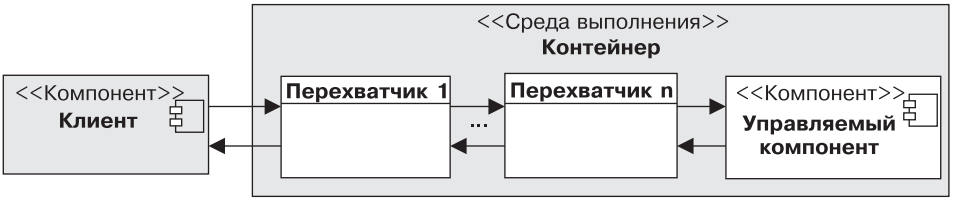


Рис. 2.2. Контейнер, перехватывающий вызов и инициирующий метод-перехватчик

Рисунок 2.2 демонстрирует количество перехватчиков, которые вызываются между клиентом и управляемым компонентом. Вы могли подумать, что EJB-контейнер представляет собой цепочку перехватчиков. Когда вы разрабатываете сеансовый объект, вы просто концентрируетесь на бизнес-коде. Но в то же самое время, когда клиент вызывает метод на вашем компоненте EJB, контейнер перехватывает вызов и применяет различные сервисы (управление жизненным циклом, транзакцию, безопасность и т. д.). Без использования перехватчиков вам приходится добавлять собственные механизмы сквозной функциональности и наиболее логичным образом применять их в вашем бизнес-коде.

Слабая связанность и строгая типизация

Перехватчики — это очень мощный способ отделения технических задач от бизнес-логики. Контекстуальное управление жизненным циклом также отделяет компоненты от управления их собственными жизненными циклами. При использовании внедрения компонент не оповещается о конкретной реализации любого компонента, с которым он взаимодействует. Но в CDI существуют и другие методы для ослабления связанности. Компоненты могут использовать уведомления о событиях для отделения производителей события от его потребителей либо применять декораторы для отделения бизнес-логики. Другими словами, слабая связанность — это ДНК, на котором построен CDI.

Все эти возможности предоставляются с сохранением безопасности типов. CDI никогда не полагается на строковые идентификаторы, чтобы определить, насколько подходят друг другу объекты. Вместо этого для скрепления компонентов CDI использует строго типизированные аннотации (например, связки квалификаторов, стереотипов и перехватчиков). Применение дескрипторов XML минимизировано до информации, связанной непосредственно с развертыванием.

Дескриптор развертывания

Почти каждая спецификация Java EE содержит опциональный дескриптор развертывания XML. Обычно он описывает, как компонент, модуль или приложение (например, корпоративное или веб-приложение) должны быть сконфигурированы. При использовании CDI дескриптор развертывания называется `beans.xml` и является обязательным. Он может применяться для конфигурирования определенного функционала (перехватчиков, декораторов, альтернатив и т. д.), но для этого необходимо активизировать CDI. Это требуется для того, чтобы CDI идентифицировал компоненты в вашем пути к классу (так называемое обнаружение компонентов).

Чудо происходит как раз на этапе обнаружения компонентов: в тот момент, когда CDI преобразует объекты POJO в компоненты CDI. Во время развертывания CDI проверяет все JAR- и WAR-файлы вашего приложения и каждый раз, когда находит дескриптор развертывания `beans.xml`, управляет всеми объектами POJO, которые впоследствии становятся компонентами CDI. Без файла `beans.xml` в пути к классу (в каталоге META-INF или WEB-INF) CDI не сможет использовать внедрение, перехват, декорацию и т. д. Без этой разметки файл CDI не будет работать. Если ваше веб-приложение содержит несколько файлов JAR и вы хотите активизировать CDI применительно ко всему приложению, то каждому файлу JAR потребуется отдельный файл `beans.xml` для инициирования CDI и обнаружения компонентов.

Обзор спецификаций по CDI

CDI стал общим основанием для нескольких спецификаций Java EE. Одни спецификации в большой степени полагаются на него (Bean Validation, JAX-RS), другие посодействовали его возникновению (EJB), а третьи связаны с ним (JSF). CDI 1.1 затрагивает несколько спецификаций, но является неполным без остальных: Dependency Injection for Java 1.0 (запрос JSR 330), Managed Bean 1.0 (запрос JSR 342), Common Annotations 1.2 (запрос JSR 250), Expression Language 3.0 (запрос JSR 341) и Interceptors 1.2 (запрос JSR 318).

Краткая история спецификаций CDI

В 2006 году Гевин Кинг (создатель Seam), вдохновленный идеями фреймворков Seam, Guise и Spring, возглавил работу над спецификацией по запросу JSR 299, позднее названной Web Beans (Веб-компоненты). Поскольку она создавалась для Java EE 6, ее пришлось переименовать в Context and Dependency Injection 1.0. При этом за основу был взят новый запрос JSR 330: Dependency Injection for Java 1.0 (также известный как `@Inject`).

Эти две спецификации взаимно дополняли друг друга и не могли использоваться в Java EE по отдельности. Внедрение зависимостей для Java определяло набор аннотаций (`@Inject`, `@Named`, `@Qualifier`, `@Scope` и `@Singleton`), используемых преимущественно для внедрения. CDI дал семантику запросу JSR 330 и добавил еще больше новых возможностей, таких как управление контекстом, события, декораторы и улучшенные перехватчики (запрос JSR 318). Более того, CDI позволил разработчику расширить платформу в рамках стандарта, что ранее не представлялось возможным. Целью CDI было заполнить все пробелы, а именно:

- придать платформе большую целостность;
- соединить веб-уровень и уровень транзакций;
- сделать внедрение зависимостей полноправным компонентом платформы;
- иметь возможность легко добавлять новые расширения.

Сегодня, с появлением Java EE 7, CDI 1.1 становится основанием для многих запросов JSR, и здесь уже появились определенные улучшения.

Что нового в CDI 1.1

В CDI 1.1 не добавилось никаких важных функций. Вместо этого новая версия концентрируется на интеграции CDI с другими спецификациями, например, благодаря активному использованию перехватчиков, добавлению диалогов в запрос сервлета либо обогащению событий жизненного цикла приложения в Java EE. Ниже перечислены новые возможности, реализованные в CDI 1.1:

- новый класс CDI обеспечивает программный доступ к средствам CDI извне управляемого компонента;
- перехватчики, декораторы и альтернативы могут быть приоритизированы (`@Priority`) и упорядочены для всего приложения;
- при добавлении аннотации `@Vetoed` к любому типу или пакету CDI перестает считать данный тип или пакет своим компонентом;
- квалификатор `@New` в CDI 1.1 считается устаревшим, и сегодня приложениям предлагается вместо этого внедрять ограниченные компоненты `@Dependent`;
- новая аннотация `@WithAnnotations` позволяет расширению фильтровать, какие типы оно будет видеть.

В табл. 2.1 перечисляются основные пакеты, относящиеся к CDI. Вы найдете аннотации и классы CDI в пакетах `javax.enterprise.inject` и `javax.decorator`.

Таблица 2.1. Основные пакеты, относящиеся к CDI

Пакет	Описание
javax.inject	Содержит базовую спецификацию по внедрению зависимостей для Java API (запрос JSR 330)
javax.enterprise.inject	Основные API для внедрения зависимостей
javax.enterprise.context	Области видимости CDI и контекстуальные API
javax.enterprise.event	События CDI и API алгоритмов наблюдения
javax.enterprise.util	Пакет утилит CDI
javax.interceptor	Содержит API перехватчика (запрос JSR 318)
javax.decorator	API декоратора CDI

Базовая реализация

Базовой реализацией CDI является Weld, свободный проект от JBoss. Есть и другие реализации, такие как Apache OpenWebBeans или CanDi (от Caucho). Важно также упомянуть проект Apache DeltaSpike, который ссылается на набор портируемых расширений CDI.

Создание компонента CDI

Компонентом CDI может быть тип любого класса, содержащий бизнес-логику. Он может вызываться напрямую из Java-кода посредством внедрения либо с помощью языка выражений (EL) со страницы JSF. Как показано в листинге 2.1,

компонент — это объект POJO, который не наследует от других объектов и не расширяет их, может внедрять ссылки на другие компоненты (`@Inject`) и имеет свой жизненный цикл, управляемый контейнером (`@PostConstruct`). Вызовы метода, выполняемые таким компонентом, могут перехватываться (здесь `@Transactional` основана на перехватчике и далее описана подробнее).

Листинг 2.1. Компонент BookService, использующий внедрение, управление жизненным циклом и перехват

```
public class BookService {
@Inject
private NumberGenerator numberGenerator;
  @Inject
  private EntityManager em;
  private Date instanciationDate;
  @PostConstruct
  private void initDate() {
    instanciationDate = new Date();
  }
  @Transactional
  public Book createBook(String title, Float price, String description) {
    Book book = new Book(title, price, description);
    book.setIsbn(numberGenerator.generateNumber());
    book.setInstanciationDate(instanciationDate);
    em.persist(book);
    return book;
  }
}
```

Внутренняя организация компонента CDI

В соответствии со спецификацией CDI 1.1 контейнер распознает как компонент CDI любой класс, если:

- он не относится к нестатичным внутренним классам;
- это конкретный класс либо класс, имеющий аннотацию `@Decorator`;
- он имеет задаваемый по умолчанию конструктор без параметров либо объявляет конструктор с аннотацией `@Inject`.

Компонент может иметь опциональную область видимости, опциональное EL-имя (EL — язык выражений), набор связок с перехватчиком и опциональное управление жизненным циклом.

Внедрение зависимостей

Java относится к объектно-ориентированным языкам программирования. Это означает, что реальный мир отображается с помощью объектов. Класс `Book` отображает копию H2G2, `Customer` замещает вас, а `PurchaseOrder` замещает то, как вы покупаете эту книгу. Эти объекты зависят друг от друга: книга может быть прочитана покупателем, а заказ на покупку может относиться к нескольким книгам. Такая зависимость — одно из достоинств объектно-ориентированного программирования.

Например, процесс создания книги (BookService) можно сократить до инстанцирования объекта Book, сгенерировав уникальный номер с использованием другого сервиса (NumberGenerator), сохраняющий книгу в базу данных. Сервис NumberGenerator может сгенерировать номер ISBN из 13 цифр либо ISBN более старого формата из восьми цифр, известный как ISSN. BookService затем будет зависеть от IsbnGenerator либо IssnGenerator. Тот или иной вариант определяется условиями работы или программным окружением.

Рисунок 2.3 демонстрирует схему класса интерфейса NumberGenerator, который имеет один метод (String generateNumber()) и реализуется посредством IsbnGenerator и IssnGenerator. Bookservice зависит от интерфейса при генерации номера книги.

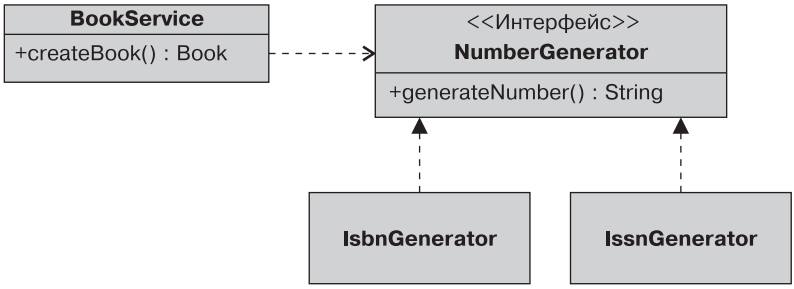


Рис. 2.3. Схема класса с интерфейсом NumberGenerator и реализациями

Как бы вы соединили BookService с ISBN-реализацией интерфейса NumberGenerator? Одно из решений — использовать старое доброе ключевое слово new, как показано в листинге 2.2.

Листинг 2.2. POJO-объект BookService, создающий зависимости с использованием ключевого слова new

```
public class BookService {
  private NumberGenerator numberGenerator;
  public BookService() {
    this.numberGenerator = new IsbnGenerator();
  }
  public Book createBook(String title, Float price, String description) {
    Book book = new Book(title, price, description);
    book.setIsbn(numberGenerator.generateNumber());
    return book;
  }
}
```

Код в листинге 2.2 достаточно прост и выполняет необходимые действия. BookService создает в конструкторе экземпляр IsbnGenerator, который затем влияет на атрибут numberGenerator. Вызов метода numberGenerator.generateNumber() сгенерирует номер из 13 цифр.

Но что, если вы хотите выбирать между реализациями, а не просто привязываться к IsbnGenerator? Одно из решений — передать реализацию конструктору и предоставить

внешнему классу возможность выбирать, какую реализацию использовать (листинг 2.3).

Листинг 2.3. Объект POJO BookService, выбирающий зависимости с использованием конструктора

```
public class BookService {
  private NumberGenerator numberGenerator;
  public BookService(NumberGenerator numberGenerator) {
    this.numberGenerator = numberGenerator;
  }
  public Book createBook(String title, Float price, String description) {
    Book book = new Book(title, price, description);
    book.setIsbn(numberGenerator.generateNumber());
    return book;
  }
}
```

Таким образом, внешний класс смог использовать `BookService` с необходимой реализацией.

```
BookService bookService = new BookService(new IsbnGenerator())
BookService bookService = new BookService(new IssnGenerator())
```

Этот пример иллюстрирует инверсию управления: инвертируется управление созданием зависимости (а не сам класс) между `BookService` и `NumberGenerator`, так как оно дается внешнему классу. Поскольку в конце вы соединяете зависимости самостоятельно, эта техника называется конструированием вручную. В предыдущем примере кода мы использовали конструктор для выбора реализации (внедрение конструктора), но еще один привычный способ состоит в использовании сеттеров (внедрение сеттера). Однако вместо конструирования зависимостей вручную вы можете перепоручить эту задачу механизму внедрения (например, CDI).

@Inject

Поскольку Java EE является управляемой средой, вам не придется конструировать зависимости вручную. Вместо вас ссылку может внедрить контейнер. Одним словом, внедрение зависимостей CDI — это возможность внедрять одни компоненты в другие с сохранением безопасности типов, что означает использование XML вместо аннотаций.

Внедрение уже существовало в Java EE 5 с такими аннотациями, как `@Resource`, `@PersistentUnit` и `EJB`. Но оно было ограничено до определенных ресурсов (баз данных, архитектура EJB) и компонентов (сервлетов, компонентов EJB, связующих компонентов JSF и т. д.). С помощью CDI вы можете внедрить практически что угодно и куда угодно благодаря аннотации `@Inject`. Обратите внимание, что в Java EE 7 разрешено использовать другие механизмы внедрения (`@Resource`), однако лучше стараться применять `@Inject` везде, где это возможно (см. подраздел «Производители данных» раздела «Создание компонента CDI» этой главы).

Листинг 2.4 показывает, как внедрять в `BookService` ссылку на `NumberGenerator` с помощью объекта CDI.

Листинг 2.4. BookService, использующий @Inject для получения ссылки на NumberGenerator

```
public class BookService {
  @Inject
  private NumberGenerator numberGenerator;
  public Book createBook(String title, Float price, String description) {
    Book book = new Book(title, price, description);
    book.setIsbn(numberGenerator.generateNumber());
    return book;
  }
}
```

Как видно из листинга 2.4, простая аннотация `@Inject` у атрибута сообщит контейнеру, что ему необходимо внедрить ссылку на реализацию `NumberGenerator`, относящуюся к свойству `NumberGenerator`. Это называется точкой внедрения (место, где находится аннотация `@Inject`). Листинг 2.5 показывает реализацию `IsbnGenerator`. Как видите, здесь нет специальных аннотаций, а класс реализует интерфейс `NumberGenerator`.

Листинг 2.5. Компонент IsbnGenerator

```
public class IsbnGenerator implements NumberGenerator {
  public String generateNumber() {
    return "13-84356-" + Math.abs(new Random().nextInt());
  }
}
```

Точки внедрения

Во время инстанцирования компонента происходит внедрение в той точке, которая указана в аннотации `@Inject`. Внедрение может происходить с помощью трех различных механизмов: свойства, сеттера или конструктора.

Во всех предыдущих примерах кода вы видели аннотацию `@Inject` к атрибутам (свойствам):

```
@Inject
private NumberGenerator numberGenerator;
```

Обратите внимание, что не обязательно создавать метод геттера и сеттера для атрибута, чтобы использовать внедрение. CDI может получить прямой доступ к полю с внедренным кодом (даже если оно приватное), что иногда помогает исключить лишний код. Но вместо аннотирования атрибутов вы можете добавить аннотацию `@Inject` к конструктору, как показано ниже:

```
@Inject
public BookService (NumberGenerator numberGenerator) {
  this.numberGenerator = numberGenerator;
}
```

Однако существует правило, что вы можете иметь только одну точку внедрения конструктора. Именно контейнер (а не вы) выполняет внедрение, так как вы не можете вызвать конструктор в управляемой среде. Поэтому для того, чтобы

контейнер мог выполнить свою работу и внедрить правильные ссылки, разрешается только один конструктор компонентов.

Другой вариант — использовать внедрение сеттера, что выглядит как внедрение конструктора. Вам просто нужно добавить к сеттеру аннотацию `@Inject`:

```
@Inject
public void setNumberGenerator(NumberGenerator numberGenerator) {
  this.numberGenerator = numberGenerator;
}
```

Вы можете спросить, когда нужно использовать поле вместо конструктора или внедрения сеттера. На этот вопрос не существует ответа с технической точки зрения, это скорее дело вашего личного вкуса. В управляемой среде только контейнер выполняет всю работу по внедрению, необходимо лишь задать правильные точки внедрения.

Внедрение по умолчанию

Предположим, `NumberGenerator` имеет только одну реализацию (`IsbnGenearator`). Тогда CDI сможет внедрить его, просто самостоятельно используя аннотацию `@Inject`:

```
@Inject
private NumberGenerator numberGenerator;
```

Это называется *внедрением по умолчанию*. Каждый раз, когда компонент или точка внедрения не объявляет очевидным образом квалификатор, контейнер по умолчанию использует квалификатор `@javax.enterprise.inject.Default`. На самом деле предыдущему отрывку кода идентичен следующий:

```
@Inject @Default
private NumberGenerator numberGenerator;
```

`@Default` — это встроенный квалификатор, сообщающий CDI, когда нужно внедрить реализацию компонента по умолчанию. Если вы определите компонент без квалификатора, ему автоматически присвоится квалификатор `@Default`. Таким образом, код в листинге 2.6 идентичен коду в листинге 2.5.

Листинг 2.6. Компонент IsbnGenerator с квалификатором @Default

```
@Default
public class IsbnGenerator implements NumberGenerator {
  public String generateNumber() {
    return "13-84356-" + Math.abs(new Random().nextInt());
  }
}
```

Если у вас есть только одна реализация компонента для внедрения, применяется поведение по умолчанию, а реализация будет внедрена непосредственно с использованием `@Inject`. Схема класса на рис. 2.4 показывает реализацию `@Default` (`IsbnGenerator`), а также точку внедрения по умолчанию (`@Inject @Default`). Но иногда приходится выбирать между несколькими реализациями. Это как раз тот случай, когда нужно использовать квалификаторы.

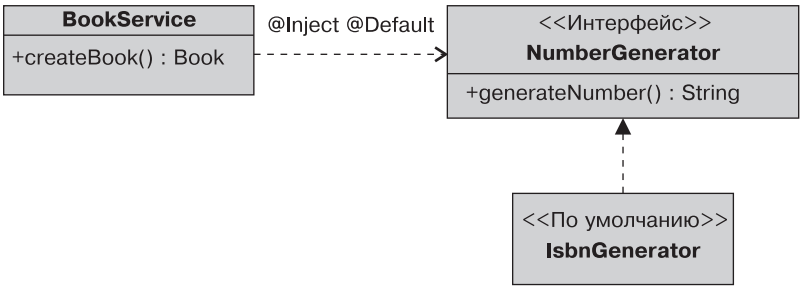


Рис. 2.4. Схема класса с квалификатором @Default

Квалификаторы

Во время инициализации системы контейнер должен подтвердить, что каждой точке внедрения соответствует строго один компонент. Это означает, что при отсутствии доступной реализации `NumberGenerator` контейнер сообщит вам о неудовлетворенной зависимости и не будет развертывать приложение. При наличии только одной реализации внедрение будет работать, используя квалификатор `@Default` (см. рис. 2.4). Если бы в наличии имелось несколько реализаций по умолчанию, то контейнер проинформировал бы вас о неоднозначной зависимости и не стал бы развертывать приложение. Это происходит потому, что в работе типобезопасного алгоритма разрешения происходит сбой, когда контейнер не может определить строго один компонент для внедрения.

Итак, как же компонент выбирает, какая реализация (`IsbnGenrator` или `IssnGenerator`) будет внедряться? При объявлении и внедрении компонентов в большинстве фреймворков активно задействуется XML-код. CDI использует квалификаторы: в основном это аннотации Java, осуществляющие типобезопасное внедрение и однозначно определяющие тип без необходимости отката к строковым именам.

Предположим, у нас есть приложение с сервисом `BookService`, который создает книги с тринадцатизначным номером ISBN, и с `LegacyBookService`, создающим книги с восьмизначным номером ISSN. Как вы можете видеть на рис. 2.5, оба сервиса внедряют ссылку на один интерфейс `NumberGenerator`. Сервисы различаются реализациями благодаря использованию квалификаторов.

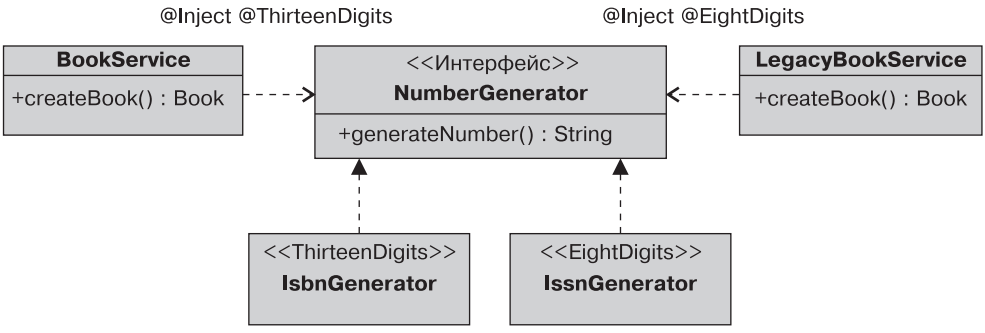


Рис. 2.5. Сервисы, использующие квалификаторы для однозначного внедрения

Квалификатор представляет определенную семантику, ассоциированную с типом и удовлетворяющую отдельной реализации этого типа. Это аннотация, которая определяется пользователем и, в свою очередь, сопровождается аннотацией `@javax.inject.Qualifer`. Например, мы можем использовать квалификаторы для замещения тринадцати- и восьмизначных генераторов чисел. Оба примера показаны в листингах 2.7 и 2.8.

Листинг 2.7. Тринадцатизначный квалификатор

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD})
public @interface ThirteenDigits { }
```

Листинг 2.8. Восьмизначный квалификатор

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD})
public @interface EightDigits { }
```

После того как вы определили требуемые квалификаторы, необходимо применить их к соответствующей реализации. Как показано в листингах 2.9 и 2.10, квалификатор `@ThirteenDigits` применяется к компоненту `IsbnGenerator`, а `@EightDigits` — к `IssnGenerator`.

Листинг 2.9. Компонент IsbnGenerator с квалификатором @ThirteenDigits

```
@ThirteenDigits
public class IsbnGenerator implements NumberGenerator {
  public String generateNumber() {
    return "13-84356-" + Math.abs(new Random().nextInt());
  }
}
```

Листинг 2.10. Компонент IssnGenerator с квалификатором @EightDigits

```
@EightDigits
public class IssnGenerator implements NumberGenerator {
  public String generateNumber() {
    return "8-" + Math.abs(new Random().nextInt());
  }
}
```

Затем эти квалификаторы применяются к точкам внедрения, чтобы отличить, какой реализации требует клиент. В листинге 2.11 `BookService` явным образом определяет тринадцатизначную реализацию посредством внедрения ссылки на генератор чисел `@ThirteenDigits`, а в листинге 2.12 `LegacyBookService` внедряет восьмизначную реализацию.

Листинг 2.11. BookService, использующий реализацию @ThirteenDigits NumberGenerator

```
public class BookService {
  @Inject @ThirteenDigits
  private NumberGenerator numberGenerator;
```

```
    public Book createBook(String title, Float price, String description) {
      Book book = new Book(title, price, description);
      book.setIsbn(numberGenerator.generateNumber());
      return book;
    }
}
```

Листинг 2.12. LegacyBookService, использующий реализацию @EightDigits NumberGenerator

```
public class LegacyBookService {
  @Inject @EightDigits
  private NumberGenerator numberGenerator;
  public Book createBook(String title, Float price, String description) {
    Book book = new Book(title, price, description);
    book.setIsbn(numberGenerator.generateNumber());
    return book;
  }
}
```

Для того чтобы это работало, вам не нужна внешняя конфигурация. Поэтому говорят, что CDI использует строгую типизацию. Вы можете как угодно переименовать ваши реализации или квалификатор — точка внедрения не изменится (так называемая слабая связанность). Как видите, CDI — это аккуратный способ произвести внедрение с сохранением безопасности типов. Но если вы начнете создавать аннотации каждый раз, когда захотите что-либо внедрить, код вашего приложения в итоге чрезмерно разрастется. В этом случае вам помогут квалификаторы, используемые с членами.

Квалификаторы, используемые с членами. Каждый раз, когда вам необходимо выбирать из нескольких реализаций, вы создаете квалификатор (то есть аннотацию). Поэтому, если вам нужны две дополнительные цифры и десятизначный генератор чисел, вы создадите дополнительные аннотации (например, `@TwoDigits`, `@EightDigits`, `@TenDigits`, `@ThirteenDigits`). Представьте, что генерируемые числа могут быть как четными, так и нечетными. В итоге у вас получится множество разных аннотаций: `@TwoOddDigits`, `@TwoEvenDigits`, `@EightOddDigits` и т. д. Один из способов избежать умножения аннотаций — использовать члены.

В нашем примере мы смогли заменить все эти квалификаторы всего одним — `@NumberOfDigits` с перечислением в качестве значения и логическими параметрами для проверки четности (листинг 2.13).

Листинг 2.13. @NumberOfDigits с DigitsEnum и проверкой четности

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD})
public @interface NumberOfDigits {
  Digits value();
  boolean odd();
}
public enum Digits {
  TWO,
```

```
    EIGHT,
    TEN,
    THIRTEEN
}
```

Способ использования этого квалификатора не отличается от тех, которые вы видели раньше. Точка внедрения квалифицирует необходимую реализацию, располагая члены аннотации следующим образом:

```
@Inject @NumberOfDigits(value = Digits.THIRTEEN, odd = false)
private NumberGenerator numberGenerator;
```

Используемая реализация сделает то же самое:

```
@NumberOfDigits(value = Digits.THIRTEEN, odd = false)
public class IsbnEvenGenerator implements NumberGenerator {...}
```

Множественные квалификаторы. Другой способ квалифицировать компонент и точку внедрения — указать множественные квалификаторы. Так, вместо множественных квалификаторов для четности (`@TwoOddDigits`, `@TwoEvenDigits`) либо квалификатора, применяемого с членами (`@NumberOfDigits`), мы могли использовать два разных набора квалификаторов: один набор для четности (`@Odd` и `@Even`), другой — для количества цифр. Ниже приводится способ, которым вы можете квалифицировать генератор 13 четных чисел:

```
@ThirteenDigits @Even
public class IsbnEvenGenerator implements NumberGenerator {...}
```

Точка внедрения использовала бы тот же самый синтаксис:

```
@Inject @ThirteenDigits @Even
private NumberGenerator numberGenerator;
```

Тогда внедрению будет подлежать только компонент, имеющий обе аннотации квалификатора. Названия квалификаторов должны быть понятными. Для приложения важно, чтобы квалификаторы имели правильные имена и степень детализации.

Альтернативы

Квалификаторы позволяют выбирать между множественными реализациями интерфейса во время развертывания. Но иногда бывает целесообразно внедрить реализацию, зависящую от конкретного сценария развертывания. Например, вы решите использовать имитационный генератор чисел в тестовой среде.

Альтернативы — это компоненты, аннотированные специальным квалификатором `javax.enterprise.inject.Alternative`. По умолчанию альтернативы отключены, и чтобы сделать их доступными для инстанцирования и внедрения, необходимо активизировать их в дескрипторе `beans.xml`. В листинге 2.14 показана альтернатива имитационного генератора чисел.

Листинг 2.14. Альтернатива имитационного генератора по умолчанию

```
@Alternative
public class MockGenerator implements NumberGenerator {
```

```
  public String generateNumber() {
    return "MOCK";
  }
}
```

Как видно из листинга 2.14, `MockGenerator`, как обычно, реализует интерфейс `NumberGenerator`. Он сопровождается аннотацией `@Alternative`, которая означает, что CDI обрабатывает его как альтернативу `NumberGenerator` по умолчанию. Как и в листинге 2.6, эта альтернатива по умолчанию могла бы использовать встроенный квалификатор `@Default` следующим образом:

```
@Alternative @Default
public class MockGenerator implements NumberGenerator {...}
```

Вместо альтернативы по умолчанию вы можете указать альтернативу с помощью квалификаторов. Например, следующий код сообщает CDI, что альтернатива тринадцатизначного генератора чисел — это имитация:

```
@Alternative @ThirteenDigits
public class MockGenerator implements NumberGenerator {...}
```

По умолчанию компоненты `@Alternative` отключены, и вам необходимо активизировать их явным образом в дескрипторе `beans.xml`, как показано в листинге 2.15.

Листинг 2.15. Активизация альтернатив в дескрипторе развертывания beans.xml

```
<beans xmlns=" http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi=" http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee ➥
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       version="1.1" bean-discovery-mode="all">
  <alternatives>
    <class>org.agoncal.book.javaee7.chapter02.MockGenerator</class>
  </alternatives>
</beans>
```

Что касается точки внедрения, ничего не меняется. Таким образом, на код вашего приложения ничто не влияет. Следующий отрывок кода содержит внедрение реализации генератора чисел по умолчанию. Если альтернатива активизирована, то `MockGenerator`, определенный в листинге 2.14, будет внедрен.

```
@Inject
private NumberGenerator numberGenerator;
```

У вас может быть несколько файлов `beans.xml`, объявляющих несколько альтернатив, в зависимости от вашей среды (разработка, поддержка готового приложения, тестирование и т. д.).

Производители данных

Я показал вам, как внедрять одни компоненты CDI в другие. Благодаря производителям данных вы также можете внедрять примитивы (такие как `int`, `long`, `float` и т. д.), массивы и любой POJO, не поддерживающий CDI. Под поддержкой CDI я имею в виду любой класс, упакованный в архив, содержащий файл `beans.xml`.

По умолчанию вы не можете внедрять такие классы, как `java.util.Date` или `java.lang.String`. Так происходит потому, что все эти классы упакованы в файл `rt.jar` (классы среды исполнения Java), а этот архив не содержит дескриптор развертывания `beans.xml`. Если в архиве в папке META-INF нет файла `beans.xml`, CDI не инициирует обнаружение компонента и POJO не будут обрабатываться как компоненты, а значит, и внедряться. Единственный способ внедрения POJO состоит в использовании полей и методов производителей данных, как показано в листинге 2.16.

Листинг 2.16. Поля и методы производителей данных

```
public class NumberProducer {
  @Produces @ThirteenDigits
  private String prefix13digits = "13-";
  @Produces @ThirteenDigits
  private int editorNumber = 84356;
  @Produces @Random
  public double random() {
    return Math.abs(new Random().nextInt());
  }
}
```

Класс `NumberProducer` в листинге 2.16 имеет несколько атрибутов и методов (все с аннотацией `javax.enterprise.inject.Produces`). Это означает, что все произведенные типы и классы теперь могут внедряться с помощью `@Inject` с использованием квалификатора (`@ThirteenDigits`, `@EightDigits` или `@Random`).

Метод производителя данных (`random()` в листинге 2.16) — это метод, выступающий в качестве фабрики экземпляров компонентов. Он позволяет внедряться возвращаемому значению. Мы даже можем указать квалификатор (например, `@Random`), область видимости и EL-имя, как вы увидите позднее. Поле производителя данных (`prefix13digits` и `editorNumber`) — более простая альтернатива методу производителя данных, не имеющая бизнес-кода. Это только свойство, которое становится внедряемым.

В листинге 2.9 `IsbnGenerator` генерирует номер ISBN с формулой `"13-84356-" + Math.abs(newRandom().nextInt())`. С помощью `NumberProducer` (см. листинг 2.16) мы можем использовать произведенные типы для изменения этой формулы. В листинге 2.17 `IsbnGenerator` теперь внедряет и строку, и целое число с аннотациями `@Inject @ThirteenDigits`, представляющими префикс (`"13-"`) и идентификатор редактора (`84356`) номера ISBN. Случайный номер внедряется с помощью аннотаций `@Inject @Random` и возвращает число с двойной точностью.

Листинг 2.17. IsbnGenerator, внедряющий произведенные типы

```
@ThirteenDigits
public class IsbnGenerator implements NumberGenerator {
  @Inject @ThirteenDigits
  private String prefix;
  @Inject @ThirteenDigits
  private int editorNumber;
  @Inject @Random
```

```
  private double postfix;
  public String generateNumber() {
    returnprefix + editorNumber + postfix;
  }
}
```

В листинге 2.17 вы можете видеть строгую типизацию в действии. Благодаря тому же синтаксису (`@Inject @ThirteenDigits`) CDI знает, что ему нужно внедрить строку, целое число или реализацию `NumberGenerator`. Преимущество применения внедряемых типов (см. листинг 2.17) вместо фиксированной формулы (см. листинг 2.9) для генерации чисел состоит в том, что вы можете использовать все возможности CDI, такие как альтернативы (и при необходимости иметь альтернативный алгоритм генератора номеров ISBN).

InjectionPoint API. В листинге 2.16 атрибуты и возвращаемое значение, полученное посредством `@Produces`, не требуют никакой информации о том, куда они внедряются. Но в определенных случаях объектам нужна информация о точке их внедрения. Это может быть способ конфигурации или изменения поведения в зависимости от точки внедрения.

Например, рассмотрим создание автоматического журнала. В JDK для создания `java.util.logging.Logger` вам необходимо задать категорию класса, владеющего им. Скажем, если вам нужен автоматический журнал для `BookService`, следует написать:

```
Logger log = Logger.getLogger(BookService.class.getName());
```

Как бы вы получили `Logger`, которому необходимо знать имя класса точки внедрения? В CDI есть `InjectionPoint` API, обеспечивающий доступ к метаданным, которые касаются точки внедрения (табл. 2.2). Таким образом, вам необходимо создать метод производителя данных, который использует `InjectionPoint` API для конфигурации правильного автоматического журнала. В листинге 2.18 показано, как метод `createLogger` получает имя класса точки внедрения.

Таблица 2.2. InjectionPoint API

Метод	Описание
Type getType()	Получает требуемый тип точки внедрения
Set<Annotation> getQualifiers()	Получает требуемые квалификаторы точки внедрения
Bean<?> getBean()	Получает объект Bean, представляющий компонент, который определяет точку внедрения
Member getMember()	Получает объект Field в случае внедрения поля
Annotated getAnnotated()	Возвращает Annotated Field или AnnotatedParameter в зависимости от того, относится точка внедрения к полю или параметру метода/конструктора
boolean isDelegate()	Определяет, происходит ли в данной точке внедрения подключение делегата декоратора
boolean isTransient()	Определяет, является ли точка временным полем

Листинг 2.18. Производитель данных автоматического журнала

```
public class LoggingProducer {
  @Produces
  private Logger createLogger(InjectionPoint injectionPoint) {
    return Logger.getLogger(injectionPoint.getMember().getDeclaringClass().
getName());
  }
}
```

Чтобы использовать произведенный автоматический журнал в любом компоненте, вы просто внедряете его и работаете с ним. Имя класса категории автоматического журнала потом будет задано автоматически:

```
@Inject Logger log;
```

Утилизаторы

В предыдущих примерах (см. листинги 2.17 и 2.18) мы использовали производителей данных для создания типов данных или объектов POJO таким образом, чтобы они могли быть внедрены. Мы создали их, и, пока они использовались, нам не требовалось разрушать или закрывать их. Но некоторые методы производителей данных возвращают объекты, требующие явного разрушения, например интерфейс Java Database Connectivity (JDBC), сеанс JMS или менеджер сущности. Для создания CDI использует производителей данных, а для разрушения — утилизаторы. Метод утилизатора позволяет приложению выполнять настраиваемую очистку объекта, возвращенного методом производителя данных.

Листинг 2.19 показывает утилитный класс, который создает и закрывает интерфейс JDBC. Метод `createConnection` берет драйвер Derby JDBC, создает соединение с определенным URL, обрабатывает исключения и возвращает открытое соединение JDBC. Это сопровождается аннотацией `@Disposes`.

Листинг 2.19. Производитель данных и утилизатор соединения JDBC

```
public class JDBCConnectionProducer {
  @Produces
  private Connection createConnection() {
    Connection conn = null;
    try {
      Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance();
      conn = DriverManager.getConnection("jdbc:derby:memory:chapter02DB", ➥
"APP", "APP");
    } catch (InstantiationException | IllegalAccessException |
ClassNotFoundException) {
      e.printStackTrace();
    }
    return conn;
  }
  private void closeConnection(@Disposes Connection conn) throws SQLException
{
```

```
      conn.close();
    }
  }
```

Уничтожение может осуществляться путем сопоставления метода утилизатора, определенного тем же классом, что и метод производителя данных. Каждый метод утилизатора, сопровождаемый аннотацией `@Disposes`, должен иметь строго один параметр такого же типа (тут `java.sql.Connection`) и квалификаторы (`@Default`), как соответствующий тип возврата метода производителя данных (с аннотацией `@Produces`). Метод утилизатора (`closeConnection()`) вызывается автоматически по окончании контекста клиента (в листинге 2.20 — контекст `@ApplicationScoped`), и параметр получает объект, порожденный методом производителя данных.

Листинг 2.20. Производитель данных и утилизатор соединения JDBS

```
@ApplicationScoped
public class DerbyPingService {
  @Inject
  private Connection conn;
  public void ping() throws SQLException {
    conn.createStatement().executeQuery("SELECT 1 FROM SYSIBM.SYSDUMMY1");
  }
}
```

Листинг 2.20 показывает, как компонент внедряет созданное соединение JDBC с аннотацией `@Inject` и использует его для проверки доступности базы данных Derby. Как видите, этот клиентский код не занимается всеми вспомогательными задачами по созданию и закрытию соединения JDBC либо обработке исключений. Производители данных и утилизаторы — хороший способ создания и закрытия ресурсов.

Области видимости

CDI имеет отношение не только к внедрению зависимостей, но и к контексту (буква C в аббревиатуре CDI означает контекст). Каждый объект, управляемый CDI, имеет строго определенные область видимости и жизненный цикл, которые связаны с конкретным контекстом. В Java область применения POJO достаточно проста: вы создаете экземпляр класса, используя ключевое слово `new`, и позволяете сборщику мусора избавиться от него, чтобы освободить некоторое количество памяти. При использовании CDI компонент связан с контекстом и остается в нем до тех пор, пока не будет разрушен контейнером. Удалить компонент из контекста вручную невозможно.

В то время как веб-уровень имеет четко определенные области видимости (приложение, сеанс, запрос), на уровне сервисов такого не было (см. также главу 7 о компонент-сеансах EJB с сохранением и без сохранения состояния). Ведь, когда компонент-сеансы или POJO используются в веб-приложениях, они не оповещаются о контекстах этих приложений. CDI соединил веб-уровень с уровнем сервисов с помощью содержательных областей видимости. Он определяет следующие встроенные области видимости и даже предлагает точки расширения, в которых вы можете создавать собственные области видимости.

- *Область видимости приложения* (`@ApplicationScoped`) — действует на протяжении всей работы приложения. Компонент создается только один раз на все время работы приложения и сбрасывается, когда оно закрывается. Эта область видимости полезна для утилитных или вспомогательных классов либо объектов, которые хранят данные, используемые совместно целым приложением. Однако необходимо проявить осторожность в вопросах конкурентного доступа, когда доступ к данным должен осуществляться по нескольким потокам.
- *Область видимости сеанса* (`@SessionScoped`) — действует на протяжении нескольких запросов HTTP или нескольких вызовов метода для одного пользовательского сеанса. Компонент создается на все время длительности HTTP-сеанса и сбрасывается, когда сеанс заканчивается. Эта область видимости предназначена для объектов, требуемых на протяжении сеанса, таких как пользовательские настройки или данные для входа в систему.
- *Область видимости запроса* (`@RequestScoped`) — соответствует единственному HTTP-запросу или вызову метода. Компонент создается на все время вызова метода и сбрасывается по его окончании. Он используется для классов обслуживания или связующих компонентов JSF, которые нужны только на протяжении HTTP-запроса.
- *Область видимости диалога* (`@ConverationScoped`) — действительна между множественными вызовами в рамках одной сессии, ее начальная и конечная точка определяются приложением. Диалоги используются среди множественных страниц как часть многоступенчатого рабочего потока.
- *Зависимая псевдообласть видимости* (`@Dependent`) — ее жизненный цикл совпадает с жизненным циклом клиента. Зависимый компонент создается каждый раз при внедрении, а ссылка удаляется одновременно с удалением целевой точки внедрения. Эта область видимости по умолчанию предназначена для CDI.

Как видите, все области видимости имеют аннотацию, которую вы можете использовать с вашими компонентами CDI (все эти аннотации содержатся в пакете `javax.enterprise.context`). Первые три области видимости хорошо известны. Например, если у вас есть компонент «Корзина», чья область видимости ограничена одним сеансом, компонент будет создан автоматически, когда начнется сессия (например, во время первой регистрации пользователя в системе), и автоматически будет разрушен по окончании сессии.

```
@SessionScoped
public class ShoppingCart implements Serializable {...}
```

Экземпляр компонента `ShoppingCart` связан с сеансом пользователя и используется совместно всеми запросами, выполняемыми в контексте этой сессии. Если вы не хотите, чтобы компонент находился в сеансе неопределенно долго, подумайте о том, чтобы задействовать другую область видимости с более коротким временем жизни, например область видимости запроса или диалога. Обратите внимание, что компоненты с областью видимости `@SessionScoped` или `@ConversationScoped` должны быть сериализуемыми, так как контейнер периодически пассивизирует их.

Если область видимости явно не обозначена, то компонент принадлежит зависимой псевдообласти видимости (`@Dependent`). Компоненты с такой областью видимости не могут совместно использоваться несколькими клиентами или через несколько точек внедрения. Их жизненный цикл связан с жизненным циклом компонента, от которого они зависят. Зависимый компонент инстанцируется, когда создается объект, к которому он относится, и разрушается, когда такой объект уничтожается. Следующий отрывок кода показывает зависимый ограниченный ISBN-генератор с квалификатором:

```
@Dependent @ThirteenDigits
public class IsbnGenerator implements NumberGenerator {...}
```

Поскольку это область видимости по умолчанию, вы можете опустить аннотацию `@Dependent` и написать следующее:

```
@ThirteenDigits
public class IsbnGenerator implements NumberGenerator {...}
```

Области видимости могут быть смешанными. Компонент с аннотацией `@SessionScoped` можно внедрить в `@RequestScoped` или `@ApplicationScoped` и наоборот.

Диалог. Область видимости диалога несколько отличается от областей видимости приложения, сеанса или запроса. Она хранит состояние, ассоциированное с пользователем, распространяется сразу на много запросов и программно отграничивается от остального кода на уровне приложения. Компонент с аннотацией `@ConversationScoped` может использоваться для длительных процессов, имеющих начало и конец, таких как навигация по мастеру или покупка товаров и подтверждение и оплата заказа.

Объекты, область видимости которых ограничена одним запросом (HTTP-запросом или вызовом метода), обычно существуют очень недолго, тогда как объекты с областью видимости в пределах сеанса существуют на протяжении всего пользовательского сеанса. Однако есть много случаев, которые не относятся к этим двум крайностям. Отдельные объекты уровня представлений могут использоваться более чем на одной странице, но не на протяжении целой сессии. Для этого в CDI есть специальная область видимости диалога (`@ConversationScoped`). В отличие от объектов в пределах сеанса, которые автоматически отключаются контейнером по истечении заданной задержки, объекты в пределах диалога имеют четко определенный жизненный цикл, который явно начинается и явно заканчивается, причем начало и окончание задаются программно с помощью API `javax.enterprise.context.Conversation`.

В качестве примера рассмотрим следующее веб-приложение: мастер для создания новой учетной записи клиента. Мастер состоит из трех шагов. В первом шаге клиент вводит данные для входа в систему, например имя пользователя и пароль. Во втором шаге пользователь вводит данные учетной записи, например имя, фамилию, почтовый адрес и адрес электронной почты. Во время последнего шага мастер подтверждает всю собранную информацию и создает учетную запись. Листинг 2.21 показывает компонент в пределах диалога, реализующий мастер для создания нового пользователя.

Листинг 2.21. Мастер для создания учетной записи пользователя с применением диалога

```
@ConversationScoped
public class CustomerCreatorWizard implements Serializable {
  private Login login;
  private Account account;
  @Inject
  private CustomerService customerService;
  @Inject
  private Conversation conversation;
  public void saveLogin() {
    conversation.begin();
    login = newLogin();
    // Задает свойства учетных данных
  }
  public void saveAccount() {
    account = new Account();
    // Задает свойства учетной записи
  }
  public void createCustomer() {
    Customer customer = new Customer();
    customer.setLogin(login);
    customer.setAccount(account);
    customerService.createCustomer(customer);
    conversation.end();
  }
}
```

В листинге 2.21 компонент `CustomerCreationWizard` сопровождается аннотацией `@ConversationScoped`. Затем он внедряет `CustomerService` для создания `Customer`, но что более важно, он внедряет `Conversation`. Этот интерфейс позволяет программно управлять жизненным циклом области видимости диалога. Обратите внимание, что при вызове метода `saveLogin` начинается диалог (`conversation.begin()`). Теперь он начинается и используется в течение всего времени работы мастера. Как только вызывается последний шаг мастера, вызывается метод `createCustomer` и диалог заканчивается (`conversation.end()`). Таблица 2.3 предлагает вам краткое обозрение API `Conversation`.

Таблица 2.3. API Conversation

Метод	Описание
void begin()	Помечает текущий кратковременный диалог как длительный
void begin(String id)	Помечает текущий кратковременный диалог как длительный, со специальным идентификатором
void end()	Помечает текущий длительный диалог как кратковременный
String getId()	Получает идентификатор текущего длительного диалога
long getTimeout()	Получает время задержки текущего диалога
void setTimeout(long millis)	Задает время задержки текущего диалога
boolean isTransient()	Определяет, помечен ли диалог как кратковременный или длительный

Компоненты в языке выражений

Одна из ключевых возможностей CDI состоит в том, что он связывает уровень транзакций и веб-уровень. Но, как вы уже могли видеть, одна из первоочередных характеристик CDI заключается в том, что внедрение зависимостей (DI) полностью типобезопасно и не зависит от символьных имен. В Java-коде это не вызывает проблем, поскольку компоненты не будут разрешимы без символьных имен за пределами Java. В частности, это касается EL-выражений на страницах JSF.

По умолчанию компонентам CDI не присваивается имя и они неразрешимы с помощью EL-связывания. Чтобы можно было присвоить компоненту имя, он должен быть аннотирован встроенным квалификатором `@javax.inject.Named`, как показано в листинге 2.22.

Листинг 2.22. Компонент BookService с символьным именем

```
@Named
public class BookService {
  private String title, description;
  private Float price;
  private Book book;
  @Inject @ThirteenDigits
  private NumberGenerator numberGenerator;
  public String createBook() {
    book = new Book(title, price, description);
    book.setIsbn(numberGenerator.generateNumber());
    return "customer.xhtml";
  }
}
```

Квалификатор `@Named` позволяет получить доступ к компоненту `BookService` через его имя (им по умолчанию является имя класса в «верблюжьем регистре» (CamelCase) с первой строчной буквой). Следующий отрывок кода показывает кнопку JSF, вызывающую метод `createBook`:

```
<h:commandButton value="Sendemail" action="#{bookService.createBook}"/>
```

Вы также можете переопределить имя компонента, добавив другое имя к квалификатору.

```
@Named("myService")
public class BookService {...}
```

Затем вы можете использовать это новое имя на странице JSF.

```
<h:commandButton value="Send email" action="#{myService.createBook}"/>
```

Перехватчики

Перехватчики позволяют добавлять к вашим компонентам сквозную функциональность. Как показано на рис. 2.2, когда клиент вызывает метод на управляемом компоненте (а значит, и на компоненте CDI, EJB либо веб-службе RESTful и т. д.),

контейнер может перехватить вызов и обработать бизнес-логику перед тем, как будет вызван метод компонента. Перехватчики делятся на четыре типа:

- *перехватчики, действующие на уровне конструктора,* — перехватчик, ассоциированный с конструктором целевого класса (`@AroundConstruct`);
- *перехватчики, действующие на уровне метода,* — перехватчик, ассоциированный со специальным бизнес-методом (`@AroundInvoke`);
- *перехватчики методов задержки* — перехватчик, помеченный аннотацией `@AroundTimeout`, вмешивается в работу методов задержки (применяется только со службой времени EJB, см. главу 8);
- *перехватчики обратного вызова жизненного цикла* — перехватчик, который вмешивается в работу обратных вызовов событий жизненного цикла целевого экземпляра (`@PostConstruct` и `@PreDestroy`).

ПРИМЕЧАНИЕ

Начиная с Java EE 6, перехватчики оформились в отдельную спецификацию (до этого они входили в состав спецификации EJB). Они могут применяться к управляемым компонентам, как вы увидите далее в этом разделе, а также к компонентам EJB и веб-службам SOAP и RESTful.

Перехватчики целевого класса

Существует несколько способов определения перехвата. Самый простой — добавить перехватчики (уровня метода, тайм-аута или жизненного цикла) к самому компоненту, как показано в листинге 2.23. Класс `CustomerService` сопровождает `logMethod()` аннотацией `@AroundInvoke`. Этот метод используется для регистрации сообщения во время входа в метод и выхода из него. Как только этот управляемый компонент развертывается, любой клиентский вызов `createCustomer()` или `findCustomerById()` будет перехватываться и начнет применяться `logMethod()`. Обратите внимание, что область видимости этого перехватчика ограничена самим компонентом (целевым классом).

Листинг 2.23. Класс CustomerService, использующий перехватчик, предшествующий вызову

```
@Transactional
public class CustomerService {
  @Inject
  private EntityManager em;
  @Inject
  private Logger logger;
  public void createCustomer(Customer customer) {
    em.persist(customer);
  }
  public Customer findCustomerById(Long id) {
    return em.find(Customer.class, id);
  }
  @AroundInvoke
  private Object logMethod(InvocationContext ic) throws Exception {
    logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
    try {
```

```
        return ic.proceed();
      } finally {
        logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
      }
    }
  }
```

Несмотря на аннотацию `@AroundInvoke`, `logMethod()` должен иметь следующий образец подписи:

```
@AroundInvoke
Object <METHOD>(InvocationContext ic) throws Exception;
```

Следующие правила относятся к методу, предшествующему вызову (а также конструктору, времени задержки или перехватчикам жизненного цикла):

- метод может иметь доступ `public`, `private`, `protected` либо доступ на уровне пакета, но не должно быть доступа `static` или `final`;
- метод должен иметь параметр `javax.interceptor.InvocationContext` и возвращать объект, который является результатом вызванного метода проб;
- метод может генерировать проверяемое исключение.

Объект `InvocationContext` позволяет перехватчикам контролировать поведение цепочки вызовов. Если несколько перехватчиков соединены в цепочку, то один и тот же экземпляр `InvocationContext` передается каждому перехватчику, который может добавить контекстуальные данные для обработки другими перехватчиками. Таблица 2.4 описывает API `InvocationContext`.

Таблица 2.4. Определение интерфейса InvocationContext

Метод	Описание
getContextData	Позволяет значениям передаваться между методами перехвата в том же экземпляре InvocationContext с использованием Map
getConstructor	Возвращает конструктор целевого класса, для которого был вызван перехватчик
getMethod	Возвращает метод класса компонентов, для которого был вызван перехватчик
getParameters	Возвращает параметры, которые будут использоваться для вызова бизнес-метода
getTarget	Возвращает экземпляр компонента, к которому относится перехватываемый метод
getTimer	Возвращает таймер, ассоциированный с методом @Timeout
proceed	Обеспечивает вызов следующего метода перехватчика по цепочке. Он возвращает результат следующего вызываемого метода. Если метод относится к типу void, то proceed возвращает null
setParameters	Модифицирует значение параметров, используемых для вызова методов целевого класса. Типы и количество параметров должны совпадать с подписью метода компонента, иначе будет сгенерировано исключение IllegalArgumentException

Чтобы понять, как работает код в листинге 2.23, взгляните на схему последовательности на рис. 2.6. Вы увидите, что происходит, когда клиент вызывает метод `createCustomer()`. Прежде всего контейнер перехватывает вызов и вместо прямой обработки `createCustomer()` сначала вызывает метод `logMethod()`. Данный метод использует интерфейс `InvocationContext` для получения имени вызываемого компонента (`ic.getTarget()`), а вызываемый метод (`ic.getMethod()`) применяет для регистрации сообщения о входе (`logger.entering()`). Затем вызывается метод `proceed()`. Вызов `InvocationContext.proceed()` очень важен, поскольку сообщает контейнеру, что тот должен обрабатывать следующий перехватчик или вызывать бизнес-метод компонента. При отсутствии вызова `proceed()` цепочка перехватчиков будет остановлена, а бизнес-метод не будет вызван. В конце вызывается метод `createCustomer()`, и как только он возвращается, перехватчик прекращает выполнение, регистрируя сообщение о выходе (`logger.exiting()`). Вызов клиентом метода `findCustomerById()` происходил бы в той же последовательности.

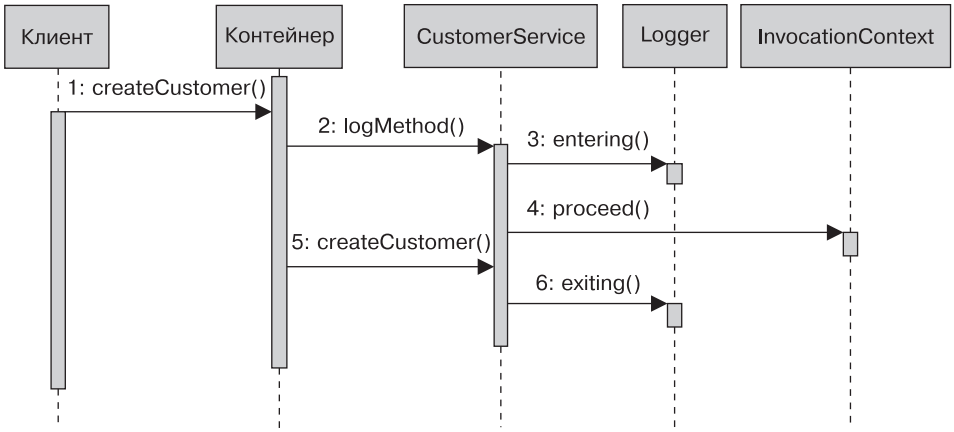


Рис. 2.6. Вызов перехватываемого бизнес-метода

ПРИМЕЧАНИЕ

Листинг 2.23 использует новую аннотацию @javax.transaction.Transactional. Она применяется для управления разграничением операций на компонентах CDI, а также сервлетах и оконечных точках сервисов JAX-RS и JAX-WS. Она обеспечивает семантику атрибутов транзакции EJB в CDI. Аннотация @Transactional реализуется с помощью перехватчика. Подробнее о транзакциях рассказывается в главе 9.

Перехватчики классов

Листинг 2.23 определяет перехватчик, доступный только для `CustomerService`. Но чаще всего вам требуется изолировать сквозную функциональность в отдельный класс и сообщить контейнеру, чтобы он перехватил вызовы нескольких компонентов. Запись информации в журнал (логирование) — типичный пример ситуации, когда вам требуется, чтобы все методы всех ваших компонентов регистрировали сообщения о входе и выходе. Для указания перехватчика класса вам необходимо

разработать отдельный класс и дать контейнеру команду применить его на определенном компоненте или методе компонента.

Чтобы обеспечить совместный доступ к коду множественным компонентам, возьмем методы `logMethod()` из листинга 2.23 и изолируем их в отдельный класс, как показано в листинге 2.24. Обратите внимание на метод `init()`, который сопровождается аннотацией `@AroundConstruct` и будет вызван только вместе с конструктором компонента.

Листинг 2.24. Класс перехватчика с Around-Invoke и Around-Construct

```
public class LoggingInterceptor {
  @Inject
  private Logger logger;
  @AroundConstruct
  private void init(InvocationContext ic) throws Exception {
    logger.fine("Entering constructor");
    try {
      ic.proceed();
    } finally {
      logger.fine("Exiting constructor");
    }
  }
  @AroundInvoke
  public Object logMethod(InvocationContext ic) throws Exception {
    logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
    try {
      return ic.proceed();
    } finally {
      logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
    }
  }
}
```

Теперь `LoggingInterceptor` может быть прозрачно обернут любым компонентом, заинтересованным в этом перехватчике. Для этого компоненту необходимо сообщить контейнеру аннотацию `@javax.interceptor.Interceptors`. В листинге 2.25 аннотация задается методом `createCustomer()`. Это означает, что любой вызов этого метода будет перехвачен контейнером, и будет вызван класс `LoggingInterceptor` (регистрация сообщения на входе в метод и выходе из него).

Листинг 2.25. CustomerService использует перехватчик в одном методе

```
@Transactional
public class CustomerService {
  @Inject
  private EntityManager em;
  @Interceptors(LoggingInterceptor.class)
  public void createCustomer(Customer customer) {
    em.persist(customer);
  }
  public Customer findCustomerById(Long id) {
```

```
    return em.find(Customer.class, id);
  }
}
```

В листинге 2.25 аннотация `@Interceptors` прикрепляется только к методу `createCustomer()`. Это означает, что, если клиент вызывает `findCustomerById()`, контейнер не будет перехватывать вызов. Если вы хотите, чтобы перехватывались вызовы обоих методов, можете добавить аннотацию `@Interceptors` либо сразу к обоим методам, либо к самому компоненту. Когда вы это делаете, перехватчик приводится в действие при вызове любого из методов. А поскольку перехватчик имеет аннотацию `@AroundConstruct`, вызов конструктора тоже будет перехвачен.

```
@Transactional
@Interceptors(LoggingInterceptor.class)
public class CustomerService {
  public void createCustomer(Customer customer) {...}
  public Customer findCustomerById(Long id) {...}
}
```

Если ваш компонент имеет несколько методов и вы хотите применить перехватчик ко всему компоненту за исключением определенного метода, можете использовать аннотацию `javax.interceptor.ExcludeClassInterceptors` для исключения перехвата вызова. В следующем отрывке кода вызов к `updateCustomer()` не будет перехвачен, а остальные будут:

```
@Transactional
@Interceptors(LoggingInterceptor.class)
public class CustomerService {
  public void createCustomer(Customer customer) {...}
  public Customer findCustomerById(Long id) {...}
  @ExcludeClassInterceptors
  public Customer updateCustomer(Customer customer) { ... }
}
```

Перехватчик жизненного цикла

В начале этой главы я рассказывал о жизненном цикле управляемого компонента (см. рис. 2.2) и событиях обратного вызова. С помощью аннотации обратного вызова вы можете дать контейнеру команду вызвать метод в определенной фазе жизненного цикла (`@PostConstruct` и `@PreDestroy`). Например, если вы хотите вносить в журнал запись каждый раз, когда создается экземпляр компонента, вам просто нужно присоединить аннотацию `@PostConstruct` к методу вашего компонента и добавить к ней некоторые механизмы записи в журнал. Но что, если вам нужно перехватывать события жизненного цикла многих типов компонентов? Перехватчики жизненного цикла позволяют изолировать определенный код в отдельный класс и вызывать его, когда приводится в действие событие жизненного цикла.

Листинг 2.26 демонстрирует класс `ProfileInterceptor` с двумя методами: `logMethod()`, который используется для постконструкции, и `profile()`, применяемый для перехвата методов (`@AroundInvoke`).

Листинг 2.26. Перехватчик с жизненным циклом и Around-Invoke

```
public class ProfileInterceptor {
  @Inject
  private Logger logger;
  @PostConstruct
  public void logMethod(InvocationContext ic) throws Exception {
    logger.fine(ic.getTarget().toString());
    try {
      ic.proceed();
    } finally {
      logger.fine(ic.getTarget().toString());
    }
  }
  @AroundInvoke
  public Object profile(InvocationContext ic) throws Exception {
    long initTime = System.currentTimeMillis();
    try {
      return ic.proceed();
    } finally {
      long diffTime = System.currentTimeMillis() - initTime;
      logger.fine(ic.getMethod() + " took " + diffTime + " millis");
    }
  }
}
```

Как видно из листинга 2.26, перехватчики жизненного цикла берут параметр `InvocationContext` и вместо `Object` возвращают `void`. Чтобы применить перехватчик, определенный в листинге 2.26, компонент `CustomerService` (листинг 2.27) должен использовать аннотацию `@Interceptors` и определять `ProfileInterceptor`. Если компонент инстанцируется контейнером, метод `logMethod()` будет вызван раньше метода `init()`. Затем, если клиент вызывает `createCustomer()` или `findCustomerById()`, будет вызван метод `profile()`.

Листинг 2.27. CustomerService, использующий перехватчик и аннотацию обратного вызова

```
@Transactional
@Interceptors(ProfileInterceptor.class)
public class CustomerService {
  @Inject
  private EntityManager em;
  @PostConstruct
  public void init() {
    // ...
  }
  public void createCustomer(Customer customer) {
    em.persist(customer);
  }
  public Customer findCustomerById(Long id) {
    return em.find(Customer.class, id);
  }
}
```

Связывание и исключение перехватчиков

Вы уже видели, как перехватываются вызовы в пределах одного компонента (с аннотацией `@Around Invoke`), а также среди множественных компонентов (с использованием аннотации `@Interceptors`). Спецификация Interceptors 1.2 также позволяет связать в цепочку несколько перехватчиков.

В действительности аннотация `@Interceptors` способна прикреплять более одного перехватчика, так как в качестве параметра она берет список перехватчиков, разделенных запятой. Когда определяются множественные перехватчики, порядок их вызова задается тем порядком, в котором они указаны в аннотации `@Interceptors`. Например, код в листинге 2.28 использует аннотацию `@Interceptors` у компонента и на уровне методов.

Листинг 2.28. CustomerService, соединяющий несколько перехватчиков

```
@Stateless
@Interceptors({I1.class, I2.class})
public class CustomerService {
  public void createCustomer(Customer customer) {...}
  @Interceptors({I3.class, I4.class})
  public Customer findCustomerById(Long id) {...}
  public void removeCustomer(Customer customer) {...}
  @ExcludeClassInterceptors
  public Customer updateCustomer(Customer customer) {...}
}
```

Когда клиент вызывает метод `updateCustomer()`, перехватчик не вызывается, так как метод аннотирован `@ExcludeClassInterceptors`. При вызове метода `createCustomer()` выполняется перехватчик `I1`, за которым следует перехватчик `I2`. При вызове метода `findCustomerById()` перехватчики `I1`, `I2`, `I3` и `I4` выполняются в соответствующем порядке.

Связывание с перехватчиком

Перехватчики определяются в своей собственной спецификации (запрос JSR 318) и могут использоваться в любых управляемых компонентах (EJB, сервлетах, вебслужбах RESTful и т. д.). Но CDI расширил исходную спецификацию, добавив к ней связывание с перехватчиком. Это означает, что связывание с перехватчиком может применяться только тогда, когда активизирован CDI.

Если вы посмотрите на листинг 2.25, то увидите, как работают перехватчики. Реализацию перехватчика необходимо указывать непосредственно на реализации компонента (например, `@Interceptors(LoggingInterceptror.class)`). Это типобезопасно, но нет слабой связи. CDI обеспечивает связывание с перехватчиком, которое представляет определенный уровень косвенности и слабой связанности. Тип связывания с перехватчиком — это определенная пользователем аннотация, также сопровождаемая аннотацией `@InterceptorBinding`, которая связывает класс перехватчика с компонентом без прямой зависимости между этими двумя классами.

Листинг 2.29 показывает связывание с перехватчиком под названием `Loggable`. Как видите, данный код очень похож на квалификатор. Связывание с перехватчи-

ком — это аннотация, также аннотированная `@InterceptorBinding`, которая может быть пустой или иметь члены (например, как в листинге 2.13).

Листинг 2.29. Связывание с перехватчиком Loggable

```
@InterceptorBinding
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Loggable { }
```

При наличии связывания с перехватчиком необходимо прикрепить его к самому перехватчику. Для этого к перехватчику добавляется аннотация `@Interceptor` и связывание с перехватчиком (`Loggable` в листинге 2.30).

Листинг 2.30. Перехватчик Loggable

```
@Interceptor
@Loggable
public class LoggingInterceptor {
  @Inject
  private Logger logger;
  @AroundInvoke
  public Object logMethod(InvocationContext ic) throws Exception {
    logger.entering(ic .getTarget().toString(), ic .getMethod().getName());
    try {
      return ic.proceed() ;
    } finally {
      logger.exiting(ic .getTarget().toString(), ic .getMethod().getName());
    }
  }
}
```

Теперь вы можете применить перехватчик к компоненту, проаннотировав класс компонента той же связкой перехватчиков, которая показана в листинге 2.31. Это дает вам слабую связанность (так как класс реализации явно не указывается) и неплохой уровень косвенности.

Листинг 2.31. CustomerService, использующий связку перехватчиков

```
@Transactional
@Loggable
public class CustomerService {
  @Inject
  private EntityManager em;
  public void createCustomer(Customer customer) {
    em.persist(customer);
  }
  public Customer findCustomerById(Long id) {
    return em.find(Customer.class, id);
  }
}
```

В листинге 2.31 связывание перехватчиков находится на компоненте. Это означает, что каждый метод будет перехватываться и записываться в журнал. Но, как

и всегда в таких случаях, можно связывать перехватчик с отдельным методом, а не с целым компонентом.

```
@Transactional
public class CustomerService {
  @Loggable
  public void createCustomer(Customer customer) {...}
  public Customer findCustomerById(Long id) {...}
}
```

Перехватчики специфичны для развертывания и отключены по умолчанию. Как и альтернативы, перехватчики необходимо активизировать, используя дескриптор развертывания `beans.xml` JAR-файла или модуля Java EE, как показано в листинге 2.32.

Листинг 2.32. Активизация перехватчика в дескрипторе развертывания beans.xml

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee ➥
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       version="1.1" bean-discovery-mode="all">
  <interceptors>
    <class>org.agoncal.book.javaee7.chapter02.LoggingInterceptor</class>
  </interceptors>
</beans>
```

Приоритизация связывания перехватчиков

Связывание перехватчиков обеспечивает определенный уровень косвенности, однако лишает возможности упорядочивать перехватчики, как показано в листинге 2.28 (`@Interceptors({I1.class, I2.class})`). Согласно CDI 1.1 вы можете приоритизировать их, используя аннотацию `@javax.annotation.Priority` (либо ее XML-эквивалент в файле `beans.xml`) вместе со значением приоритета, как показано в листинге 2.33.

Листинг 2.33. Связка перехватчиков Loggable

```
@Interceptor
@Loggable
@Priority(200)
public class LoggingInterceptor {
  @Inject
  private Logger logger;
  @AroundInvoke
  public Object logMethod(InvocationContext ic) throws Exception {
    logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
    try {
      return ic.proceed() ;
    } finally {
      logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
    }
  }
}
```

Аннотация `@Priority` берет целое число, которое может принимать любое значение. Правило состоит в том, что перехватчики с меньшим приоритетом называются первыми. Java EE 7 определяет приоритеты уровня платформы, после чего ваши перехватчики могут вызываться до или после определенных событий. Аннотация `javax.interceptor.Interceptor` определяет следующий набор констант:

- `PLATFORM_BEFORE = 0` — начинает диапазон для ранних перехватчиков, определяемых платформой Java EE;
- `LIBRARY_BEFORE = 1000` — открывает диапазон для ранних перехватчиков, задаваемых библиотеками расширения;
- `APPLICATION = 2000` — начинает диапазон для ранних перехватчиков, определяемых приложениями;
- `LIBRARY_AFTER = 3000` — открывает диапазон для поздних перехватчиков, задаваемых библиотеками расширения;
- `PLATFORM_AFTER = 4000` — начинает диапазон для поздних перехватчиков, определяемых платформой Java EE.

Поэтому, если вы хотите, чтобы ваш перехватчик выполнялся до любого перехватчика приложения, но после любого раннего перехватчика платформы, можете написать следующее:

```
@Interceptor
@Loggable
@Priority(Interceptor.Priority.LIBRARY_BEFORE + 10)
public class LoggingInterceptor {...}
```

Декораторы

Перехватчики выполняют задачи сквозной функциональности и идеальны для решения таких технических проблем, как управление транзакциями, безопасность или запись в журнал. По своей природе перехватчики не осведомлены о настоящей семантике перехватываемых действий и поэтому не подходят для отделения задач, связанных с бизнесом. Для декораторов характерно обратное.

Декораторы — общий шаблон проектирования, разработанный группой Gang of Four. Идея состоит в том, чтобы взять класс и обернуть вокруг него другой класс (то есть декорировать его). Таким образом, при вызове декорированного класса вы всегда проходите через окружающий его декоратор, прежде чем достигнете целевого класса. Декораторы позволяют добавлять бизнес-методу дополнительную логику. Они не способны решать технические задачи, которые являются сквозными и касаются многих несхожих типов. Хотя перехватчики и декораторы во многом сходны, они дополняют друг друга.

Для примера возьмем генератор чисел ISSN. ISSN — это восьмизначный номер, замещенный номером ISBN (тринадцатизначным номером). Вместо того чтобы иметь два отдельных генератора чисел (например, как в листинге 2.9 или 2.10), вы можете декорировать генератор ISSN, добавив к нему дополнительный алгоритм, превращающий восьмизначный номер в тринадцатизначный. Листинг 2.34 реализует такой алгоритм как декоратор. Класс `FromEightToThirteenDigitsDecorator`

аннотируется javax.decorator.Decorator, реализует бизнес-интерфейсы (NumberGenerator на рис. 2.3) и перезаписывает метод generateNumber. При этом декоратор может быть объявлен абстрактным классом, чтобы ему не приходилось реализовывать все бизнес-методы интерфейсов, если их несколько. Метод generateNumber() вызывает целевой компонент для генерации ISSN, добавляет бизнес-логику для трансформации такого номера и возвращает номер ISBN.

Листинг 2.34. Декоратор, преобразующий восьмизначный номер в тринадцатизначный

```
@Decorator
public class FromEightToThirteenDigitsDecorator implements NumberGenerator {
  @Inject @Delegate
  private NumberGenerator numberGenerator;
  public String generateNumber() {
    String issn = numberGenerator.generateNumber();
    String isbn = "13-84356" + issn.substring(1);
    returnisbn;
  }
}
```

Декораторы должны иметь точку внедрения делегата (аннотированную @Delegate) такого же типа, как и компоненты, которые они декорируют (здесь интерфейс NumberGenerator). Это позволяет объекту вызывать объект-делегат (например, целевой компонент IssnNumberGenerator), а затем, в свою очередь, вызывать на него любой бизнес-метод (например, numberGenerator.generateNumber() в листинге 2.34).

По умолчанию все декораторы отключены, как и альтернативы с перехватчиками. Декораторы необходимо активизировать в файле beans.xml, как показано в листинге 2.35.

Листинг 2.35. Активизация декоратора в дескрипторе развертывания beans.xml

```
<beansxmlns="http://xmlns.jcp.org/xml/ns/javaee"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee ➥
                         http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
             version="1.1" bean-discovery-mode="all">
  <decorators>
    <class>org.agoncal.book.javaee7.chapter02.FromEightToThirteenDigitsDeco-
rator</class>
  </decorators>
</beans>
```

Если в приложении присутствуют и перехватчики, и декораторы, то перехватчики вызываются в первую очередь.

События

Внедрение зависимостей, перехватчики и декораторы гарантируют слабую связанность, обеспечивая разнообразные варианты дополнительного поведения как во время развертывания, так и во время выполнения. События, кроме того, позволяют

компонентам взаимодействовать вне зависимости от времени компиляции. Один компонент может определить событие, другой — инициировать событие, а третий — обработать его. Эта базовая схема следует шаблону проектирования «Наблюдатель» (Observer), разработанному группой Gang of Four.

Производители событий запускают события, используя интерфейс `javax.enterprise.event`. Производитель инициирует события вызовом метода `fire()`, передает объект события и не зависит от наблюдателя. В листинге 2.36 `BookService` запускает событие (`bookAddedEvent`) каждый раз при создании книги. Код `bookAddedEvent.fire(book)` инициирует событие и оповещает любые методы наблюдателя, следящие за этим конкретным событием. Содержание этого события — сам объект `Book`, который будет передан от производителя потребителю.

Листинг 2.36. BookService запускает событие каждый раз, когда создается книга

```
public class BookService {
  @Inject
  private NumberGenerator numberGenerator;
  @Inject
  private Event<Book> bookAddedEvent;
  public Book createBook(String title, Float price, String description) {
    Book book = new Book(title, price, description);
    book.setIsbn(numberGenerator.generateNumber());
    bookAddedEvent.fire(book);
    return book;
  }
}
```

События инициируются производителем событий и на них подписываются наблюдатели. Наблюдатель — это компонент с одним или несколькими «отслеживающими» методами. Каждый из этих методов наблюдателя берет в качестве параметра событие определенного типа, сопровождаемое аннотацией `@Observers` и опциональными квалификаторами. Метод наблюдателя оповещается о событии, если объект события соответствует типу и всем квалификаторам. В листинге 2.37 показана служба инвентаризации, задача которой — отслеживать новые книжные поступления, дополняя информацию о книжном фонде. Там используется метод `addBook`, который наблюдает за каждым событием с типом `Book`. Аннотированный параметр называется параметром события. Поэтому, как только событие инициируется компонентом `BookService`, контейнер CDI приостанавливает выполнение и передает событие любому зарегистрированному наблюдателю. В нашем случае в листинге 2.37 будет вызван метод `addBook`, который обновит список книг, а затем контейнер продолжит выполнение кода с того места, где он остановился в компоненте `BookService`. Это означает, что события в CDI не рассматриваются асинхронно.

Листинг 2.37. InventoryService наблюдает за событием Book

```
public class InventoryService {
  @Inject
  private Logger logger;
  List<Book> inventory = new ArrayList<>();
  public void addBook(@Observes Book book) {
```

```
    logger.info("Adding book " + book.getTitle() + "to inventory");
    inventory.add(book);
  }
}
```

Как и большинство CDI, производство события и подписка являются типобезопасными и позволяют квалификаторам определять, какие наблюдатели событий будут использоваться. Событию может быть назначен один или несколько квалификаторов (с членами либо без таковых), которые позволяют наблюдателям отличить его от остальных событий такого же типа. Листинг 2.38 возвращается к компоненту `BookService`, добавив туда дополнительное событие. При создании книги инициируется событие `bookAddedEvent`, а при удалении — событие `bookRemovedEvent`, оба с типом `Book`. Чтобы можно было отличать эти события, каждое из них сопровождается аннотацией `@Added` или `@Removed`. Код этих квалификаторов идентичен коду в листинге 2.7: аннотация без членов и аннотированная `@Qualifier`.

Листинг 2.38. BookService инициирует несколько событий

```
public class BookService {
  @Inject
  private NumberGenerator numberGenerator;
  @Inject @Added
  private Event<Book> bookAddedEvent;
  @Inject @Removed
  private Event<Book> bookRemovedEvent;
  public Book createBook(String title, Float price, String description) {
    Book book = new Book(title, price, description);
    book.setIsbn(numberGenerator.generateNumber());
    bookAddedEvent.fire(book);
    return book;
  }
  public void deleteBook(Book book) {
    bookRemovedEvent.fire(book);
  }
}
```

`InventoryService` в листинге 2.39 наблюдает за обоими событиями, объявив два отдельных метода, один из которых наблюдает за событием о добавлении книги (`@Observes @Added Book`), а другой — за событием о ее удалении (`@Observes @Removed Book`).

Листинг 2.39. InventoryService наблюдает за несколькими событиями

```
public class InventoryService {
  @Inject
  private Logger logger;
  List<Book> inventory = new ArrayList<>();
  public void addBook(@Observes @Added Book book) {
    logger.warning("Книга " + book.getTitle() + " добавлена в список");
    inventory.add(book);
  }
  public void removeBook(@Observes @Removed Book book) {
```

```
        logger.warning("Книга " + book.getTitle() + " удалена из списка");
        inventory.remove(book);
    }
}
```

Поскольку модель события использует квалификаторы, вам было бы целесообразно задавать поля квалификаторов или агрегировать их. Следующий код наблюдает за всеми добавленными книгами, цена которых превышает 100:

```
void addBook(@Observes @Added @Price(greaterThan=100) Book book)
```

Все вместе

А теперь совместим некоторые из этих понятий, напишем несколько компонентов, производителей, используем внедрение, квалификаторы, альтернативы и связывание с перехватчиком. В этом примере применяется контейнер Weld для запуска класса `Main` в Java SE, а также интеграционный тест для проверки правильности внедрения.

Рисунок 2.7 показывает схему со всеми классами, необходимыми для запуска этого образца кода, и описывает все точки внедрения.

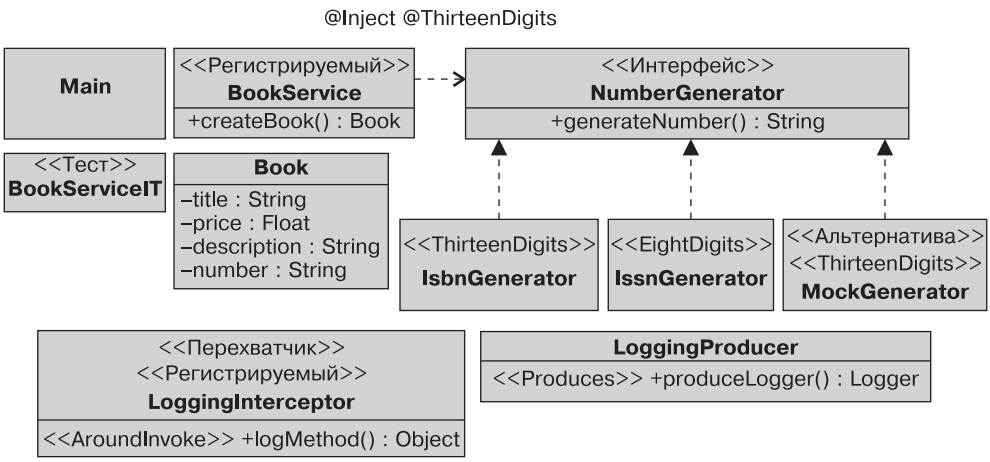


Рис. 2.7. Все вместе

- Компонент `BookService` имеет метод для создания Java-объектов `Book`.
- Интерфейс `NumberGenerator` имеет две реализации для генерации номеров ISBN и ISSN (`IsbnGenerator` и `IssnGenerator`) и одну альтернативную реализацию, чтобы генерировать имитационные номера для интеграционных тестов (`MockGenerator`).
- Реализации `NumberGenerator` используют два квалификатора, чтобы избежать неоднозначного внедрения зависимости: `@ThirteenDigits` и `@EightDigits`.
- `LoggingProducer` делает возможным внедрение `Logger` благодаря методу-производителю. `LoggingInterceptor` в паре с перехватчиком `Loggable` позволяет компонентам CDI сохранять в журнал записи о методах.

- Класс `Main` использует `BookService`, чтобы создать `Book` и сгенерировать номер с помощью `IsbnGenerator`. Интеграционный тест `BookServiceIT` использует альтернативу `MockGenerator` для генерации имитационного номера книги.

Классы, описанные на рис. 2.7, следуют стандартной структуре каталога Maven:

- src/main/java — каталог для всех компонентов, квалификаторов, перехватчиков и класса `Main`;
- src/main/resources — пустой файл `beans.xml`, поэтому мы можем инициировать CDI без альтернатив и перехватчиков;
- src/test/java — каталог для интеграционных тестов `BookServiceIT` и альтернативы `MockGenerator`;
- src/test/resources — файл `beans.xml`, обеспечивающий работу альтернативы `MockGenerator` и перехватчика `LoggingInterceptor`;
- `pom.xml` — модель объекта проекта Maven (POM), описывающая проект и его зависимости.

Написание классов Book и BookService

Приложение CD-Bookstore использует класс `BookService` для создания книг (листинг 2.40). Java-объект `Book` (листинг 2.41) имеет название, описание и цену. Номер книги (ISBN или ISSN) генерируется внешним сервисом.

Листинг 2.40. BookService, использующий внедрение зависимости и перехват

```
@Loggable
public class BookService {
  @Inject @ThirteenDigits
  private NumberGenerator numberGenerator;
  public Book createBook(String title, Float price, String description) {
    Book book = new Book(title, price, description);
    book.setNumber(numberGenerator.generateNumber());
    return book;
  }
}
```

`BookService` располагает одним методом, который берет название, цену и описание и возвращает POJO `Book`. Чтобы задать ISBN-номер книги, этот класс использует внедрение (`@Inject`) и квалификаторы (`@ThirteenDigits`) для вызова метода `generateNumber`, принадлежащего `IsbnGenerator`.

Листинг 2.41. POJO-объект Book

```
public class Book {
  private String title;
  private Float price;
  private String description;
  private String number;
  //Конструкторы, геттеры, сеттеры
}
```

В листинге 2.40 BookService аннотирован связкой с перехватчиком @Loggable (листинг 2.50). Когда эта связка действует, она регистрирует момент входа в метод и выхода из него.

Написание классов NumberGenerator

Класс BookService в листинге 2.40 зависит от интерфейса NumberGenerator (листинг 2.42). Этот интерфейс обладает методом, который генерирует и возвращает номер книги. Интерфейс реализуется классами IsbnGenerator, IssnGenerator и MockGenerator.

Листинг 2.42. Интерфейс NumberGenerator

```
public interface NumberGenerator {
  String generateNumber();
}
```

Класс IsbnGenerator (листинг 2.43) сопровождается квалификатором @ThirteenDigits. Это сообщает CDI о том, что сгенерированный номер состоит из 13 цифр. Заметьте, что класс IsbnGenerator также использует внедрение для получения java.util.logging.Logger (произведенного в листинге 2.48) и связывание с перехватчиком @Loggable для регистрации момента входа в метод и выхода из него.

Листинг 2.43. IsbnGenerator генерирует тринадцатизначный номер

```
@ThirteenDigits
public class IsbnGenerator implements NumberGenerator {
  @Inject
  private Logger logger;
  @Loggable
  public String generateNumber() {
    String isbn = "13-84356-" + Math.abs(new Random().nextInt());
    logger.info("Сгенерирован ISBN : " + isbn);
    return isbn;
  }
}
```

Класс IssnGenerator в листинге 2.44 — это восьмизначная реализация NumberGenerator.

Листинг 2.44. IssnGenerator генерирует восьмизначный номер

```
@EightDigits
public class IssnGenerator implements NumberGenerator{
  @Inject
  private Logger logger;
  @Loggable
  public String generateNumber() {
    String issn = "8-" + Math.abs(new Random().nextInt());
    logger.info("Сгенерирован ISBN : " + issn);
    return issn;
  }
}
```

Класс MockGenerator в листинге 2.45 является альтернативой IsbnGenerator (поскольку также сопровождается квалификатором @ThirteenDigits). MockGenerator используется только для интеграционных тестов, так как его можно активизировать только в файле beans.xml тестовой среды (см. листинг 2.55).

Листинг 2.45. Имитационный генератор чисел, применяемый в качестве альтернативы тринадцатизначного числа

```
@Alternative
@ThirteenDigits
public class MockGenerator implements NumberGenerator {
  @Inject
  private Logger logger;
  @Loggable
  public String generateNumber() {
    String mock = "MOCK-" + Math.abs(new Random().nextInt());
    logger.info("Сгенерирован Mock : " + mock);
    return mock;
  }
}
```

Написание квалификаторов

Поскольку существует несколько реализаций NumberGenerator, CDI необходимо квалифицировать каждый компонент и каждую точку внедрения во избежание неоднозначного внедрения. Для этого он использует два квалификатора: ThirteenDigits (листинг 2.46) и EightDigits (листинг 2.47), оба из которых аннотированы javax.inject.Qualifier и не имеют членов (просто пустые аннотации). Аннотация @ThirteenDigits применяется в компоненте IsbnGenerator (см. листинг 2.43), а также в точке внедрения BookService (см. листинг 2.40).

Листинг 2.46. Тринадцатизначный квалификатор

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD})
public @interface ThirteenDigits { }
```

Листинг 2.47. Восьмизначный квалификатор

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD})
public @interface EightDigits { }
```

Написание автоматического журнала

Демонстрационное приложение использует запись в журнал несколькими способами. Как вы могли видеть в листингах 2.43–2.45, все реализации NumberGenerator применяют внедрение для получения java.util.logging.Logger и записи в журнал. Поскольку Logger входит в состав JDK, он не является внедряемым по умолчанию (архив rt.jar не содержит файла beans.xml) и вам необходимо произвести его. Класс LoggingProducer в листинге 2.48 имеет метод продюсера (produceLogger), аннотиро-

ванный @Produces. Этот метод создаст и вернет Logger, сопровождаемый параметрами имени класса точки внедрения.

Листинг 2.48. Механизм записи в журнал

```
public class LoggingProducer {
  @Produces
  public Logger produceLogger(InjectionPoint injectionPoint) {
    return Logger.getLogger(injectionPoint.getMember().getDeclaringClass().
getName());
  }
}
```

Класс LoggingInterceptor в листинге 2.49 использует произведенный Logger для регистрации входа в методы и выхода из них. Поскольку запись в журнал может рассматриваться как сквозная функциональность, она отдельно реализуется в виде перехватчика (@AroundInvoke на logMethod). Класс LoggingInterceptor определяет связь с перехватчиком @Loggable (листинг 2.50) и может впоследствии применяться в любом компоненте (например, BookService в листинге 2.40).

Листинг 2.49. Перехватчик записывает в журнал методы при входе и при выходе

```
@Interceptor
@Loggable
public class LoggingInterceptor {
  @Inject
  private Logger logger;
  @AroundInvoke
  public Object logMethod(InvocationContext ic) throws Exception {
    logger.entering(ic.getTarget().getClass().getName(), ➥
ic.getMethod().getName());
    try {
      return ic.proceed();
    } finally {
      logger.exiting(ic.getTarget().getClass().getName(), ➥
ic.getMethod().getName());
    }
  }
}
```

Листинг 2.50. Связь с перехватчиком Loggable

```
@InterceptorBinding
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Loggable { }
```

Написание класса Main

Для запуска демонстрационного приложения нам необходим основной класс (назовем его Main), который приводит в действие контейнер CDI и вызывает метод BookService.createBook. CDI 1.1 не имеет стандартного API для начальной загрузки контейнера, поэтому код в листинге 2.51 специфичен для Weld. В первую очередь

он инициализирует WeldContainer и возвращает полностью сконструированную и внедренную сущность BookService.class. Затем вызов метода createBook станет использовать все сервисы контейнера: IsbnGenerator и Logger будут внедрены в BookService с последующим созданием и отображением Book с номером ISBN.

Листинг 2.51. Класс Main использует контейнер CDI для вызова BookService

```
public class Main {
  public static void main(String[] args) {
    Weld weld = new Weld();
    WeldContainer container = weld.initialize();
    BookService bookService =
container.instance().select(BookService.class).get();
    Book book = bookService.createBook("H2G2", 12.5f, "Интересная книга
                                       на тему высоких технологий");
    System.out.println(book);
    weld.shutdown();
  }
}
```

Код в листинге 2.51 специфичен для Weld, поэтому его нельзя портировать. Он не будет работать на других реализациях CDI, таких как OpenWebBeans (Apache) или CanDI (Caucho). Одна из целей будущего релиза CDI — стандартизировать API для начальной загрузки контейнера.

Приведение в действие CDI с beans.xml

Чтобы привести в действие CDI и позволить демонстрационному приложению заработать, нам требуется файл beans.xml в пути к классам приложения. Как следует из листинга 2.52, файл beans.xml совершенно пуст, но без него нельзя будет задействовать CDI, осуществить обнаружение компонентов либо внедрение.

Листинг 2.52. Пустой файл beans.xml для запуска CDI

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       version="1.1" bean-discovery-mode="all">
</beans>
```

Компиляция и выполнение с помощью Maven

Теперь необходимо скомпилировать все классы перед запуском класса Main и интеграционным тестом BookServiceIT. Файл pom.xml в листинге 2.53 объявляет все необходимые зависимости для компиляции кода (org.jboss.weld.se:weld-se содержит APICDI и реализацию Weld) и запуска теста (junit:junit). Указание версии 1.7 в maven-compiler-plugin означает, что вы хотите использовать Java SE 7 (<source>1.7</source>). Заметьте, что мы применяем exec-maven-plugin для того, чтобы иметь возможность выполнить класс Main с помощью фреймворка Maven.

Листинг 2.53. Файл pom.xml для компиляции, запуска и тестирования

```
<?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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.agoncal.book.javaee7</groupId>
    <artifactId>chapter02</artifactId>
    <version>1.0</version>
  </parent>

  <groupId>org.agoncal.book.javaee7.chapter02</groupId>
  <artifactId>chapter02-putting-together</artifactId>
  <version>1.0</version>

  <dependencies>
    <dependency>
      <groupId>org.jboss.weld.se</groupId>
      <artifactId>weld-se</artifactId>
      <version>2.0.0</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.5.1</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>exec-maven-plugin</artifactId>
        <version>1.2.1</version>
        <executions>
          <execution>
            <goals>
              <goal>java</goal>
```

```
            </goals>
            <configuration>
              <mainClass>org.agoncal.book.javaee7.chapter02.Main</mainClass>
            </configuration>
          </execution>
        </executions>
      </plugin>

      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>2.12.4</version>
        <executions>
          <execution>
            <id>integration-test</id>
            <goals>
              <goal>integration-test</goal>
              <goal>verify</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Для компиляции классов откройте командную строку в корневом каталоге, содержащем файл `pom.xml`, и введите следующую команду Maven:

```
$ mvn compile
```

Запуск класса Main

Благодаря `exec-maven-plugin`, сконфигурированному в файле `pom.xml` в листинге 2.53, теперь мы можем запустить класс `Main`, определенный в листинге 2.51. Откройте командную строку в корневом каталоге, содержащем файл `pom.xml`, и введите следующую команду Maven:

```
$ mvn exec:java
```

После этого начнется выполнение класса `Main`, который использует `BookService` для создания `Book`. Благодаря внедрению `Logger` будет отображать следующие выходные данные:

```
Info: Сгенерирован ISBN : 13-84356-1864341788
Book{title='H2G2', price=12.5, description='Интересная книга на тему высоких
технологий', isbn='13-84356-1864341788'}
```

Написание класса BookServiceIT

Листинг 2.54 показывает, как класс `BookServiceIT` тестирует компонент `BookService`. Он использует тот же API, специфичный для Weld, для начальной загрузки CDI

в качестве класса Main, показанного в листинге 2.51. Как только вызывается BookService.createBook, интеграционный тест проверяет, чтобы сгенерированный номер начинался с "MOCK". Это происходит потому, что интеграционный тест использует альтернативу MockGenerator (вместо IsbnGenerator).

Листинг 2.54. Интеграционный тест BookServiceIT

```
public class BookServiceIT {
  @Test
  public void shouldCheckNumberIsMOCK () {
    Weld weld = new Weld();
    WeldContainer container = weld.initialize();
    BookService bookService =
container.instance().select(BookService.class).get();
    Book book = bookService.createBook("H2G2", 12.5f, "Интересная книга
                                       на тему высоких технологий");
    assertTrue(book.getNumber().startsWith("MOCK"));
    weld.shutdown();
  }
}
```

Активизация альтернатив и перехватчиков в файлах beans.xml для интеграционного тестирования

Интеграционный тест BookServiceIT в листинге 2.54 требует, чтобы был активизирован MockGenerator. Для этого необходимо выделить отдельный файл beans.xml для тестирования (листинг 2.55) и активизации альтернатив (с тегом <alternatives>). Если вы захотите увеличить объем журналов в тестовой среде, можете активизировать LoggingInterceptor в файле beans.xml.

Листинг 2.55. Активизация альтернатив и перехватчиков в файле beans.xml

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
       http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       version="1.1" bean-discovery-mode="all">
  <alternatives>
    <class>org.agoncal.book.javaee7.chapter02.MockGenerator</class>
  </alternatives>
  <interceptors>
    <class>org.agoncal.book.javaee7.chapter02.LoggingInterceptor</class>
  </interceptors>
</beans>
```

Запуск интеграционного теста

Для выполнения интеграционного теста с помощью плагина Maven Failsafe (определенного в pom.xml в листинге 2.53) введите следующую команду Maven:

```
$ mvn integration-test
```

Класс `BookServiceIT` должен выполнить один успешный интеграционный тест. Вы также увидите несколько протоколов или регистраций входа в метод и выхода из него.

Резюме

Из этой главы вы узнали о различиях между объектом POJO, управляемым компонентом и компонентом CDI, а также о том, какие сервисы необходимо применять с каждой компонентной моделью. Благодаря спецификациям Managed Bean (запрос JSR 330) и CDI Bean (запрос JSR 299) в Java EE появилась стандартная, портируемая и типобезопасная поддержка для внедрения зависимостей. CDI предлагает дополнительные возможности, такие как области видимости и контексты, а также расширенные перехватчики, декораторы и события. В действительности CDI напрямую реализует несколько шаблонов проектирования, например «Мост» (с альтернативами), «Наблюдатель» (с событиями), «Декоратор», «Фабрика» (с производителями), и, конечно же, перехват и внедрение.

Перехватчики — это механизм Java EE, имеющий много общего с аспектно-ориентированным программированием, который позволяет контейнеру вызывать сквозные функции вашего приложения. Перехватчики легкие в использовании, мощные и могут быть сцеплены вместе либо расположены в приоритетном порядке для того, чтобы применить к вашему компоненту различную функциональность.

Будучи вертикальной спецификацией, CDI используется в других спецификациях Java EE. На самом деле в следующих главах этой книги так или иначе будут задействованы некоторые сервисы CDI.

Глава 3
Валидация компонентов

В предыдущей главе мы говорили о CDI, контексте и внедрении зависимостей, — центральной общей спецификации, действующей на всей платформе Java EE. Она помогает справляться с регулярно возникающими проблемами, связанными с внедрением, альтернативами, стереотипами, производителями данных. Разработчику приходится постоянно иметь дело с такими концепциями при решении рутинных задач. Еще одна распространенная задача, затрагивающая сразу несколько уровней современных приложений (или даже все уровни), — это валидация информации. Она выполняется повсюду, от уровня представления до базы данных. Поскольку обработка, хранение и получение валидных данных — это важнейшие задачи любого приложения, на каждом уровне правила валидации определяются по-своему. Но иногда на всех уровнях бывает реализована одна и та же логика валидации, из-за чего эта процедура становится длительной, сложной при поддержке, а также чреватой ошибками. Чтобы избежать дублирования правил валидации на каждом из уровней приложения, разработчики часто вплетают логику валидации непосредственно в предметную модель. В результате классы предметной модели оказываются переполнены валидационным кодом, который фактически представляет собой метаданные, описывающие сам класс.

Валидация компонентов решает проблему дублирования кода и излишнего запутывания классов предметной модели. Разработчику достаточно написать ограничение всего один раз, использовать его и валидировать на любом уровне. Валидация компонентов позволяет реализовать ограничение в обычном коде Java, а потом определить его с помощью аннотации (метаданных). Затем вы можете использовать эту аннотацию в своем компоненте, свойствах, конструкторах, параметрах методов и с возвращаемым значением. Валидация компонентов предоставляет простой API, позволяющий разработчикам писать и переиспользовать ограничения, связанные с бизнес-логикой.

Из этой главы вы узнаете, почему валидация имеет принципиальное значение в приложении и почему ее необходимо дублировать на разных уровнях. Вы научитесь писать ограничения: от агрегирования уже имеющихся до разработки ваших собственных. Вы увидите, как применять такие ограничения в вашем приложении от уровня представления вплоть до уровня бизнес-модели. Затем вы научитесь валидировать эти ограничения (как внутри контейнера Java EE, так и вне его).

Понятие об ограничениях и валидации

Большую часть рабочего времени программист тратит на то, чтобы гарантировать валидность (пригодность) тех данных, которые обрабатываются и сохраняются

в приложении. Разработчик пишет ограничения для допустимых значений данных, применяет эти ограничения к логике и модели приложения, а также гарантирует, что на различных уровнях валидация этих ограничительных условий происходит согласованно. Таким образом, эти ограничения должны применяться в клиентском приложении (например, в браузере, если речь идет о разработке веб-приложения), на уровне представления, бизнес-логики, бизнес-модели (она же — предметная модель), в схеме базы данных и в определенной степени на уровне интероперабельности (рис. 3.1). Разумеется, для обеспечения согласованности необходимо синхронизировать эти правила на всех используемых уровнях и в применяемых технологиях.

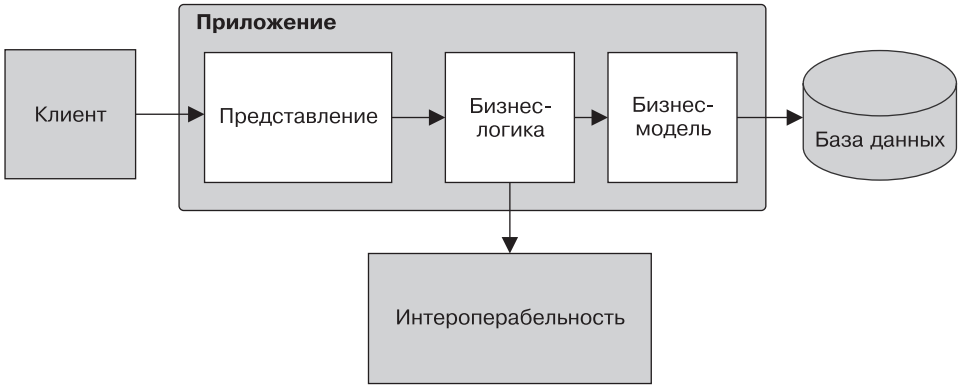


Рис. 3.1. Валидация происходит сразу на нескольких уровнях

В неоднородных приложениях разработчику приходится иметь дело сразу с несколькими технологиями и языками. Поэтому даже простое правило валидации, например «этот элемент данных является обязательным и не может быть равен нулю», приходится по-разному выражать на Java, JavaScript, в схеме базы данных или в XML-схеме.

Приложение

Независимо от того, создаете вы одноуровневое или многоуровневое приложение, вам так или иначе необходимо гарантировать, что обрабатываемые вашей программой данные корректны. Например, если адрес доставки или форма заказа товара не заполнены, вы не сможете отправить покупку клиенту. На Java вам то и дело приходится писать код, проверяющий правильность той или иной записи (`if order.getDeliveryAddress() == null`). Этот код может выдавать исключение или требовать участия пользователя для исправления внесенных данных. Валидация на уровне приложения обеспечивает довольно детализированный контроль, а также позволяет применять сравнительно сложные ограничения. (Является ли данная дата государственным праздником во Франции? Превышает ли общая среднегодовая сумма на счету клиента некоторое среднее значение?)

Валидация на уровне приложения, обеспечивающая правильность введенных данных, может проводиться в разных местах.

- **Уровень представления.** На этом уровне данные валидируются потому, что вы можете получать их от разных клиентов (браузер; инструмент для работы с командной строкой, например с URL, позволяющий отправлять команды по протоколу HTTP; нативное приложение). На этом же уровне вы стараетесь обеспечить быструю обратную связь с пользователем.
- **Уровень бизнес-логики.** Здесь координируются вызовы, направляемые к внутренним и внешним службам, к предметной модели, и так обеспечивается валидность обрабатываемых данных.
- **Уровень бизнес-модели.** Данный уровень обычно обеспечивает отображение предметной модели на базу данных, поэтому здесь валидация должна происходить до сохранения данных.

В сложном приложении зачастую приходится многократно реализовывать одно и то же ограничение сразу на нескольких уровнях, из-за чего значительная часть кода дублируется.

База данных

В конце концов, основная цель — хранить в вашей базе данных только валидную информацию, чтобы обработку можно было выполнить позднее. Строгие ограничения должны соблюдаться в реляционных базах данных, потому что здесь применяются схемы. Язык определения данных (DDL, также называемый языком описания данных) использует специальный синтаксис для определения и ограничения структур, входящих в состав базы данных. После этого вы сможете гарантировать, что данные в столбце не могут быть равны нулю (`NOT NULL`), должны быть целочисленными (`INTEGER`) или иметь ограничение максимальной длины (`VARCHAR(20)`). В последнем примере попытка вставить в столбец строку длиной 20 символов окончится неудачей.

Тем не менее выполнение валидации на уровне базы данных связано с некоторыми недостатками: негативное влияние на производительность, сообщения об ошибках выдаются вне контекста. Неверные данные должны пересечь все уровни приложения, прежде чем будут отправлены на удаленный сервер базы данных, который затем обработает валидацию и отправит обратно сообщение об ошибке. Ограничения, применяемые на уровне базы данных, учитывают только фактические данные, но не действия, осуществляемые пользователем. Поэтому сообщения об ошибках выдаются вне контекста и не могут точно описывать ситуацию. Именно по этой причине нужно стараться валидировать данные раньше — в приложении или еще на клиенте.

Клиент

Валидация данных на стороне клиента очень важна: так пользователь сможет оперативно узнавать о том, что при вводе данных он допустил ошибку. Такая валидация сокращает количество избыточных обращений к серверу, практика использования также улучшается, поскольку многие ошибки отлавливаются на раннем этапе. Эта возможность особенно важна при разработке мобильных приложений, которые обычно должны обходиться очень узкой полосой доступа к сети.

Например, в типичном веб-приложении на уровне браузера применяется код JavaScript. Он обеспечивает простую валидацию данных в отдельных полях, а на

уровне сервера проверяется соответствие более сложным бизнес-правилам. Нативные приложения, написанные на Java (Swing, мобильные приложения для Android), могут использовать весь потенциал этого языка при записи и валидации данных.

Интероперабельность

Зачастую корпоративные приложения должны обмениваться данными с внешними партнерами/системами. Такие приложения, относящиеся к категории «бизнес для бизнеса», получают информацию в любом формате, обрабатывают их, сохраняют и отправляют обратно партнеру. Валидация пользовательских (заказных) форматов — порой сложная и затратная задача. В настоящее время обмен данными между неоднородными системами обычно организуется на языке XML. Базы данных могут применять язык DDL для определения своей структуры. Аналогичным образом XML может использовать XSD (язык описания структуры XML-документа) для ограничения XML-документов. XSD выражает несколько правил, которым должен соответствовать XML-документ, чтобы удовлетворять выбранной схеме. Синтаксический разбор и валидация XML — распространенная задача, которая легко выполняется во многих фреймворках Java.

Обзор спецификации валидации компонентов

Как видите, валидация распределена по разным уровням приложения (от клиента до базы данных) и используется в разных технологиях (JavaScript, Java, DDL, XSD). Это означает, что разработчику приходится дублировать валидационный код на нескольких уровнях и писать его на разных языках. На такой распространенный подход тратится немало времени, он чреват ошибками, а в дальнейшем осложняется поддержка приложения. Кроме того, некоторые из подобных ограничений используются настолько часто, что их можно считать стандартами (проверка значения, его размера, диапазона). Было бы очень удобно централизовать эти ограничения в одном месте и совместно использовать их на разных уровнях. Вот здесь нам и пригодится валидация компонентов.

Валидация компонентов (Bean Validation) — это технология, ориентированная на Java, хотя и предпринимаются попытки интегрировать другие валидационные языки, в частности DDL или XSD. Валидация компонентов позволяет вам написать ограничение лишь однажды и использовать его на любом уровне приложения. Она не зависит от уровня, то есть одно и то же ограничение может использоваться повсюду, от уровня представления до уровня бизнес-модели. Валидация компонентов доступна как в серверных приложениях, так и в насыщенных графических клиентских интерфейсах, написанных на Java (Swing, Android). Такая валидация считается расширением объектной модели JavaBeans и, в принципе, может использоваться в качестве ядра других спецификаций (в этом вы убедитесь в следующих главах книги).

Валидация компонентов позволяет применять уже готовые, заранее определенные ограничения, а также писать собственные ограничения и также использовать их для валидации компонентов, атрибутов, конструкторов, возвращаемых типов

методов и параметров. Этот API очень прост и при этом гибок, поскольку стимулирует вас определять собственные ограничения с применением аннотаций (как вариант — с использованием XML).

Краткая история валидации компонентов

Разработчики занимаются ограничением и валидацией создаваемых моделей с тех пор, как существует язык Java. Код и фреймворки сначала были кустарными, они породили определенные практики, которые стали применяться в первых свободных проектах. Так, еще в 2000 году валидация пользовательского ввода стала использоваться в Struts — знаменитом MVC-фреймворке для Сети. Но прошло еще некоторое время, прежде чем появились целые валидационные фреймворки, предназначенные исключительно для работы с Java (а не просто для обеспечения веб-взаимодействий). Наиболее известными из таких фреймворков являются, пожалуй, Commons Validator из Apache Commons и Hibernate Validator. Другие подобные фреймворки — iScreen, OVal, а также распространенный фреймворк Spring, в котором содержится собственный валидационный пакет.

Что нового появилось в версии Bean Validation 1.1

В настоящее время валидация компонентов в версии 1.1 интегрирована в Java EE 7. В этой младшей (корректировочной) версии появились многие новые возможности, были улучшены существующие. Рассмотрим основные новые возможности.

- В этой версии ограничения могут применяться к параметрам методов и возвращаемым значениям. Поэтому валидация компонентов может использоваться для описания и валидации контракта (предусловий и постусловий) заданного метода.
- Ограничения также могут применяться с конструкторами.
- Появился новый API для получения метаданных об ограничениях и объектах, подвергаемых этим ограничениям.
- Повысилась степень интеграции со спецификацией, описывающей контекст и внедрение зависимостей (теперь можно выполнять внедрение в валидаторы).

В табл. 3.1 перечислены основные пакеты, входящие в настоящее время в спецификацию Bean Validation 1.1.

Таблица 3.1. Основные пакеты для валидации компонентов

Пакет	Описание
javax.validation	Содержит основные API для валидации компонентов
javax.validation.bootstrap	Классы, применяемые для начальной загрузки валидации компонентов и создания конфигурации, не зависящей от поставщика
javax.validation.constraints	Содержит все встроенные ограничения
javax.validation.groups	Стандартные группы для валидации компонентов

Продолжение ⇩

Таблица 3.1 *(продолжение)*

Пакет	Описание
javax.validation.metadata	Репозиторий метаданных для всех определенных ограничений и API запросов
javax.validation.spi	API, определяющие контракт между механизмом начальной загрузки валидации и движком поставщика

Справочная реализация

Hibernate Validator — это свободная справочная реализация валидации компонентов. Проект изначально был запущен в 2005 году компанией JBoss в рамках Hibernate Annotations, стал независимым в 2007 году, а статус справочной реализации приобрел в 2009 году (с выходом Hibernate Validator 4). В настоящее время Hibernate Validator 5 реализует валидацию компонентов (версия 1.1) и добавляет кое-какие собственные возможности, в числе которых политика быстрого отказа. В соответствии с этим принципом программа прерывает текущую валидацию и возвращается после первого же нарушения ограничивающих условий. К другим характерным особенностям этой реализации относится API для программного конфигурирования ограничений, а также дополнительные встроенные ограничения.

На момент написания этой книги Hibernate Validator 5 был единственной реализацией, совместимой с Bean Validation 1.1. В Apache BVal применялась спецификация Bean Validation 1.0, в настоящее время идет процесс сертификации на соответствие версии 1.1. Oval не реализует полную спецификацию Bean Validation, но умеет обрабатывать связанные с ней ограничения.

Написание ограничений

До сих пор мы говорили об ограничениях, применяемых к нескольким уровням вашего приложения. Такие ограничения могут быть написаны одновременно на нескольких языках, с применением разных технологий. Но я также упоминал и о дублировании валидационного кода. Итак, насколько сложно будет применить ограничение в ваших классах Java, использующих валидацию компонентов? В листинге 3.1 показано, насколько просто добавлять ограничения в бизнес-модель.

Листинг 3.1. Объект Book. Ограничения записаны в виде аннотаций

```
public class Book {
  @NotNull
  private String title;
  @NotNull @Min(2)
  private Float price;
  @Size(max = 2000)
  private String description;
  private String isbn;
  private Integer nbOfPage;
  // Конструкторы, геттеры, сеттеры
}
```

В листинге 3.1 показан класс Book с атрибутами, конструкторами, геттерами, сеттерами и аннотациями. Некоторые из этих атрибутов аннотированы с применением встроенных ограничений, таких как @NotNull, @Min и @Size. Так мы указываем валидационной среде исполнения, что заголовок title книги не может быть равен нулю и описание description не может превышать 2000 символов. Как видите, к атрибуту могут быть применены несколько ограничений (например, price не может быть равно нулю и его значение не может быть меньше 2).

Внутренняя организация ограничения

Ограничение определяется как комбинация ограничивающей аннотации и списка реализаций валидации ограничения. Ограничивающая аннотация применяется с типами, методами, полями или другими ограничивающими аннотациями (в случае с составными элементами). В большинстве спецификаций Java EE разработчики используют заранее определенные аннотации (например, @Entity, @Stateless и @Path). Но в случае с CDI (об этом шла речь в предыдущей главе) и при валидации компонентов программистам приходится писать собственные аннотации. Известно, что ограничение при валидации компонентов состоит из:

- аннотации, определяющей ограничение;
- списка классов, реализующих ограничивающий алгоритм с заданным типом.

В то же время аннотация выражает ограничение, действующее в предметной модели. Таким образом, реализация валидации определяет, удовлетворяет ли конкретное значение заданному ограничению.

Ограничивающая аннотация

Ограничение, применяемое с JavaBean, выражается с помощью одной или нескольких аннотаций. Аннотация считается ограничивающей, если применяемая в ней политика хранения содержит RUNTIME и если сама она аннотирована javax.validation.Constraint (эта аннотация ссылается на список реализаций валидации). В листинге 3.2 показана ограничивающая аннотация NotNull. Как видите, @Constraint(validatedBy = {}) указывает на класс реализации NotNullValidator.

Листинг 3.2. Ограничивающая аннотация NotNull

```
@Target({METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER})
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = NotNullValidator.class)
public @interface NotNull {
  String message() default "{javax.validation.constraints.NotNull.message}";
  Class<?>[] groups() default {};
  Class<? extends Payload>[] payload() default {};
}
```

Ограничивающие аннотации — это самые обычные аннотации, поэтому с ними приходится определять своеобразные метааннотации:

- `@Target({METHOD, FIELD, ...})` — указывает цель, для которой может использоваться аннотация (подробнее об этом — ниже);
- `@Retention(RUNTIME)` — определяет, как мы будем обращаться с аннотацией. Необходимо использовать как минимум `RUNTIME`, чтобы поставщик мог проверять ваш объект во время выполнения;
- `@Constraint(validatedBy = NotNullValidator.class)` — указывает класс (в случае агрегации ограничений — нуль либо список классов), в котором инкапсулирован валидационный алгоритм;
- `@Documented` — определяет, будет эта аннотация включена в Javadoc или нет. Опциональная метааннотация.

Поверх этих общих метааннотаций спецификация Bean Validation требует задавать для каждой ограничивающей аннотации три дополнительных атрибута:

- `message` — обеспечивает для аннотации возможность возвращения интернационализированного сообщения об ошибке, выдаваемого в том случае, если ограничение недопустимо. По умолчанию данный атрибут равен ключу;
- `groups` — используется для контроля за порядком, в котором интерпретируются ограничения, либо для выполнения частичной валидации;
- `payload` — применяется для ассоциирования метаинформации с ограничением.

Если ваше ограничение определяет все обязательные метааннотации и ограничения, можете добавить любой интересующий вас конкретный параметр. Например, ограничение, проверяющее длину строки, может использовать атрибут `length` для указания максимальной длины.

Реализация ограничений

Ограничение определяется комбинацией самой аннотации и нуля или более классов реализации. Классы реализации указываются элементом `validatedBy` в `@Constraint` (как показано в листинге 3.2). Листинг 3.3 демонстрирует класс реализации для аннотации `@NotNull`. Как видите, он реализует интерфейс `ConstraintValidator` и использует дженерики для передачи имени аннотации (`NotNull`) и типа, к которому применяется аннотация (в данном случае это `Object`).

Листинг 3.3. Реализация ограничения NotNull

```
public class NotNullValidator implements ConstraintValidator<NotNull, Object>
{
  public void initialize(NotNull parameters) {
  }
  public boolean isValid(Object object, ConstraintValidatorContext context) {
    return object != null;
  }
}
```

Интерфейс `ConstraintValidator` определяет два метода, которые обязательны для реализации конкретными классами.

- `initialize` — вызывается поставщиком валидации компонентов еще до применения какого-либо ограничения. Именно здесь мы обычно инициализируем любые параметры ограничения, если они имеются.
- `isValid` — здесь реализуется валидационный алгоритм. Метод интерпретируется поставщиком валидации компонентов всякий раз, когда проверяется конкретное значение. Метод возвращает `false`, если значение является недопустимым, в противном случае — `true`. Объект `ConstraintValidatorContext` несет информацию и операции, доступные в том контексте, к которому применяется ограничение.

Реализация ограничения выполняет валидацию заданной аннотации для указанного типа. В листинге 3.3 ограничение `@NotNull` типизируется к `Object` (это означает, что данное ограничение может использоваться с любым типом данных). Но у вас может быть и такая ограничивающая аннотация, которая применяет различные алгоритмы валидации в зависимости от того, об обработке какого типа данных идет речь. Например, вы можете проверять максимальное количество символов для `String`, максимальное количество знаков для `BigDecimal` или максимальное количество элементов для `Collection`. Обратите внимание: в следующей аннотации у нас несколько реализаций одной и той же аннотации (`@Size`), но она используется с разными типами данных (`String`, `BigDecimal` и `Collection<?>`):

```
public class SizeValidatorForString     implements<Size, String>     {...}
public class SizeValidatorForBigDecimal implements<Size, BigDecimal> {...}
public class SizeValidatorForCollection implements<Size, Collection<?>> {...}
```

Применение ограничения

Когда у вас есть аннотация и ее реализация, вы можете применять ограничение с элементом заданного типа (атрибут, конструктор, параметр, возвращаемое значение, компонент, интерфейс или аннотация). Это решение, которое разработчик принимает на этапе проектирования и реализует с помощью метааннотации `@Target(ElementType.*)` (см. листинг 3.2). Существуют следующие типы:

- `FIELD` — для ограниченных атрибутов;
- `METHOD` — для ограниченных геттеров и возвращаемых значений ограниченных методов;
- `CONSTRUCTOR` — для возвращаемых значений ограниченных конструкторов;
- `PARAMETER` — для параметров ограниченных методов и конструкторов;
- `TYPE` — для ограниченных компонентов, интерфейсов и суперклассов;
- `ANNOTATION_TYPE` — для ограничений, состоящих из других ограничений.

Как видите, ограничивающие аннотации могут применяться с большинством типов элементов, определяемых в Java. Только статические поля и статические методы не могут проверяться валидацией компонентов. В листинге 3.4 показан класс `Order`, где ограничивающие аннотации применяются к самому классу, атрибутам, конструктору и бизнес-методу.

Листинг 3.4. Объект, использующий ограничения с элементами нескольких типов

```
@ChronologicalDates
public class Order {
  @NotNull @Pattern(regexp = "[C,D,M][A-Z][0-9]*")
  private String orderId;
  private Date creationDate;
  @Min(1)
  private Double totalAmount;
  private Date paymentDate;
  private Date deliveryDate;
  private List<OrderLine> orderLines;
  public Order() {
  }
  public Order(@Past Date creationDate) {
    this.creationDate = creationDate;
  }
  public @NotNull Double calculateTotalAmount(@GreaterThanZero Double
changeRate) {
    // ...
  }
  // Геттеры и сеттеры
}
```

В листинге 3.4 `@ChronologicalDates` — это ограничение, действующее на уровне класса и основанное на нескольких свойствах класса `Order` (в данном случае оно гарантирует, что значения `creationDate`, `paymentDate` и `deliveryDate` являются хронологическими). Атрибут `orderId` имеет два ограничения: он не может быть равен нулю (`@NotNull`) и должен соответствовать шаблону регулярного выражения (`@Pattern`). Конструктор `Order` гарантирует, что параметр `creationDate` должен обозначать момент в прошлом. Метод `calculateTotalAmount` (рассчитывающий общую сумму заказа на покупку) проверяет, является ли `changeRate` значением больше нуля — `@GreaterThanZero`, а также гарантирует, что возвращаемая сумма будет ненулевой.

ПРИМЕЧАНИЕ

До сих пор в примерах я показывал аннотированные атрибуты, но вместо этого можно аннотировать геттеры. Вы можете определять ограничения либо для атрибута, либо для геттера, но не для обоих одновременно. Лучше оставаться последовательным и всегда использовать аннотации либо только с атрибутами, либо только с геттерами.

Встроенные ограничения

Спецификация Bean Validation позволяет вам писать собственные ограничения и валидировать их. Но в ней присутствует и несколько встроенных ограничений. Мы уже сталкивались с некоторыми из них в предыдущих примерах, но в табл. 3.2 приведен исчерпывающий список всех встроенных ограничений (таких, которые уже готовы для использования в коде и не требуют разработки аннотации или класса реализации). Все встроенные ограничения определяются в пакете `javax.validation.constraints`.

Таблица 3.2. Полный список встроенных ограничивающих аннотаций

Ограничение	Приемлемые типы	Описание
AssertFalse AssertTrue	Boolean	Аннотированный элемент должен возвращать значение true или false
DecimalMax DecimalMin	BigDecimal, BigInteger, CharSequence, byte, short, int, long и соответствующие обертки	Элемент должен быть меньше или больше указанного значения
Future Past	Calendar, Date	Аннотированный элемент должен быть датой в прошлом или будущем
Max Min	BigDecimal, BigInteger, byte, short, int, long и их обертки	Элемент должен быть меньше или больше указанного значения
Null NotNull	Object	Аннотированный элемент должен быть равен или не равен нулю
Pattern	CharSequence	Элемент должен соответствовать указанному регулярному выражению
Digits	BigDecimal, BigInteger, CharSequence, byte, short, int, long и соответствующие обертки	Аннотированный элемент должен быть числом в допустимом диапазоне
Size	Object[], CharSequence, Collection<?>, Map<?, ?>	Размер элемента должен укладываться в указанные границы

Определение собственных ограничений

Как вы уже видели, API валидации компонентов предоставляет стандартные встроенные ограничения, но они вполне могут не удовлетворить всех нужд вашего приложения. Существует несколько способов создания собственных ограничений (от агрегирования уже имеющихся до написания нового ограничения с нуля). Есть разные стили выполнения такой работы (например, создание обобщенных ограничений или ограничений, действующих на уровне класса).

Объединение ограничений

Удобный способ создания новых ограничений — агрегирование (объединение) уже имеющихся. В таком случае мы обходимся без класса реализации. Это совсем не сложно сделать, если имеющиеся ограничения обладают `@Target(ElementType.ANNOTATION_TYPE)`, то есть при работе с ними одна аннотация может быть применена к другой. Такой подход называется «объединением ограничений» и позволяет создавать высокоуровневые ограничения.

В листинге 3.5 показано, как создать ограничение `Email`, обходясь лишь встроенными ограничениями из API валидации компонентов. Это ограничение гарантирует, что адрес электронной почты является ненулевым (`@NotNull`), состоит не менее чем из семи символов (`@Size(min = 7)`) и соответствует сложному регулярному выражению (`@Pattern`). В таком объединенном ограничении также должны

определяться атрибуты `message`, `groups` и `payload`. Обратите внимание: класс реализации здесь отсутствует (`validatedBy = {}`).

Листинг 3.5. Ограничение для работы с электронной почтой, составленное из других ограничений

```
@NotNull
@Size(min = 7)
@Pattern(regexp = "[a-z0-9!#$%&'*+/=?^_`{|}~-]+ ➥
(?:\\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*" ➥
        + "@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.)+[a-z0-9](?:[a-z0-9-]* ➥
        [a-z0-9])?")
@Constraint(validatedBy = {})
@Target({METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface Email {
  String message() default "Неверный электронный адрес";
  Class<?>[] groups() default {};
  Class<? extends Payload>[] payload() default {};
}
```

Все встроенные ограничения (`@NotNull`, `@Size` и `@Pattern`) уже имеют собственные сообщения об ошибках (элемент `message()`). Таким образом, если у вас окажется нулевой адрес электронной почты, то по результатам валидации ограничение, приведенное в листинге 3.5, выдаст сообщение об ошибке `@NotNull`, а не то, что вы определили (Неверный электронный адрес). Возможно, вы захотите использовать для всех ограничений `Email` одно общее сообщение об ошибке, а не несколько разных. Для этого можно добавить аннотацию `@ReportAsSingleViolation` (как будет показано ниже, в листинге 3.24). Если вы так поступите, то валидация составного ограничения прекратится после невыполнения первого же из входящих в него ограничений и будет выдано сообщение об ошибке, соответствующее именно тому ограничению, которое оказалось невыполненным.

Объединение ограничений полезно, так как подобная практика помогает избежать дублирования кода и способствует переиспользованию сравнительно простых ограничений. В таком случае вы будете скорее создавать простые ограничения, чем составлять из них более сложные валидационные правила.

ПРИМЕЧАНИЕ

Создавая новое ограничение, обязательно дайте ему информативное имя. Если название для аннотации будет подобрано правильно, то ограничения будут лучше восприниматься в коде.

Обобщенное ограничение

Объединение простых ограничений — полезная практика, но, как правило, только ею не обойтись. Часто приходится применять сложные валидационные алгоритмы: проверять значение в базе данных, делегировать определенную валидационную работу вспомогательным классам и т. д. Именно в таких случаях приходится добавлять к вашей ограничивающей аннотации класс реализации.

В листинге 3.6 показан объект POJO, представляющий сетевое соединение с сервером, на котором находятся элементы CD-BookStore. Этот POJO имеет несколько атрибутов типа `String`, все они представляют URL. Вы хотите, чтобы URL имел допустимый формат, и даже задаете конкретный протокол (например, http, ftp...), хост и/или номер порта. Пользовательское ограничение `@URL` гарантирует, что разные строковые атрибуты класса `ItemServerConnection` соответствуют формату URL. Например, атрибут `resourceURL` может представлять собой любой допустимый URL (в частности, file://www.cdbookstore.com/item/123). С другой стороны, вы хотите ограничить атрибут `itemURL` так, чтобы он работал только с http-протоколом и с хостом, имя которого начинается с www.cdbookstore.com (например, http://www.cdbookstore.com/book/h2g2).

Листинг 3.6. Ограничивающая аннотация для работы с URL, используемая с несколькими атрибутами

```
public class ItemServerConnection {
  @URL
  private String resourceURL;
  @NotNull @URL(protocol = "http", host = "www.cdbookstore.com")
  private String itemURL;
  @URL(protocol = "ftp", port = 21)
  private String ftpServerURL;
  private Date lastConnectionDate;
  // Конструкторы, геттеры, сеттеры
}
```

Если мы хотим создать такое пользовательское ограничение для работы с URL, то первым делом должны определить аннотацию. В листинге 3.7 показана аннотация, выполняющая все предпосылки для валидации компонентов (метааннотация `@Constraint`, атрибуты `message`, `groups` и `payload`). Кроме того, она добавляет и специфические атрибуты: `protocol`, `host` и `port`, которые отображаются на имена элементов аннотации (например, `@URL(protocol = "http")`). Ограничение может использовать любой атрибут любого типа данных. Обратите также внимание, что у этих атрибутов есть значения, задаваемые по умолчанию, — например, пустая строка для протокола или хоста или `-1` для номера порта.

Листинг 3.7. Ограничивающая аннотация для работы с URL

```
@Constraint(validatedBy = {URLValidator.class})
@Target({METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER})
@Retention(RUNTIME)
public @interface URL {
  String message() default "Malformed URL";
  Class<?>[] groups() default {};
  Class<? extends Payload>[] payload() default {};
  String protocol() default "";
  String host() default "";
  int port() default -1;
}
```

В листинге 3.7 мы могли бы агрегировать уже имеющиеся ограничения, например `@NotNull`. Но основное различие между объединением ограничений и созданием обобщенного ограничения заключается в применении класса реализации, объявляемого в атрибуте `validatedBy` (здесь он ссылается на класс `URLValidator.class`).

В листинге 3.8 показан класс реализации `URLValidator`. Как видите, он реализует интерфейс `ConstraintValidator` и, следовательно, методы `initialize` и `isValid`. Здесь важно отметить, что класс `URLValidator` имеет три атрибута, определенные в аннотации (`protocol`, `host` и `port`), и инициализирует их в методе `initialize(URL url)`. Этот метод вызывается на этапе инстанцирования валидатора. В качестве параметра он получает ограничивающую аннотацию (здесь `URL`), поэтому может извлекать значения и использовать их при валидации. Так можно поступить с атрибутом `itemURL protocol`, который в листинге 3.6 имеет строковое значение `"http"`.

Листинг 3.8. Реализация ограничения для работы с URL

```
public class URLValidator implements ConstraintValidator<URL, String> {
  private String protocol;
  private String host;
  private int port;
  public void initialize(URL url) {
    this.protocol = url.protocol();
    this.host = url.host();
    this.port = url.port();
  }
  public boolean isValid(String value, ConstraintValidatorContext context) {
    if (value == null || value.length() == 0) {
    return true;
  }
  java.net.URL url;
  try {
    // Преобразуем URL в java.net.URL для проверки того,
    // имеет ли URL допустимый формат
    url = new java.net.URL(value);
  } catch (MalformedURLException e) {
    return false;
  }
  // Проверяет, имеет ли атрибут протокола допустимое значение
  if (protocol != null && protocol.length() > 0 && ➥
!url.getProtocol().equals(protocol)) {
    return false;
  }
  if (host != null && host.length() > 0 && !url.getHost().startsWith(host)) {
    return false;
  }
  if (port != -1 && url.getPort() != port) {
    return false;
  }
  return true;
  }
}
```

Метод isValid реализует алгоритм валидации URL, показанный в листинге 3.8. Параметр value содержит значение объекта, который требуется валидировать (например, file://www.cdbookstore.com/item/123). Параметр context инкапсулирует информацию о контексте, в котором осуществляется валидация (подробнее об этом ниже). Возвращаемое значение является логическим и указывает, успешно ли прошла валидация.

Основная задача валидационного алгоритма в листинге 3.8 — привести переданное значение к java.net.URL и проверить, правильно ли оформлен URL. После этого алгоритм также проверяет валидность атрибутов protocol, host и port. Если хотя бы один из них окажется невалидным, то метод вернет false. Как будет показано далее в разделе «Валидация ограничений», поставщик валидации компонентов задействует это логическое значение при создании списка ConstraintViolation.

Обратите внимание: метод isValid расценивает нуль как валидное значение (if (value == null ... return true)). Спецификация Bean Validation считает такую практику рекомендуемой. Так удается не дублировать код ограничения @NotNull. Пришлось бы одновременно использовать ограничения @URL и @NotNull, чтобы указать, что вы хотите представить валидный ненулевой URL (такой как атрибут itemURL в листинге 3.6).

Сигнатура класса определяет тип данных, с которым ассоциируется ограничение. В листинге 3.8 URLValidator реализован для типа String (ConstraintValidator<URL, String>). Это означает, что если вы примените ограничение @URL к другому типу (например, к атрибуту lastConnectionDate), то получите при валидации исключение javax.validation.UnexpectedTypeException, так как не будет найден подходящий валидатор для типа java.util.Date. Если вам требуется ограничение, которое будет применяться сразу к нескольким типам данных, то необходимо либо использовать суперклассы, когда это возможно (скажем, можно было бы определить URLValidator для CharSequence, а не для строки, выразив его так: ConstraintValidator<URL, CharSequence>), либо применить несколько классов реализации (по одному для String, CharBuffer, StringBuffer, StringBuilder) в случае иного валидационного алгоритма.

ПРИМЕЧАНИЕ

Реализация ограничения считается управляемым компонентом. Это означает, что с ней вы можете использовать все сервисы, доступные для обработки управляемых компонентов — в частности, внедрение любого вспомогательного класса, EJB или даже EntityManager (подробнее об этом — в следующих главах). Вы также можете перехватывать или декорировать методы initialize и isValid и даже задействовать управление жизненным циклом (@PostConstruct и @PreDestroy).

Множественные ограничения для одной целевой сущности

Иногда полезно применять одинаковое ограничение к одной и той же цели, используя при этом разные свойства или группы (подробнее об этом ниже). Распространенный пример такого рода — ограничение @Pattern, проверяющее соответствие целевой сущности определенному регулярному выражению. В листинге 3.9 показано, как применить два регулярных выражения к одному и тому же атрибуту. Множественные

ограничения используют оператор AND. Это означает, что для валидности атрибута orderId необходимо, чтобы он удовлетворял двум регулярным выражениям.

Листинг 3.9. Объект POJO, в котором несколько шаблонных ограничений применяются к одному и тому же атрибуту

```
public class Order {
  @Pattern.List({
      @Pattern(regexp = "[C,D,M][A-Z][0-9]*"),
      @Pattern(regexp = ".[A-Z].*?")
  })
  private String orderId;
  private Date creationDate;
  private Double totalAmount;
  private Date paymentDate;
  private Date deliveryDate;
  private List<OrderLine> orderLines;
  // Конструкторы, геттеры, сеттеры
}
```

Чтобы иметь возможность несколько раз применить одно и то же ограничение к данной цели, ограничивающая аннотация должна определить массив на основе самой себя. При валидации компонентов такие массивы ограничений обрабатываются по-особому: каждый элемент массива интерпретируется как обычное ограничение. В листинге 3.10 показана ограничивающая аннотация @Pattern, определяющая внутренний интерфейс (произвольно названный List) с элементом Pattern[]. Внутренний интерфейс должен иметь правило хранения RUNTIME и использовать в качестве исходного ограничения один и тот же набор целей (в данном случае METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER).

Листинг 3.10. Шаблонное ограничение, определяющее список шаблонов

```
@Target({METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER})
@Retention(RUNTIME)
@Constraint(validatedBy = PatternValidator.class)
public @interface Pattern {
  String regexp();
  String message() default "{javax.validation.constraints.Pattern.message}";
  Class<?>[] groups() default {};
  Class<? extends Payload>[] payload() default {};
  // Определяет несколько аннотаций @Pattern, применяемых к одному элементу
  @Target({METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER})
  @Retention(RUNTIME)
  @interface List {
    Pattern[] value();
  }
}
```

ПРИМЕЧАНИЕ

При разработке собственной ограничивающей аннотации следует добавить соответствующую ей аннотацию с множеством значений. Спецификация Bean Validation не требует этого строго, но настоятельно рекомендует определять внутренний интерфейс под названием List.

Ограничения на уровне класса

Выше мы рассмотрели различные способы разработки ограничения, которое применялось бы к атрибуту (или геттеру). Но вы также можете создать ограничение для целого класса. Идея заключается в том, чтобы выразить ограничение на основе нескольких свойств, которыми обладает заданный класс.

В листинге 3.11 показан класс для оформления заказа. Этот заказ товара следует определенному жизненному циклу бизнес-логики: создается в системе, оплачивается клиентом, а потом доставляется клиенту. Класс отслеживает все эти события, оперируя соответствующими датами: `creationDate`, `paymentDate` и `deliveryDate`. Аннотация `@ChronologicalDates` действует на уровне класса и проверяет, находятся ли эти даты в правильном хронологическом порядке.

Листинг 3.11. Ограничение, действующее на уровне класса и проверяющее верность хронологической последовательности дат

```
@ChronologicalDates
public class Order {
  private String orderId;
  private Double totalAmount;
  private Date creationDate;
  private Date paymentDate;
  private Date deliveryDate;
  private List<OrderLine> orderLines;
  // Конструкторы, геттеры, сеттеры
}
```

В листинге 3.12 показана реализация ограничения `@ChronologicalDates`. Подобно тем ограничениям, что были рассмотрены выше, оно реализует интерфейс `ConstraintValidator`, обобщенный тип которого — `Order`. Метод `isValid` проверяет, находятся ли три даты в правильном хронологическом порядке, и если это так — возвращает `true`.

Листинг 3.12. Реализация ограничения ChronologicalDates, действующего на уровне класса

```
public class ChronologicalDatesValidator implements ➥
ConstraintValidator<ChronologicalDates, Order> {
  @Override
  public void initialize(ChronologicalDates constraintAnnotation) {
  }
  @Override
  public boolean isValid(Order order, ConstraintValidatorContext context) {
    return order.getCreationDate().getTime() < ➥
           order.getPaymentDate().getTime() && ➥
           order.getPaymentDate().getTime() < ➥
           order.getDeliveryDate().getTime();
  }
}
```

Ограничение на уровне метода

Ограничения, действующие на уровне методов, появились в спецификации Bean Validation 1.1. Существуют ограничения, объявляемые для методов, а также для

конструкторов (геттеры не считаются ограниченными методами). Эти ограничения могут быть добавлены к параметрам метода (это будут «ограничения параметров») или к самому методу («ограничения возвращаемых значений»). Таким образом, спецификация Bean Validation может использоваться для описания и валидации соглашения, применяемого с заданным методом или конструктором. Так строится хорошо известный стиль «Программирование по соглашениям»:

- предусловия должны выполняться вызывающей стороной еще до вызова метода или конструктора;
- постусловия гарантированно выполняются для вызывающей стороны после возврата вызова, направленного к методу или конструктору.

В листинге 3.13 показано несколько способов использования ограничений на уровне метода. Сервис `CardValidator` проверяет кредитную карту в соответствии с конкретным валидационным алгоритмом. Для этого конструктор использует ограничение `@NotNull` с параметром `ValidationAlgorithm`. Затем два метода `validate` возвращают `Boolean` (валидна кредитная карта или нет?) с ограничением `@AssertTrue` для возвращаемого типа и ограничениями `@NotNull` и `@Future` у параметров методов.

Листинг 3.13. Сервис с конструктором и ограничениями на уровне метода

```
public class CardValidator {
  private ValidationAlgorithm validationAlgorithm;
  public CardValidator(@NotNull ValidationAlgorithm validationAlgorithm) {
    this.validationAlgorithm = validationAlgorithm;
  }
  @AssertTrue
  public Boolean validate(@NotNull CreditCard creditCard) {
    return validationAlgorithm.validate(creditCard.getNumber(),
creditCard.getCtrlNumber());
  }
  @AssertTrue
  public Boolean validate(@NotNull String number, @Future Date expiryDate,
  Integer controlNumber, String type) {
    return validationAlgorithm.validate(number, controlNumber);
  }
}
```

Наследование ограничений

Часто в бизнес-модели действует механизм наследования. При использовании валидации компонентов приходится накладывать ограничения на классы, суперклассы или интерфейсы вашей бизнес-модели. Наследование ограничений у свойств работает точно так же, как и обычное наследование в Java: оно является кумулятивным. Таким образом, если один компонент наследует от другого, то ограничения наследуемого компонента также заимствуются и будут валидироваться.

В листинге 3.15 показан класс `CD`, расширяющий `Item` (листинг 3.14). Оба класса имеют атрибуты и применяемые с ними ограничения. Если валидируется экземпляр `CD`, то валидируются не только его ограничения, но и ограничения, налагаемые на родительский класс.

Листинг 3.14. Суперкласс Item, использующий ограничения

```
Public class Item {
  @NotNull
  protected Long id;
  @NotNull @Size(min = 4, max = 50)
  protected String title;
  protected Float price;
  protected String description;
  @NotNull
  public Float calculateVAT() {
    return price * 0.196f;
  }
  @NotNull
  public Float calculatePrice(@DecimalMin("1.2") Float rate) {
    return price * rate;
  }
}
```

Листинг 3.15. Класс CD, расширяющий Item

```
public class CD extends Item {
  @Pattern(regexp = "[A-Z][a-z]{1,}")
  private String musicCompany;
  @Max(value = 5)
  private Integer numberOfCDs;
  private Float totalDuration;
  @MusicGenre
  private String genre;
  // ConstraintDeclarationException : не допускается при переопределении метода
  public Float calculatePrice(@DecimalMin("1.4") Float rate) {
    return price * rate;
  }
}
```

Такой же механизм наследования применяется и с ограничениями, действующими на уровне методов. Метод `calculateVAT`, объявляемый в `Item`, наследуется в `CD`. Но в случае переопределения метода нужно уделять особое внимание при определении ограничений для параметров. Лишь корневой метод переопределяемого метода может аннотироваться ограничениями параметров. Причина такого условия состоит в том, что предусловия нельзя ужесточать для подтипов. Напротив, в подтипах можно добавлять ограничения для возвращаемых значений в любом количестве (постусловия можно ужесточать).

Итак, если вы валидируете `calculatePrice` класса `CD` (см. листинг 3.15), среда исполнения валидации компонентов будет выдавать исключение `javax.validation.ConstraintDeclarationException`. Оно означает, что только корневой метод переопределенного метода может использовать ограничения параметров.

Сообщения

Как было показано выше (см. листинг 3.2), при определении ограничивающей аннотации есть три обязательных атрибута: `message`, `groups` и `payload`. Каждое ограничение

должно определять задаваемое по умолчанию сообщение типа String, которое используется для индикации ошибки, если при валидации компонента нарушается то или иное ограничение.

Значение такого стандартного сообщения можно жестко запрограммировать, но рекомендуется применять ключ пакета ресурсов для обеспечения интернационализации. В соответствии с действующим соглашением ключ пакета ресурсов должен быть полностью квалифицированным именем класса той ограничивающей аннотации, которая сцепляется с .message.

```
// Жестко закодированное сообщение об ошибке
String message() default "Неверный электронный адрес";
// Ключ пакета ресурсов
String message() default "{org.agoncal.book.javaee7.Email.message}";
```

По умолчанию файл пакета ресурсов называется ValidationMessages.properties и должен быть указан в пути к классам приложения. Файл построен в виде пар «ключ — значение», именно это нам и нужно для экстернализации и интернационализации сообщения об ошибке.

```
org.agoncal.book.javaee7.Email.message=Неверный электронный адрес
```

Это стандартное сообщение, заданное в ограничивающей аннотации, может быть переопределено во время объявления в зависимости от конкретных условий использования.

```
@Email(message = "Восстановленный электронный адрес не является действительным")
private String recoveryEmail;
```

Благодаря интерполяции сообщений (интерфейс javax.validation.MessageInterpolator) сообщение об ошибке может содержать джокерные элементы. Цель интерполяции — определить сообщение об ошибке, разрешая его строки и параметры, находящиеся в скобках. Следующее сообщение об ошибке интерполировано таким образом, что джокерные строки {min} и {max} заменяются значениями соответствующих элементов:

```
javax.validation.constraints.Size.message = size must be between {min} and {max}
```

В листинге 3.16 показан класс Customer, использующий сообщения об ошибках несколькими способами. Атрибут userId имеет аннотацию @Email, говорящую о том, что если значение не является действительным адресом электронной почты, то будет использоваться заданное по умолчанию сообщение об ошибке. Обратите внимание: с атрибутами firstName и age стандартные сообщения об ошибках переопределяются, вместо них используются варианты с джокерными последовательностями.

Листинг 3.16. Класс Customer, определяющий несколько сообщений об ошибках

```
public class Customer {
  @Email
  private String userId;
  @NotNull @Size(min = 4, max = 50, message = "Имя должно быть размером от
{min} до {max} символов")
```

```
  private String firstName;
  private String lastName;
  @Email(message = "Восстановленный электронный адрес не является действительным")
  private String recoveryEmail;
  private String phoneNumber;
  @Min(value = 18, message = "Покупатель слишком молод. Ему должно быть больше
{value} лет")
  Private Integer age;
  // Конструкторы, геттеры, сеттеры
}
```

Контекст ConstraintValidator

Итак, мы убедились, что классы реализации ограничений должны реализовывать `ConstraintValidator` и, следовательно, определять собственный метод `isValid`. Сигнатура метода `isValid` принимает тип данных, к которому применяется ограничение, а также `ConstraintValidationContext`. Этот интерфейс инкапсулирует данные, относящиеся к тому контексту, в котором поставщик выполняет валидацию компонентов. В табл. 3.3 перечислены методы, определяемые в интерфейсе `javax.validation.ConstraintValidatorContext`.

Таблица 3.3. Методы интерфейса ConstraintValidationContext

Метод	Описание
disableDefaultConstraintViolation	Отменяет генерацию задаваемого по умолчанию объекта ConstraintViolation
getDefaultConstraintMessageTemplate	Возвращает актуальное сообщение, заданное по умолчанию, без интерполяции
buildConstraintViolationWithTemplate	Возвращает ConstraintViolationBuilder, обеспечивающий построение пользовательского отчета о нарушении ограничения

Интерфейс `ConstraintValidatorContext` позволяет повторно определить заданное по умолчанию сообщение, связанное с ограничением. Метод `buildConstraintViolationWithTemplate` возвращает `ConstraintViolationBuilder`, опираясь на гибкий образец API, позволяющий создавать пользовательские отчеты о нарушениях. Следующий код добавляет к отчету информацию о новом нарушении ограничения:

```
context.buildConstraintViolationWithTemplate("Invalid protocol")
       .addConstraintViolation();
```

Такая техника позволяет генерировать и создавать одно или несколько пользовательских сообщений-отчетов. Если рассмотреть пример с ограничением `@URL` из листинга 3.7, то мы увидим, что всему ограничению здесь соответствует лишь одно сообщение об ошибке (Malformed URL). Но у этого ограничения несколько атрибутов (`protocol`, `host` и `port`), и нам могут понадобиться специфичные сообщения для каждого из этих атрибутов, например: `Invalid protocol` (Недопустимый протокол) или `Invalid host` (Недопустимый хост).

ПРИМЕЧАНИЕ

Интерфейс ConstraintViolation описывает нарушение ограничения. Он предоставляет контекст, в котором произошло нарушение, а также сообщение, описывающее это нарушение. Подробнее об этом читайте в разделе «Валидация ограничений» данной главы.

В листинге 3.17 мы вновь рассмотрим класс реализации ограничения URL и воспользуемся ConstraintValidatorContext, чтобы изменить сообщение об ошибке. Код полностью отключает генерацию заданного по умолчанию сообщения об ошибке (disableDefaultConstraintViolation) и отдельно определяет сообщения об ошибках для каждого атрибута.

Листинг 3.17. Ограничение URL, использующее ConstraintValidatorContext для настройки сообщений об ошибках

```
public class URLValidator implements ConstraintValidator<URL, String> {
  private String protocol;
  private String host;
  private int port;
  public void initialize(URL url) {
    this.protocol = url.protocol();
    this.host = url.host();
    this.port = url.port();
  }
  public boolean isValid(String value, ConstraintValidatorContext context) {
    if (value == null || value.length() == 0) {
      return true;
    }
    java.net.URL url;
    try {
      url = new java.net.URL(value);
    } catch (MalformedURLException e) {
      return false;
    }
    if (protocol != null && protocol.length() > 0 &&
!url.getProtocol().equals(protocol)) {
      context.disableDefaultConstraintViolation();
      context.buildConstraintViolationWithTemplate("Неверный
протокол").addConstraintViolation();
      return false;
    }
    if (host != null && host.length() > 0 && !url.getHost().startsWith(host)) {
      context.disableDefaultConstraintViolation();
      context.buildConstraintViolationWithTemplate("Неверный
хост").addConstraintViolation();
      return false;
    }
    if (port != -1 && url.getPort() != port) {
      context.disableDefaultConstraintViolation();
      context.buildConstraintViolationWithTemplate("Неверный
порт").addConstraintViolation();
      return false;
```

```
        }
        return true;
    }
}
```

Группы

Когда компонент валидируется, в ходе проверки одновременно проверяются все его ограничения. Но что, если вам требуется частично валидировать компонент (проверить часть его ограничений) или управлять порядком валидации конкретных ограничений? Здесь нам пригодятся группы. Группы позволяют сформировать набор тех ограничений, которые будут проверяться в ходе валидации.

На уровне кода группа представляет собой обычный пустой интерфейс.

```
public interface Payment {}
```

На уровне бизнес-логики группа имеет определенное значение. Например, рабочая последовательность `Payment` (Платеж) подсказывает, что атрибуты, относящиеся к этой группе, будут валидироваться на этапе оплаты в рамках заказа товара. Чтобы применить эту группу с набором ограничений, нужно использовать атрибут `groups` и передать ему этот интерфейс:

```
@Past(groups = Payment.class)
private Date paymentDate;
```

Вы можете иметь столько групп, сколько требует ваша бизнес-логика, а также использовать то или иное ограничение с несколькими группами, поскольку атрибут `groups` может принимать целый массив групп:

```
@Past(groups = {Payment.class, Delivery.class})
private Date deliveryDate;
```

В каждой ограничивающей аннотации должен определяться элемент `groups`. Если ни одна группа не указана, то считается объявленной задаваемая по умолчанию группа `javax.validation.groups.Default`. Так, следующие ограничения являются эквивалентными и входят в состав группы `Default`:

```
@NotNull
private Long id;
@Past(groups = Default.class)
private Date creationDate;
```

Вновь рассмотрим случай с предшествующим использованием, в котором мы применяли аннотацию `@ChronologicalDates`, и задействуем в нем группы. В классе `Order` из листинга 3.18 содержится несколько дат, позволяющих отслеживать ход процесса заказа: `creationDate`, `paymentDate` и `deliveryDate`. Когда вы только создаете заказ на покупку, устанавливается атрибут `creationDate`, но не `paymentDate` и `deliveryDate`. Две эти даты нам потребуется валидировать позже, на другом этапе рабочего процесса, но не одновременно с `creationDate`. Применяя группы, можно валидировать `creationDate` одновременно с группой, проверяемой по умолчанию (поскольку для этой аннотации не указана никакая группа, для нее по умолчанию действует `javax.validation.groups.Default`). Атрибут `paymentDate`

будет валидироваться на этапе Payment, а deliveryDate и @ChronologicalDates — на этапе Delivery.

Листинг 3.18. Класс, использующий несколько групп

```
@ChronologicalDates(groups = Delivery.class)
public class Order {
  @NotNull
  private Long id;
  @NotNull @Past
  private Date creationDate;
  private Double totalAmount;
  @NotNull(groups = Payment.class) @Past(groups = Payment.class)
  private Date paymentDate;
  @NotNull(groups = Delivery.class) @Past(groups = Delivery.class)
  private Date deliveryDate;
  private List<OrderLine> orderLines;
  // Конструкторы, геттеры, сеттеры
}
```

Как видите, в ходе валидации вам всего лишь следует явно указать, какие именно группы вы хотите проверить, и поставщик валидации из Bean Validation выполнит частичную проверку.

Дескрипторы развертывания

Как и большинство других технологий, входящих в состав Java EE 7, при валидации компонентов мы можем определять метаданные с помощью аннотаций (как это делалось выше) и с помощью XML. В Bean Validation у нас может быть несколько опциональных файлов, которые располагаются в каталоге META-INF. Первый файл, validation.xml, может применяться приложениями для уточнения некоторых особенностей поведения валидации компонентов (в частности, поведения действующего по умолчанию поставщика валидации компонентов, интерполятора сообщений, а также конкретных свойств). Кроме того, у вас может быть несколько файлов, описывающих объявления ограничений для ваших компонентов. Как и все дескрипторы развертывания в Java EE 7, XML переопределяет аннотации.

В листинге 3.19 показан дескриптор развертывания validation.xml, имеющий корневой XML-элемент validation-config. Гораздо важнее, что здесь определяется внешний файл для отображения ограничений: constraints.xml (показан в листинге 3.20).

Листинг 3.19. Файл validation.xml, в котором определяется файл для отображения ограничений

```
<?xml version="1.0" encoding="UTF-8"?>
<validation-config
    xmlns="http://jboss.org/xml/ns/javax/validation/configuration"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/configuration
                        validation-configuration-1.1.xsd"
    version="1.1">
  <constraint-mapping>META-INF/constraints.xml</constraint-mapping>
</validation-config>
```

Листинг 3.20. Файл, в котором определяются ограничения для компонента

```
<?xml version="1.0" encoding="UTF-8"?>
<constraint-mappings
    xmlns="http://jboss.org/xml/ns/javax/validation/mapping"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/mapping
                        validation-mapping-1.1.xsd"
    version="1.1">
  <bean class="org.agoncal.book.javaee7.chapter03.Book" ignore-annotations="false">
    <field name="title">
      <constraint annotation="javax.validation.constraints.NotNull">
        <message>Title should not be null</message>
      </constraint>
    </field>
    <field name="price">
      <constraint annotation="javax.validation.constraints.NotNull"/>
      <constraint annotation="javax.validation.constraints.Min">
        <element name="value">2</element>
      </constraint>
    </field>
    <field name="description">
      <constraint annotation="javax.validation.constraints.Size">
        <element name="max">2000</element>
      </constraint>
    </field>
  </bean>
</constraint-mappings>
```

Файл `constraints.xml` из листинга 3.20 определяет метаданные для объявления ограничений, используемых с классом `Book`. Сначала он применяет ограничение `@NotNull` к атрибуту `title` и заново задает выводимое по умолчанию сообщение об ошибке («Название не может быть пустым»). К атрибуту `price` применяются два разных ограничения, его минимальное значение равно `2`. Ситуация напоминает код из листинга 3.1, где метаданные определялись с помощью аннотаций.

Валидация ограничений

До сих пор мы плотно работали с ограничениями — определяли их, агрегировали, реализовывали наши собственные, настраивали сообщения об ошибках, возились с группами. Но без специальной валидационной среды исполнения проверка ограничений невозможна. Как и с большинством технологий Java EE 7, код в данном случае должен работать внутри контейнера или управляться поставщиком.

Ограничения можно применять к компонентам, атрибутам, геттерам, конструкторам, параметрам методов и возвращаемым значениям. Итак, валидация может выполняться для элементов всех этих типов. Можно валидировать компоненты, свойства, значения, методы и группы, а также задавать собственные ограничения для графа объектов. Чтобы все эти ограничения проверялись во время исполнения, вам потребуется валидационный API.

Валидационные API

Среда исполнения валидации использует небольшой набор API, которые позволяют ей проверять ограничения. Основной API — это интерфейс `javax.validation.Validator`. Он содержит соглашения для валидации объектов и графов объектов независимо от уровня, на котором этот интерфейс реализован (уровень представления, уровень бизнес-логики или бизнес-модели). При ошибке валидации возвращается набор интерфейсов `javax.validation.ConstraintViolation`. Он предоставляет контекст произошедшего нарушения, а также сообщение, описывающее нарушение.

Валидатор

Основной входной точкой для валидации является интерфейс `Validator`. Этот API позволяет проверять экземпляры компонентов, обходясь немногочисленными методами, перечисленными в табл. 3.4. Все эти методы объявляют каждое новое ограничение по одному и тому же образцу.

1. Устанавливается подходящая реализация `ConstraintValidator`, которая будет использоваться при определении данного ограничения (например, определяется `ConstraintValidator` для ограничения `@Size`, применяемого со строкой).
2. Выполняется метод `isValid`.
3. Если `isValid` возвращает `true`, программа переходит к следующему ограничению.
4. Если `isValid` возвращает `false`, поставщик валидации компонентов добавляет `ConstraintViolation` в список нарушений ограничений.

Таблица 3.4. Методы интерфейса Validator

Метод	Описание
<T> Set<ConstraintViolation<T>>**validate**(T object, Class<?>... groups)	Валидирует все ограничения, применяемые с объектом
<T> Set<ConstraintViolation<T>>**validateProperty**(T object, String propName, Class<?>... groups)	Валидирует все ограничения, касающиеся свойства
<T> Set<ConstraintViolation<T>>**validateValue**(Class<T> beanType, String propName, Object value,Class<?>... groups)	Валидирует все ограничения, применяемые к свойству при заданном значении
BeanDescriptor getConstraintsForClass(Class<?> clazz)	Возвращает объект дескриптора, описывающий ограничения компонента
ExecutableValidator **forExecutables**()	Возвращает делегат для валидации параметров и возвращаемых значений у методов и конструкторов

Если в ходе этой процедуры валидации происходит какая-либо неисправимая ошибка, то выдается исключение `ValidationException`. Это исключение может быть специализированным и может указывать на конкретные ситуации (недопустимое определение группы, недопустимое определение ограничения, недопустимое объявление ограничения).

Методы `validate`, `validateProperty` и `validateValue` используются соответственно для валидации целого компонента, свойства или свойства при заданном значении. Все методы принимают параметр `varargs`, позволяющий указывать группы для валидации. Метод `forExecutables` предоставляет доступ к `ExecutableValidator` для валидации методов, параметров конструктора и валидации возвращаемого значения. В табл. 3.5 описан API `ExecutableValidator`.

Таблица 3.5. Методы для интерфейса ExecutableValidator

Метод	Описание
<T> Set<ConstraintViolation<T>>**validateParameters**(T object, Method method, Object[] params, Class<?>... groups)	Валидирует все ограничения, применяемые с параметрами метода
<T> Set<ConstraintViolation<T>>**validateReturnValue**(T object, Method method, Object returnValue, Class<?>... groups)	Валидирует все ограничения возвращаемого значения, применяемые с методом
<T> Set<ConstraintViolation<T>>**validateConstructorParameters** (Constructor<T> constructor, Object[] params, Class<?>... groups)	Валидирует все ограничения, связанные с параметрами конструктора
<T> Set<ConstraintViolation<T>>**validateConstructorReturnValue** (Constructor<T> constructor, T createdObject, Class<?>... groups)	Валидирует все ограничения возвращаемых значений, связанные с конструктором

ConstraintViolation

Все валидационные методы, перечисленные в табл. 3.4 и 3.5, возвращают множество `ConstraintViolation`, которое можно перебирать и при этом просматривать, какие ошибки возникли при валидации. Если это множество пустое — значит, валидация прошла успешно. При ошибках в это множество добавляется по экземпляру `ConstraintViolation` для каждого нарушенного ограничения. `ConstraintViolation` описывает единую ошибку, связанную с ограничением, а его API дает множество полезной информации о причине ошибки. В табл. 3.6 сделан обзор этого API.

Таблица 3.6. Методы интерфейса ConstraintViolation

Метод	Описание
String getMessage()	Возвращает интерполированное сообщение об ошибке для данного нарушения
String getMessageTemplate()	Возвращает неинтерполированное сообщение об ошибке
T getRootBean()	Возвращает корневой компонент, участвующий в валидации
Class<T> getRootBeanClass()	Возвращает класс корневого компонента, участвующего в валидации

Продолжение ⇩

Таблица 3.6 *(продолжение)*

Метод	Описание
Object getLeafBean()	Возвращает дочерний компонент, к которому применяется ограничение
Path getPropertyPath()	Возвращает путь свойств к значению от корневого компонента
Object getInvalidValue()	Возвращает значение, которое не соответствует заданному ограничению
ConstraintDescriptor<?> getConstraintDescriptor()	Возвращает метаданные об ограничении

Получение валидатора

Первый этап валидации компонента — приобрести экземпляр интерфейса `Validator`. Как и в большинстве спецификаций Java EE, вы можете либо получить `Validator` программно (если ваш код выполняется вне контейнера), либо внедрить его (если код выполняется в EJB- или веб-контейнере).

Если вы делаете это программно, то необходимо начать с класса `Validation`, осуществляющего начальную загрузку поставщика валидации компонентов. Его метод `buildDefaultValidatorFactory` строит и возвращает фабрику `ValidatorFactory`, которая, в свою очередь, используется для построения `Validator`. Код выглядит следующим образом:

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();
```

Затем вам придется управлять жизненным циклом `ValidatorFactory` и программно закрывать его:

```
factory.close();
```

Если ваше приложение работает в контейнере Java EE, то контейнер должен предоставлять через JNDI следующие экземпляры:

- `ValidatorFactory` под `java:comp/ValidatorFactory`;
- `Validator` под `java:comp/Validator`.

Затем вы можете искать эти JNDI-имена и получать экземпляры `Validator`. Но вместо поиска экземпляров через JNDI вы можете запросить, чтобы они были внедрены с помощью аннотации `@Resource`:

```
@Resource ValidatorFactory validatorFactory;
@Resource Validator validator;
```

Если ваш контейнер использует CDI (а это происходит в Java EE 7 по умолчанию), то он должен допускать внедрение с помощью `@Inject`:

```
@Inject ValidatorFactory;
@Inject Validator;
```

В любом случае (как с `@Resource`, так и с `@Inject`) контейнер следит за жизненным циклом фабрики. Поэтому вам не придется вручную создавать или закрывать `ValidatorFactory`.

Валидация компонентов

Как только вы получите `Validator` программно или путем внедрения, можете использовать его методы как для валидации целого компонента, так и для работы с отдельно взятым свойством. В листинге 3.21 показан класс `CD`, в котором действуют ограничения, связанные со свойствами, параметрами методов и возвращаемым значением.

Листинг 3.21. Компонент с ограничениями, применяемыми со свойствами и методами

```
public class CD {
  @NotNull @Size(min = 4, max = 50)
  private String title;
  @NotNull
  private Float price;
  @NotNull(groups = PrintingCatalog.class)
  @Size(min = 100, max = 5000, groups = PrintingCatalog.class)
  private String description;
  @Pattern(regexp = "[A-Z][a-z]{1,}")
  private String musicCompany;
  @Max(value = 5)
  private Integer numberOfCDs;
  private Float totalDuration;
  @NotNull @DecimalMin("5.8")
  public Float calculatePrice(@DecimalMin("1.4") Float rate) {
    return price * rate;
  }
  @DecimalMin("9.99")
  public Float calculateVAT() {
    return price * 0.196f;
  }
}
```

Чтобы валидировать свойства целого компонента, мы просто должны создать экземпляр `CD` и вызвать метод `Validator.validate()`. Если экземпляр валиден, то возвращается пустое множество `ConstraintViolation`. В следующем коде показан валидный экземпляр `CD` (с заголовком и ценой), который и проверяется. После этого код удостоверяет, что множество ограничений действительно является пустым.

```
CD cd = new CD("Kind of Blue", 12.5f);
Set<ConstraintViolation<CD>> violations = validator.validate(cd);
assertEquals(0, violations.size());
```

С другой стороны, следующий код вернет два объекта `ConstraintViolation` — один будет соответствовать заголовку, а другой — цене (оба они нарушают `@NotNull`):

```
CD cd = new CD();
Set<ConstraintViolation<CD>> violations = validator.validate(cd);
assertEquals(2, violations.size());
```

Валидация свойств

В предыдущих примерах валидируются свойства целого компонента. Но существует метод `Validator.validateProperty()`, позволяющий проверять конкретное именованное свойство заданного объекта.

В показанном ниже коде создается `CD` — объект, имеющий нулевой заголовок и нулевую цену; соответственно, такой компонент невалиден. Но, поскольку мы проверяем только свойство `numberOfCDs`, валидация проходит успешно и мы получаем пустое множество нарушений ограничений:

```
CD cd = new CD();
cd.setNumberOfCDs(2);
Set<ConstraintViolation<CD>> violations = validator.validateProperty(cd,
"numberOfCDs");
assertEquals(0, violations.size());
```

С другой стороны, в следующем коде возникает нарушение ограничения, так как максимальное количество объектов `CD` должно равняться пяти, а не семи. Обратите внимание: мы используем API `ConstraintViolation` для проверки количества нарушений, интерполированного сообщения, возвращенного при нарушении, невалидного значения и шаблона сообщения:

```
CD cd = new CD();
cd.setNumberOfCDs(7);
Set<ConstraintViolation<CD>> violations = validator.validateProperty(cd,
"numberOfCDs");
assertEquals(1, violations.size());
assertEquals("must be less than or equal to 5", violations.iterator().next().
getMessage());
assertEquals(7, violations.iterator().next().getInvalidValue());
assertEquals("{javax.validation.constraints.Max.message}",
             violations.iterator().next().getMessageTemplate());
```

Валидация значений

Пользуясь методом `Validator.validateValue()`, можно проверять, удастся ли успешно валидировать конкретное свойство указанного класса при наличии у свойства заданного значения. Этот метод удобен при опережающей валидации, так как не требует даже создавать экземпляр компонента, заполнять или обновлять его значения.

Следующий код не создает объект `CD`, а просто ссылается на атрибут `numberOfCDs` класса `CD`. Он передает значение и проверяет, является ли свойство валидным (количество `CD` не должно превышать пяти):

```
Set<ConstraintViolation<CD>> constr = validator.validateValue(CD.class,
"numberOfCDs", 2);
assertEquals(0, constr.size());
Set<ConstraintViolation<CD>> constr = validator.validateValue(CD.class,
"numberOfCDs", 7);
assertEquals(1, constr.size());
```

Валидация методов

Методы для валидации параметров и возвращаемых значений методов и конструкторов вы найдете в интерфейсе javax.validation.ExecutableValidator. Метод Validator.forExecutables() возвращает этот ExecutableValidator, с которым вы можете вызывать validateParameters, validateReturnValue, validateConstructorParameters или validateConstructorReturnValue.

В следующем коде мы вызываем метод calculatePrice, передавая значение 1.2. В результате возникает нарушение ограничения, налагаемого на параметр: не выполняется условие @DecimalMin("1.4"). Для этого в коде сначала нужно создать объект java.lang.reflect.Method, нацеленный на метод calculatePrice с параметром типа Float. После этого он получает объект ExecutableValidator и вызывает validateParameters, передавая компонент, метод для вызова и значение параметра (здесь — 1.2). Затем метод удостоверяется в том, что никакие ограничения нарушены не были.

```
CD cd = new CD("Kind of Blue", 12.5f);
Method method = CD.class.getMethod("calculatePrice", Float.class);
ExecutableValidator methodValidator = validator. forExecutables();
Set<ConstraintViolation<CD>> violations = methodValidator.
validateParameters(cd, method,
                 new Object[]{new Float(1.2)});
assertEquals(1, violations.size());
```

Валидация групп

Группа определяет подмножество ограничений. Вместо валидации всех ограничений для данного компонента проверяется только нужное подмножество. При объявлении каждого ограничения указывается список групп, в которые входит это ограничение. Если явно не объявлена ни одна группа, то ограничение относится к группе Default. Что касается проверки, у всех валидационных методов есть параметр с переменным количеством аргументов, указывающий, сколько групп должно учитываться при выполнении валидации. Если этот параметр не задан, будет использоваться указанная по умолчанию валидационная группа (javax.validation.groups.Default). Если вместо Default указана другая группа, то Default не валидируется.

В листинге 3.21 все ограничения, за исключением применяемых с атрибутом description, относятся к группе Default. Описание (@NotNull @Size(min = 100, max = 5000)) требуется лишь в том случае, если диск должен быть упомянут в каталоге (группа PrintingCatalog). Итак, если мы создаем CD без заголовка, цены и описания, после чего проверяем лишь условия из группы Default, то будут нарушены всего два ограничения @NotNull, касающиеся title и price.

```
CD cd = new CD();
cd.setDescription("Best Jazz CD ever");
Set<ConstraintViolation<CD>> violations = validator.validate(cd, Default.
class);
assertEquals(2, violations.size());
```

Обратите внимание: в предыдущем коде при валидации явно упоминается группа Default, но это слово можно пропустить. Итак, следующий код идентичен предыдущему:

```
Set<ConstraintViolation<CD>> violations = validator. validate(cd);
```

С другой стороны, если бы мы решили проверить CD только для группы PrintingCatalog, то следующий код нарушал бы лишь ограничение, налагаемое на description, так как предоставляемое значение было бы слишком коротким:

```
CD cd = new CD();
cd.setDescription("Too short");
Set<ConstraintViolation<CD>> violations = validator.validate(cd,
PrintingCatalog.class);
assertEquals(1, violations.size());
```

Если бы вы хотели проверить компонент на соответствие обеим группам — Default и PrintingCatalog, то у вас было бы нарушено три ограничения (@NotNull для title и price, а также очень краткое описание):

```
CD cd = new CD();
cd.setDescription("Too short");
Set<ConstraintViolation<CD>> violations = validator.validate(cd, Default.
class, PrintingCatalog.class);
assertEquals(3, violations.size());
```

Все вместе

Теперь рассмотрим все изученные концепции вместе и напишем Java-компоненты, с которыми сможем использовать встроенные ограничения, а также разработать наше собственное. В этом примере применяются CDI и ограничения валидации компонентов на основе Java SE (пока не требуется что-либо развертывать в GlassFish). Кроме того, выполняется два интеграционных теста, проверяющих правильность использованных ограничений.

На рис. 3.2 показан класс Customer, имеющий адрес доставки (Address). Оба компонента снабжены встроенными ограничениями (@NotNull, @Size и @Past), которые применяются с их атрибутами. Но нам остается разработать еще два собственных ограничения:

- @Email — агрегированное ограничение, проверяющее правильность адреса электронной почты;
- @ZipCode — ограничение, проверяющее правильность почтового ZIP-кода (для США). В состав ограничения входит как аннотация, так и класс реализации (ZipCodeValidator). Обратите внимание: ZipCodeValidator внедряет вспомогательный класс ZipCodeChecker с аннотацией @Inject (и CDI-квалификатором @USA).

Классы, описанные на рис. 3.2, построены в соответствии со структурой каталогов Maven и должны находиться в следующих папках и файлах:

- src/main/java — папка для компонентов Customer, Address и ограничений ZipCode и Email;

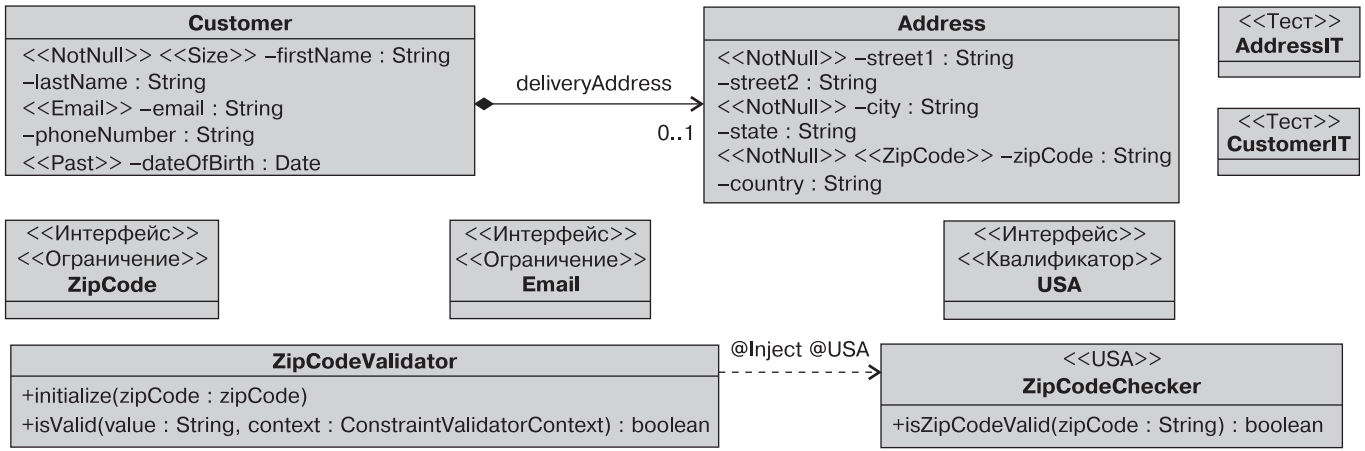


Рис. 3.2. Все вместе

- src/main/resources — содержит файл `beans.xml`, поэтому мы можем использовать как CDI, так и файл `ValidationMessages.properties` для сообщений об ошибках, связанных с нарушением ограничений;
- src/test/java — папка для интеграционных тестов `AddressIT` и `CustomerIT`;
- `pom.xml` — объектная модель проекта Maven (POM), описывающая проект и его зависимости.

Написание компонента Customer

В приложении CD-Book Store клиент покупает товары, заказанные онлайн, и эти товары доставляются на его почтовый адрес. Для обеспечения такой доставки приложению нужна верная информация об имени клиента, адрес электронной почты и адрес доставки. Поскольку у нас есть дата рождения клиента, программа может ежегодно присылать ему соответствующее поздравление. В листинге 3.22 показан компонент `Customer`, включающий в себя несколько встроенных ограничений, налагаемых на атрибуты (`firstname` не может быть нулевым, а дата `dateOfBirth` должна относиться к прошлому). Здесь также есть ограничение `@Email`, которое мы разработаем. Код проверяет, является ли строка `String` валидным адресом электронной почты.

Листинг 3.22. Компонент Customer со встроенными ограничениями и ограничением для электронной почты

```
public class Customer {
  @NotNull @Size(min = 2)
  private String firstName;
  private String lastName;
  @Email
  private String email;
  private String phoneNumber;
  @Past
  private Date dateOfBirth;
  private Address deliveryAddress;
  // Конструкторы, геттеры, сеттеры
}
```

Написание компонента Address

У `Customer` может быть ноль или один адрес доставки. `Address` — это компонент, включающий всю информацию, необходимую для доставки товара по указанному адресу: улица, город, штат, ZIP-код и страна. В листинге 3.23 показан компонент `Address` с ограничением `@NotNull`, налагаемым на важнейшие атрибуты (`street1`, `city` и `zipcode`), а также с ограничением `@ZipCode`, проверяющим валидность ZIP-кода (это ограничение будет разработано позже).

Листинг 3.23. Компонент Address со встроенными ограничениями и ограничением @ZipCode

```
public class Address {
  @NotNull
  private String street1;
```

```
  private String street2;
  @NotNull
  private String city;
  private String state;
  @NotNull @ZipCode
  private String zipcode;
  private String country;
  // Конструкторы, геттеры, сеттеры
}
```

Написание ограничения @Email

Ограничение `@Email` не встроено в систему валидации компонентов, поэтому нам самим придется его разработать. Нам не понадобится класс реализации (`@Constraint(validatedBy = {})`), так как вполне работоспособным будет обычная ограничивающая аннотация с регулярным выражением (`@Pattern`) и заданным размером. В листинге 3.24 показана ограничивающая аннотация `@Email`.

Листинг 3.24. Ограничивающая аннотация электронной почты со встроенной агрегацией ограничений

```
@Size(min = 7)
@Pattern(regexp = "[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\\." ➥
    + "[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*" ➥
    + "@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?")
@ReportAsSingleViolation
@Constraint(validatedBy = {})
@Target({METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER})
@Retention(RUNTIME)
public @interface Email {
  String message() default "{org.agoncal.book.javaee7.chapter03.Email.message}";
  Class<?>[] groups() default {};
  Class<? extends Payload>[] payload() default {};
  @Target({METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER})
  @Retention(RUNTIME)
  @interface List {
    Email[] value();
  }
}
```

Обратите внимание: в листинге 3.24 сообщение об ошибке представляет собой ключ пакета, определяемый в файле `META-INF/ValidationMessages.properties`.

```
org.agoncal.book.javaee7.chapter03.Email.message=invalid email address
```

Написание ограничения @ZipCode

Ограничение `@ZipCode` написать сложнее, чем `@Email`. ZIP-код имеет определенный формат (например, в США он состоит из пяти цифр), который не составляет труда проверить с помощью регулярного выражения. Но чтобы гарантировать, что ZIP-код не только синтаксически верен, но и валиден, необходимо прибегнуть к внешней службе, которая будет проверять, существует ли конкретный ZIP-код

в базе данных. Именно поэтому ограничивающая аннотация ZipCode в листинге 3.25 требует класса реализации (ZipCodeValidator).

Листинг 3.25. Ограничивающая аннотация @ZipCode

```
@Constraint(validatedBy = ZipCodeValidator.class)
@Target({METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER})
@Retention(RUNTIME)
public @interface ZipCode {
  String message() default "{org.agoncal.book.javaee7.chapter03.ZipCode.message}";
  Class<?>[] groups() default {};
  Class<? extends Payload>[] payload() default {};
  @Target({METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER})
  @Retention(RUNTIME)
  @interface List {
    ZipCode[] value();
  }
}
```

В листинге 3.26 показан класс реализации ограничивающей аннотации: ZipCodeValidator реализует интерфейс javax.validation.ConstraintValidator с обобщенным типом String. Метод isValid реализует алгоритм валидации, в рамках которого выполняется сопоставление с шаблоном регулярного выражения и происходит вызов внешнего сервиса: ZipCodeChecker. Код ZipCodeChecker здесь не показан, так как в данном случае он неважен. Но необходимо отметить, что он внедряется (@Inject) с квалификатором CDI (@USA, показан в листинге 3.27). Итак, здесь мы наблюдаем взаимодействие спецификаций CDI и Bean Validation.

Листинг 3.26. Реализация ограничения ZipCodeValidator

```
public class ZipCodeValidator implements ConstraintValidator<ZipCode, String> {
  @Inject @USA
  private ZipCodeChecker checker;
  private Pattern zipPattern = Pattern.compile("\\d{5}(-\\d{5})?");
  public void initialize(ZipCode zipCode) {
  }
  public boolean isValid(String value, ConstraintValidatorContext context) {
    if (value == null)
      return true;
    Matcher m = zipPattern.matcher(value);
    if (!m.matches())
    return false;
    return checker.isZipCodeValid(value);
  }
}
```

Листинг 3.27. Квалификатор USACDI

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD})
public @interface USA {
}
```

ПРИМЕЧАНИЕ

В следующих главах будет показано, как интегрировать валидацию компонентов с другими спецификациями, в частности JPA (можно добавлять ограничения к вашим сущностям) или JSF (можно ограничивать базовые компоненты).

Написание интеграционных тестов CustomerIT и AddressIT

Как мы теперь можем протестировать ограничения, налагаемые на наши компоненты? Мы ведь не можем писать модульные тесты для `@Email`, так как это аннотация, агрегирующая ограничения, равно как и для `@ZipCode`, которое может работать только для внедрения (а это контейнерная служба). Проще всего будет написать интеграционный тест, то есть использовать фабрику `ValidatorFactory` для получения `Validator`, а потом валидировать наши компоненты.

В листинге 3.28 показан класс `CustomerIT`, выполняющий интеграционное тестирование компонента `Customer`. Метод инициализирует `Validator` (с помощью `ValidatorFactory`), а метод `close()` высвобождает фабрику. Класс далее содержит два теста: в одном создается валидный объект `Customer`, а в другом создается объект с недопустимым адресом электронной почты и проверяется, окончится ли валидация ошибкой.

Листинг 3.28. Интеграционный тест CustomerIT

```
public class CustomerIT {
  private static ValidatorFactory vf;
  private static Validator validator;
  @BeforeClass
  public static void init() {
    vf = Validation.buildDefaultValidatorFactory();
    validator = vf.getValidator();
  }
  @AfterClass
  public static void close() {
    vf.close();
  }
  @Test
  public void shouldRaiseNoConstraintViolation() {
    Customer customer = new Customer("John", "Smith", "jsmith@gmail.com");
    Set<ConstraintViolation<Customer>> violations = validator.validate(customer);
    assertEquals(0, violations.size());
  }
  @Test
  public void shouldRaiseConstraintViolationCauseInvalidEmail() {
    Customer customer = new Customer("Джон", "Смит", "DummyEmail");
    Set<ConstraintViolation<Customer>> violations = validator.validate(customer);
    assertEquals(1, violations.size());
    assertEquals("invalid email address", violations.iterator().next().
getMessage());
    assertEquals("dummy", violations.iterator().next().getInvalidValue());
```

```
    assertEquals("{org.agoncal.book.javaee7.chapter03.Email.message}",
                 violations.iterator().next().getMessageTemplate());
  }
}
```

Листинг 3.29 построен по такому же принципу (`Validator` создается с помощью фабрики, происходит валидация компонента, фабрика закрывается), но проверяет компонент `Address`.

Листинг 3.29. Интеграционный тест AddressIT

```
public class AddressIT {
  @Test
  public void shouldRaiseConstraintViolationCauseInvalidZipCode() {
    ValidatorFactory vf = Validation.buildDefaultValidatorFactory();
    Validator validator = vf.getValidator();
    Address address = new Address("233 Стрит", "Нью-Йорк", "NY", ➥
"DummyZip", "США");
    Set<ConstraintViolation<Address>> violations = ➥
validator.validate(address);
    assertEquals(1, violations.size());
    vf.close();
  }
}
```

Компиляция и тестирование в Maven

Прежде чем протестировать все классы, их необходимо скомпилировать. Файл `pom.xml` в листинге 3.20 объявляет все зависимости, требуемые для компиляции кода: Hibernate Validator (справочная реализация Bean Validation) и Weld (для CDI). Обратите внимание: в `pom.xml` также объявляется плагин Failsafe, предназначенный для запуска интеграционных тестов (применяется одновременно с обоими классами — `CustomerIT` и `AddressIT`).

Листинг 3.30. Файл pom.xml для компиляции и тестирования ограничений

```
<?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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <artifactId>chapter03</artifactId>
    <groupId>org.agoncal.book.javaee7</groupId>
    <version>1.0</version>
  </parent>
  <groupId>org.agoncal.book.javaee7.chapter03</groupId>
  <artifactId>chapter03-putting-together</artifactId>
  <version>1.0</version>
  <dependencies>
    <dependency>
```

```
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-validator</artifactId>
      <version>5.0.0.Final</version>
    </dependency>
    <dependency>
      <groupId>org.jboss.weld.se</groupId>
      <artifactId>weld-se</artifactId>
      <version>2.0.0.Final</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.5.1</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>2.12.4</version>
        <executions>
          <execution>
            <id>integration-test</id>
            <goals>
              <goal>integration-test</goal>
              <goal>verify</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Чтобы скомпилировать классы, откройте командную строку в корневом каталоге, содержащем файл `pom.xml`, и введите следующую команду Maven:

```
$ mvn compile
```

Чтобы выполнить интеграционные тесты с плагином Maven Failsafe, введите в командную строку:

```
$ mvn integration-test
```

Резюме

Валидация компонентов обеспечивает комплексный подход ко всем проблемам, связанным с валидацией, и решает большинство возникающих на практике ситуаций путем валидации свойств или методов на любом уровне приложения. Если вы обнаруживаете, что какие-то случаи были проигнорированы или забыты, то API может быть расширен, чтобы соответствовать вашим запросам.

В этой главе было показано, что ограничение состоит из аннотации и отдельного класса, реализующего логику валидации. При дальнейшей работе можно агрегировать имеющиеся ограничения для создания новых или переиспользовать уже имеющиеся. Спецификация Bean Validation содержит несколько встроенных ограничений, хотя, к сожалению, некоторых ограничений в ней сильно не хватает (`@Email`, `@URL`, `@CreditCard`).

В первой версии спецификации Bean Validation можно было валидировать только методы и атрибуты. Так обеспечивалось более качественное предметно-ориентированное проектирование, при котором неглубокая доменная валидация помещалась в сам POJO, что позволяло избегать антипаттерна «анемичный объект». В версии Bean Validation 1.1 появилась валидация конструкторов, параметров методов и возвращаемых значений. В настоящее время можно выполнять валидацию предусловий и постусловий, что приближает нас к проектированию по соглашениям.

Из следующих глав вы узнаете, как валидация компонентов интегрирована в Java EE, какую роль в этом играют JPA и JSF и как ее можно использовать в большинстве спецификаций.

Глава 4
Java Persistence API

Приложения включают бизнес-логику, взаимодействие с другими системами, интерфейсы пользователя... и данные. Большую часть данных, которыми манипулируют наши приложения, приходится хранить в базах данных. Оттуда их требуется извлекать, а также анализировать их. Базы данных имеют важное значение: в них хранятся бизнес-данные, они выступают в качестве центральной точки между приложениями и обрабатывают информацию посредством триггеров или хранимых процедур. Постоянные данные встречаются повсюду, и большую часть времени они используют реляционные базы данных как основной механизм обеспечения постоянства (а не бессхемные базы данных). Информация в реляционных базах данных располагается в таблицах, состоящих из строк и столбцов. Данные идентифицируются с помощью первичных ключей, которые представляют собой особые столбцы с ограничениями уникальности и иногда индексами. Связи между таблицами предполагают использование внешних ключей и таблиц соединения с ограничениями целостности.

В таком объектно-ориентированном языке программирования, как Java, вся эта терминология неактуальна. При использовании Java мы манипулируем объектами, являющимися экземплярами классов. Объекты наследуют от других объектов, располагают ссылками на коллекции прочих объектов, а также иногда указывают на себя рекурсивным образом. У нас есть конкретные классы, абстрактные классы, интерфейсы, перечисления, аннотации, методы, атрибуты и т. д. Объекты хорошо инкапсулируют состояние и поведение, однако это состояние доступно только при работающей виртуальной машине Java (Java Virtual Machine — JVM): если виртуальная машина Java останавливается или сборщик мусора удаляет содержимое ее памяти, объекты исчезают вместе со своим состоянием. Некоторые объекты нуждаются в том, чтобы быть постоянными. Под постоянными данными я имею в виду данные, которые намеренно сохранены на перманентной основе на магнитном носителе, флеш-накопителе и т. п. Объект, который может сохранить свое состояние для его повторного использования позднее, называется постоянным.

Основная идея объектно-реляционного отображения (Object-Relational Mapping — ORM) заключается в объединении миров баз данных и объектов. Это подразумевает делегирование доступа к реляционным базам данных внешним инструментам или фреймворкам, которые, в свою очередь, обеспечивают объектно-ориентированное представление реляционных данных, и наоборот. Инструменты отображения предусматривают двунаправленное соответствие между базой данных и объектами. Несколько фреймворков обеспечивают это, например Hibernate, TopLink и Java Data Objects (JDO), а Java Persistence API (JPA) является предпочтительной технологией и частью Java EE 7.

Эта глава представляет собой введение в JPA, а в двух последующих главах я сосредоточусь на объектно-реляционном отображении, а также на выполнении запросов к объектам и управлении объектами.

Понятие сущностей

При разговоре об отображении объектов в реляционной базе данных, обеспечении постоянства объектов или выполнении запросов к ним вместо слова «объект» следует использовать термин «сущность». Объекты — это экземпляры, которые располагаются в памяти. Сущности представляют собой объекты, которые недолго располагаются в памяти, но постоянно — в базе данных. Все они могут быть отображены в базе данных. Они также могут быть конкретными или абстрактными; кроме того, они поддерживают наследование, связи и т. д. Произведя отображение этих сущностей, ими можно управлять посредством JPA. Вы можете обеспечить постоянство сущности в базе данных, удалить ее, а также выполнять запросы к этой сущности с использованием языка запросов Java Persistence Query Language, или JPQL. Объектно-реляционное отображение позволяет вам манипулировать сущностями при доступе к базе данных «за кадром». И, как вы увидите, у сущности имеется определенный жизненный цикл. JPA позволяет вам привязывать бизнес-код к событиям жизненного цикла с применением методов обратного вызова и слушателей.

В качестве первого примера начнем с простой сущности, которая только может быть у нас. В модели постоянства JPA сущность — это простой Java-объект в старом стиле (Plain Old Java Object — POJO). Это означает, что объявление сущности, создание ее экземпляра и использование осуществляются точно так же, как и в случае с любым другим Java-классом. У сущности имеются атрибуты (ее состояние), которыми можно манипулировать с помощью геттеров и сеттеров. Пример простой сущности приведен в листинге 4.1.

Листинг 4.1. Простой пример сущности Book

```
@Entity
public class Book {
  @Id @GeneratedValue
  private Long id;
  private String title;
  private Float price;
  private String description;
  private String isbn;
  private Integer nbOfPage;
  private Boolean illustrations;
  public Book() {
}
// Геттеры, сеттеры
}
```

В примере в листинге 4.1 представлена сущность `Book`, из которой я для ясности убрал геттеры и сеттеры. Как вы можете видеть, за исключением некоторых анно-

таций, эта сущность выглядит точно так же, как любой другой Java-класс: у нее есть атрибуты (`id`, `title`, `price` и т. д.) разных типов (`Long`, `String`, `Float`, `Integer` и `Boolean`), конструктор по умолчанию, при этом имеются геттеры и сеттеры для каждого атрибута. Как все это отображается в таблицу? Ответ можно получить благодаря аннотациям.

Анатомия сущности

Чтобы класс был сущностью, его нужно снабдить аннотацией `@javax.persistence.Entity`, которая позволит поставщику постоянства признать его постоянным классом, а не простым POJO. Кроме того, аннотация `@javax.persistence.Id` будет определять уникальный идентификатор этого объекта. Поскольку JPA используется для отображения объектов в реляционные таблицы, объектам необходим идентификатор, который будет отображен в первичный ключ. Остальные атрибуты в листинге 4.1 (`title`, `price`, `description` и т. д.) не снабжены аннотациями, поэтому они будут сделаны постоянными путем применения отображения по умолчанию.

Этот пример кода включает атрибуты, однако, как вы увидите позднее, сущность также может располагать бизнес-методами. Обратите внимание, что эта сущность `Book` является Java-классом, который не реализует никакого интерфейса и не расширяет какого-либо класса. Фактически, чтобы быть сущностью, класс должен придерживаться таких правил.

- Класс-сущность должен быть снабжен аннотацией `@javax.persistence.Entity` (или обозначен в XML-дескрипторе как сущность).
- Для обозначения простого первичного ключа должна быть использована аннотация `@javax.persistence.Id`.
- Класс-сущность должен располагать конструктором без аргументов, который должен быть `public` или `protected`. У класса-сущности также могут иметься другие конструкторы.
- Класс-сущность должен быть классом верхнего уровня. Перечисление или интерфейс не могут быть обозначены как сущность.
- Класс-сущность не должен быть `final`. Ни один из методов или постоянные переменные экземпляра класса-сущности тоже не могут быть `final`.
- Если экземпляр сущности надлежит передать с использованием значения как обособленный объект (например, с помощью удаленного интерфейса), то класс-сущность должен реализовывать интерфейс `Serializable`.

ПРИМЕЧАНИЕ

В предыдущих версиях Java EE постоянная компонентная модель называлась Entity Bean (Компонент-сущность EJB) или Entity Bean CMP (Container-Managed Persistence) (Сохраняемость компонентов-сущностей EJB, управляемая контейнером) и была связана с Enterprise JavaBeans. Эта модель постоянства использовалась, начиная с J2EE 1.3 и до появления версии 1.4, однако была тяжеловесной и в конце концов оказалась заменена JPA с выходом версии Java EE 5. В JPA вместо словосочетания «компонент-сущность EJB» используется термин «сущность».

Объектно-реляционное отображение

Принцип объектно-реляционного отображения заключается в возложении на внешние инструменты или фреймворки (в нашем случае JPA) задачи по обеспечению соответствия между объектами и таблицами. Тогда мир классов, объектов и атрибутов можно будет отобразить в реляционные базы данных, состоящие из таблиц, которые содержат строки и столбцы. Отображение обеспечивает объектно-ориентированное представление для разработчиков, которые смогут прозрачно использовать сущности вместо таблиц. Как именно JPA отображает объекты в базе данных? Ответ: с помощью *метаданных*.

С каждой сущностью ассоциированы метаданные, которые описывают отображение. Они позволяют поставщику постоянства распознать сущность и интерпретировать отображение. Метаданные могут быть записаны в двух разных форматах.

- *Аннотации* — код сущности непосредственно снабжается всевозможными аннотациями, описанными в пакете `javax.persistence`.
- *XML-дескрипторы* — вместо аннотаций (или в дополнение к ним) вы можете использовать XML-дескрипторы. Отображение определяется во внешнем XML-файле, который будет развернут вместе с сущностями. Это может оказаться очень полезным, если, к примеру, конфигурация базы данных будет изменяться в зависимости от среды.

В случае с сущностью `Book` (показанной в листинге 4.1) используются аннотации JPA, чтобы поставщик постоянства смог синхронизировать данные между атрибутами сущности `Book` и столбцами таблицы `BOOK`. Следовательно, если атрибут `isbn` окажется модифицирован приложением, то будет синхронизирован столбец `ISBN` (при управлении сущностью, задании контекста транзакций и т. д.).

Как показано на рис. 4.1, сущность `Book` отображается в таблице `BOOK`, а каждому столбцу присваивается имя в соответствии с именем атрибута класса (например, атрибут `isbn`, имеющий тип `String`, отображается в столбец, который имеет имя `ISBN` и тип `VARCHAR`). Эти правила отображения по умолчанию являются важной частью принципа, известного как конфигурация в порядке исключения.

В версии Java EE 5 была представлена идея конфигурации в порядке исключения (иногда называемая программированием в порядке исключения или *соглашением прежде конфигурации*), которая по-прежнему активно используется сегодня в Java EE 7. Это означает, что, если не указано иное, контейнер или поставщик должен применять правила по умолчанию. Другими словами, необходимость обеспечения конфигурации является исключением из правила. Это позволяет вам написать минимальное количество кода для того, чтобы ваше приложение работало, положившись на правила, применяемые контейнером и поставщиком по умолчанию. Если вы не хотите, чтобы поставщик применял правила по умолчанию, то можете настроить отображение в соответствии со своими нуждами. Иначе говоря, необходимость обеспечения конфигурации является исключением из правила.

Без аннотаций сущность `Book` в листинге 4.1 рассматривалась бы как простой POJO, а ее постоянство не обеспечивалось бы. Это закон: если не предусмотрено

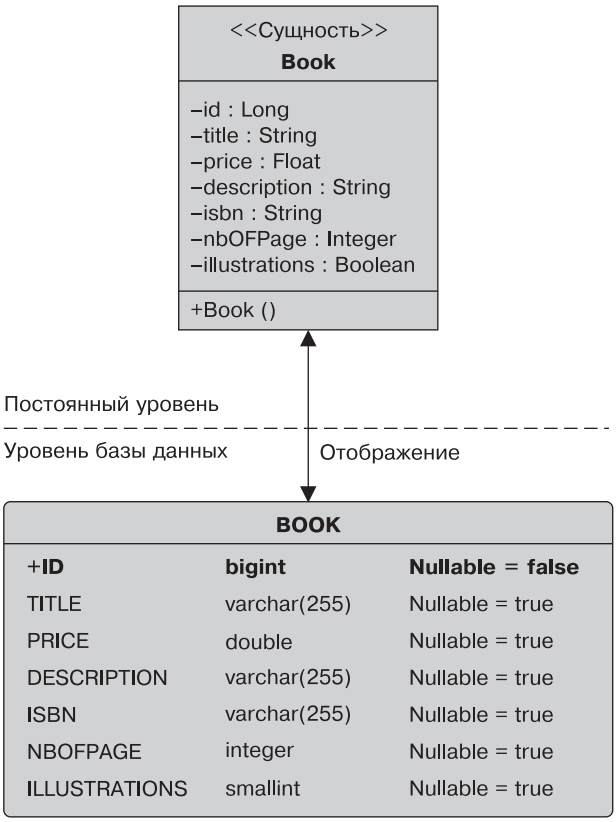


Рис. 4.1. Синхронизация данных между сущностью и таблицей

никакой специальной конфигурации, то должны применяться правила по умолчанию, а поставщик постоянства по умолчанию исходит из того, что у класса `Book` нет представления базы данных. Но, поскольку вам необходимо изменить это поведение по умолчанию, вы снабжаете класс аннотацией `@Entity`. То же самое и в случае с идентификатором. Вам нужен способ сообщить поставщику постоянства о том, что атрибут `id` требуется отобразить в первичный ключ, поэтому вы снабжаете его аннотацией `@Id`, а значение соответствующего идентификатора автоматически генерируется поставщиком постоянства с использованием опциональной аннотации `@GeneratedValue`. Решение такого рода характеризует подход «конфигурация в порядке исключения», при котором аннотации не требуются в более общих случаях, а используются, только когда необходимо переопределение. Это означает, что по отношению ко всем прочим атрибутам будут применяться следующие правила отображения по умолчанию.

- Имя сущности будет отображаться в имя реляционной таблицы (например, сущность `Book` отобразится в таблицу `BOOK`). Если вы захотите отобразить сущность в другую таблицу, то вам потребуется прибегнуть к аннотации `@Table`, как вы увидите в разделе «Элементарное отображение» в следующей главе.

- Имена атрибутов будут отображаться в имени столбца (например, атрибут `id` или метод `getId()` отобразится в столбце `ID`). Если вы захотите изменить это отображение по умолчанию, то вам придется воспользоваться аннотацией `@Column`.
- Правила JDBC применяются при отображении Java-примитивов в типах реляционных данных. `String` отобразится в `VARCHAR`, `Long` — в `BIGINT`, `Boolean` — в `SMALLINT` и т. д. Размер по умолчанию столбца, отображаемого из `String`, будет равен `255` (`String` отобразится в `VARCHAR(255)`). Однако имейте в виду, что правила отображения по умолчанию разнятся от одной базы данных к другой. Например, `String` отображается в `VARCHAR` при использовании базы данных Derby и в `VARCHAR2`, если применяется Oracle. `Integer` отображается в `INTEGER` в случае с базой данных Derby и в `NUMBER` при использовании Oracle. Информация, касающаяся основной базы данных, будет содержаться в файле `persistence.xml`, который вы увидите позднее.

Придерживаясь этих правил, сущность `Book` будет отображена в Derby-таблицу со структурой, описанной в листинге 4.2.

Листинг 4.2. Сценарий для создания структуры таблицы BOOK

```
CREATE TABLE BOOK (
  ID BIGINT NOT NULL,
  TITLE VARCHAR(255),
  PRICE FLOAT,
  DESCRIPTION VARCHAR(255),
  ISBN VARCHAR(255),
  NBOFPAGE INTEGER,
  ILLUSTRATIONS SMALLINT DEFAULT 0,
  PRIMARY KEY (ID)
)
```

JPA2.1 предусматривает API-интерфейс и стандартный механизм для автоматического генерирования баз данных из сущностей и создания сценариев вроде того, что показан в листинге 4.2. Это очень удобно, когда вы находитесь в режиме разработки. Однако бо́льшую часть времени вам понадобится подключение к унаследованной базе данных, которая уже существует.

В листинге 4.1 приведен пример очень простого отображения. Как вы увидите в следующей главе, отображение может быть намного более обширным, включая всевозможные вещи, начиная с объектов и заканчивая связями. Мир объектно-ориентированного программирования изобилует классами и ассоциациями между классами (и коллекциями классов). В случае с базами данных также моделируются связи, но по-другому — с использованием внешних ключей или таблиц соединения. В JPA имеется набор метаданных для управления отображением связей. Даже наследование может быть отображено. Оно широко применяется разработчиками для повторного использования кода, однако эта концепция изначально неизвестна в сфере реляционных баз данных (поскольку им приходится эмулировать наследование с использованием внешних ключей и ограничений). Даже если и придется прибегнуть к некоторым трюкам при отображении наследования, JPA поддерживает его и предлагает вам три разные стратегии на выбор.

ПРИМЕЧАНИЕ

JPA нацелен на реляционные базы данных. Метаданные отображения (аннотации или XML) предназначены для отображения сущностей в структурированные таблицы, а атрибутов — в столбцы. Благодаря разным структурам хранения данных началась новая эра баз данных NoSQL (Not Only SQL — «Не только SQL») (или бессхемных баз данных): ключ/значение, столбец, документ или граф. В настоящее время JPA не позволяет отображать сущности в эти структуры. Hibernate OGM — фреймворк с открытым исходным кодом, который пытается решить этот вопрос. У EclipseLink тоже имеется несколько расширений для отображения NoSQL-структур. Рассмотрение Hibernate OGM и EclipseLink-расширений выходит за рамки этой книги, однако вам следует взглянуть на них, если вы планируете использовать базы данных NoSQL.

Выполнение запросов к сущностям

JPA позволяет вам отображать сущности в базах данных, а также выполнять к ним запросы с использованием разных критериев. Мощь JPA заключается в том, что он дает возможность выполнять запросы к сущностям объектно-ориентированным путем без необходимости применения внешних ключей или столбцов, относящихся к основной базе данных. Центральным элементом API-интерфейса, отвечающего за оркестровку сущностей, является `javax.persistence.EntityManager`. Его роль состоит в управлении сущностями, их чтении и записи в определенную базу данных наряду с обеспечением возможности проведения простых операций CRUD (create, read, update, delete — «создание», «чтение», «обновление», «удаление»), а также комплексных запросов с применением JPQL. С технической точки зрения `EntityManager` представляет собой всего лишь интерфейс, реализация которого обеспечивается поставщиком постоянства (например, EclipseLink). В приведенном далее фрагменте кода показано, как получить `EntityManager` и обеспечить постоянство сущности `Book`:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("chapter04PU");
EntityManager em = emf.createEntityManager();
em.persist(book);
```

На рис. 4.2 продемонстрировано, как интерфейс менеджера сущностей может быть использован классом (который здесь имеет имя `Main`) для манипулирования сущностями (в данном случае `Book`). С помощью таких методов, как `persist()` и `find()`, `EntityManager` скрывает JDBC-вызовы базы данных и оператор SQL (Structured Query Language — язык структурированных запросов) `INSERT` или `SELECT`.

Менеджер сущностей также позволяет вам выполнять запросы к сущностям. *Запрос* в данном случае аналогичен запросу к базе данных за исключением того, что вместо использования SQL интерфейс JPA выполняет запросы к сущностям с применением JPQL. В его синтаксисе используется привычная точечная (`.`) нотация объектов. Для извлечения информации обо всех книгах, в названии которых присутствует `H2G2`, вы можете написать следующее:

```
SELECT b FROM Book b WHERE b.title = 'H2G2'
```

Следует отметить, что `title` является именем атрибута `Book`, а не именем столбца таблицы. JPQL-операторы манипулируют объектами и атрибутами, а не таблицами и столбцами. JPQL-оператор может выполняться с использованием динамических

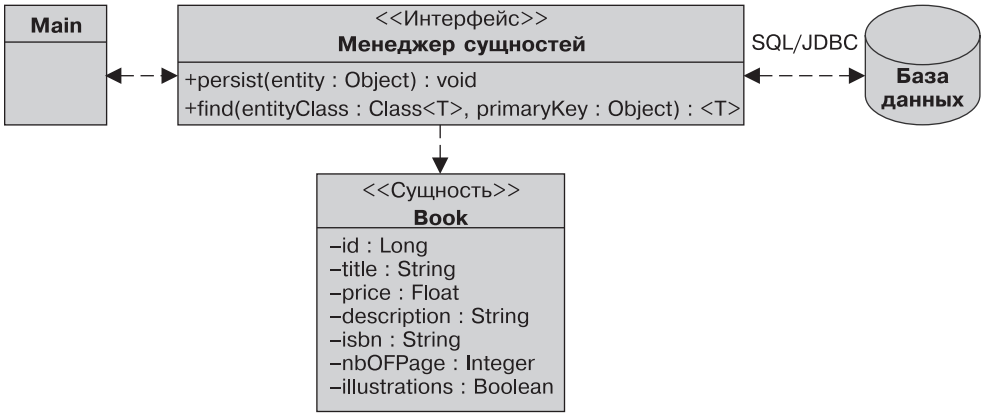


Рис. 4.2. Менеджер сущностей взаимодействует с сущностью и основной базой данных

(генерируемых динамически во время выполнения) или статических (определяемых статически во время компиляции) запросов. Вы также можете выполнять «родные» SQL-операторы и даже хранимые процедуры. Статические запросы, также известные как именованные, определяются с использованием либо аннотаций (`@NamedQuery`), либо XML-метаданных. Приведенный ранее JPQL-оператор может, к примеру, быть определен как именованный запрос в отношении сущности `Book`. В листинге 4.3 показана сущность `Book` с определением именованного запроса `findBookH2G2` с помощью аннотации `@NamedQuery` (более подробно о запросах мы поговорим в главе 6).

Листинг 4.3. Сущность Book с именованным запросом findBookH2G2

```
@Entity
@NamedQuery(name = "findBookH2G2",
            query = "SELECT b FROM Book b WHERE b.title ='H2G2'")
public class Book {
  @Id @GeneratedValue
  private Long id;
  private String title;
  private Float price;
  private String description;
  private String isbn;
  private Integer nbOfPage;
  private Boolean illustrations;
// Конструкторы, геттеры, сеттеры
}
```

`EntityManager` можно получить в стандартном Java-классе с использованием фабрики. В листинге 4.4 показан такой класс, который создает сущность `Book`, обеспечивает ее постоянство в таблице и выполняет именованный запрос. Соответствующие действия он предпринимает с этапа 1 по 5.

1. Создает экземпляр сущности `Book`. Сущности являются аннотированными POJO, управляемыми поставщиком постоянства. С точки зрения Java экземпляр класса, как и любой POJO, необходимо создавать с помощью ключевого слова

new. Важно подчеркнуть, что до этой точки в коде поставщик постоянства не осведомлен об объекте Book.

2. Получает EntityManager и транзакцию. Это важная часть кода, поскольку EntityManager необходим для манипулирования сущностями. Прежде всего создается EntityManagerFactory для единицы сохраняемости chapter04PU. После этого EntityManagerFactory задействуется для получения EntityManager (переменная em), используется повсюду в коде для получения транзакции (переменная tx) и обеспечения постоянства и извлечения Book.
3. Обеспечивает постоянство Book в базе данных. Код начинает транзакцию (tx.begin()) и использует метод EntityManager.persist() для вставки экземпляра Book. После фиксации транзакции (tx.commit()) информация сбрасывается в базу данных.
4. Выполняет именованный запрос. Опять-таки EntityManager используется для извлечения Book посредством именованного запроса findBookH2G2.
5. Закрывает EntityManager и EntityManagerFactory.

Листинг 4.4. Класс Main, который обеспечивает постоянство и извлекает сущность Book

```
public class Main {
  public static void main(String[] args) {
    // 1. Создает экземпляр Book
    Book book = new Book("H2G2", "Автостопом по Галактике", 12.5F,
"1-84023-742-2", 354, false);
    // 2. Получает EntityManager и транзакцию
    EntityManagerFactory emf =
Persistence.createEntityManagerFactory("chapter04PU");
    EntityManager em = emf.createEntityManager();
    // 3. Обеспечивает постоянство Book в базе данных
    EntityTransaction tx = em.getTransaction();
    tx.begin();
    em.persist(book);
    tx.commit();
    // 4. Выполняет именованный запрос
    book = em.createNamedQuery("findBookH2G2", Book.class).getSingleResult();
    // 5. Закрывает EntityManager и EntityManagerFactory
    em.close();
    emf.close();
  }
}
```

Обратите внимание на отсутствие SQL-запросов и JDBC-вызовов в листинге 4.4. Как показано на рис. 4.2, класс Main взаимодействует с основной базой данных с помощью интерфейса EntityManager, который обеспечивает набор стандартных методов, позволяющих вам осуществлять операции с сущностью Book. EntityManager «за кадром» полагается на поставщика постоянства для взаимодействия с базами данных. При вызове метода EntityManager поставщик постоянства генерирует и выполняет SQL-оператор с помощью соответствующего JDBC-драйвера.

Единица сохраняемости

Какой JDBC-драйвер вам следует использовать? Как вам нужно подключаться к базе данных? Каково имя базы данных? Эта информация отсутствует в приводившемся ранее коде. Когда класс `Main` (см. листинг 4.4) создает `EntityManagerFactory`, он передает имя единицы сохраняемости в качестве параметра; в данном случае она имеет имя `chapter04PU`. Единица сохраняемости позволяет `EntityManager` узнать тип базы данных для использования и параметры подключения, определенные в файле `persistence.xml`, который показан в листинге 4.5 и должен быть доступен по пути к соответствующему классу.

Листинг 4.5. Файл persistence.xml, определяющий единицу сохраняемости

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence ➥
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
             version="2.1">
  <persistence-unit name="chapter04PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>org.agoncal.book.javaee7.chapter04.Book</class>
    <properties>
      <property name="javax.persistence.schema-generation-action" ➥
value="drop-and-create"/>
      <property name="javax.persistence.schema-generation-target"
value="database"/>
      <property name="javax.persistence.jdbc.driver" ➥
                value="org.apache.derby.jdbc.ClientDriver"/>
      <property name="javax.persistence.jdbc.url" ➥
                value="jdbc:derby://localhost:1527/chapter04DB;create=true"/>
      <property name="javax.persistence.jdbc.user" value="APP"/>
      <property name="javax.persistence.jdbc.password" value="APP"/>
    </properties>
  </persistence-unit>
</persistence>
```

Единица сохраняемости `chapter04PU` определяет JDBC-подключение для базы данных Derby `chapter04DB`, которая функционирует на локальном хосте с помощью порта 1527. К ней подключается пользователь (`APP`) с применением пароля (`APP`) по заданному URL-адресу. Тег `<class>` дает указание поставщику постоянства управлять классом `Book` (существуют и другие теги для неявного или явного обозначения классов с управляемым постоянством, например `<mapping-file>`, `<jar-file>` или `<exclude-unlisted-classes>`). Без единицы сохраняемости сущностями можно манипулировать как POJO, однако их постоянство при этом обеспечиваться не будет.

Жизненный цикл сущности и обратные вызовы

Сущности представляют собой всего лишь POJO. Когда `EntityManager` управляет POJO, у них имеется идентификатор постоянства (ключ, который уникально идентифицирует экземпляр и является эквивалентом первичного ключа), а база данных

синхронизирует их состояние. Когда управление ими не осуществляется (то есть они обособлены от `EntityManager`), их можно использовать как любой другой Java-класс. Это означает, что у сущностей имеется жизненный цикл (рис. 4.3). Когда вы создадите экземпляр сущности `Book` с помощью оператора `new`, объект будет располагаться в памяти, а JPA ничего не будет знать о нем (этот объект даже может перестать существовать в результате сборки мусора) Когда им начнет управлять `EntityManager`, таблица `BOOK` отобразит и синхронизирует его состояние. Вызов метода `EntityManager.remove()` приведет к удалению соответствующей информации из базы данных, однако Java-объект продолжит находиться в памяти, пока не исчезнет в результате сборки мусора.

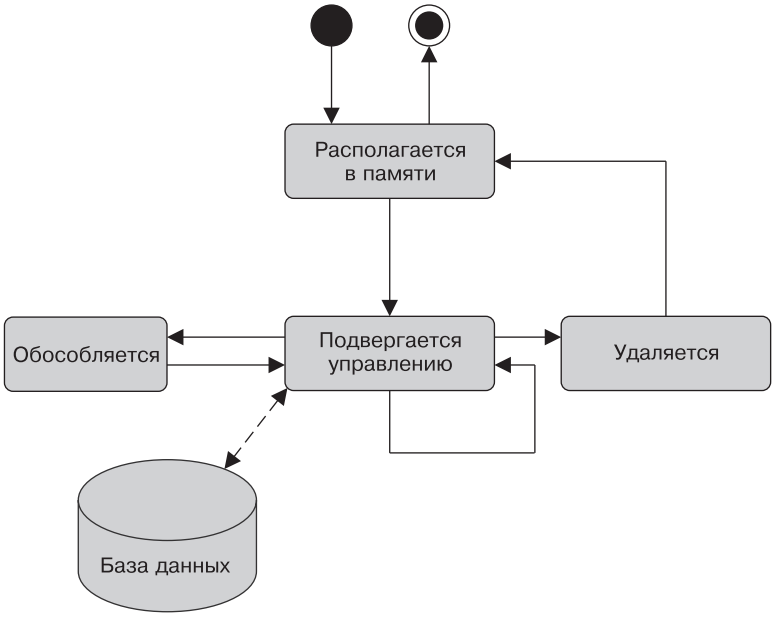


Рис. 4.3. Жизненный цикл сущности

Операции с сущностями подпадают под четыре категории: обеспечение постоянства, обновление, удаление и загрузка — аналогичные категориям операций с базами данных, к которым относятся соответственно вставка, обновление, удаление и выборка. Для каждой операции имеют место события с приставками `pre` и `post` (за исключением загрузки, для которой имеет место только событие с приставкой `post`). Эти события могут быть перехвачены `EntityManager` для вызова бизнес-метода. Как вы увидите в главе 6, в вашем распоряжении будут аннотации `@PrePersist`, `@PostPersist` и т. д. JPA позволяет вам привязывать бизнес-логику к определенной сущности, когда имеют место эти события. Упомянутые аннотации могут быть применены к методам сущностей (также известным как методы обратного вызова) или внешним классам (также известным как слушатели). Вы можете представлять себе методы обратного вызова и слушатели как аналогичные триггеры в реляционной базе данных.

Интеграция с Bean Validation

Технология Bean Validation, которая была разъяснена в предыдущей главе, связана с Java EE несколькими способами. Один из них выражается в ее интеграции с JPA и жизненным циклом сущностей. Сущности могут включать ограничения Bean Validation и автоматически подвергаться валидации. Фактически автоматическая валидация обеспечивается благодаря тому, что JPA возлагает ее проведение на реализацию Bean Validation при наступлении событий жизненного цикла сущности `pre-persist`, `pre-update` и `pre-remove`. Разумеется, при необходимости валидация по-прежнему может проводиться вручную вызовом метода `validate` класса `Validator` в отношении сущности.

В листинге 4.6 показана сущность `Book` с двумя ограничениями Bean Validation (`@NotNull` и `@Size`). Если значением атрибута `title` окажется `null`, а вы захотите обеспечить постоянство этой сущности (вызвав `EntityManager.persist()`), то во время выполнения JPA будет выброшено исключение `ConstraintViolation`, а соответствующая информация не будет помещена в базу данных.

Листинг 4.6. Сущность с аннотациями Bean Validation

```
@Entity
public class Book {
  @Id @GeneratedValue
  private Long id;
  @NotNull
  private String title;
  private Float price;
  @Size(min = 10, max = 2000)
  private String description;
  private String isbn;
  private Integer nbOfPage;
  private Boolean illustrations;
// Конструкторы, геттеры, сеттеры
}
```

Обзор спецификации JPA

Версия JPA 1.0 была создана вместе с Java EE 5 для решения проблемы обеспечения постоянства данных. Она объединила объектно-ориентированные и реляционные модели. В Java EE 7 версия JPA 2.1 идет тем же путем простоты и надежности, привнося при этом новую функциональность. Вы можете использовать этот API-интерфейс для доступа к реляционным данным Enterprise JavaBeans, веб-компонентам и приложениям Java SE и манипулирования ими.

JPA — это абстракция над JDBC, которая дает возможность быть независимым от SQL. Все классы и аннотации этого API-интерфейса располагаются в пакете `javax.persistence`. Рассмотрим основные компоненты JPA.

- Объектно-реляционное отображение, которое представляет собой механизм отображения объектов в данные, хранящиеся в реляционной базе данных.

- API менеджера сущностей для осуществления операций, связанных с базами данных, например CRUD-операций.
- JPQL, который позволяет вам извлекать данные с помощью объектно-ориентированного языка запросов.
- Транзакции и механизмы блокировки, которые предусматривает Java Transaction API (JTA) при одновременном доступе к данным. JPA также поддерживает ресурсные локальные (не-JTA) транзакции.
- Обратные вызовы и слушатели для добавления бизнес-логики в жизненный цикл того или иного постоянного объекта.

Краткая история JPA

Решения, которые позволяют выполнять объектно-реляционное отображение, существуют уже долгое время, причем они появились раньше Java. Продукты вроде TopLink начинали со Smalltalk в 1994 году, перед тем как перейти на Java. Коммерческие продукты для выполнения объектно-реляционного отображения, например TopLink, доступны с самых первых дней существования языка Java. Они стали успешными, но никогда не были стандартизированы для платформы Java. Схожий подход к объектно-реляционному отображению был стандартизирован в форме технологии JDO, которой так и не удалось значительно проникнуть на рынок.

В 1998 году появилась версия EJB 1.0, которая позднее сопутствовала J2EE 1.2. Это был тяжеловесный, распределенный компонент, использовавшийся для обработки транзакционной бизнес-логики. Технология Entity Bean CMP, представленная в EJB 1.0 как опциональная, стала обязательной в EJB 1.1 и совершенствовалась с выходом последующих версий вплоть до EJB 2.1 (J2EE 1.4). Постоянство могло обеспечиваться только внутри контейнера благодаря сложному механизму создания экземпляров с использованием домашних, локальных или удаленных интерфейсов. Возможности в плане объектно-реляционного отображения тоже оказались весьма ограниченными, поскольку наследование было сложно отображать.

Параллельно с миром J2EE существовало популярное решение с открытым исходным кодом, которое привело к удивительным изменениям в «направлении» постоянства: имеется в виду технология Hibernate, вернувшая назад легковесную, объектно-ориентированную постоянную модель.

Спустя годы жалоб на компоненты Entity CMP 2.*x* и в знак признания успеха и простоты фреймворков с открытым исходным кодом вроде Hibernate архитектура модели постоянства Enterprise Edition была полностью пересмотрена в Java EE 5. Версия JPA 1.0 была создана с использованием весьма облегченного подхода, который позаимствовал многие принципы проектирования Hibernate. Спецификация JPA 1.0 была связана с EJB 3.0 (JSR 220). В 2009 году версия JPA 2.0 (JSR 317) сопутствовала Java EE 6 и привнесла новые API-интерфейсы, расширила JPQL и добавила новую функциональность, такую, например, как кэш второго уровня, пессимистическая блокировка или Criteria API.

Сегодня, когда уже есть Java EE 7, версия JPA 2.1 стремится к легкости разработки и привносит новые функции. Она эволюционировала в JSR 338.

Что нового в JPA 2.1

Если версия JPA 1.0 стала революцией по сравнению со своим предком Entity CMP 2.*x* в силу совершенно новой модели постоянства, JPA 2.0 оказалась продолжением JPA 1.0, а сегодня версия JPA2.1 идет тем же путем и привносит множество новых опций и усовершенствований.

- Генерирование схем — JPA 2.1 включает механизм генерирования стандартизированных схем баз данных, привнося новый API-интерфейс и набор свойств (которые определяются в файле `persistence.xml`).
- Преобразователи — новые классы, которые обеспечивают преобразование между представлениями баз данных и атрибутов.
- Поддержка CDI — теперь можно внедрять зависимости в слушатели событий.
- Поддержка хранимых процедур — JPA 2.1 позволяет выполнять динамически генерируемые и именованные запросы к хранимым процедурам.
- Запросы с использованием критериев на пакетное обновление и удаление — Criteria API позволял выполнять только запросы на выборку; теперь стали возможны запросы на обновление и удаление.
- Понижающее приведение — новый оператор `TREAT` обеспечивает доступ к специфичному для подклассов состоянию в запросах.

В табл. 4.1 приведены основные пакеты, определенные в JPA 2.1 на сегодняшний день.

Таблица 4.1. Основные JPA-пакеты

Пакет	Описание
javax.persistence	API-интерфейс для управления постоянством и объектно-реляционным отображением
javax.persistence.criteria	Java Persistence Criteria API
javax.persistence.metamode	Java Persistence Metamodel API
javax.persistence.spi	SPI для поставщиков Java Persistence

Эталонная реализация

EclipseLink 2.5 представляет собой эталонную реализацию JPA 2.1, имеющую открытый исходный код. Она обеспечивает мощный и гибкий фреймворк для сохранения Java-объектов в реляционных базах данных. EclipseLink является реализацией JPA, однако также поддерживает XML-постоянство с помощью Java XML Binding (JAXB) и других средств вроде Service Data Objects (SDO). EclipseLink предусматривает поддержку не только объектно-реляционного отображения, но и технологии Object XML Mapping (OXM), постоянства объектов в информационных системах предприятий (Enterprise Information System — EIS) с использованием Java EE Connector Architecture (JCA) и веб-служб баз данных.

Истоки EclipseLink берут свое начало в продукте TopLink от компании Oracle, переданном организации Eclipse Foundation в 2006 году. EclipseLink является

эталонной реализацией JPA и фреймворком постоянства, используемым в этой книге. Его также называют поставщиком постоянства, или просто поставщиком.

На момент написания этой книги EclipseLink был лишь реализацией JPA 2.1. Однако вскоре последуют Hibernate и OpenJPA и у вас будет несколько реализаций на выбор.

Все вместе

Теперь, когда вы немного познакомились с JPA, EclipseLink, сущностями, менеджером сущностей и JPQL, сведем их воедино и напишем небольшое приложение, которое будет обеспечивать постоянство сущности в базе данных. Идея заключается в том, чтобы написать простую сущность `Book` с ограничениями Bean Validation и класс `Main`, который позаботится о ее постоянстве. Затем вы проведете компиляцию соответствующего кода с помощью Maven и выполните его с использованием EclipseLink и клиентской базы данных Derby. Чтобы продемонстрировать, насколько легко провести интеграционное тестирование сущности, я покажу вам, как написать тестовый класс (`BookIT`) с применением JUnit 4 и задействовать встроенный режим Derby для обеспечения постоянства данных с использованием базы данных в оперативной памяти.

В этом примере мы будем придерживаться структуры каталогов Maven, в силу чего классы и файлы, показанные на рис. 4.4, должны будут располагаться в следующих каталогах:

- `src/main/java` — для сущности `Book` и класса `Main`;
- `src/main/resources` — для файла `persistence.xml`, которым будут пользоваться классы `Main` и `BookIT`, а также сценарий загрузки базы данных `insert.sql`;
- `src/test/java` — для класса `BookIT`, который будет применяться для интеграционного тестирования;
- `pom.xml` — для POM-модели Maven, которая описывает проект и его зависимости от других внешних модулей и компонентов.

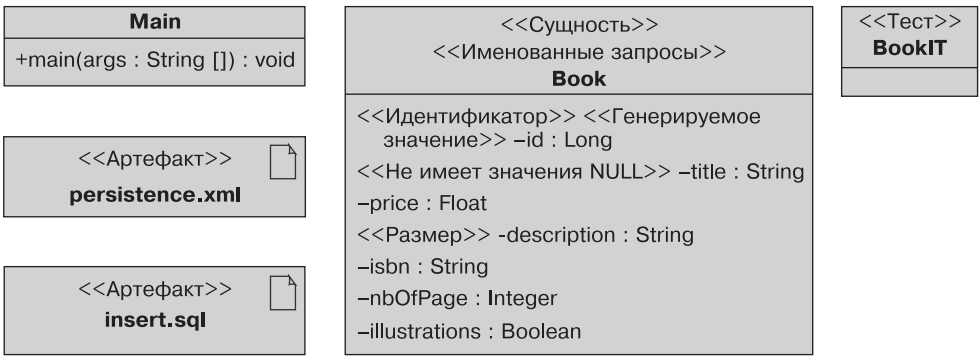


Рис. 4.4. Все вместе

Написание сущности Book

Сущность Book, показанную в листинге 4.7, необходимо создать и разместить в каталоге src/main/java. У нее будет несколько атрибутов (title, price и т. д.) с разными типами данных (String, Float, Integer и Boolean), аннотации Bean Validation (@NotNull и @Size), а также аннотации JPA.

- @Entity проинформирует поставщика постоянства о том, что этот класс является сущностью и ему следует управлять им.
- Аннотации @NamedQueries и @NamedQuery будут определять два именованных запроса, которые станут использовать JPQL для извлечения информации о книгах из базы данных.
- @Id будет определять атрибут id как первичный ключ.
- Аннотация @GeneratedValue проинформирует поставщика постоянства о необходимости автоматического генерирования первичного ключа с использованием инструмента id основной базы данных.

Листинг 4.7. Сущность Book и именованный запрос

```
package org.agoncal.book.javaee7.chapter04;
@Entity
@NamedQueries({
  @NamedQuery(name = "findAllBooks", query = "SELECT b FROM Book b"),
  @NamedQuery(name = "findBookH2G2", query = "SELECT b FROM Book b ➥
WHERE b.title ='H2G2'")
})
public class Book {
  @Id @GeneratedValue
  private Long id;
  @NotNull
  private String title;
  private Float price;
  @Size(min = 10, max = 2000)
  private String description;
  private String isbn;
  private Integer nbOfPage;
  private Boolean illustrations;
  // Конструкторы, геттеры, сеттеры
}
```

Обратите внимание, что для лучшей удобочитаемости я убрал конструктор, геттеры и сеттеры этого класса. Как видно из кода, если не считать нескольких аннотаций, Book представляет собой простой POJO. А теперь напишем класс Main, который будет обеспечивать постоянство Book в базе данных.

Написание класса Main

Класс Main, показанный в листинге 4.8, будет располагаться в том же пакете, что и сущность Book. Он начнет с того, что создаст новый экземпляр Book (с использованием ключевого слова Java new) и задаст значения для его атрибутов. Здесь не

будет ничего особенного, лишь чистый Java-код. Затем он воспользуется классом Persistence для получения экземпляра EntityManagerFactory, ссылающегося на единицу сохраняемости с именем chapter04PU, которую я охарактеризую позднее, в разделе «Написание единицы сохраняемости». EntityManagerFactory создаст экземпляр EntityManager (переменная em). Как уже отмечалось ранее, EntityManager является центральным элементом JPA в том смысле, что способен начать транзакцию, обеспечить постоянство объекта Book с помощью метода EntityManager.persist(), а затем произвести фиксацию транзакции. В конце метода main() как EntityManager, так и EntityManagerFactory будут закрыты, чтобы высвободить ресурсы поставщика.

Листинг 4.8. Класс Main, который обеспечивает постоянство сущности Book

```
package org.agoncal.book.javaee7.chapter04;
public class Main {
  public static void main(String[] args) {
    // Создает экземпляр Book
    Book book = new Book("H2G2", "Автостопом по Галактике", ➥
    12.5F, "1-84023-742-2", 354, false);
    // Получает EntityManager и транзакцию
    EntityManagerFactory emf = ➥
Persistence.createEntityManagerFactory("chapter04PU");
    EntityManager em = emf.createEntityManager();
    // Обеспечивает постоянство Book в базе данных
    EntityTransaction tx = em.getTransaction();
    tx.begin();
    em.persist(book);
    tx.commit();
    // Закрывает EntityManager и EntityManagerFactory
    em.close();
    emf.close();
  }
}
```

Опять-таки ради удобочитаемости я убрал обработку исключений. Если будет иметь место исключение постоянства, то вам придется откатить транзакцию, зарегистрировать сообщение и закрыть EntityManager в финальном блоке.

Написание интеграционного теста BookIT

Одна из жалоб на предыдущие версии Entity CMP 2.*x* заключалась в сложности интеграционного тестирования постоянных компонентов. Один из основных выигрышных моментов JPA состоит в том, что вы можете с легкостью тестировать сущности без необходимости в работающем сервере приложений или реальной базе данных. Но какие именно функции вы можете протестировать? Сущности как таковые обычно не нуждаются в тестировании в изоляции. Большинство методов сущностей представляют собой простые геттеры или сеттеры. Проверка того, что сеттер присваивает значение атрибуту, а также того, что соответствующий геттер извлекает то же значение, не имеет особой ценности (если только побочный эффект не проявится в геттерах или сеттерах). Поэтому модульное тестирование сущностей представляет ограниченный интерес.

А как насчет тестирования запросов к базе данных? Нужно ли убедиться в том, что запрос `findBookH2G2` корректен? Или вводить информацию в базу данных и тестировать комплексные запросы, возвращающие множественные значения? Такие интеграционные тесты потребовали бы реальной базы данных с реальной информацией, либо вы стали бы проводить модульное тестирование в изоляции, пытаясь симулировать запрос. Хорошее компромиссное решение — использовать базу данных в оперативной памяти и JPA-транзакции. CRUD-операции и JPQL-запросы могут быть протестированы с применением очень легковесной базы данных, которая не потребует запуска отдельного процесса (понадобится лишь добавить файл с расширением `.jar`, используя путь к соответствующему классу). Рассмотрим, как придется задействовать наш класс `BookIT` во встроенном режиме Derby.

Maven использует два разных каталога, один из которых применяется для размещения кода главного приложения, а другой — для тестовых классов. Класс `BookIT`, показанный в листинге 4.9, располагается в каталоге `src/test/java` и осуществляет тестирование на предмет того, может ли менеджер сущностей обеспечить постоянство сущности `Book` и извлекать ее из базы данных, а также удостоверяется в том, что применяются ограничения Bean Validation.

Листинг 4.9. Тестовый класс, который создает и извлекает Book из базы данных

```
public class BookIT {
  private static EntityManagerFactory emf = ➥
                Persistence.createEntityManagerFactory("chapter04TestPU");
  private EntityManager em;
  private EntityTransaction tx;
  @Before
  public void initEntityManager() throws Exception {
    em = emf.createEntityManager();
    tx = em.getTransaction();
  }
  @After
  public void closeEntityManager() throws Exception {
    if (em != null) em.close();
  }
  @Test
  public void shouldFindjavaee7Book() throws Exception {
    Book book = em.find(Book.class, 1001L);
    assertEquals("Изучаем Java EE 7", book.getTitle());
  }
  @Test
  public void shouldCreateH2G2Book() throws Exception {
    // Создает экземпляр Book
    Book book = new Book("H2G2", "Автостопом по Галактике", ➥
    12.5F, "1-84023-742-2", 354, false);
    // Обеспечивает постоянство Book в базе данных
    tx.begin();
    em.persist(book);
    tx.commit();
    assertNotNull("ID не может быть пустым", book.getId());
```

```
    // Извлекает информацию обо всех соответствующих книгах из базы данных
    book = em.createNamedQuery("findBookH2G2", Book.class).getSingleResult();
    assertEquals("Автостопом по Галактике", book.getDescription());
  }
  @Test(expected = ConstraintViolationException.class)
  public void shouldRaiseConstraintViolationCauseNullTitle() {
    Book book = new Book(null, "Пустое название, ошибка", 12.5F, ➥
    "1-84023-742-2", 354, false);
    em.persist(book);
  }
}
```

Как и классу `Main`, `BookIT` в листинге 4.9 необходимо создать экземпляр `EntityManager` с использованием `EntityManagerFactory`. Для инициализации этих компонентов можно прибегнуть к фикстурам JUnit 4. Аннотации `@Before` и `@After` позволяют выполнять некоторый код до и после выполнения теста. Это идеальное место для создания и закрытия экземпляра `EntityManager` и получения транзакции.

Вариант тестирования `shouldFindJavaEE7Book()` опирается на информацию, которая уже присутствует в базе данных (подробнее о сценарии `inset.sql` мы поговорим позднее), и при поиске `Book` с идентификатором `1001` мы убеждается в том, что названием является `"Изучаем Java EE 7"`. Метод `shouldCreateH2G2Book()` обеспечивает постоянство `Book` (с помощью метода `EntityManager.persist()`) и проверяет, был ли идентификатор автоматически сгенерирован EclipseLink (с использованием `assertNotNull`). Если да, то выполняется именованный запрос и осуществляется проверка, является ли `"Автостопом по Галактике"` описанием возвращенной сущности `Book`. Последний вариант тестирования создает `Book` с атрибутом `Nulltitle`, обеспечивает постоянство `Book` и удостоверяется в том, что было сгенерировано исключение `ConstraintViolationException`.

Написание единицы сохраняемости

Как вы можете видеть в классе `Main` (см. листинг 4.8), `EntityManagerFactory` требуется единица сохраняемости с именем `chapter04PU`. А для интеграционного теста `BookIT` (см. листинг 4.9) используется другая единица сохраняемости (`chapter04TestPU`). Эти две единицы сохраняемости должны быть определены в файле `persistence.xml`, находящемся в каталоге `src/main/resources/META-INF` (листинг 4.10). Этот файл, необходимый согласно спецификации JPA, важен, поскольку соединяет поставщика JPA (которым в нашем случае выступает EclipseLink) с базой данных (Derby). Он содержит всю информацию, необходимую для подключения к базе данных (URL-адрес, JDBC-драйвер, сведения о пользователе, пароль), и сообщает поставщику режим генерирования схемы базы данных (`drop-and-create` означает, что таблицы будут удалены, а затем снова созданы). Элемент `<provider>` определяет поставщика постоянства, которым в нашем случае является EclipseLink. Единицы сохраняемости позволяют узнать обо всех сущностях, которыми должен управлять менеджер сущностей. Здесь в теге `<class>` предусмотрена ссылка на сущность `Book`.

Две единицы сохраняемости отличаются в том смысле, что `chapter04PU` использует работающую базу данных Derby, а `chapter04TestPU` — ту, что располагается в оперативной памяти. Обратите внимание, что они обе задействуют сценарий `inset.sql` для ввода информации в базу данных во время выполнения.

Листинг 4.10. Файл persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
             version="2.1">
  <persistence-unit name="chapter04PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>org.agoncal.book.javaee7.chapter04.Book</class>
    <properties>
      <property name="javax.persistence.schema-generation-action"
value="drop-and-create"/>
      <property name="javax.persistence.schema-generation-target"
                value="database-and-scripts"/>
      <property name="javax.persistence.jdbc.driver"
                value="org.apache.derby.jdbc.ClientDriver"/>
      <property name="javax.persistence.jdbc.url"
                value="jdbc:derby://localhost:1527/chapter04DB;create=true"/>
      <property name="javax.persistence.jdbc.user" value="APP"/>
      <property name="javax.persistence.jdbc.password" value="APP"/>
      <property name="javax.persistence.sql-load-script-source"
value="insert.sql"/>
    </properties>
  </persistence-unit>
  <persistence-unit name="chapter04TestPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>org.agoncal.book.javaee7.chapter04.Book</class>
    <properties>
      <property name="javax.persistence.schema-generation-action"
value="drop-and-create"/>
      <property name="javax.persistence.schema-generation-target"
value="database"/>
      <property name="javax.persistence.jdbc.driver"
                value="org.apache.derby.jdbc.EmbeddedDriver"/>
      <property name="javax.persistence.jdbc.url"
                value="jdbc:derby:memory:chapter04DB;create=true"/>
      <property name="javax.persistence.sql-load-script-source"
value="insert.sql"/>
    </properties>
  </persistence-unit>
</persistence>
```

Написание SQL-сценария для загрузки данных

Обе единицы сохраняемости, определенные в листинге 4.10, загружают сценарий `insert.sql` (с использованием свойства `javax.persistence.sql-loadscript-source`). Это означает, что сценарий, показанный в листинге 4.11, выполняется для инициализации базы данных и вводит информацию о трех книгах.

Листинг 4.11. Файл insert.sql

```
INSERT INTO BOOK(ID, TITLE, DESCRIPTION, ILLUSTRATIONS, ISBN, NBOFPAGE,
                 PRICE) values (1000, Изучаем Java EE 6',
                 'Лучшая книга о Java EE', 1, '1234-5678', 450, 49)
INSERT INTO BOOK(ID, TITLE, DESCRIPTION, ILLUSTRATIONS, ISBN, NBOFPAGE,
                 PRICE) values (1001, 'Изучаем Java EE 7', 'Нет, это лучшая',
                 1, '5678-9012', 550, 53)
INSERT INTO BOOK(ID, TITLE, DESCRIPTION, ILLUSTRATIONS, ISBN, NBOFPAGE,
                 PRICE) values (1010, 'Властелин колец', 'Одно кольцо
                 для управления всеми остальными', 0, '9012-3456', 222, 23)
```

Если вы внимательно посмотрите на интеграционный тест BookIT (метод shouldFindJavaEE7Book), то увидите, что в случае с ним ожидается, что в базе данных будет идентификатор Book в виде 1001. Благодаря инициализации базы данных соответствующие действия предпринимаются до выполнения тестов.

Компиляция и тестирование с использованием Maven

У нас есть все составляющие для компиляции и тестирования сущности перед выполнением приложения Main: сущность Book, интеграционный тест BookIT и единицы сохраняемости, которые привязывают эту сущность к базе данных Derby. Для компиляции этого кода вы воспользуетесь Maven вместо того, чтобы прибегать непосредственно к команде компилятора javac. Вам сначала потребуется создать файл pom.xml, описывающий проект и его зависимости вроде JPA и Bean Validation API. Вам также придется проинформировать Maven о том, что вы используете Java SE 7, сконфигурировав maven-compiler-plugin, как показано в листинге 4.12.

Листинг 4.12. Maven-файл pom.xml для компиляции, тестирования и выполнения приложения

```
<?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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <artifactId>parent</artifactId>
    <groupId>org.agoncal.book.javaee7</groupId>
    <version>1.0</version>
  </parent>
  <groupId>org.agoncal.book.javaee7</groupId>
  <artifactId>chapter04</artifactId>
  <version>1.0</version>

  <dependencies>
    <dependency>
```

```
      <groupId>org.eclipse.persistence</groupId>
      <artifactId>org.eclipse.persistence.jpa</artifactId>
      <version>2.5.0</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-validator</artifactId>
      <version>5.0.0</version>
    </dependency>
    <dependency>
      <groupId>org.apache.derby</groupId>
      <artifactId>derbyclient</artifactId>
      <version>10.9.1.0</version>
    </dependency>
    <dependency>
      <groupId>org.apache.derby</groupId>
      <artifactId>derby</artifactId>
      <version>10.9.1.0</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.5.1</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>2.12.4</version>
        <executions>
          <execution>
            <id>integration-test</id>
            <goals>
              <goal>integration-test</goal>
              <goal>verify</goal>
            </goals>
          </execution>
```

```
        </executions>
      </plugin>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>exec-maven-plugin</artifactId>
        <version>1.2.1</version>
        <executions>
          <execution>
            <goals>
              <goal>java</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <mainClass>org.agoncal.book.javaee7.chapter04.Main</mainClass>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Прежде всего, чтобы вы смогли произвести компиляцию кода, вам потребуется JPA API, который определяет все аннотации и классы, имеющиеся в пакете `javax.persistence`. Все это, а также время выполнения EclipseLink (то есть поставщика постоянства) будет обеспечиваться с помощью идентификатора артефакта `org.eclipse.persistence.jpa`. Как можно было видеть в предыдущей главе, Bean Validation API заключен в артефакт `hibernate-validator`. Кроме того, вам потребуются JDBC-драйверы для подключения к Derby. Идентификатор артефакта `derbyclient` ссылается на файл с расширением `.jar`, содержащий JDBC-драйвер для подключения к Derby, работающей в серверном режиме (база данных функционирует как отдельный процесс и прослушивает порт), а идентификатор артефакта `derby` включает в себя классы для использования Derby как встраиваемой системы управления базами данных. Обратите внимание, что область этого идентификатора артефакта ограничивается тестированием (`<scope>test</scope>`), как и артефакта для JUnit 4.

Для компиляции классов откройте интерпретатор командной строки в корневом каталоге, содержащем файл `pom.xml`, и введите приведенную далее Maven-команду:

```
$ mvn compile
```

После этого вы должны будете увидеть сообщение `BUILD SUCCESS`, говорящее о том, что компиляция прошла успешно. Maven создаст подкаталог target со всеми файлами наряду с `persistence.xml`. Для выполнения интеграционных тестов снова прибегните к Maven, введя следующую команду:

```
$ mvn integration-test
```

Вы должны будете увидеть несколько журналов касаемо Derby при создании базы данных и таблиц в памяти. Затем окажется задействован класс `BookIT`, а отчет Maven проинформирует вас о том, что результаты применения трех вариантов тестирования оказались успешными:

```
Results :
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 5.192s
[INFO] Finished
[INFO] Final Memory: 18M/221M
[INFO] ------------------------------------------------------------------------
```

Применение класса Main с использованием Maven

Перед тем как применять класс `Main`, вам потребуется запустить Derby. Самый легкий способ сделать это — открыть каталог `$DERBY_HOME/bin` и выполнить сценарий `startNetworkServer`. Derby запустится и выведет в консоли следующие сообщения:

```
Security manager installed using the Basic server security policy.
Apache Derby Network Server - 10.9.1.0 - (802917) started and ready to accept
connections on port 1527
```

Процесс Derby прослушивает порт 1527 и ждет, когда JDBC-драйвер отправит какой-нибудь SQL-оператор. Для применения класса `Main` вы можете воспользоваться командой интерпретатора `java` либо прибегнуть к `exec-maven-plugin`, как показано далее:

```
$ mvn exec:java
```

В результате применения класса `Main` произойдет несколько вещей. Прежде всего Derby автоматически создаст базу данных `chapter04DB`, как только будет инициализирована сущность `Book`. Это случится потому, что в файле `persistence.xml` вы добавили свойство `create=true` в URL-адрес JDBC:

```
<property name="javax.persistence.jdbc.url" ➥
          value="jdbc:derby://localhost:1527/chapter04DB;create=true"/>
```

Это сокращение очень полезно, когда вы находитесь в режиме разработки, поскольку вам не требуются никакие SQL-сценарии для создания базы данных. Затем свойство `javax.persistence.schema-generation-action` проинформирует EclipseLink о необходимости автоматически удалить и заново создать таблицу `BOOK`. И наконец, сущность `Book` будет вставлена в таблицу (с автоматически сгенерированным идентификатором).

Воспользуемся Derby-командами для вывода на экран структуры таблицы: введем команду `ij` в консоли (каталог `$DERBY_HOME/bin` должен быть в вашей переменной `PATH`). Это приведет к запуску интерпретатора Derby, и вы сможете выполнить команды для подключения к базе данных, вывести на экран таблицы базы данных `chapter04DB` (`show tables`), проверить структуру таблицы `BOOK` (`describebook`) и даже увидеть ее содержимое, введя SQL-операторы, например `SELECT * FROM BOOK`.

```
$ ij
version 10.9.1.0
```

```
ij> connect 'jdbc:derby://localhost:1527/chapter04DB';
ij> show tables;
TABLE_SCHEM     |TABLE_NAME     |REMARKS
------------------------------------------------------------------------
APP             |BOOK           |
APP             |SEQUENCE       |
ij> describe book;
COLUMN_NAME   |TYPE_NAME|DEC&|NUM&|COLUM&|COLUMN_DEF|CHAR_OCTE&|IS_NULL&
------------------------------------------------------------------------
ID            |BIGINT   |0   |10  |19    |NULL      |NULL      |NO
TITLE         |VARCHAR  |NULL|NULL|255   |NULL      |510       |YES
PRICE         |DOUBLE   |NULL|2   |52    |NULL      |NULL      |YES
ILLUSTRATIONS |SMALLINT |0   |10  |5     |0         |NULL      |YES
DESCRIPTION   |VARCHAR  |NULL|NULL|255   |NULL      |510       |YES
ISBN          |VARCHAR  |NULL|NULL|255   |NULL      |510       |YES
NBOFPAGE      |INTEGER  |0   |10  |10    |NULL      |NULL      |YES
```

Возвращаясь к коду сущности `Book` (см. листинг 4.7), отмечу, что, поскольку вы использовали аннотацию `@GeneratedValue` (для автоматического генерирования идентификатора), EclipseLink создал таблицу последовательности для сохранения нумерации (таблица `SEQUENCE`). В случае со структурой таблицы `BOOK` JPA придерживался определенных соглашений по умолчанию, присваивая имена таблице и столбцам в соответствии с именем сущности и атрибутов (например, `String` отображается в `VARCHAR(255)`).

Проверка сгенерированной схемы

В файле `persistence.xml`, описанном в листинге 4.10, мы проинформировали EclipseLink о необходимости сгенерировать схему базы данных и сценарии для удаления и создания с помощью следующего свойства:

```
<property name="javax.persistence.schema-generation.database.action" ➥
          value="drop-and-create"/>
<property name="javax.persistence.schema-generation.scripts.action" ➥
          value="drop-and-create"/>
```

По умолчанию поставщик постоянства сгенерирует два SQL-сценария: `createDDL.jdbc` (листинг 4.13) со всеми SQL-операторами для создания всей базы данных целиком и `dropDDL.jdbc` (листинг 4.14) для удаления всех таблиц. Это удобно, когда вам необходимо выполнить сценарии с целью создания базы данных в своем процессе непрерывной интеграции.

Листинг 4.13. Сценарий createDDL.jdbc

```
CREATE TABLE BOOK (ID BIGINT NOT NULL, DESCRIPTION VARCHAR(255), ➥
                   ILLUSTRATIONS SMALLINT DEFAULT 0, ISBN VARCHAR(255), ➥
NBOFPAGE INTEGER, PRICE FLOAT, TITLE VARCHAR(255), PRIMARY KEY (ID))
CREATE TABLE SEQUENCE (SEQ_NAME VARCHAR(50) NOT NULL, SEQ_COUNT DECIMAL(15), ➥
                       PRIMARY KEY (SEQ_NAME))
INSERT INTO SEQUENCE (SEQ_NAME, SEQ_COUNT) values ('SEQ_GEN', 0)
```

Листинг 4.14. Сценарий dropDDL.jdbc

```
DROP TABLE BOOK
DELETE FROM SEQUENCE WHERE SEQ_NAME = 'SEQ_GEN'
```

Резюме

В данной главе приведен беглый обзор JPA 2.1. Как и большинство других спецификаций Java EE 7, JPA сосредоточен на простой объектной архитектуре, оставляя позади своего предка — тяжеловесную компонентную модель (также известную как EJB CMP 2.*x*). В этой главе также рассмотрены сущности, которые представляют собой постоянные объекты, отображающие метаданные с помощью аннотаций или XML.

Благодаря разделу «Все вместе» вы увидели, как следует выполнять JPA-приложения с использованием EclipseLink и Derby. Интеграционное тестирование — важная задача в проектах, а с помощью JPA и баз данных в оперативной памяти вроде Derby сейчас стало очень легко проводить тесты касаемо постоянства.

Из последующих глав вы больше узнаете об основных компонентах JPA. В главе 5 вы увидите, как отображать сущности, связи и наследование в базу данных. А в главе 6 внимание будет сосредоточено на API менеджера сущностей, синтаксисе JPQL. Вы узнаете о том, как использовать запросы и механизмы блокировки, а также будет объяснен жизненный цикл сущностей и то, как добавлять бизнес-логику, когда речь идет о методах сущностей и слушателях.

Глава 5

Объектно-реляционное отображение

В предыдущей главе я подробно рассказал вам об основах объектно-реляционного отображения (Object-Relational Mapping — ORM), которое по сути является отображением сущностей в таблицах и атрибутов в столбцах. Я также поведал вам о конфигурации в порядке исключения, которая позволяет поставщику JPA отображать сущность в таблицу базы данных с использованием всех правил по умолчанию. Однако такие правила не всегда оказываются подходящими, особенно если вы отображаете свою доменную модель в существующую базу данных. JPA сопутствует богатый набор метаданных, благодаря чему вы можете настраивать отображение.

В этой главе я рассмотрю элементарное отображение, а также сконцентрируюсь на более сложных отображениях, таких как отображение связей, композиции и наследования. Доменная модель состоит из объектов, которые взаимодействуют друг с другом. У объектов и баз данных имеются разные способы сохранения информации о связях (ссылок в объектах и внешних ключей в базах данных). Наследование не является чертой, которая от природы имеется у реляционных баз данных, и, следовательно, отображение не столь очевидно. В этой главе я тщательно разберу некоторые подробности и приведу примеры, демонстрирующие то, как следует отображать атрибуты, связи и наследования из доменной модели в базе данных.

Элементарное отображение

Способ, которым Java обрабатывает данные, значительно отличается, если сравнивать его с подходом к обработке информации, используемым реляционными базами данных. В случае с Java мы применяем классы для описания как атрибутов для размещения данных, так и методов для доступа и манипулирования данными. Определив класс, мы можем создать столько его экземпляров, сколько нам потребуется, указав ключевое слово `new`. В реляционной базе данных информация хранится в структурах (столбцах и строках), не являющихся объектными, а динамическое поведение представляет собой хранимую функциональность вроде триггеров таблиц и хранимых процедур, которые не связаны тесно со структурами данных так, как с объектами. Иногда отображение Java-объектов в основной базе данных может быть легким, при этом могут применяться правила по умолчанию.

А иной раз эти правила не отвечают вашим нуждам и приходится настраивать отображение. Аннотации элементарного отображения сосредоточены на настройке требуемой таблицы, первичного ключа и столбцов и позволяют вам модифицировать определенные соглашения об именовании или типизацию (речь может идти о не имеющем значения `null` столбце, длине и т. д.).

Таблицы

Согласно правилам отображения с использованием подхода «конфигурация в порядке исключения» имена сущности и таблицы должны совпадать (сущность `Book` будет отображаться в таблице `BOOK`, сущность `AncientBook` — в таблице `ANCIENTBOOK` и т. д.). Это, возможно, будет устраивать вас в большинстве ситуаций, однако вам может потребоваться отобразить свои данные в другой таблице или даже отобразить одну сущность в нескольких таблицах.

@Table

Аннотация `@javax.persistence.Table` дает возможность изменять значения по умолчанию, связанные с определенной таблицей. Например, вы можете указать имя таблицы, в которой будут храниться данные, каталог и схему базы данных. Вы также можете определить уникальные ограничения для таблицы с помощью аннотации `@UniqueConstraint` в сочетании с `@Table`. Если пренебречь аннотацией `@Table`, то имя таблицы будет соответствовать имени сущности. Если вы захотите изменить имя с `BOOK` на `T_BOOK`, то поступите так, как показано в листинге 5.1.

Листинг 5.1. Сущность Book, отображаемая в таблице T_BOOK

```
@Entity
@Table(name = "t_book")
public class Book {

@Id
  private Long id;
  private String title;
  private Float price;
  private String description;
  private String isbn;
  private Integer nbOfPage;
  private Boolean illustrations;

  // Конструкторы, геттеры, сеттеры
}
```

ПРИМЕЧАНИЕ

Я включил в аннотацию @Table имя таблицы в нижнем регистре (t_book). По умолчанию большинство баз данных будут отображать имя сущности в имя таблицы в верхнем регистре (именно так дело обстоит в случае с Derby), если только вы не сконфигурируете их на соблюдение регистра.

@SecondaryTable

До сих пор я исходил из того, что сущность будет отображаться в одну таблицу, также известную как *первичная таблица*. Но если у вас есть уже существующая модель данных, необходимо разбросать данные по нескольким таблицам, или *вторичным таблицам*. Для этого вам потребуется прибегнуть к аннотации `@SecondaryTable`, чтобы ассоциировать вторичную таблицу с сущностью, или `@SecondaryTables` (с буквой s на конце) в случае с несколькими вторичными таблицами. Вы можете распределить данные требуемой сущности по столбцам как первичной таблицы, так и вторичных таблиц, просто определив вторичные таблицы с использованием аннотаций, а затем указав для каждого атрибута то, к какой таблице он относится (с помощью аннотации `@Column`, более подробное описание которой я приведу в подразделе «@Column»). В листинге 5.2 показано отображение атрибутов сущности `Address` в одну первичную таблицу и две вторичные таблицы.

Листинг 5.2. Атрибуты сущности Address, отображаемые в три разные таблицы

```
@Entity
@SecondaryTables({
  @SecondaryTable(name = "city"),
 @SecondaryTable(name = "country")
})
public class Address {

  @Id
  private Long id;
  private String street1;
  private String street2;
  @Column(table = "city")
  private String city;
  @Column(table = "city")
  private String state;
  @Column(table = "city")
  private String zipcode;
  @Column(table = "country")
  private String country;

  // Конструкторы, геттеры, сеттеры
}
```

По умолчанию атрибуты сущности `Address` будут отображаться в первичную таблицу (которая по умолчанию имеет имя сущности, поэтому называется `ADDRESS`). Аннотация `@SecondaryTables` проинформирует вас о том, что в данном случае есть две вторичные таблицы: `CITY` и `COUNTRY`. Затем вам потребуется указать, какой атрибут в какой таблице будет располагаться (с использованием аннотации `@Column(table="city")` или `@Column(table="country")`). На рис. 5.1 показана структура таблиц, в которых будет отображаться сущность `Address`. Каждая таблица содержит разные атрибуты, однако у всех имеется один и тот же первичный ключ (для соединения таблиц). Опять-таки не забывайте, что Derby преобразует имена таблиц в нижнем регистре (`city`) в имена таблиц в верхнем регистре (`CITY`).

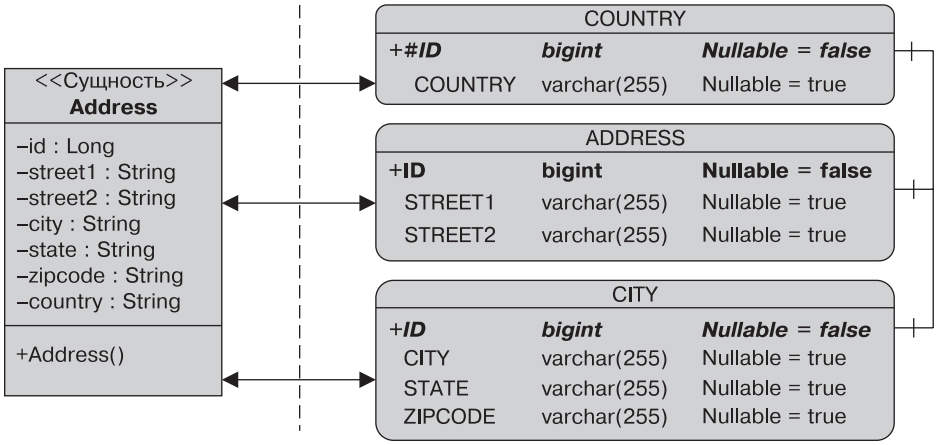


Рис. 5.1. Сущность Address отображается в трех таблицах

Как вы, вероятно, уже поняли, для одной и той же сущности может быть несколько аннотаций. Если вы захотите переименовать первичную таблицу, то можете добавить аннотацию `@Table`, как показано в листинге 5.3.

Листинг 5.3. Первичная таблица переименовывается в T_ADDRESS

```
@Entity
@Table(name = "t_address")
@SecondaryTables({
  @SecondaryTable(name = "t_city"),
  @SecondaryTable(name = "t_country")
})
public class Address {

  // Атрибуты, конструктор, геттеры, сеттеры
}
```

ПРИМЕЧАНИЕ

При использовании вторичных таблиц вы должны принимать во внимание производительность. Каждый раз, когда вы получаете доступ к сущности, поставщик постоянства обращается к нескольким таблицам, которые ему приходится соединять. С другой стороны, вторичные таблицы могут оказаться кстати, когда у вас имеются «затратные» атрибуты вроде больших двоичных объектов (Binary Large Object — BLOB), которые вы хотите изолировать в другой таблице.

Первичные ключи

Первичные ключи в реляционных базах данных уникально идентифицируют все строки таблиц. Первичный ключ охватывает либо один столбец, либо набор столбцов. Такой ключ должен быть уникальным, поскольку он идентифицирует одну строку (значение `null` не допускается). Примерами первичных ключей являются идентификатор клиента, телефонный номер, номер получения, а также ISBN-номер. JPA требует, чтобы у сущностей был идентификатор, отображаемый в пер-

вичный ключ, который будет придерживаться аналогичного правила: уникально идентифицировать сущность с помощью либо одного атрибута, либо набора атрибутов (составной ключ). Значение этого первичного ключа сущности нельзя обновить после того, как оно было присвоено.

@Id и @GeneratedValue

Простой (то есть не являющийся составным) первичный ключ должен соответствовать одному атрибуту класса-сущности. Аннотация `@Id`, которую вы видели ранее, используется для обозначения простого первичного ключа. `@javax.persistence.Id` аннотирует атрибут как уникальный идентификатор. Он может относиться к одному из таких типов, как:

- примитивные Java-типы: `byte`, `int`, `short`, `long`, `char`;
- классы-обертки примитивных Java-типов: `Byte`, `Integer`, `Short`, `Long`, `Character`;
- массивы примитивных типов или классов-адаптеров: `int[]`, `Integer[]` и т. д.;
- строки, числа и даты: `java.lang.String`, `java.math.BigInteger`, `java.util.Date`, `java.sql.Date`.

При создании сущности значение этого идентификатора может быть сгенерировано либо вручную с помощью приложения, либо автоматически поставщиком постоянства с использованием аннотации `@GeneratedValue`. Эта аннотация способна иметь четыре возможных значения:

- `SEQUENCE` и `IDENTITY` определяют использование SQL-последовательности базы данных или столбца идентификаторов соответственно;
- `TABLE` дает указание поставщику постоянства сохранить имя последовательности и ее текущее значение в таблице, увеличивая это значение каждый раз, когда будет обеспечиваться постоянство нового экземпляра сущности. Например, EclipseLink создаст таблицу `SEQUENCE` с двумя столбцами: в одном будет имя последовательности (произвольное), а в другом — значение последовательности (целое число, автоматически инкрементируемое Derby);
- генерирование ключа выполняется автоматически (`AUTO`) основным поставщиком постоянства, который выберет подходящую стратегию для определенной базы данных (EclipseLink будет использовать стратегию `TABLE`). `AUTO` является значением по умолчанию аннотации `@GeneratedValue`.

Если аннотация `@GeneratedValue` не будет определена, приложению придется создать собственный идентификатор, применив любой алгоритм, который возвратит уникальное значение. В коде, приведенном в листинге 5.4, показано, как получить автоматически генерируемый идентификатор. Будучи значением по умолчанию, `GenerationType.AUTO` позволило бы мне не включать в этот код элемент `strategy`. Обратите внимание, что атрибут `id` аннотирован дважды: один раз с использованием `@Id` и еще один раз — посредством `@GeneratedValue`.

Листинг 5.4. Сущность Book с автоматически генерируемым идентификатором

```
@Entity
public class Book {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String title;
    private Float price;
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;

    // Конструкторы, геттеры, сеттеры
}
```

Составные первичные ключи

При отображении сущностей правильным подходом будет обозначить один специально отведенный для этого столбец как первичный ключ. Однако бывают ситуации, когда требуется составной первичный ключ (скажем, если приходится выполнять отображение в унаследованную базу данных или первичные ключи должны придерживаться определенных бизнес-правил — например, необходимо включить значение или код страны и отметку времени). Для этого должен быть определен класс первичного ключа, который будет представлять составной ключ. Кроме того, у нас есть две доступные аннотации для определения этого класса в зависимости от того, как мы хотим структурировать сущность: `@EmbeddedId` и `@IdClass`. Как вы еще увидите, конечный результат будет одинаковым и в итоге у вас окажется одна и та же схема базы данных, однако способы выполнения запросов к сущности будут немного различаться.

Например, приложению CD-BookStore необходимо часто выкладывать информацию на главной странице, где вы сможете читать ежедневные новости о книгах, музыке или артистах. У новостей будет иметься содержимое, название и, поскольку они могут быть написаны на нескольких языках, код языка (`EN` для английского, `PT` — для португальского и т. д.). Первичным ключом для новостей тогда сможет быть название и код языка, поскольку статья может быть переведена на разные языки, но с сохранением ее оригинального названия. Таким образом, класс первичного ключа `NewsId` будет включать два атрибута, имеющие тип `String`: `title` и `language`. Классы первичных ключей должны включать определения методов для `equals()` и `hashCode()` для управления запросами и внутренними коллекциями (равенство в случае с этими методами должно быть таким же, как и равенство, которое имеет место в случае с базой данных), а их атрибуты должны быть допустимых типов, входящих в приведенный ранее набор. Они также должны быть открытыми, реализовывать интерфейс `Serializable`, если им потребуется пересекать архитектурные уровни (например, управление ими будет осуществляться на постоянном уровне, а использование — на уровне представления), и располагать конструктором без аргументов.

@EmbeddedId. Как вы еще увидите позднее в этой главе, JPA использует встроенные объекты разных типов. Резюмируя, отмечу, что у встроенного объекта нет какого-либо идентификатора (собственного первичного ключа), а его атрибуты в итоге окажутся столбцами таблицы содержащей их сущности.

В листинге 5.5 класс NewsId показан как встраиваемый. Он представляет собой всего лишь встроенный объект (аннотированный с использованием @Embeddable), который в данном случае включает два атрибута (title и language). У этого класса должен быть конструктор без аргументов, геттер, сеттер, а также реализации equals() и hashCode(). Это означает, что он должен придерживаться соглашений JavaBeans. У этого класса как такового нет собственного идентификатора (отсутствует аннотация @Id). Это характерно для встраиваемых классов.

Листинг 5.5. Класс первичного ключа снабжается аннотацией @Embeddable

```
@Embeddable
public class NewsId {

  private String title;
  private String language;

  // Конструкторы, геттеры, сеттеры
}
```

В случае с сущностью News, показанной в листинге 5.6, затем придется встроить класс первичного ключа NewsId с применением аннотации @EmbeddedId. Благодаря такому подходу не потребуется использовать @Id. Каждая аннотация @EmbeddedId должна ссылаться на встраиваемый класс, помеченный @Embeddable.

Листинг 5.6. Вложение класса первичного ключа с применением аннотации @EmbeddedId для сущности

```
@Entity
public class News {

  @EmbeddedId
  private NewsId id;
  private String content;

  // Конструкторы, геттеры, сеттеры
}
```

В следующей главе я опишу, как находить сущности с использованием их первичного ключа. Вот первый, но мимолетный взгляд на то, как это происходит: первичный ключ — это класс с конструктором. Вам придется создать экземпляр этого класса со значениями, формирующими ваш уникальный ключ, и передать соответствующий объект менеджера сущностей (атрибут em), как показано в этом коде:

```
NewsId pk = new NewsId("Richard Wright has died on September 2008", "EN")
News news = em.find(News.class, pk);
```

@IdClass. Другой метод объявления составного ключа — использование аннотации @IdClass. Это иной подход, в соответствии с которым каждый атрибут класса первичного ключа также должен быть объявлен в классе-сущности и аннотирован с помощью @Id.

Составной первичный ключ в примере NewsId, приведенном в листинге 5.7, является всего лишь POJO, которому не требуется какая-либо аннотация (в предыдущем

примере в листинге 5.6 класс первичного ключа приходилось аннотировать с помощью `@EmbeddedId`).

Листинг 5.7. У класса первичного ключа аннотация отсутствует

```
public class NewsId {

private String title;
  private String language;

  // Конструкторы, геттеры, сеттеры
}
```

Затем для сущности `News`, показанной в листинге 5.8, придется определить первичный ключ с использованием аннотации `@IdClass`, и аннотировать каждый ключ посредством `@Id`. Для обеспечения постоянства сущности `News` вам потребуется задать значения для атрибутов `title` и `language`.

Листинг 5.8. Определение первичного класса с использованием аннотации @IdClass для сущности

```
@Entity
@IdClass(NewsId.class)
public class News {

  @Id private String title;
  @Id private String language;
  private String content;

  // Конструкторы, геттеры, сеттеры
}
```

Оба варианта — `@EmbeddedId` и `@IdClass` — будут отображены в одну и ту же табличную структуру. Она определена в листинге 5.9 с использованием языка описания данных (Data Definition Language — DDL). Атрибуты сущности и первичный ключ в итоге окажутся в одной и той же таблице, а первичный ключ будет сформирован с использованием атрибутов составного класса (`title` и `language`).

Листинг 5.9. DDL-код таблицы NEWS с составным первичным ключом

```
create table NEWS (
    CONTENT VARCHAR(255),
    TITLE VARCHAR(255) not null,
    LANGUAGE VARCHAR(255) not null,
    primary key (TITLE, LANGUAGE)
);
```

Подход с использованием `@IdClass` более предрасположен к ошибкам, поскольку вам потребуется определить каждый атрибут первичного ключа как в `@IdClass`, так и в сущности, позаботившись об использовании одинакового имени и Java-типа. Преимущество заключается в том, что вам не придется изменять код класса первичного ключа (никаких аннотаций не потребуется). Например, вы могли бы использовать унаследованный класс, который по причинам правового характера вам

не разрешено изменять, однако при этом вам предоставляется возможность его повторного использования.

Одно из видимых отличий заключается в способе, которым вы ссылаетесь на сущность при использовании JPQL. В случае с `@IdClass` вы сделали бы что-то вроде следующего:

```
select n.title from News n
```

А в случае с `@EmbeddedId` у вас получилось бы что-то вроде показанного далее:

```
select n.newsId.title from News n
```

Атрибуты

У сущности должен иметься первичный ключ (простой или составной), чтобы у нее был идентификатор в реляционной базе данных. Она также обладает всевозможными атрибутами, которые обуславливают ее состояние и должны быть отображены в таблицу. Это состояние способно включать почти любой Java-тип, который вам может потребоваться отобразить:

- примитивные Java-типы и классы-обертки (`int`, `double`, `float` и т. д.) (`Integer`, `Double`, `Float` и т. д.);
- массивы байтов и символов (`byte[]`, `Byte[]`, `char[]`, `Character[]`);
- строковые, связанные с большими числами, и временные типы (`java.lang.String`, `java.math.BigInteger`, `java.math.BigDecimal`, `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`);
- перечислимые типы, а также определяемые пользователем типы, которые реализуют интерфейс `Serializable`;
- коллекции базовых и встраиваемых типов.

Разумеется, у сущности также могут иметься атрибуты сущности, коллекции сущностей или встраиваемые классы. Это требует рассказа о связях между сущностями (которые будут рассмотрены в разделе «Отображение связей»).

Как вы уже видели ранее, при конфигурации в порядке исключения атрибуты отображаются с использованием правил отображения по умолчанию. Однако иногда возникает необходимость настроить детали этого отображения. Именно здесь в дело вступают аннотации JPA (или их XML-эквиваленты).

@Basic

Опциональная аннотация `@javax.persistence.Basic` (листинг 5.10) является отображением в столбец базы данных, относящимся к самому простому типу, поскольку она переопределяет базовое постоянство.

Листинг 5.10. Элементы аннотации @Basic

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Basic {
  FetchType fetch() default EAGER;
  boolean optional() default true;
}
```

У этой аннотации есть два параметра: optional и fetch. Элемент optional подсказывает вам, может ли null быть значением атрибута. Однако он игнорируется для примитивных типов. Элемент fetch может принимать два значения: LAZY или EAGER. Он намекает на то, что во время выполнения поставщика постоянства выборка данных должна быть отложенной (только когда приложение запрашивает соответствующее свойство) или быстрой (когда сущность изначально загружается поставщиком).

Возьмем, к примеру, сущность Track, показанную в листинге 5.11. В CD-альбом входит несколько треков, имеющих название, описание и WAV-файл определенной длительности, который вы можете прослушать. WAV-файл представляет собой BLOB-объект, который может иметь объем несколько мегабайт. Вам не нужно, чтобы при доступе к сущности Track WAV-файл тут же быстро загружался; вы можете снабдить атрибут аннотацией @Basic(fetch = FetchType.LAZY), и извлечение информации из базы данных будет отложенным (например, только при доступе к атрибуту wav с использованием его геттера).

Листинг 5.11. Сущность Entity с отложенной загрузкой для атрибута wav

```
@Entity
public class Track {

  @Id @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  private String title;
  private Float duration;
  @Basic(fetch = FetchType.LAZY)
  @Lob
  private byte[] wav;
  private String description;

  // Конструкторы, геттеры, сеттеры
}
```

Обратите внимание, что атрибут wav типа byte[] также аннотирован с помощью @Lob для сохранения значения как большого объекта (Large Object — LOB). Столбцы базы данных, в которых могут храниться большие объекты такого типа, требуют специальных JDBC-вызовов для доступа к ним из Java-кода. Для информирования поставщика в случае с базовым отображением должна быть добавлена опциональная аннотация @Lob.

@Column

Аннотация @javax.persistence.Column, показанная в листинге 5.12, определяет свойства столбца. Вы можете изменить имя столбца (которое по умолчанию отображается в совпадающее с ним имя атрибута), а также указать размер и разрешить (или запретить) столбцу иметь значение null, быть уникальным или позволить его значению быть обновляемым или вставляемым. В листинге 5.12 приведен API-интерфейс аннотации @Column с элементами и их значениями по умолчанию.

Листинг 5.12. Элементы аннотации @Column

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Column {
```

```
  String name() default "";
  boolean unique() default false;
  boolean nullable() default true;
  boolean insertable() default true;
  boolean updatable() default true;
  String columnDefinition() default "";
  String table() default "";
  int length() default 255;
  int precision() default 0; // десятичная точность
  int scale() default 0;     // десятичная система счисления
}
```

Для переопределения отображения по умолчанию оригинальной сущности `Book` (см. листинг 5.1) вы можете использовать аннотацию `@Column` разными способами (листинг 5.13). Например, вы можете изменить имя столбца `title` и `nbOfPage` либо длину `description` и не разрешить значения `null`. Следует отметить, что допустимы не все комбинации атрибутов для типов данных (например, `length` применим только к столбцу со строковым значением, а `scale` и `precision` — только к десятичному столбцу).

Листинг 5.13. Настройка отображения сущности Book

```
@Entity
public class Book {
  @Id @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  @Column(name = "book_title", nullable = false, updatable = false)
  private String title;
  private Float price;
  @Column(length = 2000)
  private String description;
  private String isbn;
  @Column(name = "nb_of_page", nullable = false)
  private Integer nbOfPage;
  private Boolean illustrations;
  // Конструкторы, геттеры, сеттеры
}
```

Сущность `Book` из листинга 5.13 будет отображена в определение таблицы, показанное в листинге 5.14.

Листинг 5.14. Определение таблицы BOOK

```
create table BOOK (
  ID BIGINT not null,
  BOOK_TITLE VARCHAR(255) not null,
  PRICE DOUBLE(52, 0),
  DESCRIPTION VARCHAR(2000),
  ISBN VARCHAR(255),
  NB_OF_PAGE INTEGER not null,
  ILLUSTRATIONS SMALLINT,
  primary key (ID)
);
```

Большинство элементов аннотации @Column влияют на отображение. Если вы измените значение длины description на 2000, то значение длины целевого столбца тоже будет задано как равное 2000. Параметры updatable и insertable по умолчанию имеют значение true, которое подразумевает, что любой атрибут может быть вставлен или обновлен в базе данных, а также оказывают влияние во время выполнения. Вы сможете задать для них значение false, когда вам понадобится, чтобы поставщик постоянства позаботился о том, что он не вставит или не обновит данные в таблице в ответ на изменения в сущности. Следует отметить, что это не подразумевает, что атрибут сущности не изменится в памяти. Вы по-прежнему сможете изменить соответствующее значение, однако оно не будет синхронизировано с базой данных. Причина этого состоит в том, что сгенерированный SQL-оператор (INSERT или UPDATE) не будет включать столбцы. Другими словами, эти элементы не влияют на реляционное отображение, но влияют на динамическое поведение менеджера сущностей при доступе к реляционным данным.

ПРИМЕЧАНИЕ

Как можно было видеть в главе 3, Bean Validation определяет ограничения только в рамках пространства Java. Поэтому @NotNull обеспечивает считывание фрагмента Java-кода, который убеждается в том, что значением атрибута не является null. С другой стороны, аннотация JPA @Column(nullable = false) используется в пространстве базы данных для генерирования схемы базы данных. Аннотации JPA и Bean Validation могут сосуществовать в случае с тем или иным атрибутом.

@Temporal

При использовании Java вы можете задействовать java.util.Date и java.util.Calendar, чтобы сохранить данные и, кроме того, получить в свое распоряжение несколько их представлений, например, в виде даты, часа или миллисекунд. Чтобы указать это при объектно-реляционном отображении, вы можете воспользоваться аннотацией @javax.persistence.Temporal. Она принимает три возможных значения: DATE, TIME или TIMESTAMP (например, текущая дата, только время или и то и другое). В листинге 5.15 определена сущность Customer, у которой имеется атрибут dateOfBirth, а также технический атрибут, обеспечивающий сохранение значения точного времени ее создания в системе (в данной ситуации используется значение TIMESTAMP).

Листинг 5.15. Сущность Customer с двумя атрибутами @Temporal

```
@Entity
public class Customer {
  @Id @GeneratedValue
  private Long id;
  private String firstName;
  private String lastName;
  private String email;
  private String phoneNumber;
  @Temporal(TemporalType.DATE)
  private Date dateOfBirth;
  @Temporal(TemporalType.TIMESTAMP)
  private Date creationDate;
  // Конструкторы, геттеры, сеттеры
}
```

Сущность Customer из листинга 5.15 будет отображена в таблице, определенной в листинге 5.16. Атрибут dateOfBirth будет отображен в столбце типа DATE, а creationDate — в столбце типа TIMESTAMP.

Листинг 5.16. Определение таблицы CUSTOMER

```
create table CUSTOMER (
  ID BIGINT not null,
  FIRSTNAME VARCHAR(255),
  LASTNAME VARCHAR(255),
  EMAIL VARCHAR(255),
  PHONENUMBER VARCHAR(255),
  DATEOFBIRTH DATE,
  CREATIONDATE TIMESTAMP,
  primary key (ID)
);
```

@Transient

При использовании JPA, как только класс окажется снабжен аннотацией @Entity, все его атрибуты будут автоматически отображены в таблице. Если вам не нужно, чтобы атрибут отображался, вы можете воспользоваться аннотацией @javax.persistence.Transient или ключевым Java-словом transient. К примеру, возьмем упоминавшуюся выше сущность Customer и добавим атрибут age (листинг 5.17). Поскольку возраст может быть автоматически вычислен исходя из даты рождения, атрибут age не нужно отображать и, следовательно, он может быть временным.

Листинг 5.17. Сущность Customer с временным атрибутом age

```
@Entity
public class Customer {
  @Id @GeneratedValue
  private Long id;
  private String firstName;
  private String lastName;
  private String email;
  private String phoneNumber;
  @Temporal(TemporalType.DATE)
  private Date dateOfBirth;
  @Transient
  private Integer age;
  @Temporal(TemporalType.TIMESTAMP)
  private Date creationDate;
  // Конструкторы, геттеры, сеттеры
}
```

В результате атрибут age не нужно отображать в каком-либо столбце AGE.

@Enumerated

В Java SE 5 были представлены перечислимые типы, которые теперь настолько часто используются, что обычно становятся частью жизни разработчиков. Значения перечислимых типов представляют собой константы, которым неявно присваиваются порядковые номера согласно той последовательности, где они объявляются.

Такой порядковый номер нельзя модифицировать во время выполнения, однако его можно использовать для сохранения значения перечислимого типа в базе данных. В листинге 5.18 показано перечисление CreditCardType.

Листинг 5.18. Перечисление CreditCardType

```
public enum CreditCardType {
  VISA,
  MASTER_CARD,
  AMERICAN_EXPRESS
}
```

Порядковыми номерами, присвоенными значениям этого перечислимого типа во время компиляции, являются 0 для VISA, 1 для MASTER_CARD и 2 для AMERICAN_EXPRESS. По умолчанию поставщики постоянства будут отображать этот перечислимый тип в базе данных, предполагая, что соответствующий столбец имеет тип Integer. Взглянув на код, приведенный в листинге 5.19, вы увидите сущность CreditCard, которая задействует предыдущее перечисление при отображении по умолчанию.

Листинг 5.19. Отображение перечислимого типа с порядковыми номерами

```
@Entity
@Table(name = "credit_card")
public class CreditCard {
  @Id
  private String number;
  private String expiryDate;
  private Integer controlNumber;
  private CreditCardType creditCardType;
  // Конструкторы, геттеры, сеттеры
}
```

Поскольку применяются правила по умолчанию, перечисление отобразится в целочисленном столбце, и все будет отлично. Но представим, что в верхушку перечисления добавлена новая константа. Поскольку присваивание порядковых номеров осуществляется согласно последовательности, в которой значения объявляются, значения, уже хранящиеся в базе данных, больше не будут соответствовать перечислению. Лучшим решением стало бы сохранение имени значения как строки вместо сохранения порядкового номера. Это можно сделать, добавив аннотацию @Enumerated к атрибуту и указав значение STRING (значением по умолчанию является ORDINAL), как показано в листинге 5.20.

Листинг 5.20. Отображение перечислимого типа с использованием STRING

```
@Entity
@Table(name = "credit_card")
public class CreditCard {
  @Id
  private String number;
  private String expiryDate;
  private Integer controlNumber;
  @Enumerated(EnumType.STRING)
  private CreditCardType creditCardType;
```

```
  // Конструкторы, геттеры, сеттеры
}
```

Теперь столбец базы `CreditCardType` данных будет иметь тип `VARCHAR`, а информация о карточке Visa будет сохранена в строке `"VISA"`.

Тип доступа

До сих пор я показывал вам аннотированные классы (`@Entity` или `@Table`) и атрибуты (`@Basic`, `@Column`, `@Temporal` и т. д.), однако аннотации, применяемые к атрибуту (или доступ к полям), также могут быть применены к соответствующему методу-геттеру (или доступ к свойствам). Например, аннотация `@Id` может быть применена к атрибуту `id` или методу `getId()`. Поскольку это в основном вопрос личных предпочтений, я склонен использовать доступ к свойствам (аннотировать геттеры), так как, по моему мнению, код при этом получается более удобочитаемым. Это позволяет быстро изучить атрибуты сущности, не утопая в аннотациях. Чтобы код в этой книге было легко разобрать, я решил аннотировать атрибуты. Однако в некоторых случаях (например, когда речь идет о наследовании) это не просто дело личного вкуса, поскольку оно может повлиять на ваше отображение.

ПРИМЕЧАНИЕ

В Java поле определяется как атрибут экземпляра. Свойство — это любое поле с методами (геттерами или сеттерами) доступа, которые придерживаются шаблона JavaBean (в начале идет getXXX, setXXX или isXXX в случае с Boolean).

При выборе между доступом к полям (атрибуты) или доступом к свойствам (геттеры) необходимо определить *тип доступа*. По умолчанию для сущности применяется один тип доступа: это либо доступ к полям, либо доступ к свойствам, но не оба сразу (например, поставщик постоянства получает доступ к постоянному состоянию либо посредством атрибутов, либо посредством методов-геттеров). Согласно спецификации поведение приложения, в котором сочетается применение аннотаций к полям и свойствам без указания типа доступа явным образом, является неопределенным. При использовании доступа к полям (листинг 5.21) поставщик постоянства отображает атрибуты.

Листинг 5.21. Сущность Customer с аннотированными полями

```
@Entity
public class Customer {
  @Id @GeneratedValue
  private Long id;
  @Column(name = "first_name", nullable = false, length = 50)
  private String firstName;
  @Column(name = "last_name", nullable = false, length = 50)
  private String lastName;
  private String email;
  @Column(name = "phone_number", length = 15)
  private String phoneNumber;
  // Конструкторы, геттеры, сеттеры
}
```

При использовании доступа к свойствам, как показано в листинге 5.22, отображение базируется на геттерах, а не на атрибутах. Все геттеры, не снабженные аннотацией `@Transient`, являются постоянными.

Листинг 5.22. Сущность Customer с аннотированными свойствами

```
@Entity
public class Customer {
  private Long id;
  private String firstName;
  private String lastName;
  private String email;
  private String phoneNumber;
  // Конструкторы
  @Id @GeneratedValue
  public Long getId() {
    return id;
  }
  public void setId(Long id) {
    this.id = id;
  }
  @Column(name = "first_name", nullable = false, length = 50)
  public String getFirstName() {
    return firstName;
  }
  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }
  @Column(name = "last_name", nullable = false, length = 50)
  public String getLastName() {
    return lastName;
  }
  public void setLastName(String lastName) {
    this.lastName = lastName;
  }
  public String getEmail() {
    return email;
  }
  public void setEmail(String email) {
    this.email = email;
  }
  @Column(name = "phone_number", length = 15)
  public String getPhoneNumber() {
    return phoneNumber;
  }
  public void setPhoneNumber(String phoneNumber) {
    this.phoneNumber = phoneNumber;
  }
}
```

В плане отображения две сущности из листингов 5.21 и 5.22 полностью идентичны, поскольку имена атрибутов в данном случае совпадают с именами геттеров.

Однако вместо использования типа доступа по умолчанию вы можете явным образом указать тип с помощью аннотации `@javax.persistence.Access`.

Эта аннотация принимает два возможных значения — `FIELD` или `PROPERTY`, а также может быть использована в отношении сущности как таковой и/или каждого атрибута или геттера. Например, при применении `@Access(AccessType.FIELD)` к сущности поставщиком постоянства будут приниматься во внимание только аннотации отображения, которыми снабжены атрибуты. Тогда можно будет выборочно обозначить отдельные геттеры для доступа к свойствам посредством `@Access(AccessType.PROPERTY)`.

Типы явного доступа могут быть очень полезны (например, при работе со встраиваемыми объектами или наследованием), однако их смешение часто приводит к ошибкам. В листинге 5.23 показан пример того, что может произойти при смешении типов доступа.

Листинг 5.23. Сущность Customer, в случае с которой явным образом смешиваются типы доступа

```
@Entity
@Access(AccessType.FIELD)
public class Customer {
  @Id @GeneratedValue
  private Long id;
  @Column(name = "first_name", nullable = false, length = 50)
  private String firstName;
  @Column(name = "last_name", nullable = false, length = 50)
  private String lastName;
  private String email;
  @Column(name = "phone_number", length = 15)
  private String phoneNumber;
  // Конструкторы, геттеры, сеттеры
  @Access(AccessType.PROPERTY)
  @Column(name = "phone_number", length = 555)
  public String getPhoneNumber() {
    return phoneNumber;
  }
  public void setPhoneNumber(String phoneNumber) {
    this.phoneNumber = phoneNumber;
  }
}
```

В примере, показанном в листинге 5.23, тип доступа явным образом определяется как `FIELD` на уровне сущности. Это говорит интерфейсу `PersistenceManager` о том, что ему следует обрабатывать только аннотации, которыми снабжены атрибуты. Атрибут `phoneNumber` снабжен аннотацией `@Column`, ограничивающей значение его `length` величиной 15. Читая этот код, вы ожидаете, что в базе данных в итоге будет `VARCHAR(15)`, однако этого не случится. Метод-геттер показывает, что тип доступа для метода `getPhoneNumber()` был изменен явным образом, поэтому длина равна значению `length` атрибута `phoneNumber` (до 555). В данном случае сущность `AccessType.FIELD` перезаписывается `AccessType.PROPERTY`. Тогда в базе данных у вас окажется `VARCHAR(555)`.

Коллекции базовых типов

Коллекции тех или иных элементов очень распространены в Java. Из последующих разделов вы узнаете о связях между сущностями (которые могут быть коллекциями сущностей). По сути, это означает, что у одной сущности имеется коллекция других сущностей или встраиваемых объектов. Что касается отображения, то каждая сущность отображается в свою таблицу, при этом создаются ссылки между первичными и внешними ключами. Как вы уже знаете, сущность представляет собой Java-класс с идентификатором и множеством других атрибутов. Но что, если вам потребуется сохранить коллекцию Java-типов, например `String` или `Integer`? С тех пор как вышла версия JPA 2.0, это можно легко сделать без необходимости решать проблему создания отдельного класса. Для этого предназначены аннотации `@ElementCollection` и `@CollectionTable`.

Мы используем аннотацию `@ElementCollection` как индикатор того, что атрибут типа `java.util.Collection` включает коллекцию экземпляров базовых типов (то есть объектов, которые не являются сущностями) или встраиваемых объектов (подробнее на эту тему мы поговорим в разделе «Встраиваемые объекты»). Фактически этот атрибут может иметь один из следующих типов:

- `java.util.Collection` — общий корневой интерфейс в иерархии коллекций;
- `java.util.Set` — коллекция, предотвращающая вставку элементов-дубликатов;
- `java.util.List` — коллекция, которая применяется, когда требуется извлечь элементы в некоем порядке, определяемом пользователем.

Кроме того, аннотация `@CollectionTable` позволяет вам настраивать детали таблицы коллекции (то есть таблицы, которая будет соединять таблицу сущности с таблицей базовых типов), например изменять ее имя. При отсутствии этой аннотации имя таблицы будет конкатенацией имени содержащей сущности и имени атрибута коллекции, разделенных знаком подчеркивания.

Опять-таки, используя сущность-пример `Book`, взглянем на то, как добавить атрибут для сохранения тегов. Сегодня теги и облака тегов распространены повсеместно. Они обычно применяются при сортировке данных, поэтому представим в этом примере, что вы хотите добавить как можно больше тегов сущности `Book` для ее описания и быстрого поиска. Тег — это всего лишь строка, так что у сущности `Book` может быть коллекция строк для сохранения соответствующей информации, как показано в листинге 5.24.

Листинг 5.24. Сущность Book с коллекцией строк

```
@Entity
public class Book {
  @Id @GeneratedValue
  private Long id;
  private String title;
  private Float price;
  private String description;
  private String isbn;
  private Integer nbOfPage;
  private Boolean illustrations;
```

```
    @ElementCollection(fetch = FetchType.LAZY)
    @CollectionTable(name = "Tag")
    @Column(name = "Value")
    private List<String> tags = new ArrayList<>();
    // Конструкторы, геттеры, сеттеры
}
```

Аннотация `@ElementCollection`, показанная в листинге 5.24, используется для информирования поставщика постоянства о том, что атрибут `tags` представляет собой список строк, а его выборка должна быть отложенной. При отсутствии `@CollectionTable` имя таблицы по умолчанию было бы `BOOK_TAGS` (конкатенация имени содержащей сущности и имени атрибута коллекции, разделенных знаком подчеркивания), а не `TAG`, как указано в элементе `name` (`name = "Tag"`). Обратите внимание, что я добавил дополнительную аннотацию `@Column`, чтобы переименовать столбец в `VALUE` (в противном случае этот столбец получил бы имя, как у атрибута, `TAGS`). Результат можно увидеть на рис. 5.2.

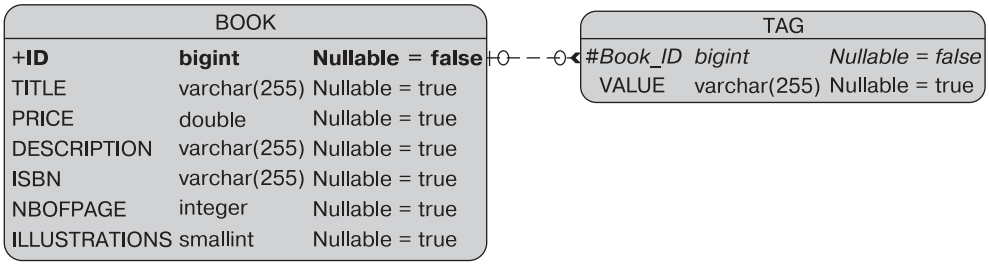


Рис. 5.2. Связь между таблицами BOOK и TAG

ПРИМЕЧАНИЕ

В JPA 1.0 этих аннотаций не было. Однако все равно можно было сохранить список примитивных типов, например BLOB, в базе данных. Почему? А потому, что java.util.ArrayList реализует Serializable, а JPA может автоматически отображать сериализуемые объекты в BLOB-объекты. Вместе с тем, если вы взамен используете тип java.util.List, то получите исключение, так как он не расширяет Serializable. Применение @ElementCollection — более элегантный и целесообразный способ сохранения списков базовых типов. Сохранение списков в недоступном двоичном формате не позволит совершать к ним запросы и сделает их непереносимыми в код на других языках (поскольку основной сериализованный объект может быть использован во время выполнения кода, написанного только на Java — не на Ruby, PHP...).

Отображение базовых типов

Как и коллекции, отображения очень полезны при сохранении данных. В JPA 1.0 ключи могли относиться только к базовому типу данных, а значения могли быть только сущностями. Сейчас отображения могут содержать любую комбинацию базовых типов, встраиваемых объектов и сущностей как ключей или значений, что привносит большую гибкость в процесс отображения. Сосредоточимся на отображениях базовых типов.

Когда отображение задействует базовые типы, аннотации `@ElementCollection` и `@CollectionTable` могут быть использованы тем же путем, который вы видели ранее в случае с коллекциями. Таблица коллекции тогда будет использоваться для хранения данных отображения.

Обратимся к примеру с CD-альбомом, содержащим треки (листинг 5.25). Трек имеет название и позицию (первый трек альбома, второй трек альбома и т. д.). Тогда у вас может быть отображение треков с целочисленным значением, говорящем о позиции определенного трека (ключ отображения), и строкой для указания названия этого трека (значение отображения).

Листинг 5.25. CD-альбом с отображением треков

```
@Entity
public class CD {
  @Id @GeneratedValue
  private Long id;
  private String title;
  private Float price;
  private String description;
  @Lob
  private byte[] cover;
  @ElementCollection
  @CollectionTable(name="track")
  @MapKeyColumn (name = "position")
  @Column(name = "title")
  private Map<Integer, String> tracks = new HashMap<>();
  // Конструкторы, геттеры, сеттеры
}
```

Я уже говорил, что аннотация `@ElementCollection` используется как индикатор объектов в отображении, хранящихся в таблице коллекции. Аннотация `@CollectionTable` изменяет имя по умолчанию таблицы коллекции на `TRACK`.

Разница в случае с коллекциями заключается во введении новой аннотации: `@MapKeyColumn`. Она используется для указания отображения ключевого столбца. Если она не будет задана, то столбец получит имя в виде конкатенации имени ссылающегося атрибута связи и `_KEY`. В листинге 5.25 видно, что аннотация переименовала его в `POSITION`, чтобы было яснее, а по умолчанию он назывался бы иначе (`TRACK_KEY`).

Аннотация `@Column` указывает на то, что столбец, содержащий значение отображения, следует переименовать в `TITLE`. Результат можно увидеть на рис. 5.3.

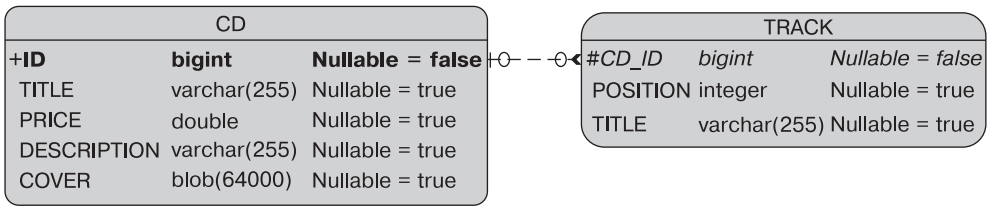


Рис. 5.3. Связь между таблицами CD и TRACK

Отображение с использованием XML

Теперь, когда вы ближе познакомились с элементарным отображением с использованием аннотаций, взглянем на XML-отображение. Если вам доводилось применять объектно-реляционный фреймворк вроде ранних версий Hibernate, то вы уже знаете, как отображать свои сущности в отдельном файле XML-дескриптора развертывания. С начала этой главы вы не видели ни одной строки XML-кода, а только аннотации. JPA также предлагает как вариант XML-синтаксис для отображения сущностей. Я не стану вдаваться в подробности XML-отображения, поскольку решил сосредоточиться на аннотациях (так как их легче использовать в книге, а большинство разработчиков выбирает их вместо XML-отображения). Имейте в виду, что у каждой аннотации, которую вы увидите в текущей главе, имеется XML-эквивалент, а этот раздел получился бы огромным, если бы я рассмотрел в нем их все. Я рекомендую вам заглянуть в главу 12 под названием «XML-дескриптор объектно-реляционного отображения» спецификации JPA 2.1, где более детально рассматриваются все XML-теги.

XML-дескрипторы развертывания — альтернатива применению аннотаций. Однако, несмотря на то что у каждой аннотации есть эквивалентный XML-тег и наоборот, разница состоит в том, что XML берет верх над аннотациями. Если вы аннотируете атрибут или сущность с использованием определенного значения и в то же время произведете развертывание XML-дескриптора с использованием другого значения, то преимущество будет за XML.

Вопрос звучит так: когда вам следует использовать аннотации вместо XML и почему? Прежде всего, это дело вкуса, поскольку оба ведут себя абсолютно одинаково. Когда метаданные действительно связаны с кодом (например, речь может идти о первичном ключе), имеет смысл использовать аннотации, поскольку метаданные являются лишь еще одним аспектом программы. Прочие виды метаданных, например длина столбца или другие детали схемы, могут быть изменены в зависимости от среды развертывания (скажем, схема базы данных может отличаться в среде разработки, тестирования или производства). Похожая ситуация возникает, когда основанному на JPA продукту необходимо обеспечить поддержку баз данных от нескольких разных поставщиков. Может потребоваться настроить генерирование определенного идентификатора, параметра столбца и т. д. в зависимости от типа используемой базы данных. Это может быть лучше выражено во внешних XML-дескрипторах развертывания (по одному на среду), благодаря чему код не придется модифицировать.

Снова обратимся к примеру сущности `Book`. На этот раз представьте, что у вас имеется две среды и вы желаете отобразить сущность `Book` в таблицу `BOOK` в среде разработки и в таблицу `BOOK_XML_MAPPING` в среде тестирования. Класс будет аннотирован только с помощью `@Entity` (листинг 5.26) и не станет включать информацию о таблице, в которую его надлежит отобразить (то есть аннотация `@Table` будет отсутствовать). Аннотация `@Id` определяет первичный ключ как генерируемый автоматически, а аннотация `@Column` задает длину описания равной 500 символам.

Листинг 5.26. Сущность Book лишь с несколькими аннотациями

```
@Entity
public class Book {
  @Id @GeneratedValue
  private Long id;
  private String title;
  private Float price;
  @Column(length = 500)
  private String description;
  private String isbn;
  private Integer nbOfPage;
  private Boolean illustrations;
  // Конструкторы, геттеры, сеттеры
}
```

В отдельном файле book_mapping.xml (листинг 5.27), придерживаясь заданной XML-схемы, вы можете изменить отображение для любых данных сущности. Тег <table> позволяет вам изменить имя таблицы, в которую будет отображаться сущность (BOOK_XML_MAPPING вместо BOOK по умолчанию). В теге <attributes> вы можете настроить атрибуты, указав не только их имена и длину, но и их связи с другими сущностями. Например, вы можете изменить отображение для столбца title и количество страниц (nbOfPage).

Листинг 5.27. Отображение файла META-INF/book_mapping.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
                 http://java.sun.com/xml/ns/persistence/orm_2_1.xsd"
                 version="2.1">
  <entity class="org.agoncal.book.javaee7.chapter05.Book">
    <table name="book_xml_mapping"/>
    <attributes>
      <basic name="title">
        <column name="book_title" nullable="false" updatable="false"/>
      </basic>
      <basic name="description">
        <column length="2000"/>
      </basic>
      <basic name="nbOfPage">
        <column name="nb_of_page" nullable="false"/>
      </basic>
    </attributes>
  </entity>
</entity-mappings>
```

Всегда следует помнить о том, что XML имеет преимущественное значение перед аннотациями. Хотя атрибут description и аннотирован посредством @Column(length = 500), используется та длина столбца, которая указана в файле book_mapping.xml

(см. листинг 5.27). Она равна 2000. Это может привести в замешательство, если вы взглянете на соответствующий код и увидите значение 500, а затем посмотрите на DDL-код и увидите значение 2000. Никогда не забывайте проверять XML-дескриптор развертывания.

Результатом слияния XML-метаданных и метаданных аннотаций является то, что сущность Book будет отображена в структуру таблицы BOOK_XML_MAPPING, определенную в листинге 5.28. Если вы решите полностью проигнорировать аннотации и определить свое отображение с использованием только XML, то сможете добавить тег <xml-mapping-metadata-complete> в файл book_mapping.xml (в данном случае все аннотации будут игнорироваться, даже если XML не будет содержать переопределение).

Листинг 5.28. Структура таблицы BOOK_XML_MAPPING

```
create table BOOK_XML_MAPPING (
  ID BIGINT not null,
  BOOK_TITLE VARCHAR(255) not null,
  DESCRIPTION VARCHAR(2000),
  NB_OF_PAGE INTEGER not null,
  PRICE DOUBLE(52, 0),
  ISBN VARCHAR(255),
  ILLUSTRATIONS SMALLINT,
  primary key (ID)
);
```

Чтобы заставить все это работать, не хватает лишь одной порции информации. Вам нужно указать ссылку на файл book_mapping.xml в своем файле persistence.xml, для чего потребуется прибегнуть к тегу <mapping-file>. Названный файл определяет контекст постоянства сущности и базу данных, в которую она должна быть отображена. Это самая важная порция информации, которая необходима поставщику постоянства для ссылки на внешний XML-файл отображения. Произведите развертывание сущности Book с обоими XML-файлами в каталоге META-INF, и все будет готово (листинг 5.29).

Листинг 5.29. Файл persistence.xml со ссылкой на внешний файл отображения

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence 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_2_1.xsd"
             version="2.1">
  <persistence-unit name="chapter05PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
      <class>org.agoncal.book.javaee7.chapter05.Book</class>
      <mapping-file>META-INF/book_mapping.xml</mapping-file>
      <properties>
        <!-- Свойства поставщика постоянства -->
      </properties>
  </persistence-unit>
</persistence>
```

Встраиваемые объекты

В приводившемся ранее в этой главе подразделе «Составные первичные ключи» вы видели, как класс может быть встроен и использован в качестве первичного ключа с применением аннотации `@EmbeddedId`. Встраиваемые объекты — это объекты, которые сами по себе не имеют постоянного идентификатора; они могут быть только встроены во владеющие сущности. Владеющая сущность может располагать коллекциями встраиваемых объектов, а также одним встраиваемым атрибутом. Они сохраняются как внутренняя часть владеющей сущности и совместно используют идентификатор этой сущности. Это означает, что каждый атрибут встроенного объекта будет отображаться в таблицу сущности. Это связь строгого владения (также называемая композицией), поэтому если удалить сущность, то встроенные объекты тоже окажутся удалены.

Эта композиция между двумя классами задействует аннотации. Для включенного класса используется аннотация `@Embeddable`, в то время как в отношении сущности, которая включает этот класс, используется `@Embedded`. Обратимся к примеру клиента, у которого имеется идентификатор, имя, адрес электронной почты, а также домашний адрес. Все соответствующие атрибуты могли бы располагаться в одной сущности `Customer` (листинг 5.31), однако по причинам объектного моделирования они разделены на два класса: `Customer` и `Address`. Поскольку у `Address` нет собственного идентификатора, и при этом он всего лишь часть состояния `Customer`, этот класс является хорошим кандидатом на то, чтобы стать встраиваемым объектом вместо сущности (листинг 5.30).

Листинг 5.30. Класс Address является встраиваемым объектом

```
@Embeddable
public class Address {
  private String street1;
  private String street2;
  private String city;
  private String state;
  private String zipcode;
  private String country;
  // Конструкторы, геттеры, сеттеры
}
```

Как вы можете видеть в листинге 5.30, класс `Address` аннотирован не как сущность, а как встраиваемый объект. Аннотация `@Embeddable` определяет, что `Address` может быть встроен в иной класс-сущность (или иной встраиваемый объект). С другой стороны, для сущности `Customer` приходится использовать аннотацию `@Embedded`, чтобы определить, что `Address` является постоянным атрибутом, который будет сохранен как внутренняя часть и станет совместно использовать его идентификатор (см. листинг 5.31).

Листинг 5.31. Сущность Customer со встроенным Address

```
@Entity
public class Customer {
  @Id @GeneratedValue
  private Long id;
```

```
  private String firstName;
  private String lastName;
  private String email;
  private String phoneNumber;
  @Embedded
  private Address address;
  // Конструкторы, геттеры, сеттеры
}
```

Каждый атрибут `Address` будет отображаться в таблицу владеющей сущности `Customer`. Будет только одна таблица со структурой, определенной в листинге 5.32. Как вы увидите позднее в подразделе «Переопределение атрибутов» раздела «Отображение наследования» данной главы, сущности могут переопределять атрибуты встраиваемых объектов (с использованием аннотации `@AttributeOverrides`).

Листинг 5.32. Структура таблицы CUSTOMER со всеми атрибутами Address

```
create table CUSTOMER (
  ID BIGINT not null,
  LASTNAME VARCHAR(255),
  PHONENUMBER VARCHAR(255),
  EMAIL VARCHAR(255),
  FIRSTNAME VARCHAR(255),
  STREET2 VARCHAR(255),
  STREET1 VARCHAR(255),
  ZIPCODE VARCHAR(255),
  STATE VARCHAR(255),
  COUNTRY VARCHAR(255),
  CITY VARCHAR(255),
  primary key (ID)
);
```

ПРИМЕЧАНИЕ

В предшествующих разделах я показывал вам, как отображать коллекции и отображения базовых типов данных. В JPA 2.1 то же самое возможно с помощью встраиваемых объектов. Вы можете отображать коллекции встраиваемых объектов, а также отображения таких объектов (встраиваемый объект может быть либо ключом, либо значением отображения).

Тип доступа встраиваемого класса. Тип доступа встраиваемого класса обуславливается типом доступа класса-сущности, в котором он располагается. Если сущность явным образом использует такой тип доступа, как доступ к свойствам, то встраиваемый объект будет неявно использовать аналогичный тип доступа. Другой тип доступа для встраиваемого класса можно указать с помощью аннотации `@Access`.

Сущности `Customer` (листинг 5.33) и `Address` (листинг 5.34) задействуют разные типы доступа.

Листинг 5.33. Сущность Customer с таким типом доступа, как доступ к полям

```
@Entity
@Access(AccessType.FIELD)
public class Customer {
  @Id @GeneratedValue
```

```
  private Long id;
  @Column(name = "first_name", nullable = false, length = 50)
  private String firstName;
  @Column(name = "last_name", nullable = false, length = 50)
  private String lastName;
  private String email;
  @Column(name = "phone_number", length = 15)
  private String phoneNumber;
  @Embedded
  private Address address;
  // Конструкторы, геттеры, сеттеры
}
```

Листинг 5.34. Встраиваемый объект с таким типом доступа, как доступ к свойствам

```
@Embeddable
@Access(AccessType.PROPERTY)
public class Address {
  private String street1;
  private String street2;
  private String city;
  private String state;
  private String zipcode;
  private String country;
  // Конструкторы
  @Column(nullable = false)
  public String getStreet1() {
    return street1;
  }
  public void setStreet1(String street1) {
    this.street1 = street1;
  }
  public String getStreet2() {
    return street2;
  }
  public void setStreet2(String street2) {
    this.street2 = street2;
  }
  @Column(nullable = false, length = 50)
  public String getCity() {
    return city;
  }
  public void setCity(String city) {
    this.city = city;
  }
  @Column(length = 3)
  public String getState() {
    return state;
  }
  public void setState(String state) {
    this.state = state;
  }
```

```
    @Column(name = "zip_code", length = 10)
    public String getZipcode() {
      return zipcode;
    }
    public void setZipcode(String zipcode) {
      this.zipcode = zipcode;
    }
    public String getCountry() {
      return country;
    }
    public void setCountry(String country) {
      this.country = country;
    }
  }
```

Настоятельно рекомендуется явным образом задавать тип доступа для встраиваемых объектов, чтобы избежать ошибок отображения, когда один встраиваемый объект окажется встроенным во множественные сущности. К примеру, расширим нашу модель, добавив сущность `Order`, как показано на рис. 5.4. `Address` теперь будет встроен в `Customer` и `Order`.

Рис. 5.4. Address встроен в Customer и Order

Для каждой сущности определяется отличающийся тип доступа: `Customer` использует доступ к полям, а `Order` — доступ к свойствам. Поскольку тип доступа встраиваемого объекта обуславливается типом доступа класса-сущности, в котором он объявлен, `Address` будет отображен двумя разными путями, что может привести к проблемам отображения. Чтобы этого не случилось, тип доступа `Address` должен быть задан явным образом.

ПРИМЕЧАНИЕ

Типы явного доступа также очень полезны, когда речь идет о наследовании. По умолчанию листовые сущности наследуют тип доступа от своей корневой сущности. В иерархии сущностей доступ к каждой из них возможен по-разному из других классов в иерархии. Включение аннотации @Access фактически приведет к тому, что режим доступа по умолчанию для иерархии будет локально переопределен.

Отображение связей

Мир объектно-ориентированного программирования изобилует классами и ассоциациями между классами. Эти ассоциации являются структурными в том смысле, что связывают объекты одного типа с объектами другого типа, позволяя одному

объекту заставлять другой объект выполнять действия от своего имени. Между классами могут существовать ассоциации нескольких типов.

Прежде всего, у ассоциации есть направление. Она может быть *однонаправленной* (один объект может осуществлять навигацию по направлению к другому объекту) и *двунаправленной* (один объект может осуществлять навигацию по направлению к другому и наоборот). В Java для навигации по объектам используется точечный (`.`) синтаксис. Например, если написать `customer.getAddress().getCountry()`, то будет осуществляться навигация от объекта `Customer` к объекту `Address`, а затем — к объекту `Country`.

В унифицированном языке моделирования (Unified Modeling Language — UML) для представления однонаправленной ассоциации между двумя классами используется стрелка, указывающая направление. На рис. 5.5 `Class1` (источник) может осуществлять навигацию по направлению к `Class2` (цель), но не наоборот.

Рис. 5.5. Однонаправленная ассоциация между двумя классами

Для индикации двунаправленной ассоциации стрелки не используются. Как показано на рис. 5.6, `Class1` может осуществлять навигацию по направлению к `Class2` и наоборот. В Java это представляется как наличие у `Class1` атрибута типа `Class2` и наличие у `Class2` атрибута типа `Class1`.

Рис. 5.6. Двунаправленная ассоциация между двумя классами

Для ассоциации также характерна *множественность* (или *кардинальность*). На каждом из концов ассоциации можно указать, сколько ссылающихся объектов вовлечено в нее. На UML-диаграмме, приведенной на рис. 5.7, показано, что `Class1` ссылается на нуль или более экземпляров `Class2`.

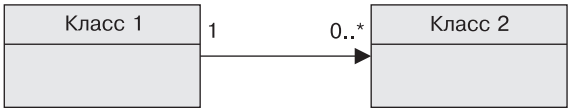


Рис. 5.7. Множественность ассоциаций классов

В UML кардинальность — это диапазон между минимальным и максимальным числами. Таким образом, `0..1` означает, что у вас будет минимум нуль объектов и максимум один объект; `1` означает, что у вас один и только один экземпляр; `1..*` означает, что у вас может быть один или много экземпляров, а `3..6` означа-

ет, что у вас может быть от трех до шести объектов. В Java ассоциация, представляющая более одного объекта, задействует коллекции типа `java.util.Collection`, `java.util.Set`, `java.util.List` или даже `java.util.Map`.

В случае со связью имеет место владение (то есть владелец связи). При однонаправленной связи предполагается владение: нет сомнения, что на рис. 5.5 владельцем является `Class1`, однако при двунаправленной связи, как показано на рис. 5.6, владельца нужно указывать явным образом. В таком случае вы показываете владельца, который определяет физическое отображение и противоположную сторону (не владеющую сторону).

В следующих разделах вы увидите, как отображать коллекции объектов с использованием аннотаций JPA.

Связи в реляционных базах данных

В реляционном мире дело обстоит по-другому, поскольку, строго говоря, реляционная база данных — это коллекция *отношений* (также называемых *таблицами*), то есть все, что вы смоделируете, будет таблицей. При моделировании ассоциации у вас не будет списков, наборов или отображений — у вас будут таблицы. В JPA при наличии ассоциации между одним классом и другим в базе данных у вас будет ссылка на таблицу. Эту ссылку можно смоделировать двумя способами: с помощью *внешнего ключа (столбца соединения)* или с использованием *таблицы соединения*. В контексте базы данных столбец, ссылающийся на ключ другой таблицы, называется *столбцом внешнего ключа*.

В качестве примера предположим, что у клиента имеется один домашний адрес. В листингах 5.33 и 5.34 мы моделировали эту связь как встраиваемый объект, однако теперь превратим ее в *связь «один к одному»*. При использовании Java у вас имелся бы класс `Customer` с атрибутом `Address`. В реляционном мире у вас могла бы быть таблица `CUSTOMER`, указывающая на `ADDRESS` с помощью столбца внешнего ключа (или столбца соединения), как показано на рис. 5.8.

Customer

Первичный ключ	Имя	Фамилия	Внешний ключ
1	Джеймс	Роррисон	11
2	Доминик	Джонсон	12
3	Мака	Макарон	13

Address

Первичный ключ	Улица	Город	Страна
11	Алигре	Париж	Франция
12	Бэлхэм	Лондон	Англия
13	Алфама	Лиссабон	Португалия

Рис. 5.8. Связь с использованием столбца соединения

Есть и второй способ моделирования — с применением таблицы соединения. В таблице `CUSTOMER`, показанной на рис. 5.9, больше нет внешнего ключа `ADDRESS`. Для размещения информации о связи с сохранением внешних ключей была создана промежуточная таблица.

Вы не стали бы использовать таблицу соединения для представления связи «один к одному», поскольку это могло бы отрицательно сказаться на производительности (вам всегда будет нужен доступ к третьей таблице для получения адреса клиента).

Рис. 5.9. Связь с использованием таблицы соединения

Таблицы соединения обычно применяются при наличии кардинальностей «один ко многим» или «многие ко многим». Как вы увидите в следующем разделе, JPA использует два режима для отображения объектных ассоциаций.

Связи между сущностями

Теперь вернемся к JPA. Большинству сущностей необходима возможность ссылаться на другие сущности или иметь связи с ними. Именно это приводит к созданию графов доменной модели, распространенных в сфере бизнес-приложений. JPA позволяет отображать ассоциации, благодаря чему одна сущность может быть связана с другой в реляционной модели. Как это имеет место в случае с аннотациями элементарного отображения, которые вы видели ранее, JPA задействует конфигурацию в порядке исключения, когда речь идет об ассоциациях. JPA предусматривает используемый по умолчанию способ сохранения связей, однако если он окажется неподходящим для вашей модели базы данных, в вашем распоряжении есть несколько аннотаций, которые вы сможете использовать для настройки отображения.

Между двумя сущностями могут быть отношения «один к одному», «один ко многим», «многие к одному» или «многие ко многим». Каждое соответствующее отображение именуется согласно типу связи источника и цели: аннотации `@OneToOne`, `@OneToMany`, `@ManyToOne` или `@ManyToMany`. Каждая аннотация может быть использована однонаправленным или двунаправленным путем. В табл. 5.1 приведены все возможные комбинации типов связи и направлений.

Таблица 5.1. Все возможные комбинации «тип связи — направление»

Тип связи	Направление
«Один к одному»	Однонаправленный подход
«Один к одному»	Двунаправленный подход
«Один ко многим»	Однонаправленный подход

Тип связи	Направление
«Многие к одному»/«один ко многим»	Двунаправленный подход
«Многие к одному»	Однонаправленный подход
«Многие ко многим»	Однонаправленный подход
«Многие ко многим»	Двунаправленный подход

Вы увидите, что названные подходы являются повторяющимися концепциями, которые применимы одинаковым образом ко всем типам связи. Далее вы узнаете, в чем заключается разница между однонаправленными и двунаправленными связями, а затем реализуете некоторые из соответствующих комбинаций. Я не стану подробно разбирать весь перечень комбинаций, а сосредоточусь на одном подмножестве. Объяснение всех комбинаций было бы повторением одного и того же. Важно, чтобы вы поняли, как отображать отношение и направление в связях.

Однонаправленные и двунаправленные ассоциации

С точки зрения объектного моделирования связь между классами естественна. При однонаправленной ассоциации объект `A` указывает только на объект `B`; при двунаправленной ассоциации оба объекта ссылаются друг на друга. Однако потребуется приложить усилия, когда речь зайдет об отображении двунаправленной связи в реляционную базу данных, как показано в следующем примере, включающем клиента, у которого имеется домашний адрес.

При однонаправленной связи у сущности `Customer` имеется атрибут типа `Address` (рис. 5.10). Эта связь является однонаправленной и предполагает навигацию от одной стороны к другой. `Customer` в данном случае называется владельцем связи. В контексте базы данных это означает, что таблица `CUSTOMER` будет располагать внешним ключом (столбцом соединения), указывающим на `ADDRESS`, а если вы будете владеть связью, то сможете настроить отображение этой связи. Например, если вам потребуется изменить имя внешнего ключа, отображение будет выполнено в сущности `Customer` (то есть владельце).

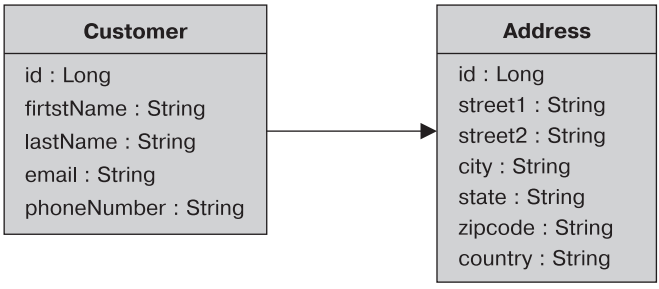


Рис. 5.10. Однонаправленная ассоциация между Customer и Address

Как уже отмечалось, связи также могут быть двунаправленными. Чтобы иметь возможность осуществлять навигацию между `Address` и `Customer`, вам потребуется преобразовать однонаправленную связь в двунаправленную, добавив атрибут

Customer в сущность Address (рис. 5.11). Обратите внимание, что на UML-диаграммах классов не приводятся атрибуты, представляющие связь.

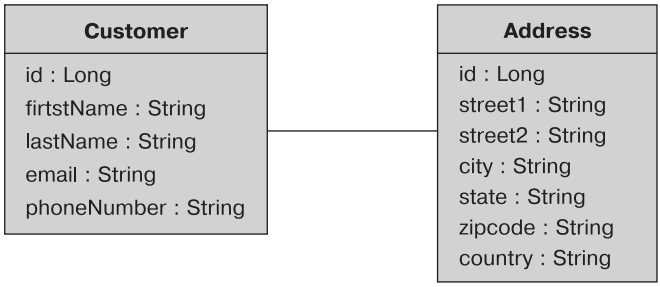


Рис. 5.11. Двунаправленная ассоциация между Customer и Address

В контексте Java-кода и аннотаций это аналогично наличию двух отдельных отображений «один к одному», по одному в каждом направлении. Вы можете представлять себе двунаправленную связь как пару однонаправленных связей, идущих соответственно туда и обратно (рис. 5.12).

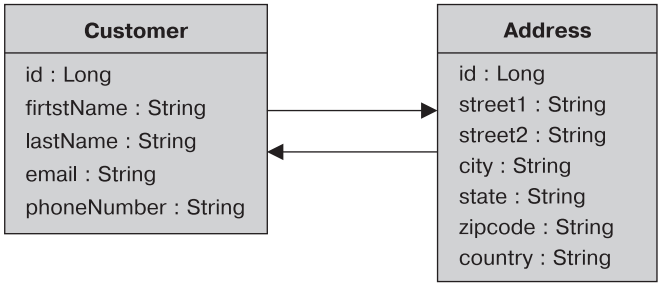


Рис. 5.12. Двунаправленная ассоциация, представленная двумя стрелками

Как осуществляется отображение пары однонаправленных связей? Кто является владельцем двунаправленной связи? Кто владеет информацией об отображении столбца соединения или таблицы соединения? Если у однонаправленных связей есть владеющая сторона, то у двунаправленных связей есть как владеющая, так и противоположная сторона, которая должна быть указана явным образом с помощью элемента mappedBy аннотаций @OneToOne, @OneToMany и @ManyToMany. Элемент mappedBy идентифицирует атрибут, который владеет связью и необходим для двунаправленных связей.

Для пояснения сравним Java-код (с одной стороны) и отображение базы данных (с другой стороны). Как вы можете видеть в левой части рис. 5.13, обе сущности указывают друг на друга с помощью атрибутов: у Customer имеется атрибут, аннотированный с использованием @OneToOne, а у сущности Address — атрибут Customer, тоже снабженный аннотацией. В правой части располагается отображение базы данных, где показаны таблицы CUSTOMER и ADDRESS. CUSTOMER является владельцем связи, поскольку содержит внешний ключ ADDRESS.

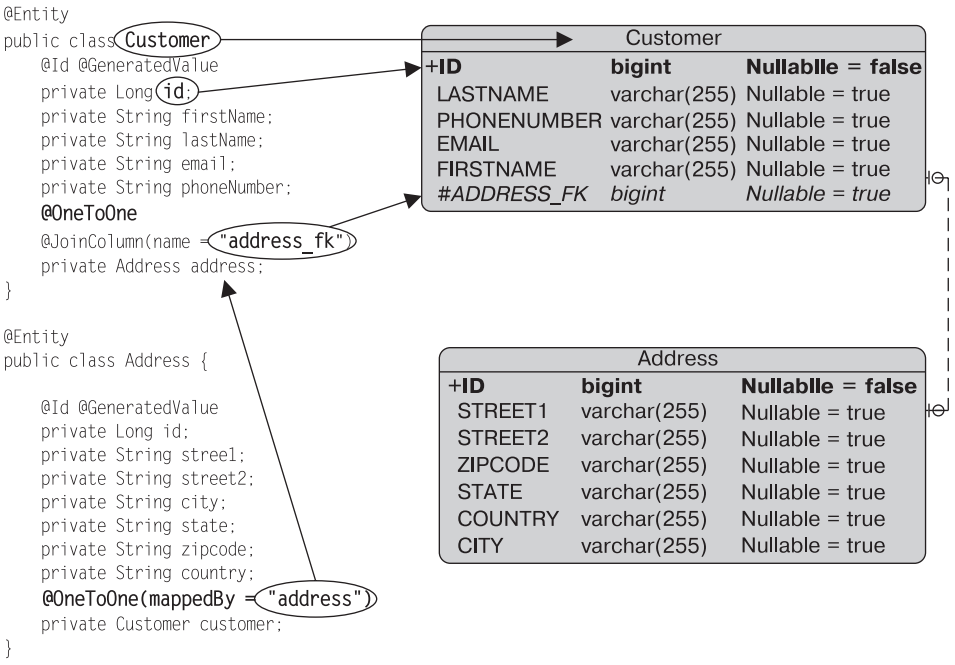


Рис. 5.13. Код Customer и Address наряду с отображением базы данных

На рис. 5.13 в аннотации `@OneToOne` сущности `Address` используется элемент `mappedBy`. `Address` в данном случае именуется противоположным владельцем связи, поскольку обладает элементом `mappedBy`. Элемент `mappedBy` говорит о том, что столбец соединения (`address`) указан на другом конце связи. Кроме того, на другом конце связи сущность `Customer` определяет столбец соединения путем использования аннотации `@JoinColumn` и переименовывает внешний ключ в `address_fk`. Сущность `Customer` представляет собой владеющую сторону связи и, как владелец, должна определять отображение столбца соединения. `Address` выступает в качестве противоположной стороны связи, где таблица владеющей сущности содержит внешний ключ (таблица `CUSTOMER` включает столбец `ADDRESS_FK`).

Элемент `mappedBy` может присутствовать в аннотациях `@OneToOne`, `@OneToMany` и `@ManyToMany`, но не в `@ManyToOne`. Атрибут `mappedBy` не может быть сразу на обеих сторонах двунаправленной ассоциации. Было бы также неправильно, если бы его не было ни на одной из сторон, поскольку поставщик воспринимал бы все это как две независимые однонаправленные связи. Это подразумевало бы, что каждая из сторон является владельцем и может определять столбец соединения.

ПРИМЕЧАНИЕ

Если вы знакомы с более ранними версиями Hibernate, то можете представлять себе JPA-параметр mappedBy как эквивалент Hibernate-атрибута inverse.

Однонаправленная связь с использованием @OneToOne

Однонаправленная связь «один к одному» между сущностями имеет ссылку кардинальности 1, которая досягаема только в одном направлении. Обратившись к примеру клиента и его домашнего адреса, предположим, что у него имеется только один домашний адрес (кардинальность 1). Важно выполнить навигацию от Customer (источник) по направлению к Address (цель), чтобы узнать, где клиент живет. Однако по некоторым причинам при нашей модели, показанной на рис. 5.14, вам не потребуется возможность навигации в противоположном направлении (например, вам не будет нужно знать, какой клиент живет по определенному адресу).

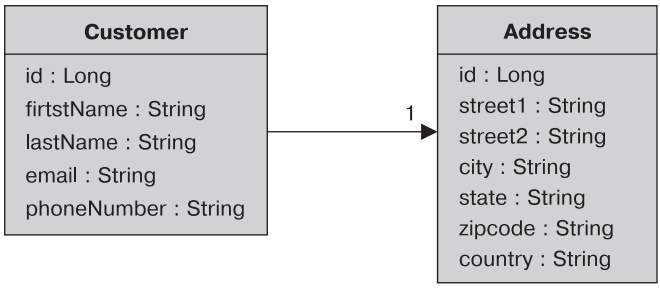


Рис. 5.14. У одного клиента имеется один домашний адрес

В Java однонаправленная связь означает, что у Customer будет иметься атрибут Address (листинг 5.35), однако у Address не будет атрибута Customer (листинг 5.36).

Листинг 5.35. Customer с одним Address

```
@Entity
public class Customer {
  @Id @GeneratedValue
  private Long id;
  private String firstName;
  private String lastName;
  private String email;
  private String phoneNumber;
  private Address address;
  // Конструкторы, геттеры, сеттеры
}
```

Листинг 5.36. Сущность Address

```
@Entity
public class Address {
  @Id @GeneratedValue
  private Long id;
  private String street1;
  private String street2;
  private String city;
  private String state;
  private String zipcode;
```

```
  private String country;
  // Конструкторы, геттеры, сеттеры
}
```

Как вы можете видеть в листингах 5.35 и 5.36, эти две сущности снабжены минимально необходимыми нотациями: `@Entity` с `@Id` и `@GeneratedValue` для первичного ключа и все. При использовании конфигурации в порядке исключения поставщик постоянства отобразит эти две сущности в две таблицы, а также внешний ключ для связи (из `customer`, указывающего на `address`). Отображение «один к одному» инициируется тем фактом, что `Address` является объявленной сущностью и включает сущность `Customer` как атрибут. Мы автоматически предполагаем связь, используя одну сущность как свойство другой, поэтому нам не нужна аннотация `@OneToOne`, поскольку она опирается на правила по умолчанию (листинги 5.37 и 5.38).

Листинг 5.37. Таблица CUSTOMER с внешним ключом к ADDRESS

```
create table CUSTOMER (
  ID BIGINT not null,
  FIRSTNAME VARCHAR(255),
  LASTNAME VARCHAR(255),
  EMAIL VARCHAR(255),
  PHONENUMBER VARCHAR(255),
  ADDRESS_ID BIGINT,
  primary key (ID),
  foreign key (ADDRESS_ID) references ADDRESS(ID)
);
```

Листинг 5.38. Таблица ADDRESS

```
create table ADDRESS (
  ID BIGINT not null,
  STREET1 VARCHAR(255),
  STREET2 VARCHAR(255),
  CITY VARCHAR(255),
  STATE VARCHAR(255),
  ZIPCODE VARCHAR(255),
  COUNTRY VARCHAR(255),
  primary key (ID)
);
```

Как вы уже знаете, при использовании JPA, если не аннотировать атрибут, будут применены правила отображения по умолчанию. Таким образом, изначально внешний ключ будет носить имя `ADDRESS_ID` (листинг 5.37), которое представляет собой конкатенацию имени атрибута связи (здесь это `address`), символа «_» и имени столбца первичного ключа целевой таблицы (здесь им будет идентификатор столбца таблицы `ADDRESS`). Обратите также внимание, что в DDL-коде столбец `ADDRESS_ID` по умолчанию является `nullable`, а это означает, что ассоциация «один к одному» отображается в нуль (значение `null`) или единицу.

Для настройки отображения вы можете использовать две аннотации. Первая — `@OneToOne` (поскольку кардинальность отношения равна единице). Она позволяет модифицировать отдельные атрибуты ассоциации как таковой, например, подход,

посредством которого должна осуществляться ее выборка. В листинге 5.39 определен API-интерфейс аннотации @OneToOne.

Листинг 5.39. API-интерфейс аннотации @OneToOne

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface OneToOne {
  Class targetEntity() default void.class;
  CascadeType[] cascade() default {};
  FetchType fetch() default EAGER;
  boolean optional() default true;
  String mappedBy() default "";
  boolean orphanRemoval() default false;
}
```

Второй аннотацией является @JoinColumn (ее API-интерфейс очень схож с API-интерфейсом аннотации @Column из листинга 5.12). Она используется для настройки столбца соединения, то есть внешнего ключа владеющей стороны. В листинге 5.40 показано, как вы применили бы две эти аннотации.

Листинг 5.40. Сущность Customer с настроенным отображением связей

```
@Entity
public class Customer {
  @Id @GeneratedValue
  private Long id;
  private String firstName;
  private String lastName;
  private String email;
  private String phoneNumber;
  @OneToOne (fetch = FetchType.LAZY)
  @JoinColumn(name = "add_fk", nullable = false)
  private Address address;
  // Конструкторы, геттеры, сеттеры
}
```

В JPA внешний ключ называется столбцом соединения. Аннотация @JoinColumn позволяет вам настраивать отображение внешнего ключа. В листинге 5.40 она используется для переименования столбца внешнего ключа в ADD_FK, а также для того, чтобы сделать связь обязательной, отклонив значение null (nullable=false). Аннотация @OneToOne подсказывает поставщику постоянства о том, что выборка связи должна быть отложенной (подробнее об этом мы поговорим позднее).

Однонаправленная связь с использованием @OneToMany

Связь «один ко многим» наблюдается, когда один объект-источник ссылается на множество объектов-целей. Например, заказ на покупку включает несколько строк заказа (рис. 5.15). Строка заказа могла бы ссылаться на заказ на покупку с использованием аннотации @ManyToOne, однако это не тот случай, поскольку связь является однонаправленной. Order — это сторона «одного» и источник связи, а OrderLine — сторона «многих» и цель.

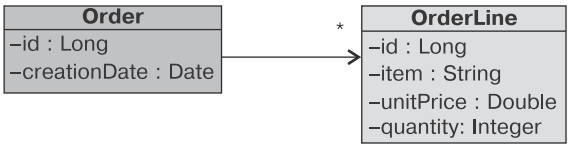


Рис. 5.15. Один заказ включает несколько строк

Отношения являются множественными, а навигация осуществляется только от `Order` по направлению к `OrderLine`. В Java эта множественность характеризуется интерфейсами `Collection`, `List` и `Set` из пакета `java.util`. В листинге 5.41 приведен код сущности `Order` с однонаправленной связью «один ко многим» относительно `OrderLine` (листинг 5.42).

Листинг 5.41. Order содержит orderLines

```
@Entity
public class Order {
  @Id @GeneratedValue
  private Long id;
  @Temporal(TemporalType.TIMESTAMP)
  private Date creationDate;
  private List<OrderLine> orderLines;
  // Конструкторы, геттеры, сеттеры
}
```

Листинг 5.42. OrderLine

```
@Entity
@Table(name = "order_line")
public class OrderLine {
  @Id @GeneratedValue
  private Long id;
  private String item;
  private Double unitPrice;
  private Integer quantity;
  // Конструкторы, геттеры, сеттеры
}
```

`Order` из листинга 5.41 не имеет каких-либо специальных аннотаций и опирается на парадигму «конфигурация в порядке исключения». Тот факт, что коллекция типа сущности используется как атрибут для этой сущности, по умолчанию инициирует отображение связи «один ко многим». Изначально однонаправленные связи «один ко многим» задействуют таблицу соединения с двумя столбцами внешнего ключа для сохранения информации о связях. Один столбец внешнего ключа будет ссылаться на таблицу `ORDER` и иметь тот же тип, что и ее первичный ключ, а другой будет ссылаться на таблицу `ORDER_LINE`. Имя этой таблицы соединения будет состоять из имен обеих сущностей, разделенных символом «_». Таблица соединения получит имя `ORDER_ORDER_LINE`, а в результате у нас будет схематическая структура, показанная на рис. 5.16.

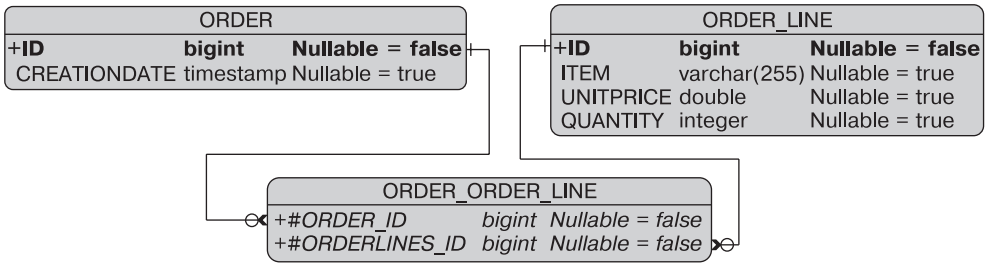


Рис. 5.16. Таблица соединения между ORDER и ORDER_LINE

Если вам не нравятся имена таблицы соединения и внешнего ключа либо если вы выполняете отображение в уже существующую таблицу, то можете воспользоваться аннотациями JPA для переопределения этих применяемых по умолчанию имен. Именем по умолчанию для столбца соединения является конкатенация имени сущности, символа «_» и имени первичного ключа, на который происходит ссылка. В то время как аннотация `@JoinColumn` может быть использована для изменения столбцов внешнего ключа, аннотация `@JoinTable` (листинг 5.43) позволяет сделать то же самое, если речь идет об отображении таблицы соединения. Вы также можете воспользоваться аннотацией `@OneToMany` (листинг 5.44), которая, как и `@OneToOne`, дает возможность настраивать саму связь (применение режима `fetch` и т. д.).

Листинг 5.43. API-интерфейс аннотации @JoinTable

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface JoinTable {
  String name() default "";
  String catalog() default "";
  String schema() default "";
  JoinColumn[] joinColumns() default {};
  JoinColumn[] inverseJoinColumns() default {};
  UniqueConstraint[] uniqueConstraints() default {};
  Index[] indexes() default {};
}
```

Листинг 5.44. Сущность Order с аннотированной связью «один ко многим»

```
@Entity
public class Order {
  @Id @GeneratedValue
  private Long id;
  @Temporal(TemporalType.TIMESTAMP)
  private Date creationDate;
  @OneToMany
  @JoinTable(name = "jnd_ord_line",
    joinColumns = @JoinColumn(name = "order_fk"),
    inverseJoinColumns = @JoinColumn(name = "order_line_fk") )
  private List<OrderLine> orderLines;
  // Конструкторы, геттеры, сеттеры
}
```

В случае с API-интерфейсом аннотации @JoinTable в листинге 5.42 вы можете видеть два атрибута типа @JoinColumn: joinColumns и inverseJoinColumns. Они различаются владеющей и противоположной сторонами. Элемент joinColumns характеризует владеющую сторону (владельца связи) и в нашем примере ссылается на таблицу ORDER. Элемент inverseJoinColumns определяет противоположную сторону, то есть цель связи, и ссылается на ORDER_LINE.

При использовании сущности Order (листинг 5.44) вы можете добавить аннотации @OneToMany и @JoinTable для атрибута orderLines, переименовав таблицу соединения в JND_ORD_LINE (вместо ORDER_ORDER_LINE), а также два столбца внешнего ключа.

Сущность Order из листинга 5.44 будет отображена в таблицу соединения, описанную в листинге 5.45.

Листинг 5.45. Структура таблицы соединения между ORDER и ORDER_LINE

```
create table JND_ORD_LINE (
  ORDER_FK BIGINT not null,
  ORDER_LINE_FK BIGINT not null,
  primary key (ORDER_FK, ORDER_LINE_FK),
  foreign key (ORDER_LINE_FK) references ORDER_LINE(ID),
  foreign key (ORDER_FK) references ORDER(ID)
);
```

Правило по умолчанию для однонаправленной связи «один ко многим» — использование таблицы соединения, однако его очень легко (и целесообразно, если речь идет об унаследованных базах данных) изменить таким образом, чтобы применялись внешние ключи. Для сущности Order необходимо предусмотреть аннотацию @JoinColumn вместо @JoinTable, что позволит изменить код, как показано в листинге 5.46.

Листинг 5.46. Сущность Order наряду со столбцом соединения

```
@Entity
public class Order {
  @Id @GeneratedValue
  private Long id;
  @Temporal(TemporalType.TIMESTAMP)
  private Date creationDate;
  @OneToMany(fetch = FetchType.EAGER)
  @JoinColumn(name = "order_fk")
  private List<OrderLine> orderLines;
  // Конструкторы, геттеры, сеттеры
}
```

Код сущности OrderLine (показанной в листинге 5.46) не изменится. Обратите внимание, что в листинге 5.46 аннотация @OneToMany переключает режим по умолчанию fetch (поменяв LAZY на EAGER). При использовании вами @JoinColumn стратегия внешнего ключа затем обеспечивает отображение однонаправленной связи. Внешний ключ переименовывается с помощью аннотации в ORDER_FK и располагается в целевой таблице (ORDER_LINE). В результате получается структура базы данных, показанная на рис. 5.17. Таблица соединения отсутствует, а ссылка между обеими таблицами осуществляется по внешнему ключу ORDER_FK.

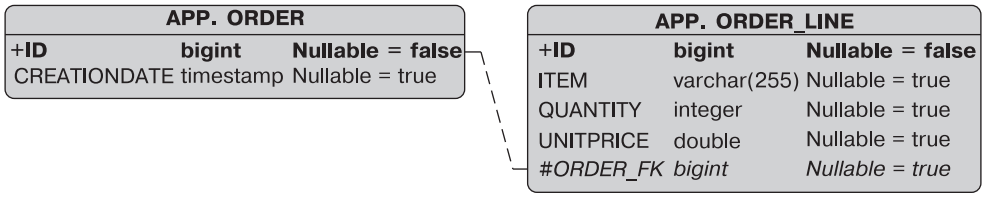


Рис. 5.17. Столбец соединения между ORDER и ORDER_LINE

Двунаправленная связь с использованием @ManyToMany

Двунаправленная связь «многие ко многим» имеет место, когда один объект-источник ссылается на много целей и когда цель ссылается на много источников. Например, CD-альбом создается несколькими артистами, а один артист принимает участие в создании нескольких CD-альбомов. В мире Java у каждой сущности будет коллекция целевых сущностей. В реляционном мире единственный способ отобразить связь «многие ко многим» — использовать таблицу соединения (столбец соединения не поможет). Кроме того, как вы уже видели ранее, при двунаправленной связи вам потребуется явным образом определить владельца (с помощью элемента `mappedBy`).

Если исходить из того, что сущность `Artist` является владельцем связи, то это будет означать, что противоположным владельцем (листинг 5.47) выступает сущность `CD`, которой необходимо, чтобы элемент `mappedBy` был использован в сочетании с ее аннотацией `@ManyToMany`. Элемент `mappedBy` сообщит поставщику постоянства о том, что `appearsOnCDs` — это имя соответствующего атрибута владеющей сущности.

Листинг 5.47. Несколько артистов приняли участие в создании одного CD-альбома

```
@Entity
public class CD {
  @Id @GeneratedValue
  private Long id;
  private String title;
  private Float price;
  private String description;
  @ManyToMany(mappedBy = "appearsOnCDs")
  private List<Artist> createdByArtists;
  // Конструкторы, геттеры, сеттеры
}
```

Таким образом, если сущность `Artist` является владельцем связи, как показано в листинге 5.48, то она будет использоваться для настройки отображения таблицы соединения посредством аннотаций `@JoinTable` и `@JoinColumn`.

Листинг 5.48. Один артист принял участие в создании нескольких CD-альбомов

```
@Entity
public class Artist {
  @Id @GeneratedValue
  private Long id;
  private String firstName;
```

```
    private String lastName;
    @ManyToMany
    @JoinTable(name = "jnd_art_cd", ➥
      joinColumns = @JoinColumn(name = "artist_fk"), ➥
      inverseJoinColumns = @JoinColumn(name = "cd_fk"))
    private List<CD>appearsOnCDs;
    // Конструкторы, геттеры, сеттеры
}
```

Как вы можете видеть в листинге 5.48, таблица соединения между `Artist` и `CD` переименовывается в `JND_ART_CD`, как и каждый столбец соединения (благодаря аннотации `@JoinTable`). Элемент `joinColumns` ссылается на владеющую сторону (`Artist`), а `inverseJoinColumns` — на противоположную владеющую сторону (`CD`). Соответствующая структура базы данных показана на рис. 5.18.

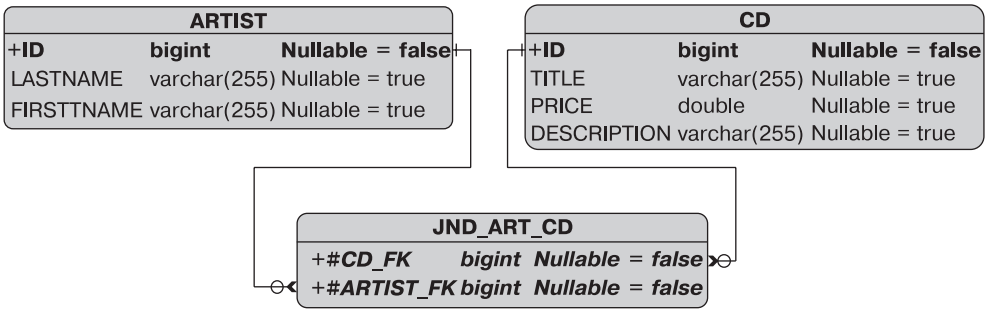


Рис. 5.18. ARTIST, CD и таблица соединения

Следует отметить, что при двунаправленной связи «многие ко многим» и «один к одному» любая из сторон может быть обозначена как владеющая. Независимо от того, какая из сторон будет обозначена как владелец, владеющая сторона должна включать элемент `mappedBy`. В противном случае поставщик будет считать, что обе стороны являются владельцами, и воспринимать все это как две отдельные однонаправленные связи «один ко многим». В результате этого могло бы получиться четыре таблицы: `ARTIST` и `CD` плюс две таблицы соединения с именами `ARTIST_CD` и `CD_ARTIST`. И недопустимым было бы наличие `mappedBy` на обеих сторонах.

Выборка связей

Все аннотации, которые вы видели ранее (`@OneToOne`, `@OneToMany`, `@ManyToOne` и `@ManyToMany`), определяют атрибут выборки, указывающий, что загрузка ассоциированных объектов должна быть незамедлительной или отложенной с результирующим влиянием на производительность. В зависимости от вашего приложения доступ к одним связям осуществляется чаще, чем к другим. В таких ситуациях вы можете оптимизировать производительность, загружая информацию из базы данных, когда сущность подвергается первоначальному чтению (незамедлительная загрузка) или когда к ней осуществляется доступ (отложенная загрузка). В качестве примера взглянем на некоторые крайние случаи.

Представим себе четыре сущности, которые все связаны между собой и имеют разные отношения («один к одному», «один ко многим»). В первом случае (рис. 5.19) между всеми сущностями будут связи EAGER. Это означает, что, как только вы загрузите Class1 (произведя поиск по идентификатору или выполнив запрос), все зависимые объекты будут автоматически загружены в память. Это может отразиться на производительности вашей системы.

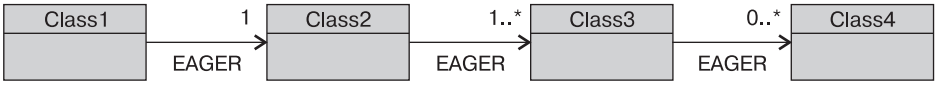


Рис. 5.19. Четыре сущности со связями EAGER

Если взять противоположный сценарий, то все связи будут задействовать режим fetch, обеспечивающий отложенную выборку (рис. 5.20). При загрузке Class1 ничего больше загружаться не будет (за исключением, конечно же, непосредственных атрибутов Class1). Вам потребуется явным образом получить доступ к Class2 (например, с помощью метода-геттера), чтобы дать команду поставщику постоянства на загрузку информации из базы данных и т. д. Если вы захотите управлять всем графом объекта, то вам потребуется явным образом вызывать каждую сущность.

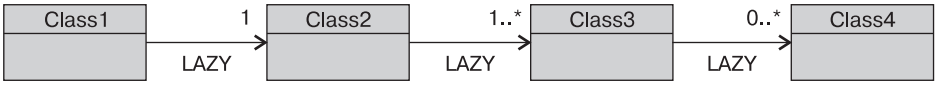


Рис. 5.20. Четыре сущности со связями LAZY

Однако не следует думать, что EAGER — это плохо, а LAZY — хорошо. EAGER поместит все данные в память с помощью небольшого количества операций доступа к базе данных (поставщик постоянства, вероятно, будет использовать запросы с соединением для связи таблиц и извлечения данных). В случае с LAZY вы не рискуете заполнить всю используемую вами память, поскольку будете контролировать, какой объект будет загружаться. Однако вам придется каждый раз осуществлять доступ к базе данных.

Параметр fetch очень важен, поскольку, если его неправильно использовать, это может привести к проблемам с производительностью. У каждой аннотации есть значение fetch по умолчанию, которое вам необходимо знать, и, если оно окажется неподходящим, изменить его (табл. 5.2).

Таблица 5.2. Стратегии выборки по умолчанию

Аннотация	Стратегия выборки по умолчанию
@OneToOne	EAGER
@ManyToOne	EAGER
@OneToMany	LAZY
@ManyToMany	LAZY

Если при разгрузке заказа вам постоянно будет нужен доступ к его строкам в вашем приложении, то, возможно, будет целесообразно изменить режим fetch по умолчанию аннотации @OneToMany на EAGER (листинг 5.49).

Листинг 5.49. Order со связью EAGER с OrderLine

```
@Entity
public class Order {
  @Id @GeneratedValue
  private Long id;
  @Temporal(TemporalType.TIMESTAMP)
  private Date creationDate;
  @OneToMany(fetch = FetchType.EAGER)
  private List<OrderLine> orderLines;
  // Конструкторы, геттеры, сеттеры
}
```

Упорядочение связей

При связях «один ко многим» или «многие ко многим» ваши сущности имеют дело с коллекциями объектов. На стороне Java эти коллекции обычно неупорядочены. В таблицах реляционных баз данных тоже не соблюдается какой-либо порядок. Следовательно, если у вас возникнет необходимость в упорядоченном списке, то вам придется либо отсортировать свою коллекцию программным путем, либо воспользоваться JPQL-запросом с предложением ORDER BY. У JPA имеются более простые механизмы, основанные на аннотациях, которые могут помочь в упорядочении связей.

@OrderBy

Динамическое упорядочение может быть обеспечено благодаря аннотации @OrderBy. «Динамическое» оно потому, что вы упорядочиваете элементы коллекции при извлечении ассоциации.

В примере приложения CD-BookStore пользователям предоставляется возможность писать новости о музыке и книгах. Эти новости затем выкладываются на сайте, а после их публикации люди могут добавлять к ним комментарии (листинг 5.50). Вам необходимо, чтобы комментарии выводились на сайте в хронологическом порядке.

Листинг 5.50. Сущность Comment с postedDate

```
@Entity
public class Comment {
  @Id @GeneratedValue
  private Long id;
  private String nickname;
  private String content;
  private Integer note;
  @Column(name = "posted_date")
  @Temporal(TemporalType.TIMESTAMP)
  private Date postedDate;
  // Конструкторы, геттеры, сеттеры
}
```

Комментарии моделируются с использованием сущности Comment, показанной в листинге 5.50. У нее имеется content, она размещается пользователем (идентифицируется параметром nickname), оставляющим примечания к новостям, кроме того, она располагает postedDate типа TIMESTAMP, который автоматически создается системой. В случае с сущностью News, показанной в листинге 5.51, вы захотите иметь возможность упорядочивать список комментариев согласно дате их размещения в убывающем порядке. Для этого вам потребуется прибегнуть к аннотации @OrderBy в сочетании с аннотацией @OneToMany.

Листинг 5.51. Комментарии, которые относятся к сущности News, упорядочиваются согласно дате размещения в убывающем порядке

```
@Entity
public class News {
  @Id @GeneratedValue
  private Long id;
  @Column(nullable = false)
  private String content;
  @OneToMany(fetch = FetchType.EAGER)
  @OrderBy("postedDate DESC")
  private List<Comment> comments;
  // Конструкторы, геттеры, сеттеры
}
```

Аннотация @OrderBy принимает имена атрибутов, которые должны быть отсортированы (атрибут postedDate), а также метод (сортировка в возрастающем или убывающем порядке). Строка ASC или DESC может быть использована для обеспечения сортировки соответственно либо в возрастающем, либо в убывающем порядке. Аннотация @OrderBy может охватывать несколько столбцов. Если вам потребуется выполнить упорядочение согласно дате размещения и примечаниям, то вы сможете воспользоваться OrderBy("postedDate DESC, note ASC").

Аннотация @OrderBy никак не влияет на отображение базы данных. Поставщик постоянства просто информируется о необходимости использовать предложение ORDER BY при извлечении коллекции во время выполнения.

@OrderColumn

Версия JPA 1.0 позволяла осуществлять динамическое упорядочение с использованием аннотации @OrderBy, однако не предусматривала возможности поддерживать постоянное упорядочение. С выходом JPA 2.0 это стало возможным благодаря добавлению аннотации @OrderColumn (ее API-интерфейс схож с API-интерфейсом аннотации @Column из листинга 5.12). Эта аннотация информирует поставщика постоянства о том, что он должен обеспечить поддержку упорядоченного списка с использованием отдельного столбца, в котором располагается индекс. @OrderColumn определяет этот отдельный столбец.

Воспользуемся примером сущностей News и Comment и немного изменим его. На этот раз у сущности Comment, показанной в листинге 5.52, не будет атрибута postedDate и, следовательно, сортировка комментариев в хронологическом порядке окажется невозможной.

Листинг 5.52. Сущность Comment без postedDate

```
@Entity
public class Comment {
  @Id @GeneratedValue
  private Long id;
  private String nickname;
  private String content;
  private Integer note;
  // Конструкторы, геттеры, сеттеры
}
```

Для сохранения сортировки без даты размещения в случае с сущностью News (показанной в листинге 5.53) можно аннотировать связь с помощью @OrderColumn. Тогда поставщик постоянства отобразит сущность News в таблицу с дополнительным столбцом для сохранения сортировки.

Листинг 5.53. Обеспечение постоянства сортировки комментариев

```
@Entity
public class News {
  @Id @GeneratedValue
  private Long id;
  @Column(nullable = false)
  private String content;
  @OneToMany(fetch = FetchType.EAGER)
  @OrderColumn(name = "posted_index")
  private List<Comment> comments;
  // Конструкторы, геттеры, сеттеры
}
```

В листинге 5.53 @OrderColumn переименовывает дополнительный столбец в POSTED_INDEX. Если это имя не будет изменено, то по умолчанию имя столбца будет представлять собой конкатенацию имени атрибута сущности и строки _ORDER (COMMENTS_ORDER в нашем примере). Этот столбец должен иметь числовой тип. Соответствующая упорядоченная связь будет отображена в отдельную таблицу соединения, как показано далее:

```
create table NEWS_COMMENT (
    NEWS_ID BIGINT not null,
    COMMENTS_ID BIGINT not null,
    POSTED_INDEX INTEGER
);
```

Есть особенности, влияющие на производительность, о которых вам следует знать; как и в случае с аннотацией @OrderColumn, поставщик постоянства должен отслеживать изменения индекса. Он отвечает за поддержку порядка при вставке, удалении и переупорядочении. Если информация окажется вставлена в середину уже существующего отсортированного списка данных, то поставщику постоянства придется переупорядочить весь индекс.

Переносимым приложениям не следует ожидать, что список будет упорядочен базой данных, предполагая, что движки некоторых баз данных автоматически

оптимизируют их индексы, благодаря чему таблица данных выглядит отсортированной. Вместо этого для них следует использовать конструкцию `@OrderColumn` либо `@OrderBy`. Нужно отметить, что вы не сможете одновременно задействовать обе эти аннотации.

Отображение наследования

Объектно-ориентированные языки задействуют парадигму наследования с момента своего появления. C++ допускает множественное наследование, а Java поддерживает наследование от одного класса. Применяя объектно-ориентированные языки, разработчики обычно повторно используют код, наследуя атрибуты и поведения корневых классов.

Вы только что изучили связи, а связи между сущностями очень просто отображаются в реляционную базу данных. Однако с наследованием дело обстоит подругому. В реляционном мире наследование — неизвестная концепция, которая изначально не реализована там. Концепция наследования предполагает использование ряда трюков при сохранении объектов в реляционную базу данных.

Как превратить иерархическую модель в плоскую реляционную модель? JPA предлагает на выбор три разные стратегии.

- *Иерархическая стратегия «одна таблица на класс»* — совокупность атрибутов всей иерархии сущностей отображается в одну таблицу (применяется по умолчанию).
- *Стратегия «соединенный подкласс»* — при этом подходе каждая сущность в иерархии, конкретная или абстрактная, отображается в свою специально отведенную для этого таблицу.
- *Стратегия «таблица на конкретный класс»* — при использовании этой стратегии каждая иерархия конкретных сущностей отображается в свою отдельную таблицу.

ПРИМЕЧАНИЕ

Поддержка стратегии отображения «таблица на конкретный класс» в версии JPA 2.1 все еще необязательна. Лучше не применять ее для переносимых приложений до тех пор, пока ее поддержка не станет обязательной официально.

Выгодно используя легкость применения аннотаций, JPA 2.1 обеспечивает поддержку определения и отображения иерархий наследования, включая сущности, абстрактные сущности, отображаемые классы и временные классы. Аннотация `@Inheritance` применяется в отношении корневой сущности с целью продиктовать стратегию отображения для нее самой и для листовых классов. JPA также переносит объектное понятие переопределения в сферу отображения, которое позволяет дочерним классам переопределять атрибуты корневого класса. В следующем разделе вы также увидите, как тип доступа может быть использован при наследовании для смешения доступа к полям с доступом к свойствам.

Стратегии наследования

В том, что касается отображения наследования, JPA поддерживает три разные стратегии. При наличии иерархии сущностей одна из них всегда выступает в каче-

стве корневой. Класс корневой сущности может определить стратегию наследования, используя элемент strategy аннотации @Inheritance, согласно одному из вариантов, определенных в перечислимом типе javax.persistence.InheritanceType. В противном случае будет задействована иерархическая стратегия по умолчанию «одна таблица на класс». Чтобы исследовать каждую стратегию, я расскажу вам о том, как отобразить сущности CD и Book, которые наследуют от сущности Item (рис. 5.21).

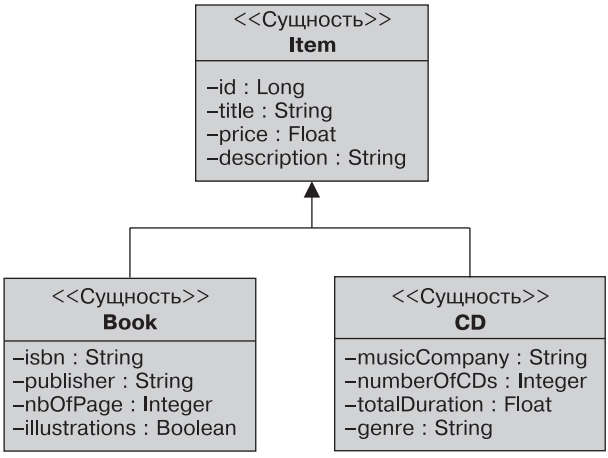


Рис. 5.21. Иерархия наследования между CD, Book и Item

Сущность Item является корневой и содержит атрибуты id, title, description и price. Обе сущности — CD и Book — наследуют от Item. Каждый из этих листовых классов привносит дополнительные атрибуты, например isbn в случае с сущностью Book или totalDuration, если вести речь о сущности CD.

Иерархическая стратегия «одна таблица на класс»

По умолчанию используется стратегия отображения наследования «одна таблица на класс». При ней все сущности в иерархии отображаются в одну таблицу. Поскольку она применяется по умолчанию, вы можете вообще не использовать аннотацию @Inheritance в сочетании с корневой сущностью (благодаря конфигурации в порядке исключения), что и было сделано с сущностью Item (листинг 5.54).

Листинг 5.54. Сущность Item задействует стратегию по умолчанию «одна таблица на класс»

```
@Entity
public class Item {
  @Id @GeneratedValue
  protected Long id;
  protected String title;
  protected Float price;
  protected String description;
  // Конструкторы, геттеры, сеттеры
}
```

Item (см. листинг 5.54) является корневой по отношению к сущностям Book (листинг 5.55) и CD (листинг 5.56). Эти сущности наследуют атрибуты от Item, а также используемую по умолчанию стратегию наследования, поэтому нет нужды в аннотации @Inheritance.

Листинг 5.55. Book расширяет Item

```
@Entity
public class Book extends Item {
  private String isbn;
  private String publisher;
  private Integer nbOfPage;
  private Boolean illustrations;
  // Конструкторы, геттеры, сеттеры
}
```

Листинг 5.56. CD расширяет Item

```
@Entity
public class CD extends Item {
  private String musicCompany;
  private Integer numberOfCDs;
  private Float totalDuration;
  private String genre;
  // Конструкторы, геттеры, сеттеры
}
```

Учитывая то, что вы видели до настоящего момента, без наследования эти три сущности были бы отображены в их собственные отдельные таблицы, однако с наследованием все будет по-другому. При использовании стратегии «одна таблица на класс» все они окажутся в одной и той же таблице базы данных, имя которой по умолчанию будет совпадать с именем корневого класса — ITEM. Ее структура показана на рис. 5.22.

ITEM		
+ID	**bigint**	**Nullable = false**
DTYPE	varchar(31)	Nullable = true
TITLE	varchar(255)	Nullable = false
PRICE	double	Nullable = false
DESCRIPTION	varchar(255)	Nullable = true
ILLUSTRATIONS	smallint	Nullable = true
ISBN	varchar(255)	Nullable = true
NBOFPAGE	integer	Nullable = true
PUBLISHER	varchar(255)	Nullable = true
MUSICCOMPANY	varchar(255)	Nullable = true
NUMBEROFCDS	integer	Nullable = true
TOTALDURATION	double	Nullable = true
GENRE	varchar(255)	Nullable = true

Рис. 5.22. Структура таблицы ITEM

Как вы можете видеть на рис. 5.22, в таблице ITEM собраны все атрибуты сущностей Item, Book и CD. Однако есть дополнительный столбец, который не относится к атрибутам какой-либо из этих сущностей — это столбец дискриминатора DTYPE.

Таблица ITEM будет наполнена информацией, касающейся элементов, книг и CD-альбомов. При доступе к данным поставщику постоянства потребуется знать, какая строка к какой сущности относится. Таким образом, поставщик создаст экземпляр соответствующего типа объекта (Item, Book или CD) при чтении таблицы ITEM. Вот почему столбец дискриминатора используется для того, чтобы явным образом указать тип в каждой строке.

На рис. 5.23 показан фрагмент таблицы ITEM с данными. Как вы можете видеть, в стратегии «одна таблица на класс» имеются некоторые бреши; не каждый столбец подходит для любой из сущностей. В первой строке располагаются данные, касающиеся сущности Item (имя этой сущности содержится в столбце DTYPE). В случае с Item в таблице имеется только название, цена и описание (см. листинг 5.53), при этом отсутствует название компании звукозаписи, ISBN-номер и т. д. Поэтому соответствующие столбцы всегда будут оставаться пустыми.

ID	DTYPE	НАЗВАНИЕ	ЦЕНА	ОПИСАНИЕ	КОМПАНИЯ ЗВУКОЗАПИСИ	ISBN	...
1	Item	Ручка	2.10	Красивая черная ручка			...
2	CD	«Соултрейн»	23.50	Потрясающий джазовый альбом	Prestige		...
3	CD	«Прелести Зут»	18	Один из лучших альбомов Заппы	Warner		...
4	Book	«Роботы зари»	22.30	Роботы повсюду		0-554-456	...
5	Book	«Автостопом по Галактике»	17.50	Интересная книга на тему высоких технологий		1-278-983	...

Рис. 5.23. Фрагмент таблицы ITEM, наполненной данными

Столбец дискриминатора по умолчанию имеет имя DTYPE, тип String (отображаемый в VARCHAR) и содержит имя сущности. Если используемые по умолчанию значения окажутся неподходящими, то аннотация @DiscriminatorColumn позволит вам изменить имя и тип данных. По умолчанию значением этого столбца является имя сущности, на которую он ссылается, однако сущность может переопределить это значение благодаря аннотации @DiscriminatorValue.

В листинге 5.57 столбец дискриминатора переименовывается в DISC (вместо DTYPE), а также изменяется его тип данных с String на Char. Тогда значение дискриминатора каждой из сущностей должно измениться соответственно с Item на I, с Book на B (листинг 5.58) и с CD на C (листинг 5.59).

Листинг 5.57. Item переопределяет столбец дискриминатора

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn (name="disc", ➥
                      discriminatorType = DiscriminatorType.CHAR)
```

```
@DiscriminatorValue("I")
public class Item {
  @Id @GeneratedValue
  protected Long id;
  protected String title;
  protected Float price;
  protected String description;
  // Конструкторы, геттеры, сеттеры
}
```

Листинг 5.58. Book переопределяет значение дискриминатора на B @Entity

```
@DiscriminatorValue("B")
public class Book extends Item {
  private String isbn;
  private String publisher;
  private Integer nbOfPage;
  private Boolean illustrations;
  // Конструкторы, геттеры, сеттеры
}
```

Листинг 5.59. CD переопределяет значение дискриминатора на C @Entity

```
@DiscriminatorValue("C")
public class CD extends Item {
  private String musicCompany;
  private Integer numberOfCDs;
  private Float totalDuration;
  private String genre;
  // Конструкторы, геттеры, сеттеры
}
```

Корневая сущность `Item` один раз определяет столбец дискриминатора для иерархии сущностей с использованием `@DiscriminatorColumn`. Затем она изменяет свое значение по умолчанию на `I` благодаря `@DiscriminatorValue`. Дочерним сущностям требуется переопределить только собственное значение дискриминатора.

Результат показан на рис. 5.24. Столбец дискриминатора и его значения отличаются от тех, что были приведены ранее на рис. 5.23.

ID	DISC	НАЗВАНИЕ	ЦЕНА	ОПИСАНИЕ	КОМПАНИЯ ЗВУКОЗАПИСИ	ISBN	...
1	I	Ручка	2.10	Красивая черная ручка			...
2	C	«Соултрейн»	23.50	Потрясающий джазовый альбом	Prestige		...
3	C	«Прелести Зут»	18	Один из лучших альбомов Заппы	Warner		...
4	B	«Роботы зари»	22.30	Роботы повсюду		0-554-456	...
5	B	«Автостопом по Галактике»	17.50	Интересная книга на тему высоких технологий		1-278-983	...

Рис. 5.24. Таблица ITEM, включающая другое имя и значения дискриминатора

Стратегия «одна таблица на класс» является самой легкой для понимания и хорошо работает, когда иерархия относительно проста и стабильна. Однако у нее имеются кое-какие недостатки: добавление новых сущностей в иерархию или атрибутов в уже существующие сущности влечет добавление новых столбцов в таблицу, миграцию данных и изменение индексов. Эта стратегия также требует, чтобы столбцы дочерних сущностей допускали значение `null`. Если столбец сущности `Book`, содержащий ISBN-номер, не окажется таковым, то вы больше не сможете вставлять данные, которые относятся к CD-альбомам, поскольку для сущности `CD` отсутствует значение такого столбца.

Стратегия «соединенный подкласс»

При использовании стратегии «соединенный подкласс» каждая сущность в иерархии отображается в свою таблицу. Корневая сущность отображается в таблицу, которая определяет первичный ключ, подлежащий использованию всеми таблицами в иерархии, а также столбец дискриминатора. Каждый подкласс представляется с помощью отдельной таблицы, содержащей его атрибуты (не унаследованные от корневого класса) и первичный ключ, который ссылается на первичный ключ корневой таблицы. Таблицы, не являющиеся корневыми, не содержат столбец дискриминатора.

Вы можете реализовать стратегию «соединенный подкласс», снабдив корневую сущность аннотацией `@Inheritance`, как показано в листинге 5.60 (код `CD` и `Book` останется таким же, как и раньше).

Листинг 5.60. Сущность Item с использованием стратегии «соединенный подкласс»

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Item {
  @Id @GeneratedValue
  protected Long id;
  protected String title;
  protected Float price;
  protected String description;
  // Конструкторы, геттеры, сеттеры
}
```

С точки зрения разработчика, стратегия «соединенный подкласс» естественна, поскольку состояния всех сущностей, абстрактных или конкретных, будут отображаться в разные таблицы. На рис. 5.25 показано, как будут отображены сущности `Item`, `Book` и `CD`.

Вы по-прежнему сможете использовать аннотации `@DiscriminatorColumn` и `@DiscriminatorValue` в случае с корневой сущностью для настройки столбца дискриминатора и изменения значений (столбец `DTYPE` располагается в таблице `ITEM`).

Стратегия «соединенный подкласс» интуитивно понятна и близка к тому, что вы знаете, исходя из механизма объектного наследования. Однако выполнение запросов может влиять на производительность. В названии этой стратегии присутствует слово «соединенный», так как для повторной сборки экземпляра подкласса таблицу подкласса необходимо соединить с таблицей корневого класса.

Чем глубже иерархия, тем больше соединений потребуется для сборки листовой сущности. Эта стратегия хорошо поддерживает полиморфные связи, однако требует, чтобы при создании экземпляров подклассов сущностей была проведена одна или несколько операций соединения. Это может привести к низкой производительности в случае с обширными иерархиями классов. Аналогичным образом запросы, которые охватывают всю иерархию классов, требуют проведения операций соединения между таблицами подклассов, приводящих к снижению производительности.

Стратегия «таблица на конкретный класс»

Если задействуется стратегия «таблица на класс» (или «таблица на конкретный класс»), то каждая сущность отображается в свою специально отведенную для этого таблицу, как при использовании стратегии «соединенный подкласс». Отличие состоит в том, что все атрибуты корневой сущности также будут отображены в столбцы таблицы дочерней сущности. С позиции базы данных эта стратегия денормализует модель и приводит к тому, что все атрибуты корневой сущности переопределяются в таблицах всех листовых сущностей, которые наследуют от нее. При стратегии «таблица на конкретный класс» нет совместно используемой таблицы, совместно используемых столбцов и столбца дискриминатора. Единственное требование состоит в том, что все таблицы должны совместно пользоваться общим первичным ключом — одинаковым для всех таблиц в иерархии.

Для отображения нашего примера с применением этой стратегии потребуется указать `TABLE_PER_CLASS` в аннотации `@Inheritance` (листинг 5.61) корневой сущности (`Item`).

Листинг 5.61. Сущность Item с использованием стратегии «таблица на конкретный класс»

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Item {
  @Id @GeneratedValue
  protected Long id;
  protected String title;
  protected Float price;
  protected String description;
  // Конструкторы, геттеры, сеттеры
}
```

На рис. 5.26 показаны таблицы `ITEM`, `BOOK` и `CD`. Как вы можете видеть, в `BOOK` и `CD` имеются дубликаты столбцов `ID`, `TITLE`, `PRICE` и `DESCRIPTION` таблицы `ITEM`. Обратите внимание, что показанные таблицы не связаны.

Разумеется, помните, что каждая таблица может быть переопределена, если снабдить каждую сущность аннотацией `@Table`.

Стратегия «таблица на конкретный класс» хорошо работает при выполнении запросов к экземплярам одной сущности, поскольку ее применение схоже с использованием стратегии «одна таблица на класс»: запрос ограничивается одной

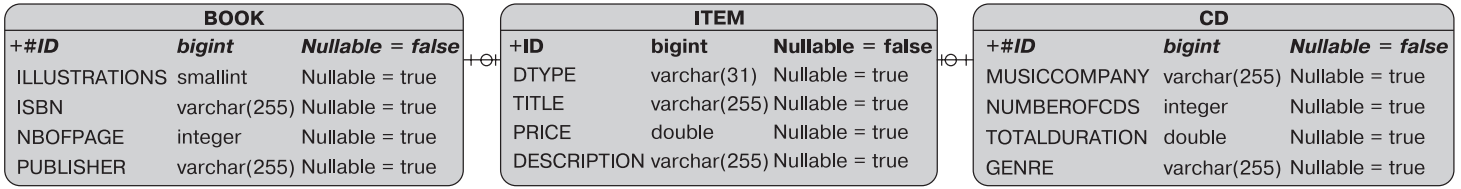


Рис. 5.25. Отображение наследования с применением стратегии «соединенный подкласс»

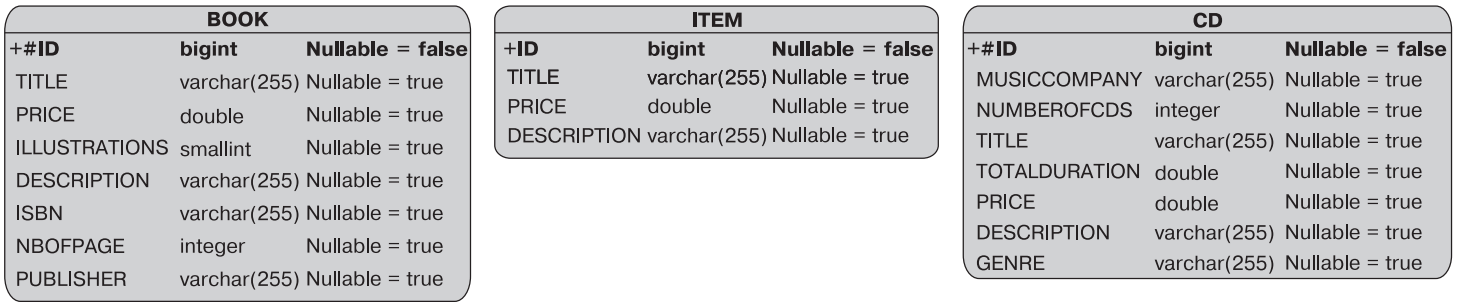


Рис. 5.26. Наличие дубликатов столбцов таблицы ITEM в таблицах BOOK и CD

таблицей. Недостаток этой стратегии заключается в том, что она делает полиморфные запросы в иерархии классов более затратными, чем другие стратегии (например, поиск всех элементов, включая CD-альбомы и книги). При применении этой стратегии запросы ко всем таблицам подклассов должны выполняться с использованием операции UNION, которая оказывается затратной, если охватывается большой объем данных. Поддержка этой стратегии в JPA 2.1 все еще необязательна.

Переопределение атрибутов

При использовании стратегии «таблица на конкретный класс» столбцы таблицы корневого класса дублируются в листовых таблицах. Они будут иметь одинаковые имена. Но что, если применять унаследованную базу данных, а столбцы будут обладать другими именами? JPA задействует аннотацию @AttributeOverride для переопределения отображения одного столбца и @AttributeOverrides, если речь идет о переопределении нескольких.

Чтобы переименовать столбцы ID, TITLE и DESCRIPTION в таблицах BOOK и CD, код сущности Item не потребуется изменять, однако придется задействовать аннотацию @AttributeOverride для сущностей Book (листинг 5.62) и CD (листинг 5.63).

Листинг 5.62. Book переопределяет некоторые столбцы, связанные с Item

```
@Entity
@AttributeOverrides({
  @AttributeOverride(name = "id", ➥
                     column = @Column(name = "book_id")),
  @AttributeOverride(name = "title", ➥
                     column = @Column(name = "book_title")),
  @AttributeOverride(name = "description", ➥
                     column = @Column(name = "book_description"))
})
public class Book extends Item {
  private String isbn;
  private String publisher;
  private Integer nbOfPage;
  private Boolean illustrations;
  // Конструкторы, геттеры, сеттеры
}
```

Листинг 5.63. CD переопределяет некоторые столбцы, связанные с Item

```
@Entity
@AttributeOverrides({
  @AttributeOverride(name = "id", ➥
                     column = @Column(name = "cd_id")),
  @AttributeOverride(name = "title", ➥
                     column = @Column(name = "cd_title")),
  @AttributeOverride(name = "description", ➥
                     column = @Column(name = "cd_description"))
})
public class CD extends Item {
  private String musicCompany;
```

```
    private Integer numberOfCDs;
    private Float totalDuration;
    private String genre;
    // Конструкторы, геттеры, сеттеры
}
```

Поскольку требуется переопределить несколько атрибутов, вам необходимо использовать аннотацию `@AttributeOverrides`, которая принимает массив аннотаций `@AttributeOverride`. После этого каждая аннотация указывает на атрибут сущности `Item` и переопределяет отображение столбца с помощью аннотации `@Column`. Таким образом, `name = "title"` ссылается на атрибут `title` сущности `Item`, а `@Column(name = "cd_title")` информирует поставщика постоянства о том, что `title` надлежит отобразить в столбец `CD_TITLE`. Результат показан на рис. 5.27.

ПРИМЕЧАНИЕ

Ранее в разделе «Встраиваемые объекты» этой главы вы видели, что встраиваемый объект может совместно использоваться несколькими сущностями (Address был встроен в Customer и Order). Поскольку встраиваемые объекты — это внутренняя часть владеющей сущности, в таблице каждой сущности также будут иметься дубликаты столбцов, связанных с этими объектами. Кроме того, @AttributeOverrides можно применять, если вам необходимо переопределить встраиваемые столбцы.

Типы классов в иерархии наследования

В примерах, приводившихся ранее для объяснения стратегий отображения, были задействованы только сущности. `Item`, как и `Book` и `CD`, является сущностью. Однако сущностям не всегда приходится наследовать от сущностей. В иерархии классов могут быть смешаны всевозможные разные классы: сущности, а также классы, которые не являются сущностями (или временные классы), абстрактные сущности и отображенные суперклассы. Наследование от этих классов разных типов будет влиять на отображение.

Абстрактная сущность

В приведенных ранее примерах сущность `Item` представляла собой конкретный класс. Она была снабжена аннотацией `@Entity` и не имела ключевого слова `abstract`, однако абстрактный класс тоже может быть определен как сущность. Абстрактная сущность отличается от конкретной только тем, что нельзя непосредственно создать ее экземпляр с помощью ключевого слова `new`. Она обеспечивает общую структуру данных для своих листовых сущностей (`Book` и `CD`) и придерживается соответствующих стратегий отображения. Для поставщика постоянства абстрактная сущность отображается как обычная сущность. Единственное отличие заключается в пространстве Java, а не в отображении.

Класс, который не является сущностью

Классы, которые не являются сущностями, также называют *временными*, а это означает, что они представляют собой POJO. Сущность может выступать подклассом по отношению к классу, который не является сущностью, либо такой класс

BOOK		
+BOOK_ID	**bigint**	**Nullable = false**
BOOK_TITLE	varchar(255)	Nullable = true
BOOK_DESCRIPTION	varchar(255)	Nullable = true
PRICE	double	Nullable = true
ILLUSTRATIONS	smallint	Nullable = true
ISBN	varchar(255)	Nullable = true
NBOFPAGE	integer	Nullable = true
PUBLISHER	varchar(255)	Nullable = true

ITEM		
+ID	**bigint**	**Nullable = false**
TITLE	varchar(255)	Nullable = true
PRICE	double	Nullable = true
DESCRIPTION	varchar(255)	Nullable = true

CD		
+CD_ID	**bigint**	**Nullable = false**
CD_TITLE	varchar(255)	Nullable = true
CD_DESCRIPTION	varchar(255)	Nullable = true
PRICE	double	Nullable = true
MUSICCOMPANY	varchar(255)	Nullable = true
NUMBEROFCDS	integer	Nullable = true
TOTALDURATION	double	Nullable = true
GENRE	varchar(255)	Nullable = true

Рис. 5.27. Таблицы BOOK и CD переопределяют столбцы таблицы ITEM

может расширять ее. Зачем вам могут понадобиться в иерархии классы, которые не являются сущностями? Объектное моделирование и наследование — это инструменты, с помощью которых совместно используются состояния и поведения. Классы, которые не являются сущностями, могут применяться для обеспечения общей структуры данных для листовых сущностей. Состояние суперкласса, не являеющегося сущностью, непостоянно, поскольку не управляется поставщиком постоянства (помните, что условием для того, чтобы класс управлялся поставщиком постоянства, является наличие аннотации `@Entity`).

Например, `Book` представляет собой сущность (листинг 5.65) и расширяет `Item`, который не является сущностью (у `Item` нет аннотаций), как показано в листинге 5.64.

Листинг 5.64. Item — простой POJO без @Entity

```
public class Item {
  protected String title;
  protected Float price;
  protected String description;
  // Конструкторы, геттеры, сеттеры
}
```

Сущность `Book` (см. листинг 5.65) наследует от `Item`, поэтому Java-код может получить доступ к атрибутам `title`, `price` и `description`, а также к любому другому методу, который определен обычным, объектно-ориентированным, путем. `Item` может быть конкретным или абстрактным и не влияет на финальное отображение.

Листинг 5.65. Сущность Book расширяет POJO

```
@Entity
public class Book extends Item {
  @Id @GeneratedValue
  private Long id;
  private String isbn;
  private String publisher;
  private Integer nbOfPage;
  private Boolean illustrations;
  // Конструкторы, геттеры, сеттеры
}
```

Класс `Book` является сущностью и наследует от `Item`. Однако в таблице были бы отображены только атрибуты `Book`. Атрибуты `Item` в структуре таблицы, определенной в листинге 5.66, отсутствуют. Чтобы обеспечить постоянство `Book`, вам потребуется создать экземпляр `Book`, задать значения для любых атрибутов по вашему желанию (`title`, `price`, `isbn`, `publisher` и т. д.), однако будет обеспечено постоянство только атрибутов `Book` (`id`, `isbn` и т. д.).

Листинг 5.66. В таблице BOOK атрибуты Item отсутствуют

```
create table BOOK (
  ID BIGINT not null,
  ILLUSTRATIONS SMALLINT,
```

```
  ISBN VARCHAR(255),
  NBOFPAGE INTEGER,
  PUBLISHER VARCHAR(255),
  primary key (ID)
);
```

Отображенный суперкласс

JPA определяет особый тип классов, называемых *отображенными суперклассами*, для совместного использования состояний и поведений, а также информации об отображении, которые от них наследуют сущности. Однако отображенные суперклассы не являются сущностями. Они не управляются поставщиком постоянства, у них нет какой-либо таблицы для отображения в нее, к ним нельзя выполнять запросы и они не могут состоять в связях, однако должны обеспечивать постоянные свойства для любых сущностей, которые расширяют их. Они аналогичны встраиваемым классам за исключением того, что могут быть использованы в сочетании с наследованием. Чтобы показать, что класс является отображенным суперклассом, нужно снабдить его аннотацией `@MappedSuperclass`.

При использовании корневого класса `Item` снабжается аннотацией `@MappedSuperclass`, а не `@Entity`, как показано в листинге 5.67. В данном случае определяется стратегия наследования (`JOINED`), а также аннотируются некоторые из его атрибутов с использованием `@Column`. Однако, поскольку отображенные суперклассы не отображаются в таблицах, не допускается применять аннотацию `@Table`.

Листинг 5.67. Item — отображенный суперкласс

```
@MappedSuperclass
@Inheritance(strategy = InheritanceType.JOINED)
public class Item {
  @Id @GeneratedValue
  protected Long id;
  @Column(length = 50, nullable = false)
  protected String title;
  protected Float price;
  @Column(length = 2000)
  protected String description;
  // Конструкторы, геттеры, сеттеры
}
```

Как вы можете видеть в листинге 5.67, атрибуты `title` и `description` снабжены аннотацией `@Column`. В листинге 5.68 показана сущность `Book`, расширяющая `Item`.

Листинг 5.68. Сущность Book расширяет Item

```
@Entity
public class Book extends Item {
  private String isbn;
  private String publisher;
  private Integer nbOfPage;
  private Boolean illustrations;
  // Конструкторы, геттеры, сеттеры
}
```

Эта иерархия будет отображена только в одну таблицу. Item не является сущностью и не содержит таблиц. Атрибуты Item и Book были бы отображены в столбцы таблицы BOOK, кроме того, отображенные суперклассы также совместно используют свою информацию об отображении. Аннотации @Column отображенного суперкласса Item будут унаследованы. Однако, поскольку такие суперклассы не являются сущностями, которые находятся под управлением, вы не смогли бы, к примеру, обеспечить их постоянство или выполнять к ним запросы. В листинге 5.69 показана структура таблицы BOOK с настроенными столбцами TITLE и DESCRIPTION.

Листинг 5.69. Таблица BOOK включает все атрибуты Item

```
create table BOOK (
  ID BIGINT not null,
  TITLE VARCHAR(50) not null,
  PRICE DOUBLE(52, 0),
  DESCRIPTION VARCHAR(2000),
  ILLUSTRATIONS SMALLINT,
  ISBN VARCHAR(255),
  NBOFPAGE INTEGER,
  PUBLISHER VARCHAR(255),
  primary key (ID)
);
```

Резюме

Благодаря конфигурации в порядке исключения немногое требуется для того, чтобы отобразить сущности в таблицах. Проинформируйте поставщика постоянства о том, что класс на самом деле является сущностью (посредством @Entity), атрибут — его идентификатором (посредством @Id), а JPA сделает все остальное. Эта глава могла бы быть намного короче, если бы мы придерживались в ней только того, что применяется по умолчанию. JPA обладает очень богатым набором аннотаций для настройки всех мелких деталей объектно-реляционного отображения (а также эквивалентного XML-отображения).

Элементарные аннотации могут быть использованы в отношении атрибутов (@Basic, @Temporal и т. д.) или классов для настройки отображения. Вы можете изменить имя таблицы либо тип первичного ключа или даже избегать отображения с помощью аннотации @Transient. Применяя JPA, вы можете отображать коллекции базовых типов или встраиваемых объектов. В зависимости от своей бизнес-модели вы можете отображать связи (@OneToOne, @ManyToMany и т. д.) с разными направлениями и множественностью. То же самое касается и наследования (@Inheritance, @MappedSuperclass и т. д.), при котором допустимо использовать разные стратегии для отображения иерархий, где смешаны сущности и классы, не являющиеся сущностями.

В этой главе внимание было сосредоточено на статической части JPA, или на том, как отображать сущности в таблицах. В следующей главе рассматриваются динамические темы: как управлять этими сущностями и выполнять к ним запросы.

Глава 6
Управление постоянными объектами

У Java Persistence API имеется две стороны. Первая — это способность отображать объекты в реляционные базы данных. Конфигурация в порядке исключения дает поставщикам постоянства возможность выполнять бóльшую часть работы с использованием малого количества кода, а функционал JPA также позволяет осуществлять настроенное отображение из объектов в таблицы с помощью аннотаций или XML-дескрипторов. JPA предлагает широкий спектр настроек, начиная с простого отображения (изменения имени столбца) и заканчивая более сложным (наследованием). Благодаря этому вы сможете отобразить почти любую объектную модель в унаследованной базе данных.

Другая сторона JPA — это способность выполнять запросы к этим отображенным объектам. В JPA централизованной службой для манипулирования экземплярами сущностей является менеджер сущностей. Это API-интерфейс для создания, поиска, удаления и синхронизации объектов с базой данных. Он также позволяет выполнять разнообразные JPQL-запросы, например динамические, статические или «родные» запросы к сущностям. При использовании менеджера сущностей также возможно применение механизмов блокировки.

Мир баз данных опирается на язык структурированных запросов. Этот язык программирования предназначен для управления реляционными данными (извлечение, вставка, обновление и удаление), а его синтаксис является табличноориентированным. Вы можете осуществлять выборку столбцов из таблицы, состоящей из строк, соединять таблицы, комбинировать результаты двух SQL-запросов посредством объединений и т. д. Здесь нет объектов, а есть только строки, столбцы и таблицы. В мире Java, где мы манипулируем объектами, язык, созданный для работы с таблицами (SQL), необходимо «изогнуть» таким образом, чтобы он сочетался с языком, который базируется на объектах (Java). Именно здесь в дело вступает язык запросов Java Persistence Query Language.

JPQL — это язык, определенный в JPA для выполнения запросов к сущностям, которые располагаются в реляционных базах данных. Синтаксис JPQL похож на синтаксис SQL, однако используется в отношении объектов-сущностей, а не взаимодействует непосредственно с таблицами баз данных. JPQL не видит структуры основной базы данных и не имеет дела с таблицами или столбцами, а работает с объектами и атрибутами. Для этого он задействует точечную (.) нотацию, которая знакома Java-разработчикам.

Из этой главы вы узнаете, как управлять постоянными объектами. Это означает, что вы научитесь проводить операции создания, чтения, обновления и удаления

(CRUD) с помощью менеджера сущностей, а также выполнять комплексные запросы с использованием JPQL. В этой главе также рассказывается о том, как JPA справляется с конкурентным доступом и работает с кэшем второго уровня. Она заканчивается объяснением жизненного цикла сущности и того, как JPA позволяет вам добавлять собственную бизнес-логику, когда в случае с сущностью имеют место определенные события.

Менеджер сущностей

Менеджер сущностей — центральный элемент JPA. Он управляет состоянием и жизненным циклом сущностей, а также позволяет выполнять запросы к сущностям в контексте постоянства. Менеджер сущностей отвечает за создание и удаление экземпляров постоянных сущностей и поиск сущностей по их первичному ключу. Он может блокировать сущности для защиты от конкурентного доступа, используя оптимистическую или пессимистическую блокировку, а также способен задействовать JPQL-запросы для извлечения сущностей согласно определенным критериям.

Когда менеджер сущностей получает ссылку на сущность, считается, что он управляет ею. До этого момента сущность рассматривается как обычный POJO-объект (то есть отсоединенный). Мощь JPA заключается в том, что сущности могут использоваться как обычные объекты на разных уровнях приложения и стать управляемыми менеджером сущностей, когда вам необходимо загрузить или вставить информацию в базу данных. Когда сущность находится под управлением, вы можете проводить операции, направленные на обеспечение постоянства, а менеджер сущностей будет автоматически синхронизировать состояние сущности с базой данных. Когда сущность оказывается отсоединенной (то есть не находится под управлением), она снова становится простым Java-объектом, который затем может быть использован на других уровнях (например, JavaServer Faces или JSF на уровне представления) без синхронизации его состояния с базой данных.

Что касается постоянства, то реальная работа здесь начинается с помощью менеджера сущностей. Он является интерфейсом, реализуемым поставщиком постоянства, который будет генерировать и выполнять SQL-операторы. Интерфейс `javax.persistence.EntityManager` представляет собой API-интерфейс для манипулирования сущностями (соответствующее подмножество приведено в листинге 6.1).

Листинг 6.1. Подмножество API-интерфейса EntityManager

```
public interface EntityManager {

  // EntityManagerFactory для создания EntityManager,
  // его закрытия и проверки того, открыт ли он
  EntityManagerFactory getEntityManagerFactory();
  void close();
  boolean isOpen();

  // Возвращает EntityTransaction
  EntityTransaction getTransaction();

  // Обеспечивает постоянство, слияние сущности в базе данных,
```

```
// а также ее удаление оттуда
void persist(Object entity);
<T> T merge(T entity);
void remove(Object entity);

// Обеспечивает поиск сущности на основе ее первичного ключа
// (с разными механизмами блокировки)
<T> T find(Class<T> entityClass, Object primaryKey);
<T> T find(Class<T> entityClass, Object primaryKey, LockModeType lockMode);
<T> T getReference(Class<T> entityClass, Object primaryKey);

// Блокирует сущность, применяя указанный тип режима блокировки
// (оптимистическая, пессимистическая...)
void lock(Object entity, LockModeType lockMode);

// Синхронизирует контекст постоянства с основной базой данных
void flush();
void setFlushMode(FlushModeType flushMode);
FlushModeType getFlushMode();

// Обновляет состояние сущности из базы данных,
// перезаписывая любые внесенные изменения
void refresh(Object entity);
void refresh(Object entity, LockModeType lockMode);

// Очищает контекст постоянства, а также проверяет, содержит ли он сущность
void clear();
void detach(Object entity);
boolean contains(Object entity);

// Задает и извлекает значение свойства EntityManager или подсказку
void setProperty(String propertyName, Object value);
Map<String, Object> getProperties();

// Создает экземпляр Query или TypedQuery для выполнения JPQL-оператора
Query createQuery(String qlString);
<T> TypedQuery<T> createQuery(CriteriaQuery<T> criteriaQuery);
<T> TypedQuery<T> createQuery(String qlString, Class<T> resultClass);

// Создает экземпляр Query или TypedQuery для выполнения именованного запроса
Query createNamedQuery(String name);
<T> TypedQuery<T> createNamedQuery(String name, Class<T> resultClass);

// Создает экземпляр TypedQuery для выполнения «родного» SQL-запроса
Query createNativeQuery(String sqlString);
Query createNativeQuery(String sqlString, Class resultClass);
Query createNativeQuery(String sqlString, String resultSetMapping);

// Создает StoredProcedureQuery для выполнения хранимой процедуры в базе данных
StoredProcedureQuery createStoredProcedureQuery(String procedureName);
```

```
    StoredProcedureQuery createNamedStoredProcedureQuery(String name);

    // Metamodel и CriteriaBuilder для запросов с использованием критериев
    // (выборка, обновление и удаление)
    CriteriaBuilder getCriteriaBuilder();
    Metamodel getMetamodel();
    Query createQuery(CriteriaUpdate updateQuery);
    Query createQuery(CriteriaDelete deleteQuery);

    // Указывает на то, что JTA-транзакция активна,
    // и соединяет с ней контекст постоянства
    void joinTransaction();
    boolean isJoinedToTransaction();

    // Возвращает объект базового поставщика для EntityManager
    <T> T unwrap(Class<T> cls);
    Object getDelegate();

    // Возвращает EntityGraph
    <T> EntityGraph<T> createEntityGraph(Class<T> rootType);
    EntityGraph<?> createEntityGraph(String graphName);
    <T> EntityGraph<T> getEntityGraph(String graphName);
    <T> List<EntityGraph<? super T>> getEntityGraphs(Class<T> entityClass);
}
```

Не стоит пугаться API-интерфейса из листинга 6.1, поскольку в этой главе рассматривается большинство соответствующих методов. В следующем разделе я объясню, как получить экземпляр `EntityManager`.

Получение менеджера сущностей

Менеджер сущностей — центральный интерфейс, используемый для взаимодействия с сущностями, однако приложению нужно сначала получить его. В зависимости от того, является ли среда управляемой контейнером (как вы увидите в главе 7 при работе с EJB-компонентами) или же управляемой приложением, код может быть совершенно другим. Например, в среде, управляемой контейнером, транзакциями управляет контейнер. Это означает, что вам не потребуется явным образом указывать `commit` или `rollback`, что вам пришлось бы сделать в среде, управляемой приложением.

Словосочетание «управляемая приложением» означает, что приложение отвечает за явное получение экземпляра `EntityManager` и управление его жизненным циклом (например, оно закрывает менеджер сущностей по окончании работы). В коде, приведенном в листинге 6.2, показано, как класс, функционирующий в среде Java SE, получает экземпляр менеджера сущностей. Он задействует класс `Persistence` для начальной загрузки экземпляра `EntityManagerFactory`, который ассоциирован с единицей сохраняемости (`chapter06PU`), используемой затем для создания `EntityManager`. Следует отметить, что в среде, управляемой приложением, за создание и закрытие менеджера сущностей (то есть за управление его жизненным циклом) отвечает разработчик.

Листинг 6.2. Класс Main, создающий менеджер сущностей с помощью фабрики EntityManagerFactory

```
public class Main {
  public static void main(String[] args) {
    // Создает экземпляр Book
    Book book = new Book ("H2G2", "Автостопом по Галактике", ➥
    12.5F, "1-84023-742-2", 354, false);

    // Получает менеджер сущностей и транзакцию
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("chap-
ter06PU");
    EntityManager em = emf.createEntityManager();

    // Обеспечивает постоянство Book в базе данных
    EntityTransaction tx = em.getTransaction();
    tx.begin();
    em.persist(book);
    tx.commit();

    // Закрывает менеджер сущностей и фабрику
    em.close();
    emf.close();
  }
}
```

Создание управляемого приложением менеджера сущностей осуществляется довольно просто с помощью фабрики, однако управляемую приложением среду отличает от управляемой контейнером среды то, каким образом происходит получение `EntityManagerFactory`. Среда, управляемая контейнером, имеет место, когда приложение эволюционирует в сервлет или EJB-контейнер. В среде Java EE самый распространенный способ получения менеджера сущностей заключается в использовании аннотации `@PersistenceContext` или в JNDI-поиске. Компоненту, работающему в контейнере (это может быть сервлет, EJB-компонент, веб-служба и т. д.), не нужно создавать или закрывать `EntityManager`, поскольку его жизненный цикл управляется контейнером. В листинге 6.3 показан код сессионного EJB-компонента без сохранения состояния, в который мы внедряем ссылку на единицу сохраняемости `chapter06PU`.

Листинг 6.3. EJB-компонент без сохранения состояния с внедренной в него ссылкой на менеджер сущностей

```
@Stateless
public class BookEJB {
  @PersistenceContext(unitName = "chapter06PU")
  private EntityManager em;
  public void createBook() {
    // Создает экземпляр Book
    Book book = new Book("H2G2", "Автостопом по Галактике", ➥
    12.5F, "1-84023-742-2", 354, false);

    // Обеспечивает постоянство Book в базе данных
    em.persist(book);
```

```
    }
}
```

По сравнению с кодом из листинга 6.2 код, приведенный в листинге 6.3, намного проще. Во-первых, в нем нет `Persistence` или `EntityManagerFactory`, поскольку контейнер внедряет экземпляр `EntityManager`. Приложение не отвечает за управление жизненным циклом менеджера сущностей (создание или закрытие). Во-вторых, поскольку EJB-компоненты без сохранения состояния управляют транзакциями, в этом коде не указан явным образом параметр `commit` или `rollback`. Такой стиль `EntityManager` демонстрируется в главе 7.

ПРИМЕЧАНИЕ

Если вы заглянете в подраздел «Производители данных» главы 2, то поймете, что также можете использовать @Inject в сочетании с EntityManager, если будете генерировать его (с применением аннотации @Produces).

Контекст постоянства

Перед тем как подробно исследовать API-интерфейс менеджера сущностей, вам необходимо понять крайне важную концепцию: *контекст постоянства*. Это набор экземпляров сущностей, находящихся под управлением. Он наблюдается в определенный момент времени для транзакции определенного пользователя: в контексте постоянства может существовать только один экземпляр сущности с одним и тем же постоянным идентификатором. Например, если экземпляр `Book` с идентификатором `12` существует в контексте постоянства, то никакой другой экземпляр `Book` с этим идентификатором не может существовать в том же самом контексте постоянства. Менеджер сущностей управляет лишь теми сущностями, которые содержатся в контексте постоянства, а это означает, что изменения будут отражаться в базе данных.

Менеджер сущностей обновляет контекст постоянства или обращается к нему при каждом вызове метода интерфейса `javax.persistence.EntityManager`. Например, когда произойдет вызов метода `persist()`, сущность, передаваемая как аргумент, будет добавлена в контекст постоянства, если ее там еще нет. Аналогичным образом при поиске сущности по ее первичному ключу менеджер сущностей сначала проверяет, не содержится ли уже в контексте постоянства запрашиваемая сущность. Контекст постоянства можно рассматривать как кэш первого уровня. Это небольшое жизненное пространство, в котором менеджер сущностей размещает сущности перед тем, как сбрасывать содержимое в базу данных. По умолчанию объекты располагаются в контексте постоянства столько времени, сколько длится соответствующая транзакция.

Чтобы резюмировать все это, взглянем на рис. 6.1, где показано, что двум пользователям требуется доступ к сущностям, данные которых хранятся в базе данных. У каждого пользователя имеется собственный контекст постоянства, в котором все сохраняется, пока длится его транзакция. Пользователь 1 получает из базы данных сущности с идентификаторами `12` и `56`, поэтому оба размещаются в его контексте постоянства. Пользователь 2 получает сущности с идентификаторами `12` и `34`. Как вы можете видеть, сущность с идентификатором `12` располагается в контексте постоянства каждого из пользователей. Пока длится транзакция, контекст постоянства

выступает в роли кэша первого уровня, где находятся сущности, которыми может управлять менеджер сущностей. Когда транзакция завершается, контекст постоянства очищается от сущностей.

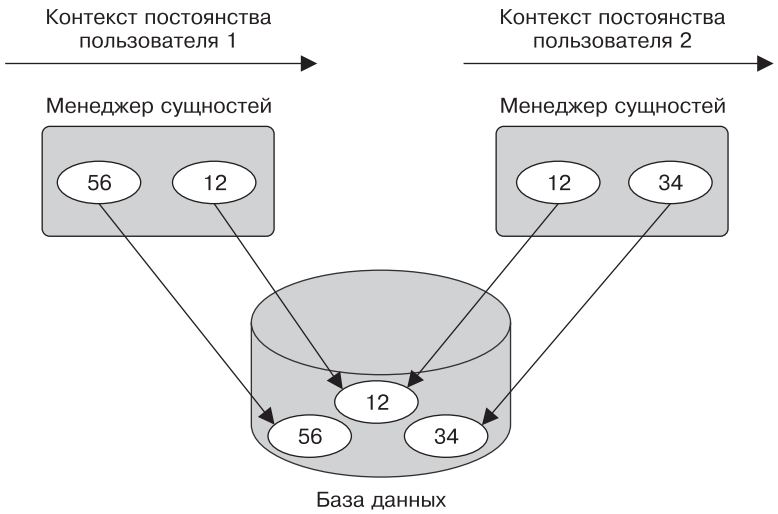


Рис. 6.1. Сущности, которые располагаются в контексте постоянства разных пользователей

Конфигурация для менеджера сущностей привязывается к экземпляру `EntityManagerFactory`, который применяется для его создания. Независимо от того, является ли среда управляемой приложением или же контейнером, эта фабрика необходима как единица сохраняемости, с использованием которой будет создаваться менеджер сущностей. Единица сохраняемости обуславливает параметры для подключения к базе данных и список сущностей, которыми можно управлять в контексте постоянства. Файл `persistence.xml` (листинг 6.4), располагающийся в каталоге META-INF, определяет единицу сохраняемости, у которой есть имя (`chapter06PU`) и набор атрибутов.

Листинг 6.4. Единица сохраняемости с набором сущностей, которые поддаются управлению

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
             version="2.1">

  <persistence-unit name="chapter06PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>org.agoncal.book.javaee7.chapter06.Book</class>
    <class>org.agoncal.book.javaee7.chapter06.Customer</class>
    <class>org.agoncal.book.javaee7.chapter06.Address</class>
```

```
    <properties>
      <property name="javax.persistence.schema-generation.database.action" ➥
                value="drop-and-create"/>
      <property name="javax.persistence.jdbc.driver" ➥
                value="org.apache.derby.jdbc.EmbeddedDriver"/>
      <property name="javax.persistence.jdbc.url" ➥
                value="jdbc:derby:memory:chapter06DB;create=true"/>
      <property name="eclipselink.logging.level" value="INFO"/>
    </properties>
  </persistence-unit>
</persistence>
```

Единица сохраняемости — это мост между контекстом постоянства и базой данных. С одной стороны, в теге `<class>` указываются все сущности, которыми можно было бы управлять в контексте постоянства, и, с другой стороны, он предоставляет всю информацию, необходимую для физического подключения к базе данных (с использованием свойств). Это происходит потому, что вы находитесь в среде, управляемой приложением (`transaction-type="RESOURCE_LOCAL"`). Как вы увидите в главе 7, в среде, управляемой контейнером, `persistence.xml` определял бы источник данных вместо свойств для подключения к базе данных и задал бы тип транзакций как JTA (`transaction-type="JTA"`).

В JPA 2.1 некоторые свойства в файле `persistence.xml` были стандартизированы (табл. 6.1). Все они начинаются с `javax.persistence`, например `javax.persistence.jdbc.url`. Требуется, чтобы поставщики JPA поддерживали эти стандартные свойства, однако они могут обеспечивать собственные свойства вроде `eclipselink` в приведенном чуть выше примере (например, `eclipselink.logging.level`).

Таблица 6.1. Стандартные свойства JPA

Свойство	Описание
javax.persistence.jdbc.driver	Полностью уточненное имя класса драйвера
javax.persistence.jdbc.url	Специфичный для драйвера URL-адрес
javax.persistence.jdbc.user	Имя пользователя, применяемое при подключении к базе данных
javax.persistence.jdbc.password	Пароль, применяемый при подключении к базе данных
javax.persistence.database-product-name	Имя целевой базы данных (например, Derby)
javax.persistence.database-major-version	Номер версии целевой базы данных
javax.persistence.database-minor-version	Дополнительный номер версии целевой базы данных
javax.persistence.ddl-create-script-source	Имя сценария, создающего базу данных
javax.persistence.ddl-drop-script-source	Имя сценария, удаляющего базу данных
javax.persistence.sql-load-script-source	Имя сценария, загружающего информацию в базу данных

Продолжение ⇩

Таблица 6.1 *(продолжение)*

Свойство	Описание
javax.persistence.schema-generation.database.action	Определяет действие, которое должно предприниматься в отношении базы данных (none, create, drop-and-create, drop)
javax.persistence.schema-generation.scripts.action	Определяет действие, которое должно предприниматься в отношении DDL-сценариев (none, create, drop-and-create, drop)
javax.persistence.lock.timeout	Значение времени ожидания в миллисекундах при пессимистической блокировке
javax.persistence.query.timeout	Значение времени ожидания в миллисекундах при запросах
javax.persistence.validation.group.pre-persist	Группы, намеченные для валидации при наступлении события pre-persist
javax.persistence.validation.group.pre-update	Группы, намеченные для валидации при наступлении события pre-update
javax.persistence.validation.group.pre-remove	Группы, намеченные для валидации при наступлении события pre-remove

Манипулирование сущностями

Мы используем менеджер сущностей как для простого манипулирования сущностями, так и для выполнения комплексных JPQL-запросов. При манипулировании одиночными сущностями интерфейс менеджера можно рассматривать как обобщенный объект доступа к данным (Data Access Object — DAO), который позволяет выполнять CRUD-операции в отношении любой сущности (табл. 6.2).

Таблица 6.2. Методы интерфейса менеджера сущностей для манипулирования сущностями

Метод	Описание
void persist(Object entity)	Делает так, что экземпляр помещается под управление, а также обеспечивает постоянство экземпляра
<T> T find(Class<T> entityClass, Object primaryKey)	Выполняет поиск сущности указанного класса и первичного ключа
<T> T getReference(Class<T> entityClass, Object primaryKey)	Извлекает экземпляр, выборка состояния которого может быть отложенной
void remove(Object entity)	Удаляет экземпляр сущности из контекста постоянства и основной базы данных
<T> T merge(T entity)	Обеспечивает слияние состояния определенной сущности с текущим контекстом постоянства
void refresh(Object entity)	Обновляет состояние экземпляра из базы данных, перезаписывая все изменения, внесенные в сущность, если таковые имеются

Метод	Описание
void flush()	Синхронизирует контекст постоянства с основной базой данных
void clear()	Очищает контекст постоянства, приводя к тому, что все сущности, которые находятся под управлением, оказываются отсоединенными
void detach(Object entity)	Убирает определенную сущность из контекста постоянства, приводя к тому, что сущность, которая находится под управлением, оказывается отсоединенной
boolean contains(Object entity)	Проверяет, является ли экземпляр сущностью, находящейся под управлением, которая относится к текущему контексту постоянства

Чтобы помочь вам лучше понять эти методы, я воспользуюсь простым примером однонаправленной связи «один к одному» между `Customer` и `Address`. Обе сущности обладают автоматически генерируемыми идентификаторами (благодаря аннотации `@GeneratedValue`), а в случае с `Customer` (листинг 6.5) имеет место отложенная выборка по отношению к `Address` (листинг 6.6).

Листинг 6.5. Сущность Customer с односторонней связью «один к одному» с Address

```
@Entity
public class Customer {
  @Id @GeneratedValue
  private Long id;
  private String firstName;
  private String lastName;
  private String email;
  @OneToOne (fetch = FetchType.LAZY)
  @JoinColumn(name = "address_fk")
  private Address address;
  // Конструкторы, геттеры, сеттеры
}
```

Листинг 6.6. Сущность Address

```
@Entity
public class Address {
  @Id @GeneratedValue
  private Long id;
  private String street1;
  private String city;
  private String zipcode;
  private String country;
  // Конструкторы, геттеры, сеттеры
}
```

Эти две сущности будут отображены в структуру базы данных, которая показана на рис. 6.2. Обратите внимание, что столбец ADDRESS_FK является внешним ключом, ссылающимся на ADDRESS.

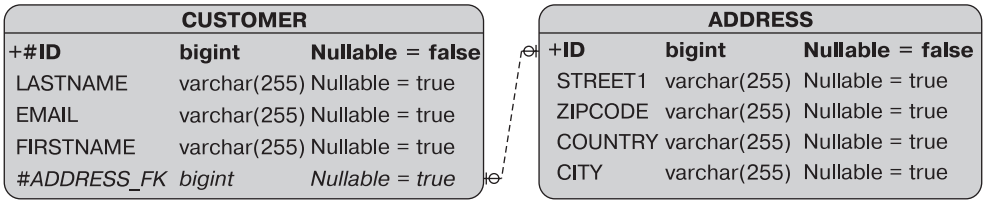


Рис. 6.2. Таблицы CUSTOMER и ADDRESS

Для лучшей удобочитаемости фрагментов кода, приведенных в следующем подразделе, предполагается, что атрибут em имеет тип EntityManager, а tx — тип EntityTransaction.

Обеспечение постоянства сущности

Обеспечение постоянства сущности подразумевает вставку в базу данных информации, которой там еще нет (в противном случае будет сгенерировано исключение EntityExistsException). Для этого необходимо создать новый экземпляр сущности с использованием оператора new, задать значения для атрибутов, привязать одну сущность к другой, если есть ассоциации, и, наконец, вызвать метод EntityManager.persist(), как показано в варианте тестирования JUnit в листинге 6.7.

Листинг 6.7. Обеспечение постоянства Customer и Address

```
Customer customer = new Customer("Энтони", "Балла", "tballa@mail.com");
Address address = new Address("Ризердаун Роуд", "Лондон", "8QE", "Англия");
customer.setAddress(address);

tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();

assertNotNull(customer.getId());
assertNotNull(address.getId());
```

В листинге 6.7 customer и address — это всего лишь два объекта, которые располагаются в памяти виртуальной машины Java. Они оба станут сущностями, которые находятся под управлением, когда менеджер сущностей (переменная em) примет их в расчет, обеспечив постоянство (em.persist(customer)). На данном этапе оба объекта окажутся подходящими для вставки в базу данных. Когда произойдет фиксация транзакции (tx.commit()), информация будет сброшена в базу данных, строка, касающаяся адреса, будет вставлена в таблицу ADDRESS, а строка, которая касается клиента, — в таблицу CUSTOMER. Поскольку Customer является владельцем связи, его таблица содержит внешний ключ, ссылающийся

на ADDRESS. Выражения assertNotNull обеспечат проверку того, что обе сущности получили сгенерированные идентификаторы (благодаря поставщику постоянства, а также аннотациям @Id и @GeneratedValue).

Обратите внимание на порядок методов persist(): сначала обеспечивается постоянство Customer, а затем — постоянство Address. Если бы этот порядок был иным, то результат все равно оказался бы тем же. Ранее менеджер сущностей был охарактеризован как кэш первого уровня. Пока не произойдет фиксации транзакции, данные остаются в памяти, а доступ к базе данных отсутствует. Менеджер сущностей кэширует данные и, когда готов, сбрасывает их в том порядке, в каком ожидает основная база данных (с соблюдением ограничений целостности). Поскольку внешний ключ располагается в таблице CUSTOMER, сначала будет выполнен оператор insert для ADDRESS, а затем — для CUSTOMER.

ПРИМЕЧАНИЕ

Большинство сущностей в этой главе не реализуют интерфейс Serializable. Причина этого заключается в том, что сущностям не требуется быть сериализуемыми для того, чтобы оказалось возможным обеспечение их постоянства в базе данных. Они передаются по ссылке от одного метода к другому, и, когда необходимо обеспечить их постоянство, вызывается метод EntityManager.persist(). Но если вам потребуется передать сущности по значению (удаленный вызов, внешний EJB-контейнер и т. д.), то они должны будут реализовывать маркерный (не содержащий методов) интерфейс java.io.Serializable. Этот интерфейс говорит компилятору, что он должен позаботиться о том, чтобы все поля, связанные с классом-сущностью, обязательно были сериализуемыми. Благодаря этому любой экземпляр можно будет сериализовать в байтовый поток и передать с использованием удаленного вызова методов (Remote Method Invocation — RMI).

Поиск по идентификатору

Для поиска сущности по ее идентификатору вы можете использовать два метода. Первый — EntityManager.find(), имеющий два параметра: класс-сущность и уникальный идентификатор (листинг 6.8). Если сущность удастся найти, то она будет возвращена; если сущность не удастся найти, то будет возвращено значение null.

Листинг 6.8. Поиск Customer по идентификатору

```
Customer customer = em.find(Customer.class, 1234L)
if (customer!= null) {
  // Обработать объект
}
```

Второй метод — getReference() (листинг 6.9). Он очень схож с операцией find, поскольку имеет те же параметры, однако извлекает ссылку на сущность (с помощью ее первичного ключа), но не извлекает ее данных. Считайте его прокси для сущности, но не самой сущностью. Он предназначен для ситуаций, в которых требуется экземпляр сущности, находящейся под управлением, но не требуется никаких данных, кроме тех, что потенциально относятся к первичному ключу сущности, к которой осуществляется доступ. При использовании getReference() выборка данных о состоянии является отложенной, а это означает, что, если вы не осуществите доступ к состоянию до того, как сущность окажется отсоединенной, данных

уже может не быть там. Если сущность не удастся найти, то будет сгенерировано исключение EntityNotFoundException.

Листинг 6.9. Поиск Customer по ссылке

```
try {
  Customer customer = em.getReference(Customer.class, 1234L)
  // Обработать объект
} catch(EntityNotFoundException ex) {
  // Сущность не найдена
}
```

Удаление сущности

Сущность можно удалить методом EntityManager.remove(). Как только сущность окажется удалена, она будет убрана из базы данных, отсоединена от менеджера сущностей и ее больше нельзя будет синхронизировать с базой данных. В плане Java-объектов эта сущность будет по-прежнему доступна, пока не окажется вне области видимости и сборщик мусора не уберет ее. В коде, приведенном в листинге 6.10, показано, как удалить объект после того, как он был создан.

Листинг 6.10. Создание и удаление сущностей Customer и Address

```
Customer customer = new Customer("Энтони", "Балла", "tballa@mail.com");
Address address = new Address("Ризердаун Роуд", "Лондон", "8QE", "Англия");
customer.setAddress(address);
tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();

tx.begin();
em.remove(customer);
tx.commit();

// Данные удаляются из базы данных, но объект по-прежнему доступен
assertNotNull(customer);
```

Код из листинга 6.10 создает экземпляр Customer и Address, связывает их (customer.setAddress(address)) и обеспечивает их постоянство. В базе данных строка, касающаяся клиента, связывается со строкой, которая касается адреса, с помощью внешнего ключа; далее в коде удаляется только Customer. В зависимости от того, как сконфигурировано каскадирование (о нем мы поговорим позднее в этой главе), Address может остаться без какой-либо другой сущности, которая ссылается на нее, а строка, касающаяся адреса, станет изолированной.

Удаление сущностей-сирот

Изолированные сущности нежелательны, поскольку они приводят к тому, что в базе данных есть строки, на которые не ссылается какая-либо другая таблица, при этом они лишены средств доступа. При использовании JPA вы можете проинформировать поставщика постоянства о необходимости автоматически удалять такие сущ-

ности или каскадировать операцию удаления. Если целевая сущность (Address) находится в частном владении источника (Customer), подразумевая, что целью никогда не может владеть несколько источников, а этот источник окажется удален приложением, то поставщик должен будет удалить и цель.

Ассоциации, которые определены как «один к одному» или «один ко многим», поддерживают использование параметра orphanRemoval. Чтобы включить этот параметр в пример, взглянем на то, как добавить элемент orphanRemoval=true в аннотацию @OneToOne (листинг 6.11).

Листинг 6.11. Сущность Customer и удаление изолированной сущности Address

```
@Entity
public class Customer {
  @Id @GeneratedValue
  private Long id;
  private String firstName;
  private String lastName;
  private String email;
  @OneToOne (fetch = FetchType.LAZY, orphanRemoval=true)
  private Address address;
  // Конструкторы, геттеры, сеттеры
}
```

При таком отображении код, приведенный в листинге 6.10, автоматически удалит сущность Address, если окажется удалена сущность Customer либо если связь будет разорвана (при присвоении атрибуту address значения null или удалении дочерней сущности из коллекции при связи «один ко многим»). Операция удаления выполняется во время операции сброса (когда имеет место фиксация транзакции).

Синхронизация с базой данных

До настоящего момента синхронизация с базой данных осуществлялась во время фиксации. Менеджер сущностей является кэшем первого уровня, ожидающим фиксации транзакции, чтобы сбросить информацию в базу данных. Но что будет, если потребуется вставить Customer и Address?

```
tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();
```

Всем ожидаемым изменениям требуется SQL-оператор. В данном случае два оператора insert генерируются и становятся постоянными, только когда происходит фиксация транзакции в базе данных. Для большинства приложений этой автоматической синхронизации данных будет достаточно. Хотя неизвестно, в какой именно момент времени поставщик на самом деле сбрасывает изменения в базу данных, вы можете быть уверены, что это случится, когда произойдет фиксация транзакции. База данных синхронизируется с сущностями в контексте постоянства, однако данные могут быть сброшены (flush) явным образом в базу либо сущности могут быть обновлены с использованием информации из базы

(refresh). Если данные будут сброшены в базу данных в определенный момент, а позднее в коде приложение вызовет метод rollback(), то сброшенные данные будут изъяты из базы.

Сброс сущности. С помощью метода EntityManager.flush() поставщика постоянства можно явным образом заставить сбрасывать данные в базу, но не будет фиксации транзакции. Это позволяет разработчику вручную инициировать тот же самый процесс, который менеджер сущностей использует внутренне для сброса контекста постоянства.

```
tx.begin();
em.persist(customer);
em.flush();
em.persist(address);
tx.commit();
```

В приведенном коде можно отметить две любопытные вещи. Первая заключается в том, что em.flush() не станет дожидаться фиксации транзакции и заставит поставщика сбросить контекст постоянства. Оператор insert будет сгенерирован и выполнен при сбросе. Вторая вещь состоит в том, что этот код не будет работать из-за ограничения целостности. Без явного сброса менеджер сущностей станет кэшировать все изменения, а также упорядочивать и последовательно вносить их в базу данных. При явном сбросе будет выполнен оператор insert в отношении CUSTOMER, однако ограничение целостности в случае с внешним ключом ADDRESS окажется нарушено (столбец ADDRESS_FK в CUSTOMER). Это приведет к откату транзакции. Откат также произойдет в случае с уже сброшенными данными. Явные сбросы следует использовать осторожно и только при необходимости.

Обновление сущности. Метод refresh() применяется для синхронизации данных в направлении, противоположном сбросу, то есть он перезаписывает текущее состояние сущности, которая находится под управлением, с использованием данных в таком виде, в каком они присутствуют в базе данных. Типичный случай — когда вы используете метод EntityManager.refresh() для отмены изменений, внесенных в сущность только в памяти. Фрагмент варианта тестирования, который приведен в листинге 6.12, обеспечивает поиск Customer по идентификатору, изменение значения его firstName и отмену этого изменения с помощью метода refresh().

Листинг 6.12. Обновление сущности Customer из базы данных

```
Customer customer = em.find(Customer.class, 1234L)
assertEquals(customer.getFirstName(), "Энтони");
customer.setFirstName("Уильям");
em.refresh(customer);
assertEquals(customer.getFirstName(), "Энтони");");
```

Содержимое контекста постоянства

В контексте постоянства содержатся сущности, которые находятся под управлением. Используя интерфейс EntityManager, вы можете проверить, находится ли сущность под управлением, отсоединить ее или очистить контекст постоянства от всех сущностей.

Contains. Сущности либо управляются менеджером сущностей, либо нет. Метод `EntityManager.contains()` возвращает логическое значение и позволяет вам проверить, находится ли экземпляр определенной сущности в настоящее время под управлением менеджера в контексте постоянства. В варианте тестирования, приведенном в листинге 6.13, показано, что обеспечивается постоянство `Customer`, и вы можете незамедлительно проверить, находится ли эта сущность под управлением (`em.contains(customer)`). Ответ в данном случае звучит как «да». Далее происходит вызов метода `remove()` и сущность удаляется из базы данных, а также из контекста постоянства (`em.contains(customer)` возвращает `false`).

Листинг 6.13. Вариант тестирования для проверки на предмет того, присутствует ли сущность Customer в контексте постоянства

```
Customer customer = new Customer("Энтони", "Балла", "tballa@mail.com");
tx.begin();
em.persist(customer);
tx.commit();
assertTrue(em.contains(customer));

tx.begin();
em.remove(customer);
tx.commit();
assertFalse(em.contains(customer));
```

Очистка и отсоединение. Метод `clear()` прост: он очищает контекст постоянства, приводя к тому, что все сущности, которые находятся под управлением, оказываются отсоединенными. Метод `detach(Object entity)` убирает определенную сущность из контекста постоянства. После такого «выселения», изменения, внесенные в эту сущность, не будут синхронизированы с базой данных. Код, приведенный в листинге 6.14, создает сущность, проверяет, находится ли она под управлением, отсоединяет ее от контекста постоянства, а также проверяет, была ли она отсоединена.

Листинг 6.14. Проверка, присутствует ли сущность Customer в контексте постоянства

```
Customer customer = new Customer("Энтони", "Балла", "tballa@mail.com");
tx.begin();
em.persist(customer);
tx.commit();
assertTrue(em.contains(customer));

em.detach(customer);
assertFalse(em.contains(customer));
```

Слияние сущности

Отсоединенная сущность больше не ассоциирована с контекстом постоянства. Если вы хотите управлять ею, то вам потребуется снова присоединить эту сущность (то есть обеспечить ее слияние). Обратимся к примеру сущности, которую необходимо вывести на JSF-странице. Сущность, сначала загружаемая из базы данных на постоянный уровень (находящаяся под управлением), возвращается из вызова локального EJB-компонента (она является отсоединенной, поскольку контекст

транзакции перестает существовать), выводится на уровне представления (все еще являясь отсоединенной), а затем возвращается для обновления в базу данных. Однако в тот момент сущность является отсоединенной, и необходимо присоединить ее снова или обеспечить слияние, чтобы синхронизировать ее состояние с базой данных.

Эта ситуация симулируется в листинге 6.15 путем очистки контекста постоянства (`em.clear()`), что приводит к отсоединению сущности.

Листинг 6.15. Очистка контекста постоянства и слияние сущности

```
Customer customer = new Customer("Энтони", "Балла", "tballa@mail.com");
tx.begin();
em.persist(customer);
tx.commit();
em.clear();
// Задает новое значение для отсоединенной сущности
customer.setFirstName("Уильям");

tx.begin();
em.merge(customer);
tx.commit();
```

Код, приведенный в листинге 6.15, создает и обеспечивает постоянство `Customer`. Вызов `em.clear()` форсирует отсоединение сущности `Customer`, однако отсоединенные сущности продолжают присутствовать, но уже вне контекста постоянства, в котором они находились, а синхронизация их состояния с состоянием базы данных больше не обеспечивается. Именно это происходит при использовании `customer.setFirstName("William")`. Код выполняется в отношении отсоединенной сущности, и данные не обновляются в базе данных. Для репликации этого изменения в базу данных вам потребуется снова присоединить сущность (то есть обеспечить ее слияние) с помощью `em.merge(customer)` в рамках транзакции.

Обновление сущности

Обновление сущности является простой операцией, но в то же время может оказаться сложной для понимания. Как вы только что видели, `EntityManager.merge()` можно использовать для присоединения сущности и синхронизации ее состояния с базой данных. Однако если сущность находится в данный момент под управлением, внесенные в нее изменения не будут автоматически отражены в базе данных. В противном случае вам потребуется явным образом вызвать `merge()`.

В листинге 6.16 демонстрируется обеспечение постоянства `Customer` с `setFirstName`, для которого задано значение `Antony`. Когда вы вызываете метод `em.persist()`, сущность находится под управлением, поэтому любые изменения, внесенные в эту сущность, будут синхронизированы с базой данных. Когда вы вызываете метод `setFirstName()`, состояние сущности изменяется. Менеджер сущностей кэширует все действия, начиная с `tx.begin()`, и обеспечивает соответствующую синхронизацию при фиксации.

Листинг 6.16. Обновление setFirstName сущности Customer

```
Customer customer = new Customer("Энтони", "Балла", "tballa@mail.com");
tx.begin();
em.persist(customer);

customer.setFirstName("Уильям");
tx.commit();
```

Каскадирование событий

По умолчанию любая операция выполняется менеджером сущностей только в отношении сущности, которая предусмотрена в качестве аргумента при этой операции. Но порой, когда операция выполняется в отношении сущности, возникает необходимость распространить ее на соответствующие ассоциации. Это называется *каскадированием события*. Приводившиеся до сих пор примеры опирались на поведение каскадирования по умолчанию, а не на настроенное поведение. В листинге 6.17 показано, что для создания Customer нужно сгенерировать экземпляры сущностей Customer и Address, связать их (customer.setAddress(address)), а затем обеспечить постоянство этих двух экземпляров.

Листинг 6.17. Обеспечение постоянства Customer и Address

```
Customer customer = new Customer("Энтони", "Балла", "tballa@mail.com");
Address address = new Address("Ризердаун Роуд", "Лондон", "8QE", "Англия");
customer.setAddress(address);

tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();
```

Поскольку между Customer и Address есть связь, вы могли бы каскадировать действие persist от Customer к Address. Это означало бы, что вызов em.persist(customer) привел бы к каскадированию события PERSIST к сущности Address, если она допускает распространение события такого типа. Тогда вы могли бы сократить код и избавиться от em.persist(address), как показано в листинге 6.18.

Листинг 6.18. Каскадирование события PERSIST к Address

```
Customer customer = new Customer("Энтони", "Балла", "tballa@mail.com");
Address address = new Address("Ризердаун Роуд", "Лондон", "8QE", "Англия");
customer.setAddress(address);

tx.begin();
em.persist(customer);
tx.commit();
```

Без каскадирования было бы обеспечено постоянство Customer, но не Address. Каскадирование события возможно при изменении отображения связи. У аннотаций @OneToOne, @OneToMany, @ManyToOne и @ManyToMany есть атрибут cascade, который принимает массив событий для каскадирования, а также событие PERSIST, которое можно каскадировать, как и событие REMOVE (широко используемое для выполнения

каскадных удалений). Для этого вам потребуется изменить отображение сущности Customer (листинг 6.19) и добавить атрибут cascade в аннотацию @OneToOne в случае с Address.

Листинг 6.19. Сущность Customer и каскадирование событий PERSIST и REMOVE

```
@Entity
public class Customer {
@Id @GeneratedValue
  private Long id;
  private String firstName;
  private String lastName;
  private String email;
  @OneToOne (fetch = FetchType.LAZY, cascade = {CascadeType.PERSIST,
CascadeType.REMOVE})
  @JoinColumn(name = "address_fk")
  private Address address;
  // Конструкторы, геттеры, сеттеры
}
```

Вы можете выбирать из нескольких событий для каскадирования к цели ассоциации (эти события приведены в табл. 6.3) и даже каскадировать их все, используя тип CascadeType.ALL.

Таблица 6.3. Возможные события для каскадирования

CascadeType	Описание
PERSIST	Каскадирует операции по обеспечению постоянства к цели ассоциации
REMOVE	Каскадирует операции удаления к цели ассоциации
MERGE	Каскадирует операции объединения к цели ассоциации
REFRESH	Каскадирует операции обновления к цели ассоциации
DETACH	Каскадирует операции отсоединения к цели ассоциации
ALL	Объявляет, что должны быть каскадированы все предыдущие операции

JPQL

Вы только что видели, как манипулировать сущностями по отдельности, используя API-интерфейс EntityManager. Вы уже знаете, как искать сущность по идентификатору, удалять ее, обновлять ее атрибуты и т. д. Однако поиск сущности по идентификатору довольно сильно ограничивает вас, поскольку вы можете извлечь только одну сущность, используя ее уникальный идентификатор. На практике вам может потребоваться извлечь сущность, исходя из иных критериев (имени, ISBN-номера и т. д.), либо извлечь набор сущностей на основе других критериев (например, все сущности, связанные с клиентами, проживающими в США). Эта возможность присуща реляционным базам данных, а у JPA есть язык, который обеспечивает это взаимодействие, — JPQL.

JPQL предназначен для определения поисков постоянных сущностей, которые не зависят от основной базы данных. JPQL — это язык запросов, укоренившийся в синтаксисе SQL. Он является стандартным языком для выполнения запросов к базам данных. Однако основное отличие состоит в том, что при использовании SQL вы получаете результаты в виде строк и столбцов (таблиц), а в случае применения JPQL — в виде сущности или коллекции сущностей. Синтаксис JPQL является объектно-ориентированным и, следовательно, более легким для понимания разработчиками, чей опыт ограничивается объектно-ориентированными языками. Разработчики управляют своей доменной моделью сущностей, а не структурой таблицы, используя точечную нотацию (например, `myClass.myAttribute`).

«За кулисами» JPQL применяет механизм отображения для преобразования JPQL-запросов в такие, которые будут понятны базам данных SQL. Запрос выполняется в отношении основной базы данных с использованием SQL- и JDBC-вызовов, после чего следует присваивание значений атрибутам экземпляров сущностей и их возврат приложению — все происходит очень простым и эффективным образом с применением богатого синтаксиса запросов.

Самый простой JPQL-запрос обеспечивает выборку всех экземпляров одной сущности:

```
SELECT b
FROM Book b
```

Если вы знаете SQL, то этот код должен показаться вам знакомым. Вместо выборки из таблицы JPQL производит выборку сущностей, в данном случае той, что носит имя `Book`. Кроме того, используется оператор `FROM` для наделения сущности псевдонимом: `b` является псевдонимом `Book`. Оператор `SELECT` запроса указывает на то, что типом результата запроса является сущность `b` (`Book`). Выполнение этого оператора приведет к получению списка, в который будет входить нуль или более экземпляров `Book`.

Чтобы ограничить результаты, добавьте критерии поиска. Вы можете воспользоваться оператором `WHERE`, как показано далее:

```
SELECT b
FROM Book b
WHERE b.title = 'H2G2'
```

Псевдоним предназначен для навигации по атрибутам сущности с применением оператора-точки. Поскольку сущность `Book` обладает постоянным атрибутом, носящим имя `title` и тип `String`, `b.title` ссылается на атрибут `title` сущности `Book`. Выполнение этого оператора приведет к получению списка, в который будет входить нуль или более экземпляров `Book` со значением `title` в виде `H2G2`.

Самый простой запрос на выборку состоит из двух обязательных частей — операторов `SELECT` и `FROM`. `SELECT` определяет формат результатов запроса. Оператор `FROM` определяет сущность или сущности, из которых будут получаться результаты, а необязательные операторы `WHERE`, `ORDER BY`, `GROUP BY` и `HAVING` могут быть использованы для ограничения или упорядочения результатов запроса. Листинг 6.20 демонстрирует упрощенный синтаксис JPQL-оператора.

Листинг 6.20. Упрощенный синтаксис JPQL-оператора

```
SELECT <оператор SELECT>
FROM <оператор FROM>
[WHERE <оператор WHERE>]
[ORDER BY <оператор ORDER BY>]
[GROUP BY <оператор GROUP BY>]
[HAVING <оператор HAVING>]
```

Листинг 6.20 определяет оператор SELECT, а операторы DELETE и UPDATE также могут быть использованы для выполнения операций удаления и обновления в отношении множественных экземпляров заданного класса-сущности.

SELECT

Оператор SELECT придерживается синтаксиса выражений путей и приводит к результату в одной из следующих форм: сущность, атрибут сущности, выражение-конструктор, агрегатная функция или их последовательность. Выражения путей являются строительными блоками запросов и используются для перемещения по атрибутам сущности или связям сущностей (либо коллекции сущностей) с помощью точечной (.) навигации с применением следующего синтаксиса:

```
SELECT [DISTINCT] <выражение> [[AS] <идентификационная переменная>]
expression ::= { NEW | TREAT | AVG | MAX | MIN | SUM | COUNT }
```

Простой оператор SELECT возвращает сущность. Например, если сущность Customer содержит псевдоним c, то SELECT возвратит сущность или список сущностей:

```
SELECT c
FROM Customer c
```

Однако оператор SELECT также может возвращать атрибуты. Если у Customer имеется firstName, то SELECT c.firstName возвратит строку или коллекцию строк с firstName.

Чтобы извлечь firstName и lastName сущности Customer, вам потребуется сгенерировать список, который будет включать два следующих атрибута:

```
SELECT c.firstName, c.lastName
FROM Customer c
```

С выходом версии JPA 2.0 появилась возможность извлекать атрибуты в зависимости от условий (с использованием выражения CASE WHEN ... THEN ... ELSE ... END). Например, вместо извлечения значения, говорящего о цене книги, оператор может возвратить расчет цены (например, с 50%-ной скидкой) в зависимости от издателя (например, с 50%-ной скидкой на книги от «Apress» и 20%-ной скидкой на все прочие книги).

```
SELECT CASE b.editor WHEN 'Apress'
                    THEN b.price * 0.5
                    ELSE b.price * 0.8
       END
FROM Book b
```

Если сущность Customer связана отношением «один к одному» с Address, то c.address будет ссылаться на адрес клиента, а в результате выполнения приведенного далее запроса будет возвращен не список клиентов, а список адресов:

```
SELECT c.address
FROM Customer c
```

Навигационные выражения можно объединять в цепочку для обхода комплексных EntityGraph. Благодаря этой методике можно создавать выражения путей, например c.address.country.code, ссылающиеся на код страны в адресе клиента:

```
SELECT c.address.country.code
FROM Customer c
```

В выражении SELECT можно применять конструктор для возврата экземпляра Java-класса, который инициализируется с использованием результата запроса. Класс не обязан быть сущностью, однако конструктор должен быть полностью уточненным и сочетаться с атрибутами.

```
SELECT NEW org.agoncal.javaee7.CustomerDTO(c.firstName, c.lastName, c.address.
street1)
FROM Customer c
```

Результатом этого запроса будет список объектов CustomerDTO, экземпляры которых были созданы с применением оператора new и инициализированы с использованием имен, фамилий и улиц проживания клиентов.

Выполнение показанных далее запросов приведет к возврату одного значения или коллекции, в которую будет входить нуль и более сущностей (или атрибутов), включая дубликаты. Для удаления дубликатов необходимо воспользоваться оператором DISTINCT.

```
SELECT DISTINCT c
FROM Customer c

SELECT DISTINCT c.firstName
FROM Customer c
```

Результатом запроса может оказаться результат выполнения агрегатной функции, примененной в выражении пути. В операторе SELECT могут быть использованы следующие агрегатные функции: AVG, COUNT, MAX, MIN, SUM. Результаты можно сгруппировать оператором GROUP BY и профильтровать оператором HAVING.

```
SELECT COUNT(c)
FROM Customer c
```

Скалярные выражения тоже могут быть использованы в операторе SELECT запроса, равно как и в операторах WHERE и HAVING. Эти выражения можно применять в отношении числовых (ABS, SQRT, MOD, SIZE, INDEX) и строковых значений (CONCAT, SUBSTRING, TRIM, LOWER, UPPER, LENGTH, LOCATE), а также значений даты/времени (CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP).

FROM

Оператор FROM определяет сущности путем объявления идентификационных переменных. *Идентификационная переменная*, или *псевдоним*, — это идентификатор, который может быть использован в других операторах (SELECT, WHERE и т. д.). Синтаксис оператора FROM включает сущность и псевдоним. В приведенном далее примере Customer является сущностью, а c — идентификационной переменной:

```
SELECT c
FROM Customer c
```

WHERE

Оператор WHERE состоит из условного выражения, используемого для ограничения результатов выполнения оператора SELECT, UPDATE или DELETE. Оператор WHERE может быть простым выражением либо набором условных выражений, применяемых для фильтрации результатов запроса.

Самый простой способ ограничить результаты запроса — использовать атрибут сущности. Например, приведенный далее запрос обеспечивает выборку всех клиентов, для которых firstName имеет значение Винсент:

```
SELECT c
FROM Customer c
WHERE c.firstName = 'Винсент'
```

Вы можете еще больше ограничить результаты запросов, используя логические операторы AND и OR. В приведенном далее примере AND задействуется для выборки всех клиентов, для которых firstName имеет значение Винсент, а country — значение Франция:

```
SELECT c
FROM Customer c
WHERE c.firstName = 'Винсент' AND c.address.country = 'Франция'
```

В операторе WHERE также могут применяться операторы сравнения: =, >, >=, <, <=, <>, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF]. Далее приведен пример использования двух из этих операторов:

```
SELECT c
FROM Customer c
WHERE c.age > 18

SELECT c
FROM Customer c
WHERE c.age NOT BETWEEN 40 AND 50

SELECT c
FROM Customer c
WHERE c.address.country IN ('США', 'Португалия')
```

Выражение LIKE состоит из строки и опциональных знаков переключения кода, которые определяют условия соответствия: знак подчеркивания (_) для метасим-

волов, состоящих из одного знака, и знак процента (%), если речь идет о метасимволах, состоящих из множества знаков.

```
SELECT c
FROM Customer c
WHERE c.email LIKE '%mail.com'
```

Привязка параметров

Операторы WHERE, приводившиеся в этой книге до настоящего момента, задействовали только фиксированные значения. В приложениях запросы часто зависят от параметров. JPQL поддерживает синтаксис для привязки параметров двух типов, делая возможным внесение динамических изменений в ограничительный оператор запроса. Это позиционные и именованные параметры.

Позиционные параметры обозначаются знаком вопроса (?), за которым следует целочисленное значение (например, ?1). При выполнении запроса необходимо указать номера параметров, которые нужно заменить.

```
SELECT c
FROM Customer c
WHERE c.firstName = ?1 AND c.address.country = ?2
```

Именованные параметры обозначаются строковым идентификатором с префиксом в виде знака двоеточия (:). При выполнении запроса необходимо указать имена параметров, которые следует заменить.

```
SELECT c
FROM Customer c
WHERE c.firstName = :fname AND c.address.country = :country
```

В разделе «Запросы», приведенном далее в этой главе, вы увидите, как приложение осуществляет привязку параметров.

Подзапросы

Подзапрос — это запрос SELECT, который вложен в условное выражение WHERE или HAVING. Результаты подзапроса оцениваются и интерпретируются в условном выражении главного запроса. Для извлечения информации о самых молодых клиентах из базы данных сначала нужно выполнить подзапрос с использованием MIN(age), а затем его результаты будут оценены в главном запросе.

```
SELECT c
FROM Customer c
WHERE c.age = (SELECT MIN(cust. age) FROM Customer cust ))
```

ORDER BY

Оператор ORDER BY позволяет упорядочивать сущности или значения, возвращаемые в результате выполнения запроса SELECT. Упорядочение применяется для атрибута сущности, который указан в этом предложении и за которым следует ключевое слово ASC или DESC. Ключевое слово ASC определяет, что будет применяться упорядочение по возрастанию; ключевое слово DESC, которое является его противоположностью,

определяет, что применяется упорядочение по убыванию. Сортировка по возрастанию задействуется по умолчанию, и ее можно не указывать.

```
SELECT c
FROM Customer c
WHERE c.age > 18
ORDER BY c.age DESC
```

Для уточнения порядка сортировки также можно использовать множественные выражения:

```
SELECT c
FROM Customer c
WHERE c.age > 18
ORDER BY c.age DESC, c.address.country ASC
```

GROUP BY и HAVING

Конструкция GROUP BY делает возможной агрегацию результирующих значений согласно набору свойств. Сущности делятся на группы в зависимости от значений связанных с ними полей, которые указаны в предложении GROUP BY. Чтобы сгруппировать клиентов в соответствии со страной их проживания и сосчитать их, используйте приведенный далее запрос:

```
SELECT c.address.country, count(c)
FROM Customer c
GROUP BY c.address.country
```

GROUP BY определяет выражения группировки (c.address.country), с использованием которых будет осуществляться агрегация и подсчет результатов (count(c)). Следует отметить, что выражения, присутствующие в операторе GROUP BY, также должны быть в операторе SELECT.

Оператор HAVING определяет соответствующий фильтр, который применяется после того, как будут сгруппированы результаты запроса, подобно вторичному оператору WHERE, обеспечивая фильтрацию результатов GROUP BY. Если воспользоваться предыдущим запросом, добавив HAVING, то могут быть возвращены результаты, в случае с которыми country будет иметь значение, отличное от Англия.

```
SELECT c.address.country, count(c)
FROM Customer c
GROUP BY c.address.country
HAVING c.address.country <> 'Англия'
```

GROUP BY и HAVING могут быть использованы только в операторе SELECT (а не в DELETE или UPDATE).

Массовое удаление

Вы уже знаете, как удалить сущность с помощью метода EntityManager.remove() и выполнить запрос к базе данных для извлечения списка сущностей, которые соответствуют определенным критериям. Чтобы удалить список сущностей, вы можете выполнить запрос, произвести итерацию по этому списку и удалить каждую сущ-

ность по отдельности. Хотя это допустимый алгоритм, он ужасен в плане производительности (слишком много раз придется получать доступ к базе данных). Есть более подходящий способ решить эту задачу — произвести массовое удаление.

JPQL выполняет операции массового удаления в отношении множественных экземпляров определенного класса-сущности. Такой подход используется для удаления большого количества сущностей в рамках одной операции. Оператор DELETE похож на оператор SELECT, поскольку может включать ограничительное выражение WHERE и задействовать параметры. В результате возвращается ряд экземпляров сущности, затронутых операцией. Синтаксис оператора DELETE выглядит следующим образом:

```
DELETE FROM <имя сущности> [[AS] <идентификационная переменная>]
[WHERE <выражение WHERE>]
```

Чтобы в качестве примера удалить всех клиентов моложе 18 лет, вы можете прибегнуть к массовому удалению с помощью оператора DELETE.

```
DELETE FROM Customer c
WHERE c.age < 18
```

Массовое обновление

Для массового обновления сущностей необходимо обратиться к оператору UPDATE, задав значение для одного или нескольких атрибутов субъекта сущности согласно условиям в выражении WHERE. Синтаксис оператора UPDATE выглядит следующим образом:

```
UPDATE <имя сущности> [[AS] <идентификационная переменная>]
SET <оператор UPDATE> {, <оператор UPDATE>}*
[WHERE <выражение WHERE>]
```

Вместо того чтобы удалять всех молодых клиентов, для них можно изменить значение firstName на TOO YOUNG с помощью приведенного далее оператора:

```
UPDATE Customer c
SET c.firstName = 'TOO YOUNG'
WHERE c.age < 18
```

Запросы

Вы уже видели синтаксис JPQL и поняли, как описывать задачи с использованием разных операторов (SELECT, FROM, WHERE и т. д.). Но как включить JPQL-оператор в свое приложение? Ответ: с помощью запросов. В JPA 2.1 имеется пять отличающихся типов запросов, которые могут быть использованы в коде, причем каждый из них — с разным назначением.

- *Динамические запросы* — это самая простая форма, куда входит всего лишь строка JPQL-запроса, динамически генерируемого во время выполнения.
- *Именованные запросы* — являются статическими и неизменяемыми.
- *Criteria API* — в JPA 2.0 была представлена концепция объектно-ориентированного Query API.

- *«Родные» запросы* — запросы этого типа пригодны для выполнения «родных» SQL-операторов вместо JPQL-операторов.
- *Запросы к хранимым процедурам* — JPA 2.1 привносит новый API для вызова хранимых процедур.

Центральной точкой, обуславливающей выбор из этих пяти типов запросов, является интерфейс менеджера сущностей, который обладает несколькими фабричными методами, приведенными в табл. 6.4, возвращая либо интерфейс `Query`, либо `TypedQuery`, либо `StoredProcedureQuery` (`TypedQuery` и `StoredProcedureQuery` расширяют `Query`). Интерфейс `Query` используется в тех случаях, когда типом результата является `Object`, а `TypedQuery` применяется, когда предпочтителен типизированный результат. `StoredProcedureQuery` задействуется для контроля выполнения запросов к хранимым процедурам.

Таблица 6.4. Методы менеджера сущностей для генерирования запросов

Метод	Описание
Query createQuery(String jpqlString)	Создает экземпляр Query для выполнения JPQL-оператора для динамических запросов
Query createNamedQuery(String name)	Создает экземпляр Query для выполнения именованного запроса (с использованием JPQL или «родного» SQL)
Query createNativeQuery(String sqlString)	Создает экземпляр Query для выполнения «родного» SQL-оператора
Query createNativeQuery(String sqlString, Class resultClass)	«Родной» запрос, передающий класс ожидаемых результатов
Query createNativeQuery(String sqlString, String resultSetMapping)	«Родной» запрос, передающий отображение результирующего набора
<T> TypedQuery<T> createQuery(CriteriaQuery<T> criteriaQuery)	Создает экземпляр TypedQuery для выполнения запроса с использованием критериев
<T> TypedQuery<T> createQuery(String jpqlString, Class<T> resultClass)	Типизированный запрос, передающий класс ожидаемых результатов
<T> TypedQuery<T> createNamedQuery(String name, Class<T> resultClass)	Типизированный запрос, передающий класс ожидаемых результатов
StoredProcedureQuery createStoredProcedureQuery(String procedureName)	Создает StoredProcedureQuery для выполнения хранимой процедуры в базе данных
StoredProcedureQuery createStoredProcedureQuery(String procedureName, Class... resultClasses)	Запрос к хранимой процедуре, передающий классы, в которые будут отображаться результирующие наборы
StoredProcedureQuery createStoredProcedureQuery(String procedureName, String... resultSetMappings)	Запрос к хранимой процедуре, передающий отображение результирующих наборов
StoredProcedureQuery createNamedStoredProcedureQuery(String name)	Генерирует запрос к именованной хранимой процедуре

При получении реализации интерфейса Query, TypedQuery или StoredProcedureQuery с помощью одного из фабричных методов в интерфейсе менеджера сущностей она будет контролироваться богатым API. API Query, показанный в листинге 6.21, задействуется для выполнения статических (то есть именованных) и динамических запросов с применением JPQL, а также «родных» запросов с использованием SQL. Кроме того, API Query поддерживает привязку параметров и управление разбиением на страницы.

Листинг 6.21. Query API

```
public interface Query {
  // Выполняет запрос и возвращает результат
  List getResultList();
  Object getSingleResult();
  int executeUpdate();

  // Задает параметры для запроса
  Query setParameter(String name, Object value);
  Query setParameter(String name, Date value, TemporalType temporalType);
  Query setParameter(String name, Calendar value, TemporalType temporalType);
  Query setParameter(int position, Object value);
  Query setParameter(int position, Date value, TemporalType temporalType);
  Query setParameter(int position, Calendar value, TemporalType temporalType);
  <T> Query setParameter(Parameter<T> param, T value);
  Query setParameter(Parameter<Date> param, Date value, TemporalType
temporalType);
  Query setParameter(Parameter<Calendar> param, Calendar value, TemporalType
temporalType);

  // Извлекает параметры посредством запроса
  Set<Parameter<?>> getParameters();
  Parameter<?> getParameter(String name);
  Parameter<?> getParameter(int position);
  <T> Parameter<T> getParameter(String name, Class<T> type);
  <T> Parameter<T> getParameter(int position, Class<T> type);
  boolean isBound(Parameter<?> param);
  <T> T getParameterValue(Parameter<T> param);
  Object getParameterValue(String name);
  Object getParameterValue(int position);

  // Ограничивает количество результатов, возвращаемых запросом
  Query setMaxResults(int maxResult);
  int getMaxResults();
  Query setFirstResult(int startPosition);
  int getFirstResult();

  // Задает и извлекает подсказки в запросах
  Query setHint(String hintName, Object value);
  Map<String, Object> getHints();

  // Задает тип режима сброса для использования при выполнении запроса
```

```
    Query setFlushMode(FlushModeType flushMode);
    FlushModeType getFlushMode();

    // Задает тип режима блокировки для использования при выполнении запроса
    Query setLockMode(LockModeType lockMode);
    LockModeType getLockMode();

    // Разрешает доступ к API, специфичному для поставщика
    <T> T unwrap(Class<T> cls);
}
```

Методы, которые главным образом используются в этом API, обеспечивают выполнение запроса как такового. Чтобы выполнить запрос SELECT, вам придется сделать выбор между двумя методами в зависимости от требуемого результата.

- Метод `getResultList()` выполняет запрос и возвращает список результатов (сущностей, атрибутов, выражений и т. д.).
- Метод `getSingleResult()` выполняет запрос и возвращает одиночный результат (генерирует исключение `NonUniqueResultException` при обнаружении нескольких результатов).

Для осуществления операции обновления или удаления метод `executeUpdate()` выполняет массовый запрос и возвращает несколько сущностей, затронутых при выполнении запроса.

Как вы уже видели в разделе «JPQL» ранее, при запросе могут использоваться параметры, которые являются либо именованными (например, `:myParam`), либо позиционными (например, `?1`). API `Query` определяет несколько методов `setParameter` для задания параметров перед выполнением запроса.

Когда вы выполняете запрос, он может возвратить большое количество результатов. В зависимости от приложения они могут быть обработаны все вместе либо порциями (например, веб-приложение выводит только десять строк за один раз). Для управления разбиением на страницы интерфейс `Query` определяет методы `setFirstResult()` и `setMaxResults()`, позволяющие указывать соответственно первый получаемый результат (с нумерацией, начинающейся с нуля) и максимальное количество результатов для возврата относительно этой точки.

Режим сброса является для поставщика постоянства индикатором того, как следует поступать с ожидаемыми изменениями и запросами. Есть две возможные настройки режима сброса: `AUTO` и `COMMIT`. `AUTO` (используемая по умолчанию) означает, что поставщик постоянства обеспечивает, что ожидаемые изменения будут видимыми при обработке запроса. `COMMIT` используется, когда эффект от обновления сущностей не перекрывается измененными данными в контексте постоянства.

Запросы можно блокировать с помощью метода `setLockMode(LockModeType)`.

В последующих разделах демонстрируется пять разных типов запросов с использованием описанных здесь методов.

Динамические запросы

Динамические запросы генерируются на лету по мере того, как это требуется приложению. Для создания динамического запроса используйте метод `EntityMana-`

ger.createQuery(), принимающий в качестве параметра строку, которая представляет JPQL-запрос.

В приведенном далее коде JPQL-запрос обеспечивает выборку всех клиентов из базы данных. Результатом этого запроса является список, так что, когда вы вызываете метод getResultList(), он возвращает список экземпляров сущности Customer (List<Customer>). Однако если вы знаете, что ваш запрос возвращает только одну сущность, используйте метод getSingleResult(). Он возвращает одну сущность и избегает работы по извлечению данных в виде списка.

```
Query query = em.createQuery("SELECT c FROM Customer c");
List<Customer> customers = query.getResultList();
```

Этот JPQL-запрос возвращает объект Query. Когда вы вызываете метод query.getResultList(), он возвращает список нетипизированных объектов. Если вы хотите, чтобы этот же запрос возвратил список типа Customer, то вам нужно использовать TypedQuery следующим образом:

```
TypedQuery<Customer> query =
    em.createQuery("SELECT c FROM Customer c", Customer.class);
List<Customer> customers = query.getResultList();
```

Эта строка запроса также может динамически создаваться приложением, которое затем способно сгенерировать заранее не известный комплексный запрос во время выполнения. Конкатенация строк используется для динамического генерирования запроса в зависимости от соответствующих критериев.

```
String jpqlQuery = "SELECT c FROM Customer c";
if (someCriteria)
    jpqlQuery += "WHERE c.firstName = 'Бетти'";
query = em.createQuery(jpqlQuery);
List<Customer> customers = query.getResultList();
```

Предыдущий запрос извлекает клиентов, для которых firstName имеет значение Бетти, однако вы можете захотеть передать параметр для firstName. Есть два возможных варианта для этого: с использованием имен или позиций. В приведенном далее примере я использую именованный параметр :fname (обратите внимание на символ «:») в запросе и привязываю его с помощью метода setParameter:

```
query = em.createQuery("SELECT c FROM Customer c where c.firstName = :fname");
query.setParameter("fname", "Бетти");
List<Customer> customers = query.getResultList();
```

Обратите внимание, что имя параметра fname не содержит двоеточия, используемого в запросе. Код, включающий в себя позиционный параметр, выглядел бы следующим образом:

```
query = em.createQuery("SELECT c FROM Customer c where c.firstName = ?1");
query.setParameter(1, "Бетти");
List<Customer> customers = query.getResultList();
```

Если вам потребуется прибегнуть к разбиению на страницы для вывода списка клиентов частями по десять человек, то можете воспользоваться методом setMaxResults, как показано далее:

```
query = em.createQuery("SELECT c FROM Customer c", Customer.class);
query.setMaxResults(10);
List<Customer> customers = query.getResultList();
```

В случае с динамическими запросами необходимо принимать во внимание, во что обходится преобразование JPQL-строк в SQL-операторы во время выполнения. Поскольку запрос генерируется динамически и его нельзя предсказать, поставщику постоянства приходится разбирать JPQL-строку, извлекать метаданные объектно-реляционного отображения и генерировать эквивалентный SQL-оператор. Обработка каждого из таких динамических запросов может отрицательно сказаться на производительности. Если вам потребуется выполнить статические запросы, которые являются неизменяемыми, и вы захотите избежать этих накладных расходов, то можете вместо этого обратиться к именованным запросам.

Именованные запросы

Именованные запросы отличаются от динамических тем, что являются статическими и неизменяемыми. В дополнение к их статической природе, которая не обеспечивает гибкости динамических запросов, выполнение именованных запросов может быть более эффективным, поскольку поставщик постоянства получает возможность преобразовать JPQL-строки в SQL-операторы, как только запустится приложение, а не делать это каждый раз при выполнении запроса.

Именованные запросы — это статические запросы, выражаемые метаданными внутри либо аннотации `@NamedQuery`, либо XML-эквивалента. Для определения этих запросов, пригодных для повторного использования, снабдите сущность аннотацией `@NamedQuery`, которая принимает два элемента: имя запроса и его содержимое. Итак, внесем изменения для сущности `Customer` и статически определим три запроса с использованием аннотаций (листинг 6.22).

Листинг 6.22. Определение именованных запросов для сущности Customer

```
@Entity
@NamedQueries({
  @NamedQuery(name = "findAll", query="select c from Customer c"),
  @NamedQuery(name = "findVincent",
              query="select c from Customer c where c.firstName = 'Винсент'"),
  @NamedQuery(name = "findWithParam",
              query="select c from Customer c where c.firstName = :fname")
})
public class Customer {
  @Id @GeneratedValue
  private Long id;
  private String firstName;
  private String lastName;
  private Integer age;
  private String email;
  @OneToOne
  @JoinColumn(name = "address_fk")
  private Address address;
  // Конструкторы, геттеры, сеттеры
}
```

Поскольку для сущности Customer определяется более одного именованного запроса, здесь применяется аннотация @NamedQueries, которая принимает массив @NamedQuery. Первый запрос findAll обеспечивает выборку всех клиентов из базы данных без каких-либо ограничений (оператор WHERE отсутствует). Запрос findWithParam задействует параметр :fname для ограничения выборки клиентов согласно значению firstName. В листинге 6.22 показан массив @NamedQuery, однако если бы в случае с Customer имел место один запрос, то он выглядел бы следующим образом:

```
@Entity
@NamedQuery(name = "findAll ", query="select c from Customer c")
public class Customer {...}
```

Способ выполнения таких именованных запросов схож с тем, что применяется для выполнения динамических запросов. Вызывается метод EntityManager.createNamedQuery(), которому передается имя запроса, определяемое аннотациями. Этот метод возвращает Query или TypedQuery, который может быть использован для задания параметров, максимального количества результатов, режимов выборки и т. д. Чтобы выполнить запрос findAll, напишите следующий код:

```
Query query = em.createNamedQuery("findAll");
```

Опять-таки если у вас возникнет необходимость ввести запрос для возврата списка объектов Customer, то придется воспользоваться TypedQuery, как показано далее:

```
TypedQuery<Customer> query = em.createNamedQuery("findAll", Customer.class);
```

Следующий фрагмент кода вызывает именованный запрос findWithParam с передачей параметра :fname и присваиванием setMaxResults значения 3:

```
Query query = em.createNamedQuery("findWithParam");
query.setParameter("fname", "Винсент");
query.setMaxResults(3);
List<Customer> customers = query.getResultList();
```

Поскольку большинство методов Query API возвращает объект Query, вы можете использовать элегантное сокращение при написании запросов. Вызывайте методы один за другим (setParameter().setMaxResults() и т. д.):

```
Query query = em.createNamedQuery("findWithParam").setParameter("fname",
                                                    "Винсент")
                                                    .setMaxResults(3);
```

Именованные запросы пригодны для организации определений запросов и являются эффективным средством повышения производительности приложений. Это происходит благодаря тому, что именованные запросы для сущностей определяются статически и обычно располагаются в классе-сущности, который соответствует непосредственно результату запроса (здесь запрос findAll возвращает всех клиентов, поэтому он должен быть определен в сущности Customer).

Есть ограничение: областью видимости для имени запроса является та, что имеет место в случае с единицей сохраняемости. При этом имя должно быть уникальным в этой области видимости, то есть может присутствовать только один метод findAll.

Запрос findAll, который касается клиентов, и запрос findAll, касающийся адресов, должны именоваться по-разному. Общепринятая практика — снабжение имени запроса префиксом в виде имени сущности. Например, запрос findAll к сущности Customer имел бы имя Customer.findAll.

Другая проблема заключается в том, что имя запроса, являющееся строкой, подвергается манипуляциям, и, если вы допустите опечатку или выполните реорганизацию своего кода, то могут быть сгенерированы исключения, говорящие о том, что запрос не существует. Чтобы ограничить риски, вы можете заменить имя запроса константой. В листинге 6.23 показано, как произвести реорганизацию сущности Customer.

Листинг 6.23. Сущность Customer и определение именованного запроса с использованием константы

```
@Entity
@NamedQuery(name = Customer.FIND_ALL, query="select c from Customer c"),
public class Customer {
  public static final String FIND_ALL = "Customer.findAll";
  // Атрибуты, конструкторы, геттеры, сеттеры
}
```

Константа FIND_ALL однозначно идентифицирует запрос findAll путем снабжения его имени префиксом в виде имени сущности. Та же константа затем используется в аннотации @NamedQuery, и вы можете задействовать ее для выполнения запроса, как показано далее:

```
TypedQuery<Customer> query = em.createNamedQuery(Customer.FIND_ALL,
      Customer.class);
```

Criteria API (или объектно-ориентированные запросы)

До сих пор при написании JPQL-операторов (для выполнения динамических или именованных запросов) я использовал строки. Такой подход удобен тем, что позволяет писать краткие запросы к базам данных. Однако он обладает и недостатками, к которым относятся предрасположенность к ошибкам и сложность манипулирования с помощью внешнего фреймворка. Это строка, а в результате вы прибегаете к конкатенации строк, из-за чего можете допустить множество опечаток. Например, вы можете допустить опечатки в ключевых словах JPQL (SLECT вместо SELECT), именах классов (Custmer вместо Customer) или атрибутов (firstname вместо firstName). Вы также можете написать синтаксически неправильный оператор (SELECT c WHERE c.firstName = 'John' FROM Customer). Любая из этих ошибок будет обнаружена во время выполнения, а иногда может быть сложно выяснить, откуда происходит определенный дефект.

С выходом JPA 2.0 появился новый API-интерфейс под названием *Criteria API*, определенный в пакете javax.persistence.criteria. Он позволяет вам писать любые запросы объектно-ориентированным и синтаксически правильным образом. Большинство ошибок, которые разработчики могут допустить при написании операторов, выявляется во время компиляции, а не во время выполнения. Идея заключается

в том, что все ключевые слова JPQL (SELECT, UPDATE, DELETE, WHERE, LIKE, GROUP BY...) определены в этом API-интерфейсе. Другими словами, Criteria API поддерживает все, что может сделать JPQL, но с использованием основанного на объектах синтаксиса. Впервые взглянем на запрос, который извлекает всех клиентов, для которых firstName имеет значение Vincent. На JPQL он выглядел бы следующим образом:

```
SELECT c FROM Customer c WHERE c.firstName = 'Винсент'
```

Этот JPQL-оператор был переписан в листинге 6.24 объектно-ориентированным образом с использованием Criteria API.

Листинг 6.24. Запрос с использованием критериев, обеспечивающий выборку всех клиентов, для которых firstName имеет значение Vincent

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = builder.createQuery(Customer.class);
Root<Customer> c = criteriaQuery.from(Customer.class);
criteriaQuery.select(c).where(builder.equal(c.get("firstName"), "Винсент"));
Query query = em.createQuery(criteriaQuery).getResultList();
List<Customer> customers = query.getResultList();
```

Не вдаваясь в подробности, вы можете видеть, что у ключевых слов SELECT, FROM и WHERE имеется API-представление с помощью методов select(), from() и where(). Это правило относится ко всем ключевым словам JPQL. Запросы с использованием критериев генерируются благодаря интерфейсу CriteriaBuilder, который получает менеджер сущностей (атрибут em в листингах 6.24 и 6.25). Он включает методы для генерирования определения запроса (этот интерфейс определяет такие ключевые слова, как desc(), asc(), avg(), sum(), max(), min(), count(), and(), or(), greaterThan(), lowerThan()...). Еще одна роль CriteriaBuilder — выступать в качестве главной фабрики запросов с использованием критериев (CriteriaQuery) и элементов запросов с применением критериев. Этот интерфейс определяет такие методы, как select(), from(), where(), orderBy(), groupBy() и having(), которые имеют эквивалентное значение в JPQL. В листинге 6.24 получение псевдонима c (как в SELECT c FROM Customer) осуществляется с помощью интерфейса Root (Root<Customer> c). Таким образом, для того чтобы написать любой необходимый SQL-оператор, вам потребуется лишь воспользоваться CriteriaBuilder, CriteriaQuery и Root: от самого простого (выборка всех сущностей из базы данных) до самого сложного (соединения, подзапросы, выражения CASE, функции...).

Листинг 6.25. Запрос с использованием критериев, обеспечивающий выборку всех клиентов старше 40 лет

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = builder.createQuery(Customer.class);
Root<Customer> c = criteriaQuery.from(Customer.class);
criteriaQuery.select(c).where(builder.greaterThan(c.get("age").as(Integer.
class), 40));
Query query = em.createQuery(criteriaQuery).getResultList();
List<Customer> customers = query.getResultList();
```

Обратимся к другому примеру. В листинге 6.25 приведен запрос, который извлекает всех клиентов старше 40 лет. Выражение c.get("age") извлекает атрибут age из сущности Customer и проверяет, больше ли 40 его значение.

Я начал этот раздел, сказав, что Criteria API позволяет вам писать операторы без ошибок. Однако это не совсем так. Взглянув на листинги 6.24 и 6.25, вы тем не менее сможете увидеть строки (`"firstName"` и `"age"`), которые представляют атрибуты сущности `Customer`. Таким образом, опечатки все равно возможны. В листинге 6.25 нам даже потребовалось преобразовать `age` в `Integer` (`(c.get("age").as(Integer.class))`), поскольку по-другому никак нельзя было бы понять, что атрибут `age` имеет тип `Integer`. Для решения этих проблем Criteria API предусматривает класс статической метамодели для каждой сущности, что обеспечивает безопасность типов в данном API-интерфейсе.

Типобезопасный Criteria API. Листинги 6.24 и 6.25 почти типобезопасны: каждое ключевое слово JPQL может быть представлено в методе интерфейсов `CriteriaBuilder` и `CriteriaQuery`. Единственной отсутствующей частью являются атрибуты сущности, которые основаны на строках: способ ссылки на атрибут `firstName` сущности `Customer` заключается в вызове `c.get("firstName")`. Метод `get` принимает строку в качестве параметра. Типобезопасный Criteria API решает проблему, переопределяя этот метод с использованием выражения пути из классов API метамодели, что обеспечивает безопасность типов.

В листинге 6.26 показана сущность `Customer` с несколькими атрибутами разных типов (`Long`, `String`, `Integer`, `Address`).

Листинг 6.26. Сущность Customer с несколькими атрибутами разных типов

```
@Entity
public class Customer {
  @Id @GeneratedValue
  private Long id;
  private String firstName;
  private String lastName;
  private Integer age;
  private String email;
  private Address address;
  // Конструкторы, геттеры, сеттеры
}
```

Для обеспечения безопасности типов JPA 2.1 может генерировать класс статической метамодели для каждой сущности. В соответствии с соглашением каждая сущность `X` будет обладать классом метаданных с именем `X_` (со знаком подчеркивания). Таким образом, у сущности `Customer` будет свое представление метамодели, описанное в классе `Customer_`, который приведен в листинге 6.27.

Листинг 6.27. Класс Customer_, описывающий метамодель Customer

```
@Generated("EclipseLink")
@StaticMetamodel(Customer.class)
public class Customer_ {
  public static volatile SingularAttribute<Customer, Long> id;
  public static volatile SingularAttribute<Customer, String> firstName;
  public static volatile SingularAttribute<Customer, String> lastName;
  public static volatile SingularAttribute<Customer, Integer> age;
```

```
    public static volatile SingularAttribute<Customer, String> email;
    public static volatile SingularAttribute<Customer, Address> address;
}
```

В классе статической метамодели каждый атрибут сущности `Customer` определяется подклассом `javax.persistence.metamodel.Attribute` (`CollectionAttribute`, `ListAttribute`, `MapAttribute`, `SetAttribute` или `SingularAttribute`). Каждый из этих атрибутов использует обобщения и является строго типизированным (например, `SingularAttribute<Customer, Integer>, age`). В листинге 6.28 приведен точно такой же код, что и в листинге 6.25, но пересмотренный с использованием класса статической метамодели (`c.get("age")` превратилось в `c.get(Customer_.age)`). Еще одно преимущество безопасности типов заключается в том, что метамодель определяет атрибут `age` как имеющий тип `Integer`, поэтому нет необходимости преобразовывать его в `Integer` с помощью `as(Integer.class)`.

Листинг 6.28. Типобезопасный запрос с использованием критериев, обеспечивающий выборку всех клиентов старше 40 лет

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = builder.createQuery(Customer.class);
Root<Customer> c = criteriaQuery.from(Customer.class);
criteriaQuery.select(c).where(builder.greaterThan(c.get(Customer_.age), 40));
Query query = em.createQuery(criteriaQuery).getResultList();
List<Customer> customers = query.getResultList();
```

Опять-таки это лишь примеры того, что вы можете сделать с помощью Criteria API. Это очень богатый API-интерфейс, который всесторонне охарактеризован в главах 5 и 6 спецификации JPA 2.1.

ПРИМЕЧАНИЕ

Классы, используемые в случае со статической метамоделью, например Attribute или SingularAttribute, являются стандартными и определены в пакете javax.persistence.metamodel. Однако генерирование классов статической метамодели зависит от реализации. EclipseLink использует внутренний класс CanonicalModelProcessor. Этот класс может вызываться вашей интегрированной средой разработки, пока вы разрабатываете Java-команду, Ant-задание или Maven-плагин.

«Родные» запросы

JPQL обладает очень богатым синтаксисом, который позволяет обрабатывать сущности в любой форме и обеспечивает переносимость между базами данных. JPA дает возможность использовать специфические особенности баз данных благодаря «родным» запросам. «Родные» запросы принимают «родной» SQL-оператор (`SELECT`, `UPDATE` или `DELETE`) в качестве параметра и возвращают экземпляр `Query` для выполнения этого оператора. Однако нельзя рассчитывать на переносимость «родных» запросов между базами данных.

Если код не является переносимым, то почему бы не использовать JDBC-вызовы? Главная причина, в силу которой следует задействовать «родные» запросы JPA, состоит в том, что результат запроса будет автоматически преобразован

обратно в сущности. Если вы захотите извлечь все экземпляры сущности `Customer` из базы данных с использованием SQL, то вам потребуется прибегнуть к методу `EntityManager.createNativeQuery()`, принимающему в качестве параметров SQL-запрос и класс-сущность, в который должен быть отображен результат.

```
Query query = em.createNativeQuery("SELECT * FROM t_customer", Customer.
class);
List<Customer> customers = query.getResultList();
```

Как вы можете видеть в приведенном фрагменте кода, SQL-запрос является строкой, которая динамически генерируется во время выполнения (точно так же, как динамические JPQL-запросы). Опять-таки запрос может быть комплексным, и, поскольку поставщик не будет заранее знать о нем, он станет каждый раз интерпретировать его. Подобно именованным запросам, «родные» могут задействовать аннотации для определения статических SQL-запросов. Именованные «родные» запросы определяются с помощью аннотации `@NamedNativeQuery`, которую необходимо поместить в код любой сущности (см. код ниже). Как и в случае с именованными JPQL-запросами, имя запроса должно быть уникальным в единице сохраняемости.

```
@Entity
@NamedNativeQuery(name = "findAll", query="select * from t_customer")
@Table(name = "t_customer")
public class Customer {...}
```

Запросы к хранимым процедурам

До сих пор у всех разных запросов (JPQL или SQL) было одно и то же назначение: отправка запроса от вашего приложения к базе данных, которая выполнит его и отошлет назад результат. Хранимые процедуры отличаются в том смысле, что они фактически хранятся в самой базе данных и выполняются в ее рамках.

Хранимая процедура — это подпрограмма, имеющаяся в распоряжении приложений, которые осуществляют доступ к реляционной базе данных. Хранимые процедуры обычно используются для экстенсивной или комплексной обработки, которая требует выполнения нескольких SQL-операторов либо для решения повторяющихся задач, связанных с работой с большими объемами данных. Как правило, хранимые процедуры пишутся на том или ином языке, близком к SQL, и, следовательно, не являются легко переносимыми между базами данных от разных поставщиков. Однако сохранение кода в базе данных даже в непереносимой форме обеспечивает многие преимущества.

- Лучшую производительность благодаря предварительной компиляции хранимой процедуры, а также повторного использования плана ее выполнения.
- Сохранение статистики, касающейся кода, для поддержания его оптимизированным.
- Снижение количества данных, передаваемых по сети, благодаря сохранению кода на сервере.
- Изменение кода в центральной локации без репликации в нескольких разных программах.

- Хранимые процедуры, которые могут использоваться множеством программ, написанных на разных языках (а не только на Java).
- Скрытие необработанных данных путем предоставления доступа к информации только хранимым процедурам.
- Усиление мер безопасности путем предоставления пользователем разрешения на выполнение той или иной хранимой процедуры независимо от разрешений, связанных с базовой таблицей.

Взглянем на пример из практики — архивирование старых книг и компакт-дисков. После определенной даты книги и компакт-диски должны помещаться в архив на конкретном складе, а это означает, что затем их придется физически перевозить со склада к перекупщику. Архивирование книг и компакт-дисков может отнимать много времени, поскольку потребуется обновлять несколько таблиц (с именами, например, `T_Inventory`, `T_Warehouse`, `T_Book`, `T_CD`, `T_Transport` и т. д.). Таким образом, мы можем написать хранимую процедуру для перегруппировки нескольких SQL-операторов и повышения производительности. Хранимая процедура `sp_archive_books`, определенная в листинге 6.29, принимает `archiveDate` и `warehouseCode` в качестве параметров и обновляет таблицы `T_Inventory` и `T_Transport`.

Листинг 6.29. Абстракция хранимой процедуры, используемой при архивировании книг

```
CREATE PROCEDURE sp_archive_books @archiveDate DATE, @warehouseCode VARCHAR AS
  UPDATE T_Inventory
  SET Number_Of_Books_Left - 1
  WHERE Archive_Date < @archiveDate AND Warehouse_Code = @warehouseCode;

  UPDATE T_Transport
  SET Warehouse_To_Take_Books_From = @warehouseCode;
END
```

Хранимая процедура из листинга 6.29 помещается в базу данных, а затем может вызываться по своему имени (`sp_archive_books`). Как вы можете видеть, хранимая процедура принимает данные в виде входных или выходных параметров. Входные параметры (`@archiveDate` и `@warehouseCode` в нашем примере) используются при выполнении хранимой процедуры, которая, в свою очередь, может генерировать выходной результат. Этот результат возвращается приложению при использовании результирующего набора.

В JPA 2.1 интерфейс `StoredProcedureQuery` (который расширяет `Query`) поддерживает хранимые процедуры. В отличие от динамических, именованных или «родных» запросов, этот API-интерфейс позволяет вам вызывать только ту хранимую процедуру, которая уже присутствует в базе данных, но не определять ее. Вы можете вызывать хранимую процедуру с помощью аннотаций (`@NamedStoredProcedureQuery`) либо динамически.

В листинге 6.30 показана сущность `Book`, для которой объявляется хранимая процедура `sp_archive_books` с использованием аннотаций именованных запросов. Аннотация `@NamedStoredProcedureQuery` определяет имя хранимой процедуры для вызова, типы всех параметров (`Date.class` и `String.class`), их соответствующие направления параметров (`IN`, `OUT`, `INOUT`, `REF_CURSOR`), а также то, как должны быть отображены

результирующие наборы (при наличии таковых). Аннотацией `@StoredProcedureParameter` необходимо снабдить каждый параметр.

Листинг 6.30. Сущность, для которой объявляется именованная хранимая процедура

```
@Entity
@NamedStoredProcedureQuery(name = "archiveOldBooks", procedureName = ➥
                           "sp_archive_books",
  parameters = {
    @StoredProcedureParameter(name = "archiveDate", mode = IN, type = Date.class),
    @StoredProcedureParameter(name = "warehouse", mode = IN, ➥
                              type = String.class)
  }
)
public class Book {
  @Id @GeneratedValue
  private Long id;
  private String title;
  private Float price;
  private String description;
  private String isbn;
  private String editor;
  private Integer nbOfPage;
  private Boolean illustrations;
  // Конструкторы, геттеры, сеттеры
}
```

Для вызова хранимой процедуры `sp_archive_books` вам потребуется прибегнуть к менеджеру сущностей и сгенерировать запрос к именованной хранимой процедуре, передав ее имя (`archiveOldBooks`). Этот запрос возвратит `StoredProcedureQuery`, для которого вы сможете задать параметры и выполнить его, как показано в листинге 6.31.

Листинг 6.31. Вызов StoredProcedureQuery

```
StoredProcedureQuery query = ➥
em.createNamedStoredProcedureQuery("archiveOldBooks");
query.setParameter("archiveDate", new Date());
query.setParameter("maxBookArchived", 1000);
query.execute();
```

Если хранимая процедура не определена с использованием метаданных (`@NamedStoredProcedureQuery`), то вы можете прибегнуть к API-интерфейсу для динамического генерирования запроса к хранимой процедуре. Это означает, что параметры и информацию касаемо результирующего набора потребуется обеспечить программным путем. Это можно сделать с помощью метода `registerStoredProcedureParameter` интерфейса `StoredProcedureQuery`, как показано в листинге 6.32.

Листинг 6.32. Регистрация и вызов StoredProcedureQuery

```
StoredProcedureQuery query = ➥
em.createStoredProcedureQuery("sp_archive_old_books");
query.registerStoredProcedureParameter("archiveDate", Date.class,
ParameterMode.IN);
```

```
query.registerStoredProcedureParameter("maxBookArchived", Integer.class,
ParameterMode.IN);

query.setParameter("archiveDate", new Date());
query.setParameter("maxBookArchived", 1000);
query.execute();
```

Cache API

В большинстве спецификаций (а не только в спецификации Java EE) много внимания уделено функциональным требованиям, а нефункциональным требованиям, таким как производительность, масштабируемость или кластеризация, отводится роль деталей реализации. Реализации должны строго придерживаться спецификации, однако также могут привносить свои особенности. Идеальным примером в случае с JPA будет кэширование.

До выхода версии JPA 2.0 в спецификации кэширование не упоминалось. Менеджер сущностей является кэшем первого уровня, используемым для всесторонней обработки информации для базы данных, а также для кэширования сущностей, которые живут короткий период времени. Этот кэш первого уровня задействуется отдельно в случае с каждой транзакцией для уменьшения количества SQL-запросов в рамках определенной транзакции. Например, если объект будет модифицирован несколько раз в рамках одной и той же транзакции, то менеджер сущностей сгенерирует только один оператор `UPDATE` в конце этой транзакции. Кэш первого уровня не является производительным кэшем.

Однако все реализации JPA используют производительный кэш (также называемый кэшем второго уровня) для оптимизации доступа к базам данных, запросов, соединений и т. д. Как показано на рис. 6.3, кэш второго уровня располагается между менеджером сущностей и базой данных с целью уменьшения трафика путем сохранения объектов загруженными в память и доступными для всего приложения.

Каждая реализация наделяется своим подходом к кэшированию объектов путем либо разработки собственного механизма, либо повторного использования уже существующих решений (с открытым исходным кодом или коммерческих). Кэширование может быть распределено по кластеру либо нет — все возможно, когда спецификация игнорирует соответствующую тему. Создатели JPA 2.0 признали, что кэш второго уровня необходим, и добавили операции кэширования в стандартный API-интерфейс. Приведенный в листинге 6.33 API-интерфейс крайне минималистичен (поскольку цель JPA не заключается в том, чтобы стандартизировать полнофункциональный кэш), однако позволяет коду выполнять запросы к некоторым сущностям и удалять их из кэша второго уровня стандартным образом. Как и менеджер сущностей, `javax.persistence.Cache` является интерфейсом, реализуемым системой кэширования поставщика постоянства.

Листинг 6.33. Cache API

```
public interface Cache {
  // Содержит ли кэш определенную сущность
```

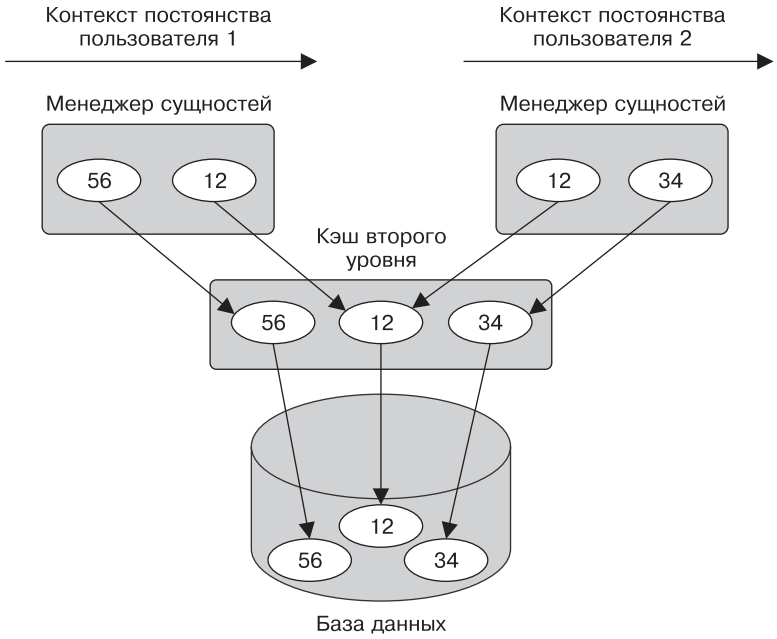


Рис. 6.3. Кэш второго уровня

```
    public boolean contains(Class cls, Object id);

    // Удаляет сущность из кэша
    public void evict(Class cls, Object id);
    // Удаляет сущности указанного класса (и его подклассы) из кэша
    public void evict(Class cls);
    // Очищает кэш
    public void evictAll();

    // Возвращает реализацию кэша, специфичную для поставщика
    public <T> T unwrap(Class<T> cls);
}
```

Вы можете использовать этот API для проверки того, содержится ли определенная сущность в кэше второго уровня, а также для очищения всего кэша. Задействуя этот API, вы можете явным образом проинформировать поставщика постоянства о том, является ли сущность кэшируемой, с помощью аннотации `@Cacheable`, как показано в листинге 6.34. Если у сущности отсутствует аннотация `@Cacheable`, то эта сущность и ее состояние не должны кэшироваться поставщиком.

Листинг 6.34. Сущность Customer является кэшируемой

```
@Entity
@Cacheable(true)
public class Customer {
  @Id @GeneratedValue
  private Long id;
```

```
  private String firstName;
  private String lastName;
  private String email;
  // Конструкторы, геттеры, сеттеры
}
```

Аннотация `@Cacheable` принимает логическое значение. Как только вы решите, какая сущность должна быть кэшируемой, вам придется проинформировать поставщика о том, какой механизм кэширования следует использовать. Это можно сделать с помощью JPA, указав атрибут `shared-cache-mode` в файле `persistence.xml`. Далее приведены возможные значения:

- `ALL` — будут кэшироваться все сущности, а также связанные с ними состояния и данные;
- `DISABLE_SELECTIVE` — будут кэшироваться все сущности за исключением тех, что снабжены аннотацией `@Cacheable(false)`;
- `ENABLE_SELECTIVE` — будут кэшироваться все сущности, снабженные аннотацией `@Cacheable(true)`;
- `NONE` — кэширование будет отключено для единицы сохраняемости;
- `UNSPECIFIED` — поведение кэширования будет неопределенным (могут быть применены правила по умолчанию, специфичные для поставщика).

Если не задать ни одно из этих значений, то поставщик будет сам решать, какой механизм кэширования применять. В коде, приведенном в листинге 6.35, показано, как использовать соответствующий механизм кэширования. Сначала мы создаем объект `Customer` и обеспечиваем его постоянство. Поскольку сущность `Customer` является кэшируемой (см. листинг 6.34), она должна быть размещена в кэше второго уровня (с помощью метода `EntityManagerFactory.getCache().contains()`). Вызов метода `ache.evict(Customer.class)` удаляет эту сущность из кэша.

Листинг 6.35. Сущность Customer является кэшируемой

```
Customer customer = new Customer("Патриция", "Джейн", "plecomte@mail.com");

tx.begin();
em.persist(customer);
tx.commit();

// Использует EntityManagerFactory для получения Cache
Cache cache = emf.getCache();

// Сущность Customer должна располагаться в кэше
assertTrue(cache.contains(Customer.class, customer.getId()));

// Удаляет сущность Customer из кэша
cache.evict(Customer.class);

// Сущность Customer больше не должна располагаться в кэше
assertFalse(cache.contains(Customer.class, customer.getId()));
```

Конкурентный доступ

JPA можно использовать для изменения постоянных данных, а JPQL — для извлечения данных, соответствующих определенным критериям. Все это может происходить в рамках приложения, выполняющегося в кластере с множеством узлов, множеством потоков и одной базой данных, благодаря чему осуществление конкурентного доступа к сущностям стало обычным явлением. Когда это имеет место, синхронизация должна контролироваться приложением с использованием механизма блокировки. Независимо от того, является приложение сложным или простым, есть вероятность, что вы решите применять блокировку где-нибудь в своем коде.

Чтобы проиллюстрировать проблему конкурентного доступа к базе данных, обратимся к примеру приложения с двумя конкурирующими потоками, показанными на рис. 6.4. Один поток ищет книгу по ее идентификатору и повышает цену книги на $2. Другой поток делает то же самое, но повышает цену книги на $5. Если вы выполните эти два потока одновременно в отдельных транзакциях, осуществляя манипуляции с одной и той же книгой, то не сможете предсказать окончательную цену книги. В этом примере начальная цена книги составляет $10. В зависимости от того, какая транзакция финиширует последней, цена может повыситься до $12 или 15.

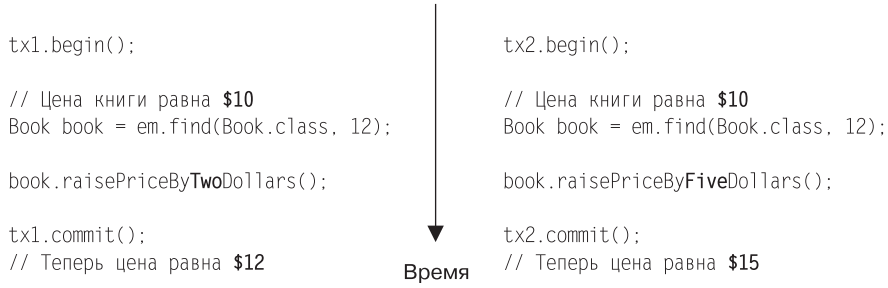


Рис. 6.4. Транзакции № 1 (tx1) и № 2 (tx2), одновременно обновляющие цену книги

Эта проблема конкурентного доступа, при которой «победителем» становится транзакция, которая фиксируется последней, неспецифична для JPA. При работе с базами данных эту проблему приходится решать с давних пор, и были найдены разные решения, позволяющие изолировать одну транзакцию от другой. Распространенный механизм, задействуемый базами данных, — это блокировка строки, в отношении которой выполняется SQL-оператор.

JPA 2.1 использует два разных механизма блокировки (версия JPA 1.0 поддерживала только оптимистическую блокировку).

- *Оптимистическая блокировка* основана на предположении, согласно которому большинство транзакций в базах данных не конфликтует с другими транзакциями, обеспечивая как можно более свободный конкурентный доступ, когда разрешается выполнение транзакций.
- *Пессимистическая блокировка* основана на противоположном предположении, в результате блокировка будет применяться к ресурсу до начала операций с ним.

В качества примера из повседневной жизни, подкрепляющего эти концепции, представьте себе «оптимистический и пессимистический переходы через улицу». Там, где движение транспорта совсем неинтенсивно, вы сможете перейти через дорогу, не проверяя, нет ли приближающихся автомобилей. Но вы не сможете так поступить в центре города!

JPA использует разные механизмы блокировки на разных уровнях API-интерфейса. Применение как пессимистической, так и оптимистической блокировки возможно с помощью методов `EntityManager.find` и `EntityManager.refresh` (в дополнение к методу `lock`), а также благодаря JPQL-запросам. Иначе говоря, блокировка может быть обеспечена на уровне менеджера сущностей и на уровне `Query` с помощью методов, приведенных в табл. 6.5 и 6.6.

Таблица 6.5. Методы менеджера сущностей для блокировки сущностей

Метод	Описание
<T> T find(Class<T> entityClass, Object primaryKey, LockModeType lockMode)	Выполняет поиск сущности указанного класса и первичного ключа, а затем блокирует ее согласно заданному типу режима блокировки
void lock(Object entity, LockModeType lockMode)	Блокирует экземпляр сущности, который содержится в контексте постоянства, согласно заданному типу режима блокировки
void refresh(Object entity, LockModeType lockMode)	Обновляет состояние экземпляра из базы данных, перезаписывая изменения, внесенные в сущность, при наличии таковых, и блокирует ее согласно заданному типу режима блокировки
LockModeType getLockMode(Object entity)	Извлекает значение текущего режима блокировки для экземпляра сущности

Таблица 6.6. Методы Query для блокировки JPQL-запросов

Метод	Описание
LockModeType getLockMode()	Извлекает значение текущего режима блокировки для запроса
Query setLockMode(LockModeType lockMode)	Задает тип режима блокировки для использования при выполнении запроса

Каждый из этих методов принимает `LockModeType` в качестве параметра, который, в свою очередь, может хранить разные значения:

- `OPTIMISTIC` — применяет оптимистическую блокировку;
- `OPTIMISTIC_FORCE_INCREMENT` — применяет оптимистическую блокировку и форсирует инкрементирование значения столбца `version`, связанного с сущностью (см. следующий подраздел «Контроль версий»);
- `PESSIMISTIC_READ` — применяет пессимистическую блокировку без необходимости в повторном чтении данных в конце транзакции для обеспечения блокировки;

- PESSIMISTIC_WRITE — применяет пессимистическую блокировку и форсирует сериализацию между транзакциями, которые пытаются обновить сущность;
- PESSIMISTIC_FORCE_INCREMENT — применяет пессимистическую блокировку и форсирует инкрементирование значения столбца version, связанного с сущностью (см. подраздел «Контроль версий» далее);
- NONE — определяет, что не должен использоваться никакой механизм блокировки.

Вы можете задавать эти параметры во многих местах в зависимости от того, как необходимо указать блокировки. Вы можете обеспечить чтение, а *затем* — блокировку.

```
Book book = em.find(Book.class, 12);
// Блокировать, чтобы повысить цену
em.lock(book, LockModeType.OPTIMISTIC_FORCE_INCREMENT);
book.raisePriceByTwoDollars();
```

Либо вы можете обеспечить чтение *и* блокировку.

```
Book book = em.find(Book.class, 12, LockModeType.OPTIMISTIC_FORCE_INCREMENT);
// Блокировка уже применена к сущности Book, повысить цену
book.raisePriceByTwoDollars();
```

Конкурентный доступ и блокировки являются ключевыми мотиваторами для контроля версий.

Контроль версий

В случае со спецификациями Java применяется контроль версий: Java SE 5.0, Java SE 6.0, EJB 3.1, JAX-RS 1.0 и т. д. При выходе новой версии JAX-RS ее номер увеличивается и вы переходите на JAX-RS 1.1. JPA использует этот точный механизм, когда вам необходимо присвоить номера версий сущностям. Таким образом, когда вы в первый раз обеспечите постоянство сущности в базе данных, она получит номер версии, равный 1. Позднее, если вы обновите атрибут и зафиксируете это изменение в базе данных, номер версии сущности будет равен уже 2 и т. д. Номер версии будет модифицироваться при каждом внесении изменений в сущность.

Чтобы это происходило, у сущности должен иметься атрибут для сохранения номера версии, снабженный аннотацией @Version. Он в дальнейшем отображается в столбец в базе данных. К числу типов атрибутов, поддерживающих контроль версий, относятся int, Integer, short, Short, long, Long и Timestamp. В листинге 6.36 показано, как добавить атрибут version в код сущности Book.

Листинг 6.36. Сущность Book с аннотацией @Version в случае с Integer

```
@Entity
public class Book {
  @Id @GeneratedValue
  private Long id;
  @Version
  private Integer version;
  private String title;
  private Float price;
```

```
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;
    // Конструкторы, геттеры, сеттеры
}
```

Сущность может получить доступ к значению своего свойства `version`, но не имеет возможности модифицировать его. Только поставщику постоянства разрешено задавать или обновлять значение атрибута `version` при записи или обновлении объекта в базе данных. Взглянем на пример, иллюстрирующий поведение, которое наблюдается при этом контроле версий. В листинге 6.37 обеспечивается постоянство новой сущности `Book` в базе данных. Как только происходит фиксация транзакции, поставщик задает для `version` значение `1`. Далее цена книги обновляется и, как только информация сбрасывается в базу данных, номер версии инкрементируется, становясь равным `2`.

Листинг 6.37. Транзакции tx1 и tx2, одновременно обновляющие цену книги

```
Book book = new Book("H2G2", 21f, "Лучшая IT-книга", "123-456", 321, false);
tx.begin();
em.persist(book);
tx.commit();
assertEquals(1, book.getVersion());

tx.begin();
book.raisePriceByTwoDollars();
tx.commit();
assertEquals(2, book.getVersion());
```

Атрибут `version` необязателен для использования, но его рекомендуется применять, когда сущность может быть одновременно модифицирована несколькими процессами или потоками. Контроль версий представляет собой ядро оптимистической блокировки и обеспечивает защиту при редких одновременных модификациях сущности. Фактически к сущности может быть автоматически применена оптимистическая блокировка, если у нее имеется свойство, снабженное аннотацией `@Version`.

Оптимистическая блокировка

Как видно из названия, оптимистическая блокировка основана на том факте, что транзакции в базах данных не конфликтуют друг с другом. Другими словами, высока вероятность того, что транзакция, обновляющая сущность, окажется единственной, которая в действительности будет обновлять сущность в этот промежуток времени. Следовательно, решение о применении блокировки к сущности на самом деле принимается в конце транзакции. Это гарантирует, что обновления сущности будут согласовываться с текущим состоянием базы данных. Результатом транзакций, которые привели бы к нарушению этого ограничения, стало бы генерирование исключения `OptimisticLockException`, а эти транзакции оказались бы помечены как подлежащие откату.

Как можно было бы сгенерировать исключение OptimisticLockException? Это можно сделать, либо применив явным образом блокировку к сущности (с помощью методов lock или find, которые вы видели ранее, передав LockModeType), либо разрешив поставщику постоянства проверить атрибут, снабженный аннотацией @Version. Использование специальной аннотации @Version в случае с сущностью позволяет менеджеру сущностей применить оптимистическую блокировку, просто сравнив значение атрибута version в экземпляре сущности со значением соответствующего столбца в базе данных. Если атрибут не будет аннотирован с использованием @Version, то менеджер сущностей не сможет применять оптимистическую блокировку автоматически (неявно).

Снова взглянем на пример повышения цены книги. Обе транзакции, tx1 и tx2, получают экземпляр одной и той же сущности Book. В тот момент номер версии сущности Book равен 1. Первая транзакция повышает цену книги на $2 и фиксирует это изменение. Когда информация сбрасывается в базу данных, поставщик постоянства увеличивает номер версии, делая его равным 2. В тот момент вторая транзакция поднимает цену на $5 и фиксирует это изменение. Менеджер сущностей для tx2 понимает, что номер версии в базе данных отличается от того, что имеется у сущности. Это означает, что номер версии был изменен в результате выполнения другой транзакции, из-за чего генерируется исключение OptimisticLockException, как показано на рис. 6.5.

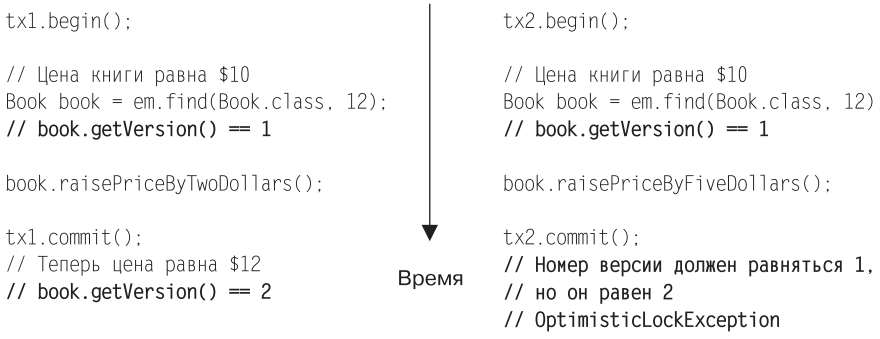


Рис. 6.5. Исключение OptimisticLockException, генерируемое в результате выполнения транзакции tx2

Это поведение по умолчанию, которое наблюдается при использовании аннотации @Version: исключение OptimisticLockException генерируется при сбросе данных (во время фиксации либо с помощью явного вызова метода em.flush()). Вы также можете решать, где вам требуется добавить оптимистическую блокировку, обеспечивая чтение, а затем — блокировку либо чтение и блокировку. Например, код для обеспечения чтения и блокировки выглядел бы следующим образом:

```
Book book = em.find(Book.class, 12);
// Блокировать, чтобы повысить цену
em.lock(book, LockModeType.OPTIMISTIC);
book.raisePriceByTwoDollars();
```

При оптимистической блокировке LockModeType, передаваемый вами в качестве параметра, может принимать два значения: OPTIMISTIC и OPTIMISTIC_FORCE_INCREMENT (или соответственно READ и WRITE, однако эти значения устарели). Единственное отличие заключается в том, что OPTIMISTIC_FORCE_INCREMENT форсирует обновление (инкрементирование) значения столбца version, связанного с сущностью.

При работе с приложениями настоятельно рекомендуется делать возможной оптимистическую блокировку всех сущностей, к которым разрешен конкурентный доступ. Неиспользование механизма блокировки может привести к несогласованному состоянию сущности, потерянным обновлениям и другим нарушениям состояния. Оптимистическая блокировка — полезная оптимизация производительности, освобождающая от работы, которую в ином случае потребовалось бы выполнить базе данных. Она представляет собой альтернативу пессимистической блокировке, которая подразумевает применение низкоуровневой блокировки базы данных.

Пессимистическая блокировка

Пессимистическая блокировка основана на предположении, противоположном тому, которое действует для оптимистической блокировки, так как пессимистическая блокировка быстро применяется к сущности до начала операций с ней. Это очень ограничивает ресурсы и приводит к значительному снижению производительности, так как блокировка базы данных поддерживается с использованием SQL-оператора SELECT ... FOR UPDATE для чтения данных.

Базы данных обычно предлагают службу для обеспечения пессимистической блокировки. Такая служба позволяет менеджеру сущностей блокировать строку таблицы для предотвращения обновления этой же строки другим потоком. Это эффективный механизм, который гарантирует, что два клиента не смогут одновременно модифицировать одну и ту же строку, однако он требует проведения затратных низкоуровневых проверок в базе данных. Результатом транзакций, которые привели бы к нарушению этого ограничения, стало бы генерирование исключения PessimisticLockException, а эти транзакции были бы помечены как подлежащие откату.

Оптимистическая блокировка целесообразна, когда вы сталкиваетесь с умеренным соперничеством между конкурирующими транзакциями. Однако в некоторых приложениях с более высокой степенью риска соперничества уместнее может оказаться пессимистическая блокировка, поскольку блокировка базы данных применяется незамедлительно в противоположность сбоям оптимистических транзакций, случающимся позднее. Например, во времена экономических кризисов на фондовые рынки поступает огромное количество поручений на продажу. Если одновременно 100 миллионам американцев потребуется продать ценные бумаги, то системе придется прибегнуть к пессимистическим блокировкам для обеспечения согласованности данных. Следует отметить, что в настоящее время рынок настроен довольно пессимистично, а не оптимистично, однако это никак не связано с JPA.

Пессимистическая блокировка может применяться к сущностям, которые не включают аннотированный атрибут @Version.

Жизненный цикл сущности

Вам уже известно большинство секретов сущностей, поэтому взглянем на их жизненный цикл. Созданный (с использованием оператора `new`) экземпляр сущности виртуальная машина Java рассматривает как простой Java-объект (то есть отсоединенный), который может быть использован приложением в качестве простого объекта. Затем, когда менеджер сущностей обеспечивает постоянство этой сущности, считается, что она находится под управлением. Когда сущность будет находиться под управлением, менеджер сущностей автоматически синхронизирует значения ее атрибутов с основной базой данных (например, если вы измените значение атрибута с помощью метода `set`, а сущность при этом будет находиться под управлением, то это новое значение окажется автоматически синхронизировано с базой данных).

Чтобы лучше понять этот процесс, взгляните на рис. 6.6, где приведена UML-диаграмма состояний. На ней показаны переходы между всеми состояниями сущности `Customer`.

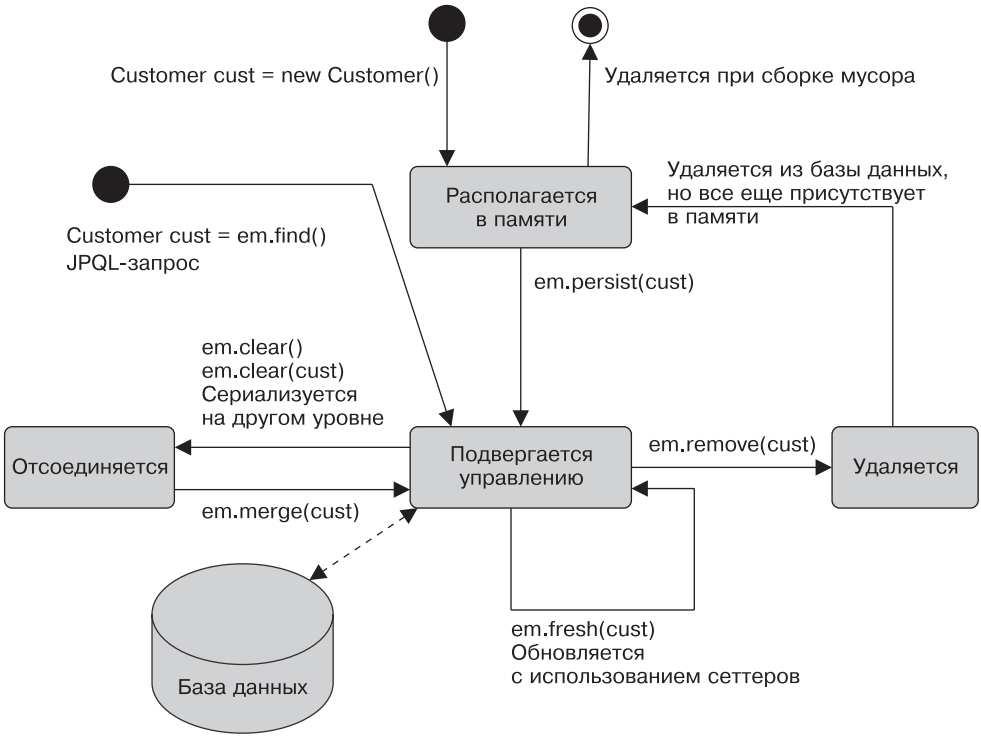


Рис. 6.6. Жизненный цикл сущности

Чтобы создать экземпляр сущности `Customer`, вы используете оператор `new`. Этот объект затем располагается в памяти, хотя JPA ничего о нем не знает. Если вы не станете что-либо делать с объектом, то он окажется вне области видимости и в ито-

ге будет удален при сборке мусора, что станет концом его жизненного цикла. Далее вы можете обеспечить постоянство `Customer` с помощью метода `EntityManager.persist()`. В тот момент сущность оказывается под управлением, а ее состояние синхронизируется с базой данных. Пока сущность пребывает в этом состоянии, в котором она подвергается управлению, вы можете обновить атрибуты с использованием методов-сеттеров (например, `customer.setFirstName()`) или обновить содержимое с помощью метода `EntityManager.refresh()`. Все эти изменения будут синхронизированы между сущностью и базой данных. Если вы вызовете метод `EntityManager.contains(customer)`, пока сущность пребывает в этом состоянии, то он возвратит `true`, поскольку сущность `Customer` содержится в контексте постоянства (то есть находится под управлением).

Сущность также может оказаться под управлением при загрузке из базы данных. Когда вы используете метод `EntityManager.find()` или генерируете JPQL-запрос для извлечения списка сущностей, все сущности автоматически подвергаются управлению и вы можете начать обновлять или удалять их атрибуты.

Пока сущность пребывает в состоянии, в котором она подвергается управлению, вы можете вызвать метод `EntityManager.remove()`, в результате чего эта сущность окажется удалена из базы данных и больше не будет находиться под управлением. Однако Java-объект продолжит располагаться в памяти, и вы все еще сможете использовать его до тех пор, пока сборщик мусора не избавится от этого объекта.

Теперь взглянем на состояние, в котором сущность является отсоединенной. Вы уже видели в предыдущей главе, как явный вызов метода `EntityManager.clear()` или `EntityManager.detach(customer)` приводит к удалению сущности из контекста постоянства; она оказывается отсоединенной. Однако есть другой, более тонкий способ отсоединить сущность: ее сериализация.

Во многих примерах в этой книге сущности ничего не реализуют, однако если им потребуется пересечь сеть при удаленном вызове или пересечь уровни для вывода на уровне представления, то им потребуется реализовать интерфейс `java.io.Serializable`. Это не JPA-, а Java-ограничение. Когда находящаяся под управлением сущность сериализуется, пересекает сеть и десериализуется, она рассматривается как отсоединенный объект. Чтобы снова присоединить сущность, вам потребуется вызвать метод `EntityManager.merge()`. Распространенный сценарий использования — вариант, когда сущность задействуется в JSF-странице. Допустим, сущность `Customer` выводится в форме на удаленной JSF-странице, подлежащей обновлению. Будучи удаленной, сущность нуждается в сериализации на стороне сервера перед отправкой на уровень представления. В тот момент сущность автоматически отсоединяется. После вывода, если какие-либо данные изменятся и их понадобится обновить, форма будет отправлена, а сущность — отослана обратно на сервер, десериализована. Она потребует слияния, чтобы снова присоединиться.

Методы обратного вызова и слушатели позволяют вам добавлять собственную бизнес-логику, когда с сущностью происходят определенные события жизненного цикла.

Обратные вызовы

Операции, выполняемые с сущностями во время их жизненного цикла, подпадают под четыре категории: обеспечение постоянства, обновление, удаление и загрузка. При этом они аналогичны категориям операций с базами данных, к которым относятся соответственно вставка, обновление, удаление и выборка. Во время каждого жизненного цикла имеют место события с приставками `pre` и `post`, которые могут быть перехвачены менеджером сущностей для вызова бизнес-метода. Такие бизнес-методы должны быть снабжены одной из аннотаций, описанных в табл. 6.7. Эти аннотации могут применяться к методам класса-сущности, отображенного суперкласса или класса-слушателя обратных вызовов.

Таблица 6.7. Аннотации обратных вызовов жизненного цикла

Аннотация	Описание
@PrePersist	Помечает метод для вызова до выполнения EntityManager.persist()
@PostPersist	Помечает метод для вызова после того, как будет обеспечено постоянство сущности. Если сущность будет автоматически генерировать свой первичный ключ (с использованием @GeneratedValue), то значение окажется доступно в соответствующем методе
@PreUpdate	Помечает метод для вызова до выполнения операции обновления в отношении базы данных (путем вызова сеттеров сущности или метода EntityManager.merge())
@PostUpdate	Помечает метод для вызова после выполнения операции обновления в отношении базы данных
@PreRemove	Помечает метод для вызова до выполнения EntityManager.remove()
@PostRemove	Помечает метод для вызова после того, как сущность будет удалена
@Postload	Помечает метод для вызова после того, как сущность будет загружена (посредством JPQL-запроса или EntityManager.find()) либо обновлена из основной базы данных. Аннотации @Preload не существует, поскольку в предварительной загрузке данных для сущности, которая еще не создана, нет смысла

Результатом добавления аннотаций обратных вызовов в UML-диаграмму состояний, показанную на рис. 6.6, является диаграмма, которую можно увидеть на рис. 6.7.

Перед вставкой сущности в базу данных менеджер сущностей вызывает метод, снабженный аннотацией `@PrePersist`. Если вставка не приведет к генерированию исключения, то будет обеспечено постоянство сущности, инициализирован ее идентификатор, а затем вызван метод, снабженный аннотацией `@PostPersist`. Аналогичное поведение наблюдается при операциях обновления (`@PreUpdate`, `@PostUpdate`) и удаления (`@PreRemove`, `@PostRemove`). Метод, аннотированный с использованием `@PostLoad`, вызывается после загрузки сущности из базы данных (посредством `EntityManager.find()` или JPQL-запроса). Когда сущность отсоединяется и требуется произвести ее слияние, менеджер сущностей сначала должен проверить, имеются ли какие-либо отличия от информации, содержащейся в базе данных (`@PostLoad`). Если да, то ему надлежит обновить данные (`@PreUpdate`, `@PostUpdate`).

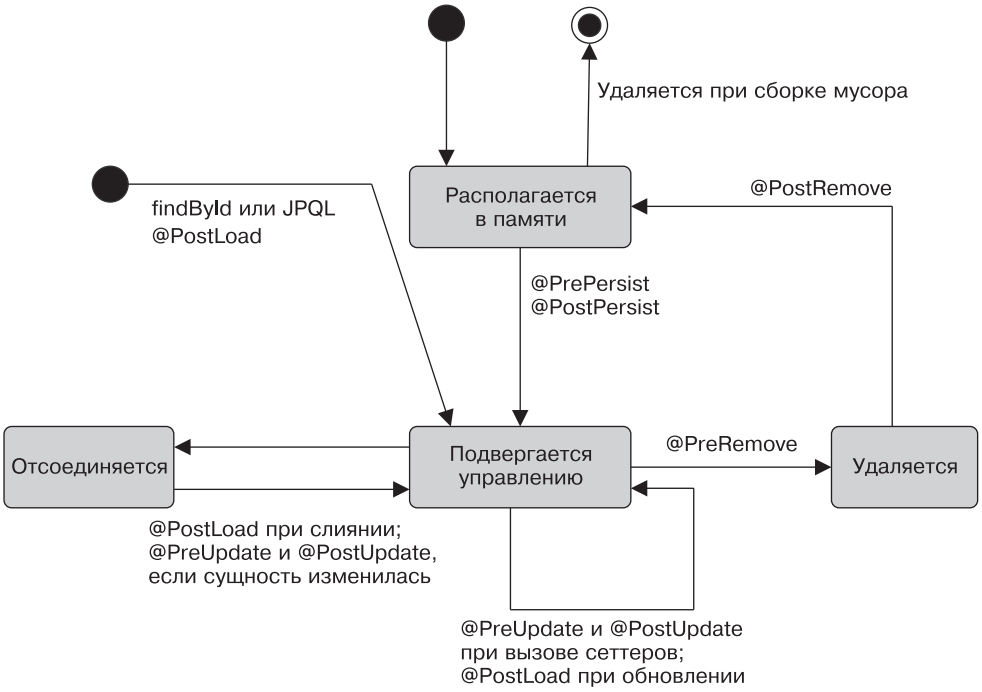


Рис. 6.7. Жизненный цикл сущности с аннотациями обратных вызовов

Как все это выглядит в коде? Сущности могут включать не только атрибуты, конструкторы, геттеры и сеттеры, но и бизнес-логику, используемую для валидации их состояния или выполнения вычислений для некоторых их атрибутов. Сюда могут входить обычные Java-методы, вызываемые другими классами, или аннотации обратных вызовов (которые также называются методами обратного вызова), как показано в листинге 6.38. Менеджер сущностей вызывает их автоматически в зависимости от инициируемого события.

Листинг 6.38. Сущность Customer с аннотациями обратных вызовов

```
@Entity
public class Customer {
  @Id @GeneratedValue
  private Long id;
  private String firstName;
  private String lastName;
  private String email;
  private String phoneNumber;
  @Temporal(TemporalType.DATE)
  private Date dateOfBirth;
  @Transient
  private Integer age;
  @Temporal(TemporalType.TIMESTAMP)
```

```
  private Date creationDate;

  @PrePersist
  @PreUpdate
  private void validate() {
    if (firstName == null || "".equals(firstName))
      throw new IllegalArgumentException("Неверное имя");
    if (lastName == null || "".equals(lastName))
      throw new IllegalArgumentException("Неверная фамилия");
  }

  @PostLoad
  @PostPersist
  @PostUpdate
  public void calculateAge() {
    if (dateOfBirth == null) {
      age = null;
      return;
    }

    Calendar birth = new GregorianCalendar();
    birth.setTime(dateOfBirth);
    Calendar now = new GregorianCalendar();
    now.setTime(new Date());
    int adjust = 0;
    if (now.get(DAY_OF_YEAR) - birth.get(DAY_OF_YEAR) < 0) {
      adjust = -1;
    }
    age = now.get(YEAR) - birth.get(YEAR) + adjust;
  }
  // Конструкторы, геттеры, сеттеры
}
```

В листинге 6.38 сущность `Customer` включает метод для валидации ее данных (он проверяет атрибуты `firstName` и `lastName`). Этот метод аннотирован с использованием `@PrePersist` и `@PreUpdate` и будет вызываться до вставки данных или обновления информации в базе данных. Если данные не смогут успешно пройти валидацию, то будет сгенерировано исключение времени выполнения, а вставка или обновление будут подвергнуты откату для гарантии того, что данные, вставленные или обновленные в базе данных, окажутся валидными.

Метод `calculateAge()` вычисляет возраст клиента. Атрибут `age` временный и не отображается в базе данных. После загрузки сущности, обеспечения ее постоянства или обновления метод `calculateAge()` принимает значение даты рождения клиента, вычисляет его возраст и задает значение для соответствующего атрибута.

Приведенные далее правила применяются к методам обратного вызова жизненного цикла.

- Методы могут иметь доступ `public`, `private`, `protected` или доступ на уровне пакета, однако не должны быть `static` или `final`. Обратите внимание в листинге 6.38 на то, что метод `validate()` является `private`.

- Метод может быть снабжен множественными аннотациями событий жизненного цикла (метод `validateData()` аннотирован с использованием `@PrePersist` и `@PreUpdate`). Однако для класса-сущности может присутствовать только одна аннотация жизненного цикла определенного типа (например, для одной и той же сущности не может быть две аннотации `@PrePersist`).
- Метод может генерировать непроверяемые исключения (времени выполнения), но не может — проверяемые. Генерирование исключения времени выполнения приведет к откату транзакции при наличии таковой.
- Метод может вызывать JNDI, JDBC, JMS и EJB-компоненты, но не может осуществлять какие-либо операции `EntityManager` и `Query`.
- При наследовании, если метод указан в суперклассе, он будет вызываться раньше соответствующего метода в дочернем классе. Например, если бы в листинге 6.38 сущность `Customer` наследовала от сущности `Person`, то метод `Person` с аннотацией `@PrePersist` вызывался бы раньше метода `Customer` с аннотацией `@PrePersist`.
- Если при работе со связями используется каскадирование событий, то метод обратного вызова тоже будет вызываться каскадным образом. Допустим, сущность обладает коллекцией адресов, а в случае со связью задано каскадное удаление. При удалении `Customer` произошел бы вызов метода `Address` с аннотацией `@PreRemove`, а также метода `Customer` с аннотацией `@PreRemove`.

Слушатели

Методы обратного вызова, которыми располагает сущность, хорошо работают при наличии бизнес-логики, связанной только с этой сущностью. Слушатели сущностей применяются для извлечения бизнес-логики в отдельный класс и обеспечения ее совместного использования другими сущностями. Слушатель сущности — это простой Java-объект, в случае с которым вам требуется определить один или несколько методов обратного вызова жизненного цикла. Для регистрации слушателя сущности необходимо задействовать аннотацию `@EntityListeners`.

Используя пример `Customer`, извлечем методы `calculateAge()` и `validate()` в отдельные классы-слушатели: соответственно `AgeCalculationListener` (листинг 6.39) и `DataValidationListener` (листинг 6.40).

Листинг 6.39. Слушатель, используемый для вычисления значения age сущности Customer

```
public class AgeCalculationListener {
  @PostLoad
  @PostPersist
  @PostUpdate
  public void calculateAge(Customer customer) {
    if (customer.getDateOfBirth() == null) {
      customer.setAge(null);
      return;
    }

    Calendar birth = new GregorianCalendar();
```

```
    birth.setTime(customer.getDateOfBirth());
    Calendar now = new GregorianCalendar();
    now.setTime(new Date());
    int adjust = 0; if (now.get(DAY_OF_YEAR) - birth.get(DAY_OF_YEAR) < 0) {
      adjust = -1;
    }
    customer.setAge(now.get(YEAR) - birth.get(YEAR) + adjust);
  }
}
```

Листинг 6.40. Слушатель, используемый для валидации атрибутов сущности Customer

```
public class DataValidationListener {
  @PrePersist
  @PreUpdate
  private void validate(Customer customer) {
    if (customer.getFirstName() == null || "".equals(customer.getFirstName()))
      throw new IllegalArgumentException("Неверное имя");
    if (customer.getLastName() == null || "".equals(customer.getLastName()))
      throw new IllegalArgumentException("Неверная фамилия");
  }
}
```

К классу-слушателю применяются только простые правила. Первое правило состоит в том, что класс должен располагать конструктором `public` без аргументов. Второе правило заключается в том, что подписи методов обратного вызова немного отличаются от тех, что приведены в листинге 6.38. Когда вы вызываете метод обратного вызова в слушателе, этому методу необходимо иметь доступ к состоянию сущности (например, к `firstName` и `lastName` сущности `Customer`, которые необходимо подвергнуть валидации). Методы должны располагать параметром, имеющим тип, который совместим с типом сущности, поскольку сущность, связанная с соответствующим событием, передается в обратный вызов. Метод обратного вызова, определенный в сущности, будет иметь такую подпись без параметров:

```
void <МЕТОД>();
```

Методы обратного вызова, определенные для слушателя сущности, могут иметь подписи двух разных типов. Если метод будет использоваться в нескольких сущностях, то у него должен быть аргумент `Object`:

```
void <МЕТОД>(Object anyEntity)
```

Если он предназначен только для одной сущности или ее подклассов (при наследовании), то параметр может иметь тип сущности:

```
void <МЕТОД>(Customer customerOrSubclasses)
```

Для обозначения того, что эти два слушателя будут уведомляться о событиях жизненного цикла сущности `Customer`, вам необходимо использовать аннотацию `@EntityListeners` (листинг 6.41). Она может принимать в качестве параметра один слушатель сущности либо массив слушателей. Если будет определено несколько

слушателей и произойдет событие жизненного цикла, то поставщик постоянства произведет итерацию по каждому слушателю в том порядке, в котором они указаны, и вызовет метод обратного вызова, передав ссылку на сущность, к которой относится соответствующее событие. Затем он вызовет методы обратного вызова в самой сущности (при наличии таковых).

Листинг 6.41. Сущность Customer, для которой определяется два слушателя

```
@EntityListeners({DataValidationListener.class, AgeCalculationListener.class})
@Entity
public class Customer {
  @Id @GeneratedValue
  private Long id;
  private String firstName;
  private String lastName;
  private String email;
  private String phoneNumber;
  @Temporal(TemporalType.DATE)
  private Date dateOfBirth;
  @Transient
  private Integer age;
  @Temporal(TemporalType.TIMESTAMP)
  private Date creationDate;
  // Конструкторы, геттеры, сеттеры
}
```

Результат выполнения этого кода будет точно таким же, что и кода из листинга 6.38. Сущность `Customer` валидирует свои данные перед вставкой или обновлением с использованием метода `DataValidationListener.validate()` и вычислит значение своего `age` с помощью метода слушателя `AgeCalculationListener.calculateAge()`.

Правила, которых должны придерживаться методы слушателя сущности, аналогичны правилам для методов обратного вызова сущности, за исключением нескольких деталей.

- Могут генерироваться только непроверяемые исключения. Это приводит к тому, что остальные слушатели и методы обратного вызова не вызываются, а транзакция подвергается откату (при наличии таковой).
- В иерархии наследования, если слушатели определены для множественных сущностей, слушатели, определенные в суперклассе, вызываются раньше слушателей, определенных в подклассах. Если не требуется, чтобы сущность наследовала слушателей суперкласса, то можно явным образом исключить их, используя аннотацию `@ExcludeSuperclassListeners` (или ее XML-эквивалент).

В листинге 6.41 показан код сущности `Customer`, где определяется два слушателя, однако слушатель может быть определен и для нескольких сущностей. Это может оказаться полезным в ситуациях, где слушатель обеспечивает общую логику, из которой могут извлечь выгоду многие сущности. Например, вы могли бы создать `DebugListener`, который будет выводить имена отдельных инициируемых событий, как показано в листинге 6.42.

Листинг 6.42. DebugListener, который может быть использован любой сущностью

```
public class DebugListener {
  @PrePersist
  void prePersist(Object object) {
    System.out.println("prePersist");
  }
  @PreUpdate
  void preUpdate(Object object) {
    System.out.println("preUpdate");
  }
  @PreRemove
  void preRemove(Object object) {
    System.out.println("preRemove");
  }
}
```

Обратите внимание, что каждый метод принимает `Object` в качестве параметра, а это означает, что сущность любого типа может использовать этот слушатель при добавлении класса `DebugListener` в свою аннотацию `@EntityListeners`. Чтобы любая сущность вашего приложения применяла этот слушатель, вам пришлось бы пройтись по каждой и добавить их вручную в аннотацию. На этот случай в JPA предусмотрено такое понятие, как слушатели по умолчанию, которые могут охватывать все сущности в контексте постоянства. Поскольку аннотация, нацеленная на всю область видимости единицы сохраняемости, отсутствует, слушатели по умолчанию могут быть объявлены только в файле отображения XML.

В предыдущей главе вы видели, как использовать файлы отображения XML вместо аннотаций. Для определения `DebugListener` в качестве слушателя по умолчанию придется выполнить те же самые шаги. Потребуется создать файл отображения с XML, определенным в листинге 6.43, и произвести его развертывание с использованием приложения.

Листинг 6.43. DebugListener, определенный в качестве слушателя по умолчанию

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
                 xsi:schemaLocation=" http://xmlns.jcp.org/xml/ns/persistence/orm
                 http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd"
                 version="2.1">

   <persistence-unit-metadata>
      <persistence-unit-defaults>
         <entity-listeners>
            <entity-listener class="org.agoncal.book.javaee7.chapter06.
DebugListener"/>
         </entity-listeners>
      </persistence-unit-defaults>
   </persistence-unit-metadata>
</entity-mappings>
```

В этом файле тег <persistence-unit-metadata> определяет все метаданные, у которых нет какого-либо эквивалента в виде аннотаций. Тег <persistence-unit-defaults> задает все настройки по умолчанию для единицы сохраняемости, а тег <entity-listener> определяет слушателя по умолчанию. В persistence.xml должна иметься ссылка на этот файл, развертывание которого необходимо выполнить с использованием приложения. Тогда DebugListener будет автоматически вызываться для каждой сущности.

Если вы объявите список слушателей сущности по умолчанию, то каждый из них будет вызываться в том порядке, в котором они указаны в файле отображения XML. Слушатели сущности по умолчанию всегда вызываются раньше любых слушателей сущности, определенных в аннотации @EntityListeners. Если потребуется, чтобы для сущности не применялись слушатели по умолчанию, то можно использовать аннотацию @ExcludeDefaultListeners, как показано в листинге 6.44.

Листинг 6.44. Сущность Customer, для которой исключаются слушатели по умолчанию

```
@ExcludeDefaultListeners
@Entity
public class Customer {
  @Id @GeneratedValue
  private Long id;
  private String firstName;
  private String lastName;
  private String email;
  private String phoneNumber;
  @Temporal(TemporalType.DATE)
  private Date dateOfBirth;
  @Transient
  private Integer age;
  @Temporal(TemporalType.TIMESTAMP)
  private Date creationDate;
  // Конструкторы, геттеры, сеттеры
}
```

Когда будет инициировано событие, слушатели станут выполняться в следующем порядке.

1. Указанные в @EntityListeners слушатели для определенной сущности или суперкласса в порядке расположения в массиве.
2. Слушатели сущности для суперклассов (в первую очередь станут выполняться те, что располагаются в самом верху списка).
3. Слушатели сущности для сущности.
4. Обратные вызовы суперклассов (в первую очередь будут выполняться те, что располагаются в самом верху списка).
5. Обратные вызовы сущности.

Резюме

Из этой главы вы узнали, как выполнять запросы к сущностям. Менеджер сущностей — это основной инструмент, используемый для взаимодействия с постоянными

сущностями. Он может создавать, обновлять сущности, выполнять поиск по идентификатору, удалять и синхронизировать сущности с базой данных с помощью контекста постоянства, который играет роль кэша первого уровня. JPA также сопутствует очень мощный язык запросов JPQL, который не зависит от поставщика базы данных. Вы можете извлекать сущности благодаря богатому синтаксису, задействуя операторы `WHERE`, `ORDER BY` или `GROUP BY`, а при конкурентном доступе к своим сущностям вы будете знать, как использовать контроль версий и когда следует применять оптимистическую или пессимистическую блокировку.

В этой главе также описан жизненный цикл сущности и то, как менеджер сущностей перехватывает события, чтобы запускать методы обратного вызова. Такие методы могут определяться в одной сущности и снабжаться несколькими аннотациями (`@PrePersist`, `@PostPersist` и т. д.). Метод также может извлекаться в классы-слушатели и использоваться несколькими или всеми сущностями (с применением слушателей сущностей по умолчанию). Благодаря методам обратного вызова вы понимаете, что сущности не просто анемичные объекты (объекты без бизнес-логики, располагающие только атрибутами, геттерами и сеттерами). Сущности способны включать бизнес-логику, которая может вызываться другими объектами в приложении, либо вызываться автоматически с помощью менеджера сущностей в зависимости от жизненного цикла сущности.

Глава 7

Корпоративные EJB-компоненты

В предыдущей главе было показано, как реализовать постоянные объекты с использованием JPA и как осуществлять к ним запросы с помощью JPQL. Сущности могут включать методы для валидации их атрибутов, однако они не предназначены для решения сложных задач, которые зачастую требуют взаимодействия с другими компонентами (другими постоянными объектами, внешними службами и т. д.). Уровень постоянства не подходит для обработки бизнес-данных. Равно как и интерфейс пользователя не следует применять для выполнения бизнес-логики, особенно когда имеется много интерфейсов (Web, Swing, портативные устройства и т. д.). Для разделения уровня постоянства и уровня представления, обеспечения управления транзакциями и усиления безопасности приложениям необходим бизнес-уровень. В Java EE мы реализуем его с использованием корпоративных EJB-компонентов (Enterprise JavaBeans — EJB).

Разделение на уровни важно для большинства приложений. Придерживаясь восходящего подхода, в предыдущих главах о JPA мы моделировали доменные классы, обычно определяя существительные (`Artist`, `CD`, `Book`, `Customer` и т. д.). На бизнес-уровне, располагающемся над доменным уровнем, моделируются действия (или глаголы) приложения (создать книгу, купить книгу, распечатать заказ, доставить книгу и т. д.). Этот бизнес-уровень часто взаимодействует с внешними веб-службами (такими как веб-службы SOAP или RESTful), обеспечивает отправку асинхронных сообщений в другие системы (с использованием JMS) или сообщений по электронной почте. Он осуществляет оркестровку компонентов из базы данных во внешние системы и выступает в роли центрального места для ограничения транзакций и безопасности, а также точки входа для клиентов любого типа вроде веб-интерфейсов (сервлетов или EJB-компонентов JSF, являющихся подложками), пакетной обработки или внешних систем. Это логическое разделение сущностей и сессионных EJB-компонентов придерживается парадигмы «разделение ответственности», при использовании которой приложение разбивается на отдельные компоненты. Их функции как можно меньше перекрывают друг друга.

В этой главе вы сначала познакомитесь с EJB-компонентами, а затем узнаете о трех разных типах сессионных EJB-компонентов: без сохранения состояния, с сохранением состояния и одиночных. EJB-компоненты без сохранения состояния являются наиболее масштабируемыми из этих типов, поскольку не сохраняют состояние и обеспечивают выполнение бизнес-логики с помощью одного вызова метода. EJB-компоненты с сохранением состояния поддерживают диалоговое состояние

для одного клиента. Версия EJB 3.1 добавила сеансовые одиночные EJB-компоненты (по одному экземпляру на приложение) в предыдущий релиз. Вы также увидите, как выполнять эти EJB-компоненты во встроенном контейнере и вызывать их синхронно или асинхронно.

Понятие корпоративных EJB-компонентов

EJB-компоненты — это серверные компоненты, которые инкапсулируют бизнес-логику, заботятся о транзакциях и безопасности. У них также имеется интегрированный стек для обмена сообщениями, планирования, удаленного доступа, конечных точек веб-служб (SOAP и REST), внедрения зависимостей, управления жизненным циклом компонентов, аспектно-ориентированного программирования (Aspect-Oriented Programming — AOP) с использованием перехватчиков и т. д. Кроме того, EJB-компоненты «бесшовно» интегрируются с другими технологиями Java SE и Java EE вроде JDBC, JavaMail, JPA, Java Transaction API (JTA), Java Messaging Service (JMS), сервиса аутентификации и авторизации Java (Java Authentication and Authorization Service — JAAS), Java Naming and Directory Interface (JNDI) и удаленного вызова методов (Remote Method Invocation — RMI). Вот почему они используются для создания уровня бизнес-логики (рис. 7.1), располагаются над уровнем базы данных и осуществляют оркестровку на уровне бизнес-модели. EJB-компоненты выступают в качестве точки входа для технологий уровня представления, например JavaServer Faces (JSF), а также для всех внешних служб (JMS или веб-служб).

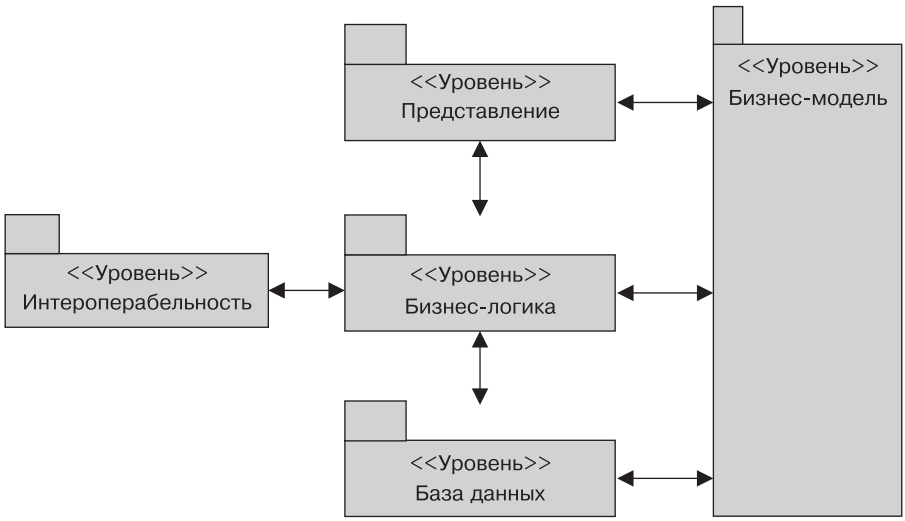


Рис. 7.1. Разделение архитектуры на уровни

EJB-компоненты — это очень мощная модель программирования, которая объединяет в себе легкость использования и надежность. Сегодня EJB-компоненты представляют собой простую модель разработки на стороне сервера с использованием Java, которая облегчает работу, одновременно привнося возможность повтор-

ного использования и масштабируемость в критически важные корпоративные приложения. Все это является результатом снабжения аннотацией одного Java-объекта, развертывание которого будет производиться в контейнере. EJB-контейнер — это среда выполнения, которая обеспечивает службы, например, для управления транзакциями, управления конкурентным доступом, организации пула и проверки прав на доступ. Исторически сложилось так, что серверы приложений привнесли другие особенности вроде кластеризации, балансировки нагрузки и обхода отказа. Кроме того, EJB-разработчики могут сосредоточиться на реализации бизнес-логики, пока контейнер занимается решением всех технических вопросов.

Сегодня как никогда, с выходом версии 3.2, EJB-компоненты можно написать один раз, а затем развертывать их в любом контейнере, который соответствует требуемой спецификации. Стандартные API-интерфейсы, переносимые JNDI-имена, легковесные компоненты, CDI-интеграция и конфигурация в порядке исключения позволяют с легкостью развертывать EJB-компоненты в реализациях с открытым кодом, а также в коммерческих реализациях. Базовая технология была создана более 12 лет назад, что привело к появлению EJB-приложений, которые выигрывают от использования доказанных концепций.

Типы EJB-компонентов

Сессионные EJB-компоненты отлично подходят для реализации бизнес-логики, процессов и потока работ. А поскольку корпоративные приложения могут быть сложными, платформа Java EE определяет несколько типов EJB-компонентов.

- *Без сохранения состояния* — не поддерживает диалоговое состояние между методами, и любой экземпляр может быть использован для любого клиента. Он применяется для решения задач, с которыми можно справиться одним вызовом метода.
- *С сохранением состояния* — поддерживает диалоговое состояние, которое может сохраняться между методами для одного пользователя. Он полезен для решения задач, с которыми можно справиться в несколько этапов.
- *Одиночный* — один EJB-компонент совместно применяется клиентами и поддерживает конкурентный доступ. Контейнер позаботится о том, чтобы для всего приложения имелся только один экземпляр.

Разумеется, у всех трех типов EJB-компонентов имеются специфические особенности, однако у них также есть много общего. Прежде всего, они обладают одинаковой моделью программирования. Как вы увидите позднее, у EJB-компонента может иметься локальный и/или удаленный интерфейс либо не быть вообще никакого интерфейса. Сессионные EJB-компоненты управляются контейнером, поэтому необходимо упаковать их в архив (файл с расширением JAR, WAR или EAR) и произвести их развертывание в контейнере.

EJB-компоненты, управляемые сообщениями (Message-Driven Beans — MDB), применяются для интеграции с внешними системами путем получения асинхронных сообщений с использованием JMS. Хотя EJB-компоненты, управляемые сообщениями, являются частью спецификации EJB, я рассматриваю их отдельно (в главе 13), поскольку соответствующая компонентная модель главным образом

применяется для интеграции с промежуточным программным обеспечением, ориентированным на обработку сообщений (Message-Oriented Middleware — MOM). EJB-компоненты, управляемые сообщениями, обычно делегируют бизнес-логику сессионным EJB-компонентам.

EJB-компоненты также используются в качестве конечных точек веб-служб. В главах 14 и 15 демонстрируются веб-службы SOAP и RESTful, которые могут быть либо простыми Java-объектами, развернутыми в веб-контейнере, либо сессионными EJB-компонентами, развернутыми в EJB-контейнере.

ПРИМЕЧАНИЕ

Для совместимости спецификация EJB 3.1 все еще включала Entity CMP. Эта постоянная компонентная модель была удалена и теперь стала необязательной в EJB 3.2. Технология JPA предпочтительна для отображения и выполнения запросов к реляционным базам данных. В этой книге не рассматривается Entity CMP.

Процесс и встроенный контейнер

С самого момента их изобретения (EJB 1.0) EJB-компоненты надлежало выполнять в контейнере, функционирующем на виртуальной машине Java. Подумайте о GlassFish, JBoss, Weblogic и т. д., и вы вспомните, что сначала нужно запустить сервер приложений, а затем производить развертывание и приступать к использованию своих EJB-компонентов. Этот внутрипроцессный контейнер подходит для среды производства, где сервер работает непрерывно. Однако подобный подход отнимает много времени в среде разработки, где вам часто требуется выполнять такие операции с контейнером, как запуск, развертывание, отладка и остановка. Другая проблема с серверами, работающими в разных процессах, заключается в том, что возможности тестирования ограниченны. Либо вы будете имитировать все контейнерные службы для модульного тестирования, либо вам потребуется произвести развертывание своего EJB-компонента на реальном сервере для осуществления интеграционного тестирования. Для решения этих проблем некоторые реализации серверов приложений предусматривают встроенные контейнеры, однако они зависят от реализации. Начиная с EJB 3.1 экспертная группа стандартизировала встроенные контейнеры, которые являются переносимыми между серверами.

Идея встроенного контейнера заключается в том, чтобы иметь возможность выполнять EJB-приложения в среде Java SE, позволяя клиентам применять для работы одну и ту же виртуальную машину Java и загрузчик классов. Это обеспечивает лучшую поддержку интеграционного тестирования, автономной обработки (например, пакетной обработки) и использования EJB-компонентов в настольных приложениях. API-интерфейс, связанный со встраиваемыми контейнерами, обеспечивает ту же самую управляемую среду, что и контейнер времени выполнения Java EE, и включает те же самые службы. Теперь вы можете задействовать встроенный контейнер на той же самой виртуальной машине Java, на которой функционирует ваша интегрированная среда разработки, и выполнять отладку своих EJB-компонентов без необходимости какой-либо разработки для отделения сервера приложений.

Службы, обеспечиваемые контейнером

Независимо то того, является контейнер встроенным или работает в отдельном процессе, он обеспечивает базовые службы, общие для многих корпоративных приложений, например такие, как те, что приведены далее.

- *Удаленные клиентские коммуникации* — EJB-клиент (другой EJB-компонент, интерфейс пользователя, пакетный процесс и т. д.) может вызывать методы удаленно с использованием стандартных протоколов без необходимости написания какого-либо сложного кода.
- *Внедрение зависимостей* — контейнер может внедрять некоторые ресурсы в EJB-компонент (пункты назначения и фабрики JMS, другие EJB-компоненты, источники данных, переменные среды и т. д.), а также любые POJO благодаря CDI.
- *Управление состоянием* — контейнер прозрачно управляет состоянием сессионных EJB-компонентов с сохранением состояния. Вы можете поддерживать состояние для определенного клиента, как если бы вы разрабатывали настольное приложение.
- *Организация пула* — в случае с EJB-компонентами без сохранения состояния и EJB-компонентами, управляемыми сообщениями, контейнер создает пул экземпляров, которые могут совместно применяться множественными клиентами. Будучи вызванным, EJB-компонент возвращает пул, который будет повторно использоваться, а не окажется уничтожен.
- *Управление жизненным циклом компонентов* — контейнер отвечает за управление жизненным циклом каждого компонента.
- *Обмен сообщениями* — контейнер позволяет EJB-компонентам, управляемым сообщениями, прослушивать пункты назначения и получать сообщения без чрезмерного использования JMS.
- *Управление транзакциями* — благодаря управлению декларативными транзакциями EJB-компонент может задействовать аннотации для информирования контейнера о том, какую политику тот должен использовать по отношению к транзакциям. Контейнер заботится о выполнении фиксации или отката.
- *Безопасность* — в случае с EJB-компонентами можно определить управление доступом на уровне класса или метода, чтобы форсировать авторизацию пользователей и ролей.
- *Поддержка конкурентного доступа* — за исключением одиночных EJB-компонентов, в случае с которыми требуется объявление конкурентного доступа, EJB-компоненты всех остальных типов потокобезопасны по своей природе. Вы можете разрабатывать высокопроизводительные приложения, не беспокоясь о проблемах, связанных с потоками.
- *Перехватчики* — в перехватчики можно заложить сквозную функциональность, которая будет автоматически вызываться контейнером.

- *Асинхронные вызовы методов* — начиная с EJB 3.1, теперь можно выполнять асинхронные вызовы без обмена сообщениями.

После того как будет произведено развертывание EJB-компонента, контейнер позаботится о применении описанных выше служб, позволив разработчику сосредоточиться на бизнес-логике, одновременно извлекая выгоду из этих служб без добавления какого-либо кода системного уровня.

EJB-компоненты являются управляемыми объектами. Фактически они считаются управляемыми MBean-компонентами (Managed Beans). Когда клиент вызывает EJB-компонент, он не работает непосредственно с экземпляром этого EJB-компонента, а взаимодействует с прокси, что наблюдается в случае с экземпляром. Каждый раз, когда клиент вызывает метод в EJB-компоненте, этот вызов на самом деле идет через прокси и перехватывается контейнером, который обеспечивает службы от имени экземпляра EJB-компонента. Разумеется, все это абсолютно прозрачно для клиента. С момента своего создания и до уничтожения корпоративный EJB-компонент располагается в контейнере.

В приложении Java EE EJB-контейнер обычно взаимодействует с другими контейнерами — контейнером сервлетов (отвечающим за управление выполнением сервлетов и применением JSF-страниц), контейнером клиентского приложения (Application Client Container — ACC) (для управления автономными приложениями), а также с брокером сообщений (для отправки, постановки в очередь и получения сообщений), поставщиком постоянства и т. д.

Контейнеры обеспечивают для EJB-компонентов набор служб. С другой стороны, EJB-компоненты не могут создавать потоки или управлять ими, осуществлять доступ к файлам с использованием `java.io`, генерировать `ServerSocket`, загружать «родные» библиотеки или применять библиотеку AWT (Abstract Window Toolkit) либо API-интерфейсы Swing для взаимодействия с пользователем.

EJB Lite

Корпоративные EJB-компоненты стали доминирующей компонентной моделью в Java EE 7, будучи самым простым средством обработки транзакций, а также безопасной обработки бизнес-данных. Однако EJB 3.2 все еще определяет сложные технологии, которые сегодня используются в меньшей степени, например интероперабельность IIOP (Internet InterOrb Protocol — Межброкерный протокол для Интернета), а это означает, что любому новому поставщику, реализующему спецификацию EJB 3.2, придется реализовать и эти технологии. При знакомстве с EJB-компонентами разработчики также оказались бы отягощенными многими технологиями, которые они в ином случае никогда бы не использовали.

По этим причинам спецификация определяет минимальное подмножество полной версии EJB API под названием EJB Lite. Сюда входит небольшая, но удачная подборка EJB-функций, подходящих для написания переносимой транзакционной и безопасной бизнес-логики. Любое приложение EJB Lite может быть развернуто в любом продукте Java EE, который реализует EJB 3.2. EJB Lite состоит из подмножества EJB API, приведенного в табл. 7.1.

Таблица 7.1. Сравнение EJB Lite и полной версии EJB

Функция	EJB Lite	Полная версия EJB 3.2
Сессионные EJB-компоненты (без сохранения состояния, с сохранением состояния, одиночные)	Да	Да
Представление без интерфейса	Да	Да
Локальный интерфейс	Да	Да
Перехватчики	Да	Да
Поддержка транзакций	Да	Да
Безопасность	Да	Да
Embeddable API	Да	Да
Асинхронные вызовы	Нет	Да
EJB-компоненты, управляемые сообщениями	Нет	Да
Удаленный интерфейс	Нет	Да
Веб-службы JAX-WS	Нет	Да
Веб-службы JAX-RS	Нет	Да
TimerService	Нет	Да
Интероперабельность RMI/IIOP	Нет	Да

ПРИМЕЧАНИЕ

С самого начала спецификации EJB требовали наличия возможности использовать RMI/IIOP для экспорта EJB-компонентов и доступа к EJB-компонентам через сеть. Это требование сделало возможной интероперабельность между продуктами Java EE. Однако оно исчезло с выходом EJB 3.2 и может стать необязательным в будущих релизах, поскольку RMI/IIOP в значительной мере вытесняются современными веб-технологиями, обеспечивающими поддержку интероперабельности, к числу которых относятся, например, SOAP и REST.

Обзор спецификации EJB

Версия EJB 1.0 появилась еще в 1998 году, а релиз EJB 3.2 состоялся в 2013 году с выходом Java EE 7. На протяжении этих 15 лет спецификация EJB претерпела много изменений, однако по-прежнему следует своим продуманным принципам. От тяжеловесных компонентов до аннотированных POJO-объектов, от Entity Bean CMP до JPA EJB-компоненты переосмысливались, чтобы соответствовать требованиям разработчиков и современных архитектур.

Спецификация EJB 3.2 как никогда помогает избежать зависимости от продукции того или иного поставщика, предусматривая функции, которые ранее были нестандартными (например, нестандартные JNDI-имена или встроенные контейнеры). Сегодня версия EJB 3.2 стала намного более переносимой, чем те, что выходили в прошлом.

Краткая история спецификации EJB

Вскоре после того, как был создан язык Java, индустрия ощутила необходимость в технологии, которая отвечала бы требованиям крупномасштабных приложений, задействуя технологию RMI или JTA. Возникла идея создания фреймворка для разработки распределенных и транзакционных бизнес-компонентов, а в итоге компания IBM первой приступила к созданию того, что в конечном счете стало известным под названием «EBJ-компоненты».

Версия EJB 1.0 поддерживала EJB-компоненты с сохранением состояния и без, а также предусматривала необязательную поддержку компонентов-сущностей EJB. Модель программирования задействовала домашние и удаленные интерфейсы в дополнение к сессионным EJB-компонентам как таковым. EJB-компоненты делались доступными с помощью интерфейса: он обеспечивал удаленный доступ с использованием аргументов, передаваемых по значению.

Версия EJB 1.1 сделала поддержку компонентов-сущностей EJB обязательной, а также представила XML-дескрипторы развертывания для сохранения метаданных (которые затем сериализовались как двоичные в файл). Эта версия обеспечила лучшую поддержку сборки и развертывания приложений, представив роли.

В 2001 году EJB 2.0 стала первой версией, которая была стандартизирована Java Community Process (как JSR 19). Она позволила решить проблему накладных расходов, связанных с передачей аргументов по значению, представив локальные интерфейсы. Клиент, работающий в контейнере, осуществлял бы доступ к EJB-компонентам с помощью их локального интерфейса (с использованием аргументов, передаваемых по ссылке). Эта версия представила EJB-компоненты, управляемые сообщениями, а компоненты-сущности EJB обрели поддержку связей и языка запросов (EJB QL).

Спустя два года версия EJB 2.1 (JSR 153) привнесла поддержку веб-служб, позволив вызывать сессионные EJB-компоненты с помощью SOAP/HTTP. Был создан инструмент `TimerService`, чтобы EJB-компоненты можно было вызывать в точно установленное время или через определенные промежутки времени.

Между появлением EJB 2.1 и EJB 3.0 прошло три года, что позволило экспертной группе переделать всю конструкцию. Вышедшая в 2006 году спецификация EJB 3.0 (JSR 220) порвала отношения с предыдущими версиями, поскольку была сосредоточена на легкости использования, при этом EJB-компоненты больше напоминали POJO. Компоненты-сущности EJB были заменены совершенно новой спецификацией (JPA), а сессионным EJB-компонентам больше не требовались домашние или EJB-специфичные интерфейсы компонентов. Были представлены внедрение ресурсов, перехватчики и обратные вызовы жизненного цикла.

В 2009 году спецификация EJB 3.1 (JSR 318) сопровождала Java EE 6 и шла путем предыдущих версий, даже еще больше упрощая модель программирования. Версия 3.1 привнесла удивительное количество новых функций, например представление без интерфейса, встроенные контейнеры, одиночные EJB-компоненты, `TimerService` с более богатыми возможностями, асинхронность, переносимые JNDI-имена и EJB Lite.

Что нового в EJB 3.2

Спецификация EJB 3.2 (JSR 345) является менее претенциозной, чем предыдущий релиз. Чтобы упростить будущее принятие этой спецификации, экспертная группа Java EE 6 составила список функций, которые могут быть ликвидированы в дальнейшем. Фактически ни одна из приведенных далее функций не была исключена из EJB 3.1, однако все они стали необязательными в версии 3.2:

- компоненты-сущности EJB 2.*x*;
- клиентское представление компонентов-сущностей EJB 2.*x*;
- EJB QL (язык запросов для CMP);
- конечные точки веб-служб на основе JAX-RPC;
- клиентское представление веб-служб JAX-RPC.

Вот почему сама спецификация состоит из двух разных документов:

- *«Основные контракты и требования EJB-компонентов» (EJB Core Contracts and Requirements)* — основной документ, определяющий EJB-компоненты;
- *«Необязательные функции EJB-компонентов» (EJB Optional Features)* — документ, описывающий приведенные ранее функции, поддержка которых стала необязательной.

Спецификация EJB 3.2 включает следующие небольшие обновления и улучшения.

- Транзакции теперь могут использоваться MBean-компонентами (ранее только EJB-компоненты могли применять транзакции; подробнее об этом мы поговорим в главе 9).
- Могут добавляться методы обратного вызова жизненного цикла EJB-компонентов с сохранением состояния, чтобы сделать их транзакционными.
- Пассивизации EJB-компонентов с сохранением состояния теперь больше нет.
- Были упрощены правила определения всех локальных/удаленных представлений EJB-компонентов.
- Было снято ограничение на получение текущего загрузчика классов, кроме того, теперь разрешено использовать пакет `java.io`.
- Корректировки JMS 2.0.
- Встраиваемый контейнер реализует `Autocloseable`, чтобы соответствовать Java SE 7.
- RMI/IIOP отсутствуют в этом релизе. Это означает, что поддержка этих технологий может быть помечена как необязательная в Java EE 8. Удаленные вызовы можно осуществлять с помощью всего лишь RMI (без интероперабельности IIOP).

В табл. 7.2 приведены основные пакеты, определенные в EJB 3.2 на сегодняшний день.

Таблица 7.2. Основные EJB-пакеты

Пакет	Описание
javax.ejb	Классы и интерфейсы, которые определяют контракты между EJB-компонентом и его клиентами, а также между EJB-компонентом и контейнером
javax.ejb.embeddable	Классы для Embeddable API
javax.ejb.spi	Интерфейсы, реализуемые EJB-контейнером

Эталонная реализация

GlassFish — это проект сервера приложений с открытым исходным кодом под руководством компании Oracle для платформы Java EE. Корпорация Sun запустила этот проект в 2005 году, и он стал эталонной реализацией Java EE 5 в 2006 году. Сегодня GlassFish версии 4 включает эталонную реализацию для EJB 3.2. Внутренне этот продукт строится вокруг модульности (исходя из времени выполнения Apache Felix OSGi), что обеспечивает очень короткое время запуска и использование различных контейнеров приложений (разумеется, Java EE 7, а также Ruby, PHP и т. д.).

На момент написания этой книги GlassFish представлял собой реализацию, совместимую только с EJB 3.2. Но скоро очередь дойдет и до остальных: OpenEJB, JBoss, Weblogic, Websphere...

Написание корпоративных EJB-компонентов

Сессионные EJB-компоненты инкапсулируют бизнес-логику и опираются на контейнер, который отвечает за организацию пула, многопоточность, безопасность и т. д. Какие артефакты нам нужны для того, чтобы создать такой мощный компонент? Один Java-класс и одна аннотация — вот и все. В листинге 7.1 показано, насколько просто контейнеру распознать, что класс является сессионным EJB-компонентом, и применить все корпоративные службы.

Листинг 7.1. Простой EJB-компонент без сохранения состояния

```
@Stateless
public class BookEJB {
  @PersistenceContext(unitName = "chapter07PU")
  private EntityManager em;
  public Book findBookById(Long id) {
    return em.find(Book.class, id);
  }
  public Book createBook(Book book) {
    em.persist(book);
    return book;
  }
}
```

Предыдущие версии J2EE вынуждали разработчиков создавать ряд артефактов для того, чтобы сгенерировать тот или иной сессионный EJB-компонент: локальный либо удаленный интерфейс (или и тот и другой), локальный домашний либо удаленный домашний интерфейс (или и тот и другой) и дескриптор развертывания. Java EE 5 и EJB 3.0 существенно упростили модель до такой степени, что стало достаточно лишь одного класса и одного или нескольких бизнес-интерфейсов, и вам не потребуется какая-либо XML-конфигурация. Как показано в листинге 7.1, с выходом EJB 3.1 классу даже не нужно реализовывать какой-либо интерфейс. Мы используем только одну аннотацию для того, чтобы превратить Java-класс в транзакционный и безопасный компонент — `@Stateless`. Затем, применяя менеджер сущностей (как можно было видеть в предыдущих главах), `BookEJB` создает и извлекает экземпляры `Book` из базы данных простым, но все же эффективным способом.

Анатомия EJB-компонента

В листинге 7.1 показана самая простая модель программирования для сессионных EJB-компонентов — аннотированный Java-объект без интерфейса. Однако, в зависимости от ваших нужд, сессионные EJB-компоненты могут обеспечить намного более богатую модель, позволив вам выполнять удаленные вызовы, внедрение зависимостей или асинхронные вызовы. EJB-компонент состоит из таких элементов, как:

- *класс EJB-компонента*, который содержит реализацию бизнес-методов и может реализовывать нуль или несколько бизнес-интерфейсов. Сессионный EJB-компонент должен быть снабжен аннотацией `@Stateless`, `@Stateful` или `@Singleton` в зависимости от своего типа;
- *бизнес-интерфейсы*, которые содержат объявление бизнес-методов, видимых для клиента и реализуемых классом EJB-компонента. Сессионный EJB-компонент может обладать локальными интерфейсами, удаленными интерфейсами либо не иметь вообще никакого интерфейса (представление без интерфейса только с локальным доступом).

Как показано на рис. 7.2, клиентское приложение может получить доступ к сессионному EJB-компоненту посредством одного из его интерфейсов (локального или удаленного) либо напрямую, вызвав сам класс EJB-компонента.

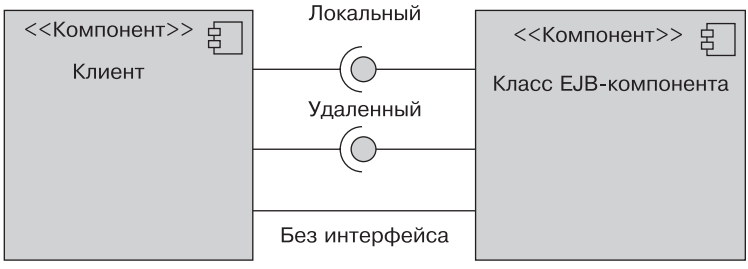


Рис. 7.2. Класс EJB-компонента обладает бизнес-интерфейсами нескольких типов

Класс EJB-компонента

Класс сессионного EJB-компонента — это любой Java-класс, который реализует бизнес-логику. Требования для разработки класса сессионного EJB-компонента таковы:

- класс должен быть снабжен аннотацией `@Stateless`, `@Stateful`, `@Singleton` или XML-эквивалентом в дескрипторе развертывания;
- он должен реализовывать методы своих интерфейсов при наличии таковых;
- класс должен быть определен как `public` и не должен быть `final` или `abstract`;
- класс должен располагать конструктором `public` без аргументов, который контейнер будет использовать для создания экземпляров;
- класс не должен определять метод `finalize()`;
- имена бизнес-методов не должны начинаться с `ejb`, при этом они не могут быть `final` или `static`;
- аргумент и возвращаемое значение удаленного метода должны относиться к допустимым типам RMI.

Удаленные и локальные представления, а также представление без интерфейса

В зависимости от того, откуда клиент станет вызывать сессионный EJB-компонент, классу EJB-компонента потребуется реализовывать удаленный или локальный интерфейсы либо не реализовывать вообще никакого интерфейса. Если ваша архитектура включает клиентов, которые находятся вне экземпляра виртуальной машины Java EJB-контейнера, то они должны использовать удаленный интерфейс. Как показано на рис. 7.3, это касается клиентов, работающих на отдельной виртуальной машине Java, в контейнере клиентского приложения (Application Client Container — ACC), или во внешнем веб-контейнере, или в EJB-контейнере. В этом случае клиентам придется вызывать методы сессионных EJB-компонентов с использованием удаленного вызова методов. Вы можете прибегнуть к локальному вызову, если EJB-компонент и клиент будут функционировать на одной и той же виртуальной машине Java. Это может быть вызов одним EJB-компонентом другого EJB-компонента или веб-компонента (сервлета, JSF), работающего в веб-контейнере на той же самой виртуальной машине Java. Кроме того, ваше приложение может использовать как удаленные, так и локальные вызовы в случае с одним и тем же сессионным EJB-компонентом.

Сессионный EJB-компонент может реализовывать несколько интерфейсов или не реализовывать ни одного. Бизнес-интерфейс — это стандартный Java-интерфейс, который не расширяет никаких EJB-специфичных интерфейсов. Как и любой Java-интерфейс, бизнес-интерфейсы определяют список методов, которые будут доступны для клиентского приложения. Для них могут использоваться следующие аннотации:

- `@Remote` — обозначает удаленный бизнес-интерфейс. Параметры методов передаются по значению и нуждаются в том, чтобы быть сериализуемыми как часть протокола RMI;

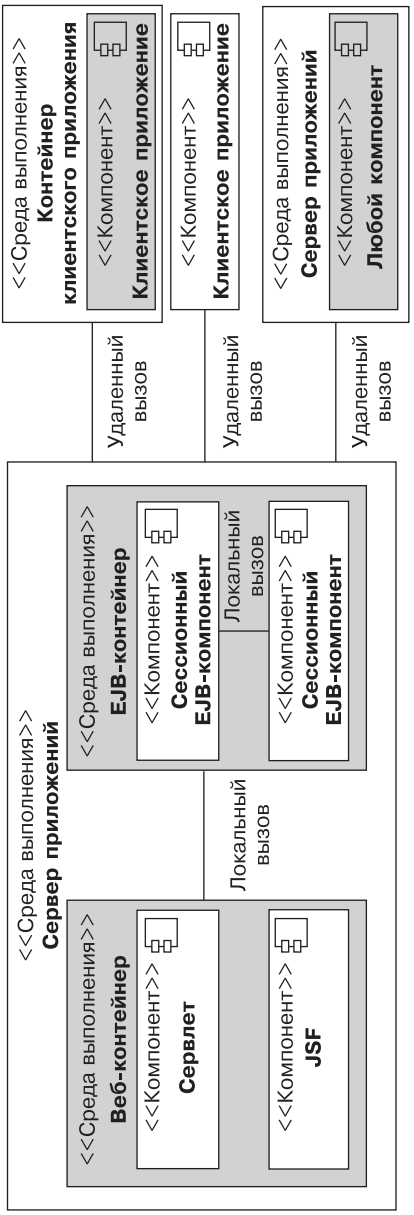


Рис. 7.3. Сессионные EJB-компоненты, вызываемые клиентами нескольких типов

- `@Local` — обозначает локальный бизнес-интерфейс. Параметры методов передаются по ссылке от клиента к EJB-компоненту.

Вы не сможете пометить один и тот же интерфейс несколькими аннотациями. Сессионные EJB-компоненты, которые вы видели до сих пор в этой главе, не имеют интерфейса. Представление без интерфейса является вариацией локального представления, которая обеспечивает все открытые бизнес-методы класса EJB-компонента локально без использования отдельного бизнес-интерфейса.

В листинге 7.2 показаны локальный (`ItemLocal`) и удаленный интерфейс (`ItemRemote`), реализуемые сессионным EJB-компонентом без сохранения состояния `ItemEJB`. Благодаря этому коду клиенты смогут вызывать метод `findCDs()` локально или удаленно, поскольку он определен в обоих интерфейсах. Метод `createCd()` будет доступен только удаленно с помощью RMI.

Листинг 7.2. Сессионный EJB-компонент без сохранения состояния, реализующий удаленный и локальный интерфейсы

```
@Local
public interface ItemLocal {
  List<Book> findBooks();
  List<CD> findCDs();
}
@Remote
  public interface ItemRemote {
  List<Book> findBooks();
  List<CD> findCDs();
  Book createBook(Book book);
  CD createCD(CD cd);
}
@Stateless
public class ItemEJB implements ItemLocal, ItemRemote {
  // ...
}
```

В качестве альтернативы коду из листинга 7.2 вы могли бы указать интерфейс в классе EJB-компонента. При этом вам пришлось бы включить имя интерфейса в аннотации `@Local` и `@Remote`, как показано в листинге 7.3. Это удобно, когда у вас имеются унаследованные интерфейсы, для которых вы не можете добавить аннотации, поэтому вынуждены использовать их в своем сессионном EJB-компоненте.

Листинг 7.3. Класс EJB-компонента, определяющий удаленный и локальный интерфейсы, а также представление без интерфейса

```
public interface ItemLocal {
  List<Book> findBooks();
  List<CD> findCDs();
}
public interface ItemRemote {
  List<Book> findBooks();
  List<CD> findCDs();
  Book createBook(Book book);
```

```
  CD createCD(CD cd);
}
@Stateless
@Remote(ItemRemote.class)
@Local(ItemLocal.class)
@LocalBean
public class ItemEJB implements ItemLocal, ItemRemote {
  // ...
}
```

Если EJB-компонент обеспечивает хотя бы один интерфейс (локальный или удаленный), то он автоматически теряет представление без интерфейса. Тогда потребуется явным образом указать, что он обеспечивает представление без интерфейса, с помощью аннотации `@LocalBean` в отношении класса EJB-компонента. Как вы можете видеть в листинге 7.3, `ItemEJB` сейчас обладает локальным и удаленным интерфейсами, а также представлением без интерфейса.

Интерфейсы веб-служб

В дополнение к удаленному вызову посредством RMI EJB-компоненты без сохранения состояния также могут вызываться удаленно как веб-службы SOAP или RESTful. Главы 14 и 15 посвящены веб-службам, поэтому я не стану описывать их здесь. Я лишь хочу показать вам, как может осуществляться доступ к сессионному EJB-компоненту без сохранения состояния в различных формах благодаря простой реализации разных аннотированных интерфейсов. В листинге 7.4 приведен EJB-компонент без сохранения состояния с локальным интерфейсом, а также конечные точки веб-служб SOAP (`@WebService`) и RESTful (`@Path`). Следует отметить, что эти аннотации происходят из JAX-WS (см. главу 14) и JAX-RS (см. главу 15) соответственно и не являются частью EJB.

Листинг 7.4. EJB-компонент без сохранения состояния, реализующий несколько интерфейсов

```
@Local
public interface ItemLocal {
  List<Book> findBooks();
  List<CD> findCDs();
}
@WebService
public interface ItemSOAP {
  List<Book> findBooks();
  List<CD> findCDs();
  Book createBook(Book book);
  CD createCD(CD cd);
}
@Path(/items)
public interface ItemRest {
    List<Book> findBooks();
}
@Stateless
```

```
public class ItemEJB implements ItemLocal, ItemSOAP, ItemRest {
  // ...
}
```

Переносимое JNDI-имя

JNDI существует уже долгое время. Соответствующий API-интерфейс определяется для серверов приложений и является переносимым между ними. Однако этого нельзя было сказать о JNDI-имени, которое зависело от реализации. При развертывании EJB-компонента в GlassFish или JBoss его имя в службе каталогов оказывалось другим и, таким образом, было непереносимым. Клиенту пришлось бы искать EJB-компонент, используя одно имя для GlassFish и другое имя — для JBoss. Начиная с EJB 3.1, JNDI-имена уже были определены, благодаря чему код мог быть переносимым. Таким образом, теперь каждый раз при развертывании сессионного EJB-компонента с его интерфейсами в контейнере каждый EJB-компонент/интерфейс автоматически привязывается к переносимому JNDI-имени. Спецификация Java EE определяет переносимые JNDI-имена с использованием следующего синтаксиса:

```
java:<область видимости>[/<имя приложения>]/<имя модуля>/<имя EJB-компо-
нента>[!<полностью уточненное имя интерфейса>]
```

У каждого фрагмента JNDI-имени есть следующее значение:

- `<область видимости>` — определяет последовательность пространств имен, которые отображаются в разные области видимости приложения Java EE;
- `global` — префикс `java:global` позволяет компоненту, выполняющемуся вне приложения Java EE, получить доступ к глобальному пространству имен;
- `app` — префикс `java:app` позволяет компоненту, выполняющемуся в рамках приложения Java EE, получить доступ к пространству имен, специфичному для приложения;
- `module` — префикс `java:module` позволяет компоненту, выполняющемуся в рамках приложения Java EE, получить доступ к пространству имен, специфичному для модуля;
- `comp` — префикс `java:comp` — это закрытое пространство имен, специфичное для компонента и недоступное для других компонентов;
- `<имя приложения>` — требуется, только если сессионный EJB-компонент упакован в файл с расширением EAR или WAR. Если дело обстоит именно так, то в `<имя приложения>` автоматически будет указано имя файла EAR или WAR (без указания расширения);
- `<имя модуля>` — это имя модуля, в который упакован сессионный EJB-компонент. Это может быть EJB-модуль в отдельном файле с расширением JAR или веб-модуль в файле с расширением WAR. В `<имя модуля>` по умолчанию указывается базовое имя архива без расширения файла;
- `<имя EJB-компонента>` — имя сессионного EJB-компонента;
- `<полностью уточненное имя интерфейса>` — это полностью уточненное имя каждого определенного бизнес-интерфейса. В случае с представлением без интерфейса именем может быть полностью уточненное имя класса EJB-компонента.

Чтобы проиллюстрировать это соглашение об именовании, обратимся к примеру ItemEJB (определенному в листинге 7.5), который располагает удаленным интерфейсом, локальным интерфейсом и представлением без интерфейса (с помощью аннотации @LocalBean). Все эти классы и интерфейсы относятся к пакету org.agoncal.book.javaee7. ItemEJB — это <имя EJB-компонента>, а соответствующий EJB-компонент упакован в cdbookstore.jar (<имя модуля>).

Листинг 7.5. Сессионный EJB-компонент без сохранения состояния, реализующий несколько интерфейсов

```
package org.agoncal.book.javaee7;
@Stateless
@Remote(ItemRemote.class)
@Local(ItemLocal.class)
@LocalBean
public class ItemEJB implements ItemLocal, ItemRemote {
  // ...
}
```

После развертывания контейнер сгенерирует три JNDI-имени, чтобы внешний компонент смог получить доступ к ItemEJB, используя следующие глобальные JNDI-имена:

```
java:global/cdbookstore/ItemEJB!org.agoncal.book.javaee7.ItemRemote
java:global/cdbookstore/ItemEJB!org.agoncal.book.javaee7.ItemLocal
java:global/cdbookstore/ItemEJB!org.agoncal.book.javaee7.ItemEJB
```

Следует отметить, что если бы ItemEJB был развернут в файле с расширением EAR (например, myapplication.ear), то вам пришлось бы указать в <имя приложения> следующее:

```
java:global/myapplication/cdbookstore/ItemEJB!org.agoncal.book.javaee7.ItemRemote
java:global/myapplication/cdbookstore/ItemEJB!org.agoncal.book.javaee7.ItemLocal
java:global/myapplication/cdbookstore/ItemEJB!org.agoncal.book.javaee7.ItemEJB
```

Контейнер также требуется для того, чтобы сделать JNDI-имена доступными при использовании пространств имен java:app и java:module. Таким образом, компонент, развернутый в том же приложении, что и ItemEJB, сможет осуществлять его поиск с использованием следующих JNDI-имен:

```
java:app/cdbookstore/ItemEJB!org.agoncal.book.javaee7.ItemRemote
java:app/cdbookstore/ItemEJB!org.agoncal.book.javaee7.ItemLocal
java:app/cdbookstore/ItemEJB!org.agoncal.book.javaee7.ItemEJB
java:module/ItemEJB!org.agoncal.book.javaee7.ItemRemote
java:module/ItemEJB!org.agoncal.book.javaee7.ItemLocal
java:module/ItemEJB!org.agoncal.book.javaee7.ItemEJB
```

Такое переносимое JNDI-имя применимо ко всем сессионным EJB-компонентам: без сохранения состояния, с сохранением состояния и одиночным.

EJB-компоненты без сохранения состояния

EJB-компоненты без сохранения состояния — это самые популярные среди разработчиков приложений Java EE сессионные EJB-компоненты. Они простые,

мощные, эффективные. Кроме того, они позволяют решать общую задачу, которая заключается в обработке бизнес-данных без сохранения состояния. Что означает словосочетание «без сохранения состояния»? Оно означает, что задача должна быть выполнена одним вызовом метода.

В качестве примера мы можем вернуться к корням объектно-ориентированного программирования, где объект инкапсулирует свое состояние и поведение. Чтобы обеспечить постоянство Book в базе данных при объектном моделировании, вы поступили бы примерно так: создали экземпляр объекта Book (с использованием ключевого слова new), задали кое-какие значения и вызвали метод, чтобы этот объект смог обеспечить свое постоянство в базе данных (book.persistToDatabase()). В приведенном далее коде вы можете видеть, что с самой первой и до последней строки объект Book вызывается несколько раз и поддерживает свое состояние:

```
Book book = new Book();
book.setTitle("Автостопом по Галактике");
book.setPrice(12.5F);
book.setDescription("Научно-фантастический комедийный сериал, созданный
Дугласом Адамсом");
book.setIsbn("1-84023-742-2");
book.setNbOfPage(354);
book.persistToDatabase();
```

В архитектуре служб вы делегировали бы бизнес-логику внешней службе. Службы без сохранения состояния идеально подходят, когда вам необходимо выполнить задачу, с которой можно справиться одним вызовом метода (передав при этом все необходимые параметры). Службы без сохранения состояния независимы, обособленны и не требуют информации или состояния от одного запроса к другому. Таким образом, если вы возьмете предыдущий код и добавите службу без сохранения состояния, то вам потребуется создать объект Book, задать кое-какие значения, а затем воспользоваться службой без сохранения состояния для вызова метода, который обеспечит постоянство Book от его имени одним вызовом. Состояние будет поддерживать Book, а не служба без сохранения состояния:

```
Book book = new Book();
book.setTitle("Автостопом по Галактике");
book.setPrice(12.5F);
book.setDescription("Научно-фантастический комедийный сериал, созданный
Дугласом Адамсом.");
book.setIsbn("1-84023-742-2");
book.setNbOfPage(354);
statelessService.persistToDatabase(book);
```

Сессионные EJB-компоненты без сохранения состояния придерживаются архитектуры служб и являются самой эффективной компонентной моделью, поскольку могут быть помещены в пул и совместно использоваться несколькими клиентами. Это означает, что для каждого EJB-компонента без сохранения состояния контейнер оставляет определенное количество экземпляров в памяти (то есть в пуле) и по-

зволяет клиентам совместно использовать их. Поскольку EJB-компоненты без сохранения состояния не обладают клиентским состоянием, все экземпляры являются одинаковыми. Когда клиент вызывает метод в EJB-компоненте без сохранения состояния, контейнер берет экземпляр из пула и присваивает его клиенту. По завершении выполнения клиентского запроса экземпляр возвращается в пул для повторного использования. Это означает, что вам потребуется лишь небольшое количество EJB-компонентов для обслуживания нескольких клиентов, как показано на рис. 7.4. Контейнер не гарантирует одного и того же экземпляра для одного и того же клиента.

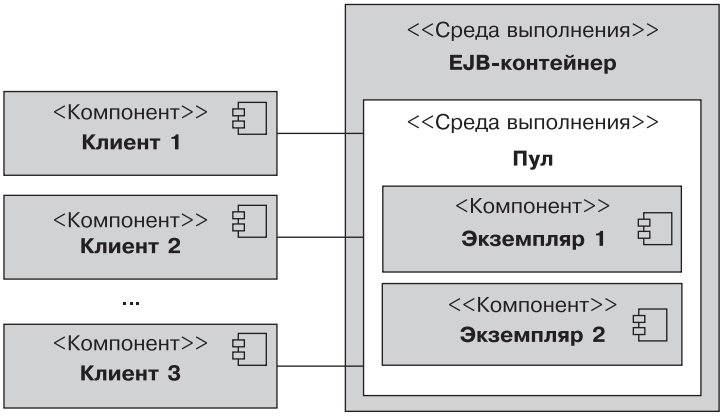


Рис. 7.4. Клиенты, осуществляющие доступ к EJB-компонентам без сохранения состояния в пуле

В листинге 7.6 показано, как выглядит EJB-компонент без сохранения состояния: стандартный Java-класс со всего одной аннотацией `@Stateless`. Поскольку он располагается в контейнере, у него есть возможность использовать любые службы, управляемые контейнером, одна из которых позволяет внедрять зависимости. Мы применяем аннотацию `@PersistenceContext` для внедрения ссылки на менеджер сущностей. В случае с сессионными EJB-компонентами без сохранения состояния контекст постоянства является транзакционным, а это означает, что любой метод, вызываемый в этом EJB-компоненте (`createBook()`, `createCD()` и т. д.), будет транзакционным. Этот процесс более подробно объясняется в главе 9. Следует отметить, что у всех методов есть необходимые параметры для обработки бизнес-логики одним вызовом. Например, метод `createBook()` принимает объект `Book` в качестве параметра и обеспечивает его постоянство без расчета на другое состояние.

Листинг 7.6. Сессионный EJB-компонент без сохранения состояния ItemEJB

```
@Stateless
public class ItemEJB {
  @PersistenceContext(unitName = "chapter07PU")
  private EntityManager em;
  public List<Book> findBooks() {
```

```
    TypedQuery<Book> query = em.createNamedQuery(Book.FIND_ALL, Book.class);
    return query.getResultList();
  }
  public List<CD> findCDs() {
    TypedQuery<CD> query = em.createNamedQuery(CD.FIND_ALL, CD.class);
    return query.getResultList();
  }
  public Book createBook(Book book) {
    em.persist(book);
    return book;
  }
  public CD createCD(CD cd) {
    em.persist(cd);
    return cd;
  }
}
```

EJB-компоненты без сохранения состояния зачастую содержат несколько тесно связанных бизнес-методов. Например, `ItemEJB` из листинга 7.5 определяет методы, связанные с элементами, продажа которых осуществляется в приложении CD-BookStore. Таким образом, вы найдете методы для создания, обновления или поиска экземпляров `Book` и `CD`, а также другой связанной бизнес-логики.

Аннотация `@Stateless` помечает Java-объект `ItemEJB` как EJB-компонент без сохранения состояния, тем самым превращая простой Java-класс в компонент, поддерживающий контейнер. В листинге 7.7 описывается спецификация аннотации `@javax.ejb.Stateless`.

Листинг 7.7. API-интерфейс аннотации @Stateless

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface Stateless {
    String name() default "";
    String mappedName() default "";
    String description() default "";
}
```

Параметр `name` определяет имя EJB-компонента и по умолчанию имеет значение в виде имени класса (`ItemEJB` в листинге 7.6). Этот параметр можно задействовать, например, для поиска EJB-компонента с применением JNDI. Параметр `description` — это строка, которая может быть использована для описания EJB-компонента. Атрибут `mappedName` определяет глобальное JNDI-имя, присваиваемое контейнером. Следует отметить, что это JNDI-имя зависит от поставщика и, следовательно, не является переносимым. `mappedName` не имеет связи с переносимым *глобальным* JNDI-именем, которое я описывал ранее.

Сессионные EJB-компоненты без сохранения состояния могут поддерживать большое количество клиентов, минимизируя объем любых необходимых ресурсов. Наличие приложений без сохранения состояния — один из способов улучшить масштабируемость (поскольку контейнеру не придется сохранять состояние и управлять им).

EJB-компоненты с сохранением состояния

EJB-компоненты без сохранения состояния обеспечивают бизнес-методы для своих клиентов, но не поддерживают для них диалоговое состояние. EJB-компоненты с сохранением состояния, наоборот, поддерживают диалоговое состояние. Они полезны для решения задач, с которыми необходимо справиться в несколько этапов. При этом каждый из этапов полагается на состояние, сохраненное на предыдущем этапе. Обратимся к примеру корзины на сайте для электронной торговли. Клиент входит в систему (его сессия начинается), выбирает первую книгу, добавляет ее в свою корзину, затем выбирает вторую книгу и добавляет ее в свою корзину. В конце клиент подсчитывает стоимость всех выбранных книг, оплачивает их и выходит из системы (сессия завершается). Корзина поддерживает состояние того, сколько книг клиент выбрал по ходу всего взаимодействия (что может занять некоторое время, в частности время сессии клиента). Код этого взаимодействия с компонентом с сохранением состояния мог бы выглядеть следующим образом:

```
Book book = new Book();
book.setTitle("Автостопом по Галактике");
book.setPrice(12.5F);
book.setDescription("Научно-фантастический комедийный сериал, созданный
Дугласом Адамсом.");
book.setIsbn("1-84023-742-2");
book.setNbOfPage(354);
statefulComponent.addBookToShoppingCart(book);
book.setTitle("Роботы зари");
book.setPrice(18.25F);
book.setDescription("Айзек Азимов, серия про роботов");
book.setIsbn("0-553-29949-2");
book.setNbOfPage(276);
statefulComponent.addBookToShoppingCart(book);
statefulComponent.checkOutShoppingCart();
```

В приведенном чуть выше коде показано, как именно работает сессионный EJB-компонент с сохранением состояния. Создаются два экземпляра `Book`, а затем они добавляются в корзину, которую представляет компонент с сохранением состояния. В конце метод `checkOutShoppingCart()` опирается на сохраненное состояние и может подсчитать стоимость двух выбранных книг.

Когда клиент вызовет EJB-компонент с сохранением состояния на сервере, EJB-контейнеру потребуется обеспечить тот же самый экземпляр при каждом последующем вызове метода. EJB-компоненты с сохранением состояния не могут повторно использоваться другими клиентами. На рис. 7.5 показана корреляция «один к одному» между экземпляром EJB-компонента и клиентом. С точки зрения разработчика, никакого дополнительного кода здесь не требуется, поскольку EJB-контейнер автоматически управляет этой корреляцией «один к одному».

За корреляцию «один к одному» придется кое-чем заплатить, поскольку, как вы, возможно, уже догадались, если у вас будет один миллион клиентов, то в вашем случае в памяти окажется один миллион EJB-компонентов с сохранением состояния. Чтобы избежать такого большого объема занимаемой памяти, контейнер временно удаляет EJB-компоненты из памяти до того, как следующий запрос от клиента

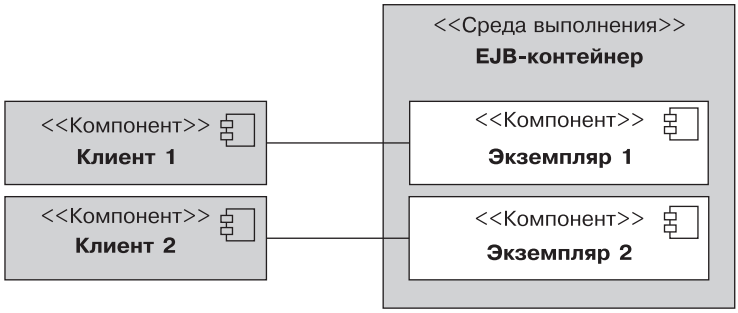


Рис. 7.5. Клиенты, осуществляющие доступ к EJB-компонентам с сохранением состояния

вернет их назад. Эта методика называется *пассивизацией* и *активизацией*. Пассивизация — это процесс удаления экземпляра из памяти и сохранения его в постоянной локации (в файле на диске, в базе данных и т. д.). Активизация — это обратный процесс, который заключается в восстановлении состояния и применении его к экземпляру. Пассивизация и активизация выполняются автоматически контейнером; вам не нужно беспокоиться о том, чтобы осуществить их самостоятельно, поскольку об этом заботится контейнерная служба. Вам следует побеспокоиться о высвобождении всех ресурсов (используемых, например, для подключения к базе данных; либо это могут быть фабрики JMS, тоже обеспечивающие подключение, и т. д.), прежде чем EJB-компонент подвергнется пассивизации. С выходом EJB 3.2 также появилась возможность отключать пассивизацию, как вы увидите в следующей главе, с использованием аннотаций жизненного цикла и обратных вызовов.

Вернемся к примеру корзины и применим его для EJB-компонента с сохранением состояния (листинг 7.8). Клиент входит в систему на сайте, просматривает каталог элементов и добавляет две книги в корзину (с помощью метода `addItem()`). Атрибут `cartItems` включает все содержимое корзины. Затем клиент решает сделать себе кофе. Пока он занят этим, контейнер может осуществить пассивизацию экземпляра, чтобы высвободить некоторый объем памяти, что, в свою очередь, приводит к сохранению содержимого корзины на постоянном запоминающем устройстве. Спустя несколько минут клиент возвращается и хочет узнать общую цену (с помощью метода `getTotal()`) товаров в своей корзине, прежде чем что-либо покупать. Контейнер активизирует EJB-компонент и восстанавливает данные в корзину. После этого клиент может подсчитать стоимость всех выбранных книг (с помощью метода `checkout()`) и купить их. Как только клиент выходит из системы, его сессия завершается и контейнер высвобождает память, навсегда удаляя экземпляр EJB-компонента с сохранением состояния.

Листинг 7.8. Сессионный EJB-компонент с сохранением состояния ShoppingCartEJB

```
@Stateful
@StatefulTimeout(value = 20, unit = TimeUnit.SECONDS)
public class ShoppingCartEJB {
  private List<Item> cartItems = new ArrayList<>();
  public void addItem(Item item) {
```

```
    if (!cartItems.contains(item))
      cartItems.add(item);
  }
  public void removeItem(Item item) {
    if (cartItems.contains(item))
      cartItems.remove(item);
  }
  public Integer getNumberOfItems() {
    if (cartItems == null || cartItems.isEmpty())
      return 0;
    return cartItems.size();
  }
  public Float getTotal() {
    if (cartItems == null || cartItems.isEmpty())
      return 0f;
    Float total = 0f;
    for (Item cartItem : cartItems) {
      total += (cartItem.getPrice());
    }
    return total;
  }
  public void empty() {
    cartItems.clear();
  }
  @Remove
  public void checkout() {
    // Выполнить некоторую бизнес-логику
    cartItems.clear();
  }
}
```

Рассмотренная нами ситуация с корзиной — это стандартный подход к использованию EJB-компонентов с сохранением состояния, при котором контейнер автоматически обеспечивает поддержание диалогового состояния. Единственная необходимая аннотация — `@javax.ejb.Stateful`, которая обладает тем же API-интерфейсом, что и аннотация `@Stateless`, описанная в листинге 7.7.

Обратите внимание на опциональные аннотации `@javax.ejb.StatefulTimeout` и `@javax.ejb.Remove`. Аннотацией `@Remove` снабжен метод `checkout()`. Это приводит к тому, что экземпляр EJB-компонента навсегда удаляется из памяти после вызова метода `checkout()`. Аннотация `@StatefulTimeout` присваивает значение времени ожидания, в течение которого EJB-компоненту разрешено оставаться незадействованным (не принимающим никаких клиентских вызовов), прежде чем он будет удален контейнером. Единицей времени в случае с этой аннотацией является `java.util.concurrent.TimeUnit`, поэтому значение может быть начиная с `DAYS`, `HOURS`... до `MILLISECONDS` (по умолчанию — `MINUTES`). В качестве альтернативы вы можете обойтись без этих аннотаций и положиться на контейнер, который автоматически удалит экземпляр, когда сессия клиента завершится или ее время истечет. Однако обеспечение удаления экземпляра в соответствующий момент способно уменьшить потребление памяти. Это может быть критически важным для приложений с высокой степенью конкуренции.

Одиночные EJB-компоненты

Одиночный EJB-компонент — это сессионный EJB-компонент, экземпляр которого создается по одному на приложение. Он реализует широко используемый шаблон Singleton («Одиночка») из знаменитой книги «Банды четырех» под названием «Приемы объектно-ориентированного проектирования. Паттерны проектирования» *(Design Patterns: Elements of Reusable Object-Oriented Software)*, авторами которой выступили Эрих Гамма, Ричард Хелм, Ральф Джонсон и Джон М. Влиссидес (Addison-Wesley, 1995). Использование одиночного EJB-компонента гарантирует, что во всем приложении будет только один экземпляр класса, и обеспечивает глобальную точку для доступа к нему. Бывает много ситуаций, в которых требуются одиночные объекты, то есть когда вашему приложению нужен только один экземпляр объекта: это может быть мышь, оконный менеджер, спулер принтера, файловая система и т. д.

Другой распространенный сценарий применения — система кэширования, при которой все приложение совместно использует один кэш (например, `Hashmap`) для размещения объектов. В среде, управляемой приложением, вам потребуется немного изменить свой код, чтобы превратить класс в одиночный EJB-компонент, как показано в листинге 7.9. Прежде всего вам понадобится предотвратить создание нового экземпляра с помощью закрытого конструктора. Открытый статический метод `getInstance()` возвращает один экземпляр класса `CacheSingleton`. Если клиентскому классу понадобится добавить объект в кэш при использовании одиночного EJB-компонента, то ему потребуется вызвать:

```
CacheSingleton.getInstance().addToCache(myObject);
```

Если вы хотите, чтобы ваш код был потокобезопасным, то вам придется воспользоваться ключевым словом `synchronized` для предотвращения интерференции потоков и появления несогласованных данных. Вместо `Map` вы также можете задействовать `java.util.concurrent.ConcurrentMap`, что приведет к намного более конкурентному и масштабируемому поведению. Такой подход может оказаться полезным, если эти особенности будут критически важными.

Листинг 7.9. Java-класс, следующий шаблону проектирования Singleton («Одиночка»)

```
public class Cache {
  private static Cache instance = new Cache();
  private Map<Long, Object> cache = new HashMap<>();
  private Cache() {}
  public static synchronized Cache getInstance() {
    return instance;
  }
  public void addToCache(Long id, Object object) {
    if (!cache.containsKey(id))
      cache.put(id, object);
}
  public void removeFromCache(Long id) {
    if (cache.containsKey(id))
      cache.remove(id);
  }
  public Object getFromCache(Long id) {
```

```
    if (cache.containsKey(id))
      return cache.get(id);
    else
      return null;
  }
}
```

В EJB 3.1 был представлен одиночный сессионный EJB-компонент, который следует шаблону проектирования Singleton («Одиночка»). После создания экземпляра контейнер убеждается в том, что в течение времени выполнения приложения будет присутствовать единственный экземпляр одиночного EJB-компонента. Экземпляр совместно используется несколькими клиентами, как показано на рис. 7.6. Одиночные EJB-компоненты поддерживают свое состояние между клиентскими вызовами.

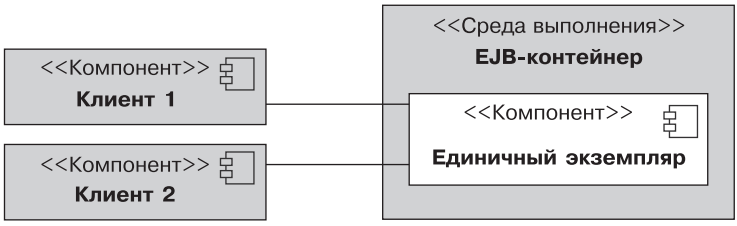


Рис. 7.6. Клиенты, осуществляющие доступ к одиночному EJB-компоненту

ПРИМЕЧАНИЕ

Одиночные EJB-компоненты не поддерживают кластер. Кластер — это группа контейнеров, которые тесно работают друг с другом (совместно используют одни и те же ресурсы, EJB-компоненты и т. д.). Таким образом, в ситуациях, когда несколько распределенных контейнеров будут образовывать кластер, функционируя на нескольких машинах, каждый контейнер будет располагать своим экземпляром одиночного EJB-компонента.

Чтобы превратить приведенный в листинге 7.9 код одиночного Java-класса в код одиночного сессионного EJB-компонента (листинг 7.10), не потребуется много усилий. Фактически вам придется лишь снабдить класс аннотацией `@Singleton`, при этом не придется беспокоиться о закрытом конструкторе или статическом методе `getInstance()`. Контейнер убедится в том, что вы создали только один экземпляр. Аннотация `@javax.ejb.Singleton` обладает тем же самым API-интерфейсом, что и аннотация `@Stateless`, описанная ранее в листинге 7.7.

Листинг 7.10. Одиночный сессионный EJB-компонент

```
@Singleton
public class CacheEJB {
  private Map<Long, Object> cache = new HashMap<>();
  public void addToCache(Long id, Object object) {
    if (!cache.containsKey(id))
      cache.put(id, object);
  }
```

```
    public void removeFromCache(Long id) {
      if (cache.containsKey(id))
        cache.remove(id);
    }
    public Object getFromCache(Long id) {
      if (cache.containsKey(id))
        return cache.get(id);
      else
        return null;
    }
}
```

Как вы можете видеть, разработка сессионных EJB-компонентов без сохранения состояния, с сохранением состояния и одиночных очень легка: вам потребуется лишь одна аннотация. Вместе с тем об одиночных EJB-компонентах можно сказать еще немного. Можно инициализировать их при запуске, объединять в цепочку и настраивать их конкурентный доступ.

Инициализация при запуске

Когда клиентскому классу требуется доступ к методу в одиночном сессионном EJB-компоненте, контейнер обеспечивает либо создание его экземпляра, либо использование того, что уже имеется в этом контейнере. Однако инициализация одиночного EJB-компонента иногда может отнимать много времени. Представим, что `CacheEJB` (приведенному в листинге 7.10) необходим доступ к базе данных для загрузки в свой кэш тысяч объектов. Первый вызов EJB-компонента окажется затратным, и первому клиенту придется ожидать завершения инициализации.

Чтобы избежать такой задержки, вы можете дать указание контейнеру инициализировать одиночный EJB-компонент при запуске. Если снабдить класс EJB-компонента аннотацией `@Startup`, то контейнер инициализирует его во время запуска приложения, а не в тот момент, когда клиент вызовет его. В приведенном далее коде показано, как следует использовать эту аннотацию:

```
@Singleton
@Startup
public class CacheEJB {...}
```

ПРИМЕЧАНИЕ

В Java EE 7 экспертная группа попыталась изъять аннотацию @Startup из спецификации EJB, чтобы она могла быть использована с любым управляемым MBean-компонентом или сервлетом. Сделать это не получилось, однако теоретически это окажется возможным в Java EE 8.

Объединение одиночных EJB-компонентов в цепочку

В некоторых случаях, когда у вас есть несколько одиночных EJB-компонентов, может быть важно явно задать порядок инициализации. Представим, что `CacheEJB` необходимо сохранить данные, которые исходят от другого одиночного EJB-компонента

(скажем, CountryCodeEJB, который возвращает ISO-коды стран). Тогда CountryCodeEJB потребуется инициализировать раньше CacheEJB. Между несколькими одиночными EJB-компонентами могут существовать зависимости, которые выражаются аннотацией @javax.ejb.DependsOn. Использование этой аннотации демонстрируется в приведенном далее примере:

```
@Singleton
public class CountryCodeEJB {...}

@DependsOn("CountryCodeEJB")
@Singleton
public class CacheEJB {...}
```

@DependsOn содержит одну или несколько строк, каждая из которых определяет имя целевого одиночного EJB-компонента. В приведенном далее коде показано, как CacheEJB зависит от инициализации CountryCodeEJB и ZipCodeEJB. @DependsOn("CountryCodeEJB", "ZipCodeEJB") дает указание контейнеру позаботиться о том, чтобы CountryCodeEJB и ZipCodeEJB были инициализированы раньше CacheEJB.

```
@Singleton
public class CountryCodeEJB {...}

@Singleton
public class ZipCodeEJB {...}
@DependsOn("CountryCodeEJB", "ZipCodeEJB")
@Startup
@Singleton
public class CacheEJB {...}
```

Как видно в этом коде, вы даже можете комбинировать зависимости, когда речь идет об инициализации при запуске. CacheEJB быстро инициализируется при запуске (поскольку снабжен аннотацией @Startup), и, следовательно, CountryCodeEJB и ZipCodeEJB тоже будут инициализированы при запуске, но раньше CacheEJB.

Вы также можете использовать полностью уточненные имена для ссылки на одиночный EJB-объект, упакованный в другой модуль в рамках одного и того же приложения. Допустим, оба CacheEJB и CountryCodeEJB упакованы в рамках одного приложения (в один и тот же файл с расширением .ear), но в разные файлы с расширением .jar (technical.jar и business.jar соответственно). В приведенном далее коде показано, как CacheEJB зависел бы от CountryCodeEJB:

```
@DependsOn("business.jar#CountryCodeEJB")
@Singleton
public class CacheEJB {...}
```

Следует отметить, что ссылка такого рода влечет зависимость кода от деталей упаковки (которыми в данном случае являются имена файлов модулей).

Конкурентный доступ

Как вы уже понимаете, имеется единственный экземпляр одиночного сессионного EJB-компонента, который совместно используется множественными клиентами.

Таким образом, конкурентный доступ для клиентов разрешается, при этом им можно управлять с помощью аннотации `@ConcurrencyManagement` на основе двух разных подходов.

- *Конкурентный доступ, управляемый контейнером (CMC),* — контейнер управляет конкурентным доступом к экземпляру EJB-компонента, исходя из метаданных (аннотации или XML-эквивалента).
- *Конкурентный доступ, управляемый EJB-компонентом (BMC),* — контейнер разрешает полный конкурентный доступ и возлагает ответственность за синхронизацию на EJB-компонент.

Если не указать, на основе какого подхода должно осуществляться управление доступом, то по умолчанию будет использоваться конкурентный доступ, управляемый контейнером. В случае с одиночным EJB-компонентом может предусматриваться использование либо одного, либо другого вида доступа, но не обоих сразу. Как вы увидите в последующих разделах, аннотацию `@AccessTimeout` можно применять для запрета конкурентного доступа (иначе говоря, если клиент вызовет бизнес-метод, который уже используется другим клиентом, то конкурентный вызов приведет к генерированию исключения `ConcurrentAccessException`).

Конкурентный доступ, управляемый контейнером

При таком доступе (он является подходом по умолчанию) контейнер отвечает за управление конкурентным доступом к экземпляру одиночного EJB-компонента. Тогда вы сможете использовать аннотацию `@Lock` для указания того, как контейнер должен управлять доступом при вызове метода клиентом. Эта аннотация может принимать значение `READ` (общая блокировка) или `WRITE` (эксклюзивная блокировка).

- `@Lock(LockType.WRITE)` — метод, ассоциированный с эксклюзивной блокировкой, не позволит выполнять конкурентные вызовы до тех пор, пока не завершится обработка, осуществляемая этим методом. Например, если клиент C1 вызовет метод с эксклюзивной блокировкой, то клиент C2 не сможет вызвать этот метод, пока клиент C1 не закончит.
- `@Lock(LockType.READ)` — метод, ассоциированный с общей блокировкой, позволит выполнять любое количество других конкурентных вызовов для экземпляра EJB-компонента. Например, два клиента — C1 и C2 — смогут одновременно получить доступ к методу с общей блокировкой.

Аннотацией `@Lock` можно снабдить классы, методы либо и те и другие сразу. Если снабдить ею класс, то это будет означать, что она распространяется на все методы. Если вы не укажете атрибут блокировки конкурентного доступа, то по умолчанию будет предполагаться `@Lock(WRITE)`. В коде, приведенном в листинге 7.11, показан `CacheEJB` с блокировкой `WRITE` в классе EJB-компонента. Это подразумевает, что для всех методов будет иметь место конкурентный доступ `WRITE` за исключением `getFromCache()`, по отношению к которому осуществляется переопределение с использованием `READ`.

Листинг 7.11. Одиночный сессионный EJB-компонент с конкурентным доступом, управляемым контейнером

```
@Singleton
@Lock(LockType.WRITE)
@AccessTimeout(value = 20, unit = TimeUnit.SECONDS)
public class CacheEJB {
  private Map<Long, Object> cache = new HashMap<>();
  public void addToCache(Long id, Object object) {
    if (!cache.containsKey(id))
      cache.put(id, object);
  }
  public void removeFromCache(Long id) {
    if (cache.containsKey(id))
      cache.remove(id);
  }
  @Lock(LockType.READ)
  public Object getFromCache(Long id) {
    if (cache.containsKey(id))
      return cache.get(id);
    else
      return null;
  }
}
```

В листинге 7.11 класс снабжен аннотацией `@AccessTimeout`. При блокировке конкурентного доступа можно указать время ожидания для отклонения запроса, если блокировка не будет применена в течение определенного времени. Если вызов `addToCache()` станет блокироваться более 20 секунд, то для клиента будет сгенерировано исключение `ConcurrentAccessTimeoutException`.

Конкурентный доступ, управляемый EJB-компонентом

При использовании такого подхода, как конкурентный доступ, управляемый EJB-компонентом, контейнер разрешает полный конкурентный доступ к экземпляру одиночного EJB-компонента. Тогда вы будете отвечать за защиту его состояния от ошибок синхронизации вследствие конкурентного доступа. В этом случае вам будет разрешено использовать Java-примитивы синхронизации, например `synchronized` и `volatile`. В коде, приведенном в листинге 7.12, показан `CacheEJB` с конкурентным доступом, управляемым EJB-компонентом (`@ConcurrencyManagement(BEAN)`), при этом для методов `addToCache()` и `removeFromCache()` используется ключевое слово `synchronized`.

Листинг 7.12. Одиночный сессионный EJB-компонент с конкурентным доступом, управляемым EJB-компонентом

```
@Singleton
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class CacheEJB {
```

```
  private Map<Long, Object> cache = new HashMap<>();

  public synchronized void addToCache(Long id, Object object) {
    if (!cache.containsKey(id))
      cache.put(id, object);
  }
  public synchronized void removeFromCache(Long id) {
    if (cache.containsKey(id))
      cache.remove(id);
  }
  public Object getFromCache(Long id) {
    if (cache.containsKey(id))
      return cache.get(id);
    else
      return null;
  }
}
```

Время ожидания конкурентного доступа и запрет конкурентного доступа

Попытка конкурентного доступа, при которой не может быть незамедлительно применена соответствующая блокировка, будет блокироваться до тех пор, пока не получится продвинуться вперед. Аннотация `@AccessTimeout` используется для определения периода, на протяжении которого должна блокироваться попытка доступа, прежде чем истечет время ожидания. Значение `@AccessTimeout`, равное `-1`, показывает, что клиентский запрос будет блокироваться неопределенно долго до тех пор, пока не получится продвинуться вперед. Значение `@AccessTimeout`, равное `0`, говорит о том, что конкурентный доступ запрещен. В результате будет сгенерировано исключение `ConcurrentAccessException`, если клиент вызовет метод, используемый в текущий момент. Это может сказаться на производительности, поскольку клиентам, возможно, придется обработать исключение, снова попытаться получить доступ к EJB-компоненту, после чего вероятны генерирование еще одного исключения, новая попытка и т. д. В листинге 7.13 `CacheEJB` запрещает конкурентный доступ к методу `addToCache()`. Это означает, что если клиент A будет добавлять объект в кэш, а клиент B захочет сделать то же самое одновременно с ним, то для клиента B будет сгенерировано исключение и ему придется попытаться снова позднее (либо поступить с исключением по-другому).

Листинг 7.13. Одиночный сессионный EJB-компонент, запрещающий конкурентный доступ к методу

```
@Singleton
public class CacheEJB {
  private Map<Long, Object> cache = new HashMap<>();
  @AccessTimeout(0)
  public void addToCache(Long id, Object object) {
    if (!cache.containsKey(id))
      cache.put(id, object);
  }
```

```
    public void removeFromCache(Long id) {
      if (cache.containsKey(id))
        cache.remove(id);
    }
    @Lock(LockType.READ)
    public Object getFromCache(Long id) {
      if (cache.containsKey(id))
        return cache.get(id);
      else
        return null;
    }
  }
```

Внедрение зависимостей

Я уже говорил о внедрении зависимостей ранее, и вы столкнетесь с этим механизмом несколько раз в последующих главах. Это простой, но все же мощный механизм, используемый Java EE 7 для внедрения ссылок на ресурсы в атрибуты. Вместо того чтобы приложению искать ресурсы с использованием JNDI, контейнер сам внедряет их. Внедрение осуществляется во время развертывания. Если есть вероятность того, что данные не будут использоваться, EJB-компонент может избежать затрат, связанных с внедрением ресурсов, выполнив JNDI-поиск. JNDI является альтернативой внедрению. При использовании JNDI код выталкивает данные из стека, только если они необходимы, вместо того чтобы принимать подвергнутые проталкиванию в стек данные, которые могут вообще не потребоваться.

Контейнеры могут внедрять ресурсы различных типов в сессионные EJB-компоненты с помощью разных аннотаций (или дескрипторов развертывания):

- `@EJB` — внедряет ссылку на локальное представление, удаленное представление и представление без интерфейса EJB-компонента в аннотированную переменную;
- `@PersistenceContext` и `@PersistenceUnit` — выражают зависимость от `EntityManager` и `EntityManagerFactory` соответственно (см. подраздел «Получение менеджера сущностей» раздела «Менеджер сущностей» главы 6);
- `@WebServiceRef` — внедряет ссылку на веб-службу;
- `@Resource` — внедряет ряд ресурсов, например источники данных JDBC, `SessionContext`, пользовательские транзакции, фабрики подключений и пункты назначения JMS, записи окружения, `TimerService` и т. д.;
- `@Inject` — внедряет почти все с использованием `@Inject` и `@Produces`, как было объяснено в главе 2.

В листинге 7.14 приведен фрагмент кода сессионного EJB-компонента без сохранения состояния. В нем для внедрения различных ресурсов в атрибуты используются разные аннотации. Следует отметить, что этими аннотациями можно снабдить переменные экземпляра, а также методы-сеттеры.

Листинг 7.14. EJB-компонент без сохранения состояния, для которого используется внедрение

```
@Stateless
public class ItemEJB {
  @PersistenceContext(unitName = "chapter07PU")
  private EntityManager em;
  @EJB
  private CustomerEJB customerEJB;
  @Inject
  private NumberGenerator generator;
  @WebServiceRef
  private ArtistWebService artistWebService;
  private SessionContext ctx;
  @Resource
  public void setCtx(SessionContext ctx) {
    this.ctx = ctx;
  }
  //...
}
```

API-интерфейс SessionContext

Сессионные EJB-компоненты являются бизнес-компонентами, располагающимися в контейнере. Обычно они не обращаются к контейнеру и не используют контейнерные службы напрямую (управление транзакциями, безопасность, внедрение зависимостей и т. д.). Предусматривается, что контейнер будет прозрачно взаимодействовать с этими службами от имени EJB-компонента (это называется *инверсией управления*). Однако иногда EJB-компоненту требуется явно использовать контейнерные службы в коде (например, чтобы пометить транзакцию как подлежащую откату). Это можно сделать с помощью интерфейса `javax.ejb.SessionContext`. API `SessionContext` разрешает программный доступ к контексту времени выполнения, который обеспечивается для экземпляра сессионного EJB-компонента. Он расширяет интерфейс `javax.ejb.EJBContext`. В табл. 7.3 приведено описание некоторых методов API-интерфейса `SessionContext`.

Таблица 7.3. Методы интерфейса SessionContext

Метод	Описание
etCallerPrincipal	Возвращает java.security.Principal, ассоциированный с вызовом
getRollbackOnly	Проверяет, была ли текущая транзакция помечена как подлежащая откату
getTimerService	Возвращает интерфейс javax.ejb.TimerService. Только EJB-компоненты без сохранения состояния и одиночные EJB-компоненты могут задействовать этот метод. Сессионные EJB-компоненты с сохранением состояния не могут быть синхронизированными объектами
getUserTransaction	Возвращает интерфейс javax.transaction.UserTransaction для ограничения транзакций. Только сессионные EJB-компоненты, для

Метод	Описание
	которых имеет место транзакция, управляемая EJB-компонентом (Bean-Managed Transaction — BMT), могут задействовать этот метод
isCallerInRole	Проверяет, имеется ли у вызывающего оператора определенная роль безопасности
lookup	Дает возможность сессионному EJB-компоненту осуществлять поиск его записей окружения в контексте именования JNDI
setRollbackOnly	Позволяет EJB-компоненту пометить текущую транзакцию как подлежащую откату
wasCancelCalled	Проверяет, вызвал ли клиент метод cancel() в клиентском объекте Future, который соответствует выполняющемуся в текущий момент асинхронному бизнес-методу

Как показано в листинге 7.15, сессионный EJB-компонент может получить доступ к своему контексту среды путем внедрения ссылки на `SessionContext` с использованием аннотации `@Resource`. Здесь метод `createBook` удостоверяется в том, что только администраторы могут создавать экземпляры `Book`. Он также осуществляет откат, если уровень запасов книг оказывается слишком высоким.

Листинг 7.15. EJB-компонент без сохранения состояния, осуществляющий доступ к API-интерфейсу SessionContext

```
@Stateless
public class ItemEJB {
  @PersistenceContext(unitName = "chapter07PU")
  private EntityManager em;
  @Resource
  private SessionContext context;
  public Book createBook(Book book) {
    if (!context.isCallerInRole("admin"))
      throw new SecurityException("Только администратор может создавать
                                   книги");
    em.persist(book);
    if (inventoryLevel(book) == TOO_MANY_BOOKS)
      context.setRollbackOnly();
    return book;
  }
}
```

Асинхронные вызовы

По умолчанию вызовы сессионных EJB-компонентов посредством удаленного представления, локального представления и представления без интерфейса являются асинхронными. Клиент вызывает метод, после чего он блокируется в течение периода, на протяжении которого выполняется метод, пока не завершится обработка и не будет возвращен результат, а затем клиент сможет продолжить свою работу. Однако асинхронная обработка является обычным требованием во многих приложениях, имеющих дело с задачами, на решение которых нужно длительное

время. Например, печать заказа может оказаться задачей, на решение которой уйдет очень много времени в зависимости от того, подключен ли принтер, достаточно ли бумаги, не ожидает ли масса документов в спуле принтера своего вывода на печать. Когда клиент вызывает метод для печати документа, ему нужно инициировать процесс «выстрелил и забыл», который позволит напечатать документ, а клиент в это время сможет продолжить свою обработку.

До выхода EJB 3.1 асинхронная обработка могла осуществляться только благодаря JMS и EJB-компонентам, управляемым сообщениями (см. главу 13). Вам приходилось создавать администрируемые объекты (фабрики и пункты назначения JMS), иметь дело с низкоуровневым JMS API для отправки сообщения в пункт назначения, а затем разрабатывать EJB-компонент, управляемый сообщениями, который принял и обработал бы ваше сообщение. JMS сопутствуют хорошие механизмы обеспечения надежности (постоянное хранилище сообщений, гарантии доставки, интеграция с другими системами и т. д.), однако все это может оказаться тяжеловесным при таких сценариях, когда вам требуется просто вызвать метод асинхронно.

С выходом EJB 3.1 у вас появилась возможность вызывать методы асинхронно, просто снабдив метод сессионного EJB-компонента аннотацией `@javax.ejb.Asynchronous`. В листинге 7.16 приведен `OrderEJB`, у которого имеется один метод для отправки клиенту сообщений по электронной почте и другой метод для печати заказа. Поскольку выполнение обоих методов занимает много времени, они оба снабжены аннотацией `@Asynchronous`.

Листинг 7.16. OrderEJB, для которого объявляются асинхронные методы

```
@Stateless
public class OrderEJB {
  @Asynchronous
  public void sendEmailOrderComplete(Order order, Customer customer) {
    // Задача, на решение которой требуется очень много времени
  }

  @Asynchronous
  public void printOrder(Order order) {
    // Задача, на решение которой требуется очень много времени
  }
}
```

Когда клиент вызывает метод `printOrder()` либо `sendEmailOrderComplete()`, контейнер незамедлительно возвращает управление клиенту и продолжает обработку вызова в отдельном потоке выполнения. Как вы можете видеть в листинге 7.16, возвращаемым типом двух асинхронных методов является `void`. Возможно, это окажется подходящим при подавляющем большинстве сценариев использования, однако иногда возникает необходимость возвратить значение. Асинхронный метод может возвратить `void`, а также объект `java.util.concurrent.Future<V>`, где `V` представляет результирующее значение. Объекты `Future` позволяют вам получить возвращаемое значение с помощью метода, выполняемого в отдельном потоке. Кроме того, клиент может прибегнуть к API-интерфейсу `Future`, чтобы получить результат или даже отметить вызов.

В листинге 7.17 приведен пример метода, который возвращает Future<Integer>. Метод sendOrderToWorkflow() использует поток операций для обработки объекта Order. Представим, что он вызывает несколько корпоративных компонентов (сообщений, веб-служб и т. д.) и на каждом этапе возвращается значение status (целочисленное). Когда клиент вызывает метод sendOrderToWorkflow() асинхронно, он ожидает получить значение status потока операций. Клиент может извлечь результат с помощью метода Future.get() или, если по какой-либо причине ему понадобится отменить вызов, у него есть возможность прибегнуть к Future.cancel(). Когда клиент вызовет Future.cancel(), контейнер попытается отменить асинхронный вызов, если выполнение этого вызова еще не началось. Обратите внимание, что метод sendOrderToWorkflow() использует метод SessionContext.wasCancelCalled() для проверки того, задал ли клиент отмену запроса. В результате метод возвращает javax.ejb.AsyncResult<V>, который является удобной реализацией Future<V>. Имейте в виду, что AsyncResult используется как средство передачи результирующего значения контейнеру, а не напрямую вызывающему оператору.

Листинг 7.17. Асинхронный метод, возвращающий Future

```
@Stateless
@Asynchronous
public class OrderEJB {
  @Resource
  SessionContext ctx;
  public Future<Integer> sendOrderToWorkflow(Order order) {
    Integer status = 0;
    // обработка
    status = 1;
    if (ctx. wasCancelCalled()) {
      return new AsyncResult<>(2);
    }
    // обработка
    return new AsyncResult<>(status);
  }
}
```

Обратите внимание в листинге 7.17 на то, что вы также можете применить аннотацию @Asynchronous на уровне класса. Она определяет все методы как асинхронные. Когда клиент вызывает метод sendOrderToWorkflow(), ему необходимо вызвать Future.get(), чтобы извлечь результирующее значение.

```
Future<Integer> status = orderEJB.sendOrderToWorkflow (order);
Integer statusValue = status.get();
```

Дескриптор развертывания

Для компонентов Java EE 7 задействуется конфигурация в порядке исключения, а это означает, что контейнер, поставщик постоянства или брокер сообщений будут применять в их отношении набор служб по умолчанию. Конфигурирование этих служб по умолчанию является исключением. Если у вас возникнет необходимость в поведении, которое не обеспечивается по умолчанию, то вам потребуется явно

указать аннотацию либо ее XML-эквивалент. Вы уже видели это при работе с сущностями JPA, где набор аннотаций позволяет настраивать отображение по умолчанию. Аналогичный принцип применяется к сессионным EJB-компонентам. Одной аннотации (`@Stateless`, `@Stateful` и т. д.) достаточно для того, чтобы проинформировать контейнер о необходимости применить определенные службы (управление транзакциями, управление жизненным циклом, безопасность, перехватчики, конкурентный доступ, асинхронность и т. д.), однако если вам потребуется сменить их, используйте аннотации или XML. Аннотации присоединяют дополнительную информацию к классу, интерфейсу, методу или переменной, а XML-дескриптор развертывания делает то же самое.

XML-дескриптор развертывания — альтернатива аннотациям, а это означает, что у любой аннотации имеется эквивалентный XML-тег. При одновременном использовании аннотаций и дескрипторов развертывания настройки в дескрипторе развертывания возьмут верх над аннотациями во время процесса развертывания. Я не стану вдаваться в подробности, описывая структуру XML-дескриптора развертывания (с именем `ejb-jar.xml`), поскольку он является необязательным и может оказаться очень многословным. В качестве примера в листинге 7.18 показано, как мог бы выглядеть файл `ejb-jar.xml`, связанный с `ItemEJB` (который был показан ранее в листинге 7.2). Он определяет класс EJB-компонента, удаленный и локальный интерфейсы, его тип (`Stateless`). Кроме того, здесь имеет место транзакция, управляемая контейнером (CMT) (`Container`).

Листинг 7.18. Файл ejb-jar.xml

```
<ejb-jar xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee ➥
         http://xmlns.jcp.org/xml/ns/javaee/ejb-jar_3_2.xsd"
         version="3.2">
  <enterprise-beans>
    <session>
      <ejb-name>ItemEJB</ejb-name>
      <remote>org.agoncal.book.javaee7.chapter07.ItemRemote</remote>
      <local>org.agoncal.book.javaee7.chapter07.ItemLocal</local>
      <local-bean/>
      <ejb-class>org.agoncal.book.javaee7.chapter07.ItemEJB</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Если вы решите произвести развертывание сессионного EJB-компонента в файле с расширением `.jar`, то вам потребуется сохранить дескриптор развертывания в файле `META-INF/ejb-jar.xml`. А при развертывании сессионного EJB-компонента в файле с расширением `.war` вам придется сохранить дескриптор развертывания в файле `WEB-INF/ejb-jar.xml`. XML-конфигурация описывает детали, специфичные для среды, и не должна указываться в коде с помощью аннотаций (например, если EJB-компонент необходимо задействовать одним путем в среде разработки и другим путем — в средах тестирования/производства).

Контекст именования среды

При работе с корпоративными приложениями возникают ситуации, в которых параметры вашего приложения изменяются от одного развертывания к другому (в зависимости от страны, в которой вы его развертываете, версии приложения и т. д.). Например, в приложении CD-BookStore `ItemEJB` (листинг 7.19) преобразует цену элемента в стоимость в валюте страны, в которой развернуто приложение (с применением `changeRate` на основе курса доллара). Если вы решите произвести развертывание этого EJB-компонента где-то в Европе, то вам потребуется умножить цену элемента на 0,80 и изменить валюту на евро.

Листинг 7.19. Сессионный EJB-компонент без сохранения состояния, преобразующий цены

```
@Stateless
public class ItemEJB {
  public Item convertPrice(Item item) {
    item.setPrice(item.getPrice() * 0.80F);
    item.setCurrency("Euros");
    return item;
  }
}
```

Как вы можете понять, проблема жесткого кодирования этих параметров заключается в том, что вам придется изменить свой код, перекомпилировать его и заново произвести развертывание компонента для каждой страны, в которой валюта будет уже другой. Иной вариант заключается в доступе к базе данных каждый раз, когда вы вызываете метод `convertPrice()`. Однако это будет расточительством ресурсов. Что вам действительно нужно сделать, так это сохранить соответствующие параметры где-нибудь, чтобы вы смогли изменить их во время развертывания. Дескриптор развертывания — идеальное место для задания этих параметров.

Может, дескриптор развертывания (`ejb-jar.xml`) и опционален в EJB 3.2, но его допустимо использовать в случае с записями окружения. Они указываются в дескрипторе развертывания и доступны с помощью внедрения зависимостей (или JNDI). Записи окружения поддерживают следующие Java-типы: `String`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Boolean`, `Double` и `Float`. В листинге 7.20 приведен файл `ejb-jar.xml`, связанный с `ItemEJB`, который определяет две записи: `currencyEntry` типа `String` со значением `Euros` и `changeRateEntry` типа `Float` со значением `0.80`.

Листинг 7.20. Записи окружения ItemEJB в ejb-jar.xml

```
<ejb-jar xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee ➥
         http://xmlns.jcp.org/xml/ns/javaee/ejb-jar_3_2.xsd"
         version="3.2">
  <enterprise-beans>
    <session>
      <ejb-name>ItemEJB</ejb-name>
      <env-entry>
        <env-entry-name>currencyEntry</env-entry-name>
```

```
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>Euros</env-entry-value>
      </env-entry>
      <env-entry>
        <env-entry-name>changeRateEntry</env-entry-name>
        <env-entry-type>java.lang.Float</env-entry-type>
        <env-entry-value>0.80</env-entry-value>
      </env-entry>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Теперь, когда параметры приложения располагаются в дескрипторе развертывания, `ItemEJB` может использовать внедрение зависимостей, чтобы извлечь значение каждой записи окружения. В листинге 7.21 `@Resource(name = "currencyEntry")` внедряет значение `currencyEntry` в атрибут `currency`. Следует отметить, что типы данных записи окружения и внедряемой переменной должны быть совместимыми; в противном случае контейнер сгенерирует исключение.

Листинг 7.21. ItemEJB, использующий записи окружения

```
@Stateless
public class ItemEJB {
  @Resource(name = "currencyEntry")
  private String currency;
  @Resource(name = "changeRateEntry")
  private Float changeRate;

  public Item convertPrice(Item item) {
    item.setPrice(item.getPrice() * changeRate);
    item.setCurrency(currency);
    return item;
  }
}
```

Упаковка

Как и большинство компонентов Java EE (сервлеты, JSF-страницы, веб-службы и т. д.), EJB-компоненты необходимо упаковать, прежде чем развертывать их в контейнере времени выполнения. В том же самом архиве обычно будет присутствовать класс корпоративного EJB-компонента, его интерфейсы, любые необходимые суперклассы или суперинтерфейсы, исключения, вспомогательные классы и необязательный дескриптор развертывания (`ejb-jar.xml`). Упаковав эти артефакты в файл с расширением `.jar`, вы сможете произвести их развертывание непосредственно в контейнере. Еще один вариант заключается в том, что вы также можете встроить файл с расширением `.jar` в файл с расширением `.ear`, а затем произвести его развертывание.

Файл с расширением `.ear` используется для того, чтобы упаковать один или несколько модулей (EJB-файлов с расширением `.jar` или веб-приложений) в один архив, благодаря чему развертывание на сервере приложений будет происходить одновременно и согласованно. Например, как показано на рис. 7.7, если вам по-

требуется произвести развертывание веб-приложения, то, возможно, вы захотите упаковать свои EJB-компоненты и сущности в отдельные файлы с расширением `.jar`, свои сервлеты — в файл с расширением `.war`, а затем все это — в файл с расширением `.ear`. Произведите развертывание EAR-файла на сервере приложений, и вы сможете манипулировать сущностями из сервлета с использованием соответствующего EJB-компонента.

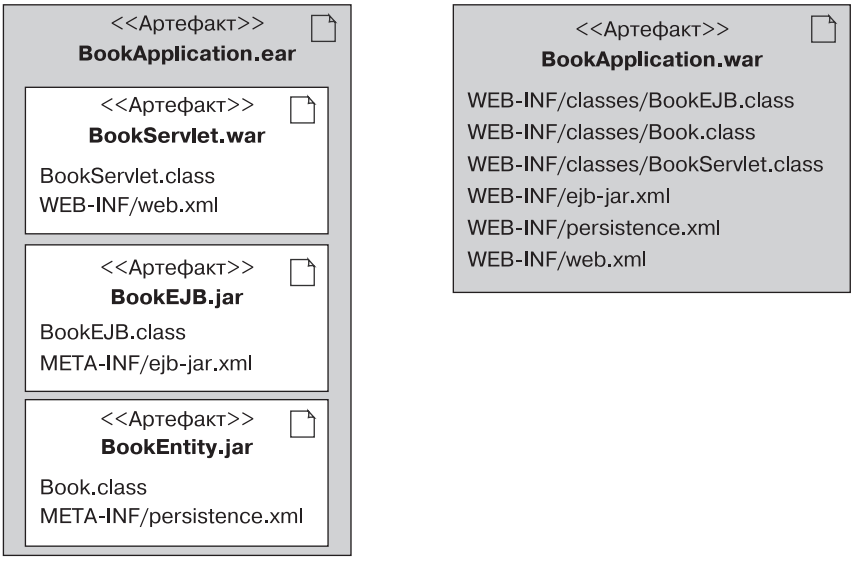


Рис. 7.7. Упаковка EJB-компонентов

С выходом EJB 3.1 появилась возможность упаковать EJB Lite непосредственно в веб-модуль (WAR-файл). Справа на рис. 7.7 сервлет, EJB-компонент и сущность упакованы в один и тот же WAR-файл со всеми дескрипторами развертывания. Обратите внимание в EJB-модуле на то, что дескриптор развертывания располагается в файлах `META-INF/ejb-jar.xml` и `WEB-INF/ejb-jar.xml`. EJB Lite можно упаковать непосредственно в WAR- или JAR-файл. Если у вас возникнет необходимость использовать полную спецификацию EJB (например, удаленный интерфейс, JMS, асинхронные вызовы...), то придется упаковать все в файл с расширением `.jar`, а не `.war`.

Развертывание EJB-компонента

Сессионные EJB-компоненты — это компоненты, управляемые контейнерами, в чем и заключается их преимущество. Контейнер сам взаимодействует со всевозможными службами (управление транзакциями, управление жизненным циклом, асинхронность, перехватчики и т. д.), позволяя вам сосредоточиться на бизнес-коде. Недостаток состоит в том, что вам всегда нужно выполнять свои EJB-компоненты в контейнере. Исторически сложилось так, что эти контейнеры функционировали в отдельном процессе, из-за чего тестирование было немного утомительным занятием. Вам приходилось запускать свой контейнер (также называемый сервером),

упаковывать свои EJB-компоненты, развертывать и тестировать их и в конце концов останавливать сервер... чтобы снова запустить его позднее.

Эта проблема была решена, начиная с EJB 3.1, созданием встраиваемого EJB-контейнера. Версия EJB 3.1 привнесла стандартный API-интерфейс для выполнения EJB-компонентов в среде Java SE. Embeddable API (пакет `javax.ejb.embeddable`) позволяет клиенту создать экземпляр EJB-контейнера, который будет функционировать в рамках своей виртуальной машины Java. Встраиваемый контейнер обеспечивает управляемую среду с поддержкой тех же базовых служб, которые имеют место в контейнере Java EE: внедрение, управление транзакциями, управление жизненным циклом и т. д. Встраиваемые EJB-контейнеры работают только с API-интерфейсом подмножества EJB Lite (никаких EJB-компонентов, управляемых сообщениями, удаленных вызовов и т. д.), то есть они обладают теми же возможностями, что и контейнер EJB Lite (но не контейнер, отвечающий полной версии EJB).

В листинге 7.22 приведен класс `Main`, который задействует API-интерфейс для начальной загрузки, чтобы запустить контейнер (абстрактный класс `javax.ejb.embeddable.EJBContainer`), осуществляет поиск EJB-компонента и вызывает методы в нем.

Листинг 7.22. Класс Main, использующий встраиваемый контейнер

```
public class Main {
  public static void main(String[] args) throws NamingException {
    // Задает путь к классам контейнера
    Map<String, Object> properties = new HashMap<>();
    properties.put(EJBContainer.MODULES, new File("target/classes"));

    // Создает встроенный контейнер и получает JNDI-контекст
    try (EJBContainer ec = EJBContainer.createEJBContainer(properties)) {
      Context ctx = ec.getContext();
      // Создает экземпляр Book
      Book book = new Book();
      book.setTitle("Автостопом по Галактике");
      book.setPrice(12.5F);
      book.setDescription("Научно-фантастическая комедийная книга");
      book.setIsbn("1-84173-742-2");
      book.setNbOfPage(354);
      book.setIllustrations(false);

      // Осуществляет поиск EJB-компонента с представлением без интерфейса
      ItemEJB itemEJB = (ItemEJB) ctx.lookup("java:global/classes/ItemEJB");

      // Обеспечивает постоянство Book в базе данных
      itemEJB.createBook(book);

      // Извлекает информацию обо всех соответствующих книгах из базы данных
      for (Book aBook : itemEJB.findBooks()) {
        System.out.println(aBook);
      }
    }
  }
}
```

Как вы можете видеть в листинге 7.22, `EJBContainer` содержит фабричный метод (`createEJBContainer()`) для создания экземпляра контейнера. По умолчанию встраиваемый контейнер осуществляет поиск, используя путь к классам клиента, чтобы отыскать набор EJB-компонентов для инициализации (либо вы можете задать путь к классам в свойствах). После инициализации контейнера приложение получает JNDI-контекст контейнера (`EJBContainer.getContext()`, который возвращает `javax.naming.Context`) для поиска `ItemEJB` (с использованием синтаксиса переносимых глобальных JNDI-имен).

Следует отметить, что `ItemEJB` (приведенный ранее в листинге 7.1) является сессионным EJB-компонентом без сохранения состояния, обеспечивающим бизнес-методы с помощью представления без интерфейса. Он использует внедрение, транзакции, управляемые контейнером, и сущность JPA с именем `Book`. Встраиваемый контейнер заботится о внедрении менеджера сущностей и фиксации или откате любой транзакции. `EJBContainer` реализует `java.lang.AutoCloseable`, поэтому блок `try-with-resources` автоматически вызовет метод `EJBContainer.close()` для закрытия экземпляра встраиваемого контейнера.

В листинге 7.22 я использовал класс `Main`, чтобы показать вам, как применять встраиваемый EJB-контейнер. Однако имейте в виду, что в настоящее время EJB-компоненты можно использовать в любой среде Java SE: от тестовых классов до Swing-приложений, и даже при пакетной обработке.

Вызов корпоративных EJB-компонентов

Теперь, когда вы уже видели примеры сессионных EJB-компонентов и их различных интерфейсов, вам, возможно, хочется взглянуть на то, как клиент вызывает эти EJB-компоненты. Клиентом сессионного EJB-компонента может быть любой компонент: POJO, графический интерфейс (Swing), управляемый CDI MBean-компонент, сервлет, EJB-компонент JSF, являющийся подложкой, веб-служба (SOAP или REST) либо другой EJB-компонент (развернутый в том же или другом контейнере).

На стороне клиента также все просто. Чтобы вызвать метод в сессионном EJB-компоненте, клиент не создает экземпляр соответствующего EJB-компонента непосредственным образом (с использованием оператора `new`). Ему нужна ссылка на этот EJB-компонент (или на его интерфейсы). Он может получить ее при внедрении зависимостей (с использованием аннотации `@EJB` или `@Inject`) либо с помощью JNDI-поиска. Внедрение зависимостей позволяет контейнеру автоматически внедрять ссылку на EJB-компонент во время развертывания. Если не предусмотрено иное, то клиент будет вызывать сессионный EJB-компонент синхронно.

Вызов с использованием внедрения

Java EE задействует несколько аннотаций для внедрения ссылок на ресурсы (`@Resource`), менеджера сущностей (`@PersistenceContext`), веб-службы (`@WebServiceRef`) и т. д. Аннотация `@javax.ejb.EJB` специально предназначена для внедрения ссылок на сессионные EJB-компоненты в клиентский код. Внедрение зависимостей возможно только в управляемых средах вроде EJB-контейнеров, веб-контейнеров и контейнеров клиентских приложений.

Обратимся к нашим исходным примерам, в которых сессионные EJB-компоненты не обладали интерфейсом. Чтобы клиент смог вызвать представление без интерфейса сессионного EJB-компонента, ему необходимо получить ссылку на сам класс EJB-компонента. Например, в приведенном далее коде клиент получает ссылку на класс `ItemEJB` с использованием аннотации `@EJB`:

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

// Клиентский код с внедрением ссылки на EJB-компонент
@EJB ItemEJB itemEJB;
```

Если сессионный EJB-компонент реализует несколько интерфейсов, клиенту придется указать, на какой из них ему требуется ссылка. В приведенном далее коде `ItemEJB` реализует два интерфейса и благодаря аннотации `@LocalBean` также обеспечивает представление без интерфейса. Клиент может вызвать EJB-компонент с помощью его локального представления, удаленного представления или представления без интерфейса:

```
@Stateless
@Remote(ItemRemote.class)
@Local(ItemLocal.class)
@LocalBean
public class ItemEJB implements ItemLocal, ItemRemote {...}

// Клиентский код с внедрением нескольких ссылок на EJB-компонент или интерфейсы
@EJB ItemEJB itemEJB;
@EJB ItemLocal itemEJBLocal;
@EJB ItemRemote itemEJBRemote;
```

У API `@EJB` имеется несколько атрибутов. Один из них — JNDI-имя EJB-компонента, который вы хотите внедрить. Оно может быть особенно полезно при работе с удаленными EJB-компонентами, располагающимися на другом сервере:

```
@EJB(lookup = "java:global/classes/ItemEJB") ItemRemote itemEJBRemote;
```

Вызов с использованием CDI

Как вы только что видели, для вызова метода в EJB-компоненте требуется только аннотация `@EJB`, позволяющая внедрить ссылку в клиентский код. Все довольно просто: вы получаете ссылку во время развертывания. Однако `@EJB` не обеспечивает, например, чего-то подобного CDI-альтернативам. Взамен вам придется использовать аннотацию `@Inject`.

В большинстве случаев вы можете просто заменять `@EJB` аннотацией `@Inject`, и ваш клиентский код будет работать. Сделав это, вы получите все преимущества CDI, которые видели в главе 2. Таким образом, если мы обратимся к приведенным ранее примерам, то вот как обеспечивалось бы EJB-внедрение с использованием CDI в случае с клиентом:

```
@Stateless
@Remote(ItemRemote.class)
```

```
@Local(ItemLocal.class)
@LocalBean
public class ItemEJB implements ItemLocal, ItemRemote {...}

// Клиентский код с внедрением нескольких ссылок на EJB-компонент или интер-
фейсы посредством @Inject
@Inject ItemEJB itemEJB;
@Inject ItemLocal itemEJBLocal;
@Inject ItemRemote itemEJBRemote;
```

В случае с удаленными EJB-компонентами, как вы только что видели, вам может потребоваться JNDI-строка для их поиска. Аннотация `@Inject` не может обладать строковым параметром, так что при этом вам придется создать удаленный EJB-компонент, чтобы вы смогли внедрить его:

```
// Код для создания удаленного EJB-компонента
@Produces @EJB(lookup = "java:global/classes/ItemEJB") ItemRemote ➥
  itemEJBRemote;

// Клиентский код с внедрением созданного удаленного EJB-компонента
@Inject ItemRemote itemEJBRemote;
```

В зависимости от клиентской среды у вас может отсутствовать возможность использовать внедрение (если компонент не управляется контейнером). В этом случае попробуйте обратиться к JNDI для поиска сессионных EJB-компонентов с помощью их переносимых JNDI-имен.

Вызов с использованием JNDI

Поиск сессионных EJB-компонентов также можно осуществлять с использованием JNDI, который главным образом применяется для удаленного доступа, когда клиент не управляется контейнером и не может применять внедрение зависимостей. Однако JNDI также может задействоваться локальными клиентами, хотя внедрение зависимостей приводит к тому, что код становится более простым. Внедрение осуществляется во время развертывания. Если есть вероятность того, что данные не будут использоваться, EJB-компонент может избежать затрат, связанных с внедрением ресурсов, выполнив JNDI-поиск. JNDI является альтернативой внедрению. При использовании JNDI код выталкивает данные из стека, только если они необходимы, вместо того чтобы принимать подвергнутые проталкиванию в стек данные, которые могут вообще не потребоваться.

JNDI — это API-интерфейс для доступа к службам каталогов, позволяющий клиентам осуществлять привязку и поиск объектов по имени. JNDI определяется в Java SE и не зависит от базовой реализации, то есть вы можете выполнять поиск объектов в каталоге Lightweight Directory Access Protocol (LDAP) или системе доменных имен (DNS), используя стандартный API-интерфейс.

В качестве альтернативы предшествующему коду можно использовать `InitialContext`, связанный с JNDI, и осуществлять поиск развернутого EJB-компонента с помощью его переносимого JNDI-имени, которое я определил ранее в подразделе «Переносимое JNDI-имя» раздела «Написание корпоративных

EJB-компонентов» данной главы. Клиентский код будет выглядеть следующим образом:

```
Context ctx = new InitialContext();
ItemRemote itemEJB = (ItemRemote)
  ctx.lookup("java:global/cdbookstore/ItemEJB!org.agoncal.book.javaee7.
ItemRemote");
```

Резюме

Спецификация EJB эволюционировала годами, начиная с версии EJB 2.*x*, — от тяжеловесной модели, при которой домашние и удаленные/локальные интерфейсы приходилось укомплектовывать множеством XML-кода, к простому Java-классу без интерфейса и с одной аннотацией. Базовая функциональность всегда оставалась неизменной: транзакционная и безопасная бизнес-логика (подробнее на эту тему мы поговорим в следующих главах).

Сессионные EJB-компоненты являются компонентами, управляемыми контейнером, которые используются для разработки бизнес-уровней. Есть три типа сессионных EJB-компонентов: без сохранения состояния, с сохранением состояния и одиночные. Первые — легко масштабируемы, поскольку не поддерживают никакого состояния, располагаются в пуле и решают задачи, с которыми можно справиться одним вызовом метода. Между EJB-компонентами с сохранением состояния и клиентом имеет место корреляция «один к одному», кроме того, они могут быть временно удалены из памяти с использованием пассивизации и активизации. Одиночные EJB-компоненты обладают уникальным экземпляром, который совместно применяется несколькими клиентами и может инициализироваться во время запуска, объединяться в цепочку с другими экземплярами и настраиваться при осуществлении клиентами конкурентного доступа.

Несмотря на эти различия, сессионные EJB-компоненты совместно используют общую модель программирования. Они могут обладать локальным представлением, удаленным представлением или представлением без интерфейса, задействовать аннотации или быть развернутыми с применением дескриптора развертывания. Сессионные EJB-компоненты могут применять внедрение зависимостей для получения ссылок на несколько ресурсов (источники данных JDBC, контекст постоянства, записи окружения и т. д.), а также на их контекст среды выполнения (объект `SessionContext`). С выходом EJB 3.1 у вас появилась возможность вызывать методы асинхронно, выполнять поиск EJB-компонентов с использованием переносимых JNDI-имен и задействовать встраиваемый EJB-контейнер в среде Java SE. Версия EJB 3.2 идет по пути своей предшественницы, привнося небольшие улучшения.

В следующей главе объясняется жизненный цикл разных сессионных EJB-компонентов и описывается, как можно осуществлять взаимодействие с использованием аннотаций обратных вызовов. Кроме того, вы узнаете, как применять расширенный инструмент `TimerService` для планирования задач, а также как задавать проверку прав на доступ.

Глава 8

Функции обратного вызова, служба таймера и авторизация

Из предыдущей главы вы узнали, что сеансовые компоненты управляются контейнером. Они располагаются в контейнере EJB, который незаметно обертывает бизнес-логику с помощью некоторых служб (внедрение зависимостей, управление транзакциями и т. д.). Рассмотрим три такие службы: управление жизненным циклом, планирование и авторизацию.

Жизненный цикл означает, что компонент-сеанс проходит через предопределенный набор переходов между состояниями. В зависимости от вида вашего компонента (хранящие/не хранящие состояние либо синглтоны) жизненный цикл будет состоять из различных состояний. Каждый раз, когда контейнер изменяет состояние жизненного цикла, он может вызывать методы, которые аннотированы как методы обратного вызова.

Служба таймера EJB является стандартным решением вопросов планирования задач. Корпоративные приложения, зависящие от уведомлений, основанных на календаре, используют эту службу для моделирования бизнес-процессов.

Защита данных также важна. Вы хотите, чтобы ваш бизнес-уровень действовал как брандмауэр и разрешал выполнять некоторые действия определенным группам пользователей и запрещал доступ остальным (например, только сотрудники имеют право сохранять данные, но и пользователи, и авторизованные сотрудники имеют право их считывать).

В этой главе вы узнаете, что сеансовый компонент, не хранящий состояние, а также синглтон могут иметь общий жизненный цикл, а жизненный цикл компонента, хранящего состояние, немного отличается. Вы также увидите, как компонент, не хранящий состояние, и синглтон могут использовать службу таймера для создания распределения задач по времени декларативно или программно. Эта глава заканчивается демонстрацией принципа работы авторизации в контейнере EJB. Кроме того, вы узнаете, как можно с легкостью позволить пользователям с определенными ролями получить доступ к вашему коду.

Жизненный цикл сеансовых компонентов

Как вы видели в предыдущей главе, клиент не создает экземпляр сеансового компонента с помощью оператора `new`. Он получает ссылку на сеансовый компонент либо с помощью внедрения зависимостей, либо через поиск JNDI. Созданием и разрушением экземпляров занимается контейнер. Это означает, что ни клиент,

ни компонент не отвечают за определение момента, когда создается экземпляр компонента, когда внедряются зависимости или когда экземпляр уничтожается. Контейнер отвечает за управление жизненным циклом компонента.

Все сеансовые компоненты имеют две очевидные фазы их жизненного цикла: создание и разрушение. Кроме того, компоненты, сохраняющие состояние, имеют фазы пассивизации и активизации, упомянутые в предыдущей главе. EJB аналогично методам обратного вызова, используемым в сущностях, которые вы видели в предыдущих главах, позволяет контейнеру автоматически вызывать аннотированные методы (`@PostConstruct`, `@PreDestroy` и т. д.) на определенных этапах его жизненного цикла. Эти методы могут инициализировать информацию о состоянии компонента, искать ресурсы с помощью JNDI и освобождать соединения с базой данных.

Компоненты, не сохраняющие состояния, и синглтоны

Компоненты, не сохраняющие состояния, и синглтоны похожи друг на друга тем, что они не поддерживают состояние диалога с выделенным клиентом. Оба типа компонентов позволяют получить доступ к любому клиенту — компонент, не хранящий состояние, делает это для каждого экземпляра, в то время как синглтон предоставляет доступ к одному экземпляру. Компоненты обоих типов имеют одинаковые жизненные циклы (рис. 8.1), которые описываются следующим образом.

1. Жизненный цикл компонента без сохранения состояния и синглтона начинается, когда клиент запрашивает ссылку на компонент (с помощью внедрения зависимостей или поиска JNDI). Для синглтона он также может начаться, когда загружается контейнер (при этом используется аннотация `@Startup`). Контейнер создает новый экземпляр сеансового компонента.
2. Если вновь созданный экземпляр использует внедрение зависимостей через аннотации (`@Inject`, `@Resource`, `@EJB`, `@PersistenceContext` и т. д.) или развертывание дескрипторов, то контейнер внедряет все необходимые ресурсы.
3. Если экземпляр имеет метод с аннотацией `@PostContruct`, то контейнер вызывает его.
4. Экземпляр компонента обрабатывает вызов, созданный клиентом, и остается в режиме готовности к обработке будущих вызовов. Компоненты, не сохраняющие состояние, остаются в этом режиме до тех пор, пока контейнер не освободит место в хранилище. Синглтоны остаются в режиме готовности до тех пор, пока контейнер не будет выключен.
5. Объект больше не нужен контейнеру. Последний вызывает метод с аннотацией `@PreDestroy`, если таковой имеется, и заканчивает жизненный цикл экземпляра компонента.

Компоненты, не хранящие состояния, и синглтоны имеют одинаковый жизненный цикл, но существуют различия в том, как они создаются и уничтожаются.

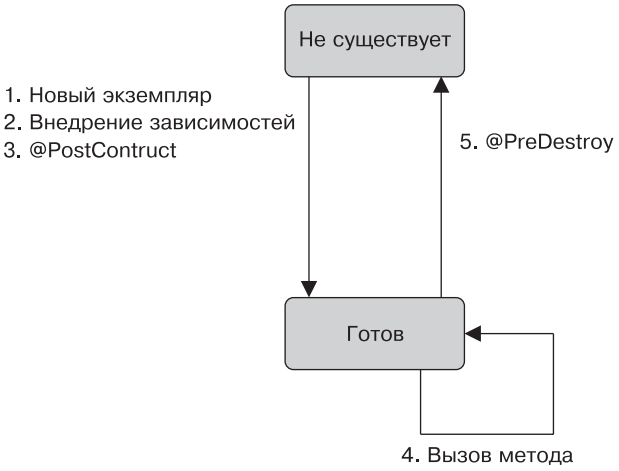


Рис. 8.1. Жизненный цикл сеансовых компонентов, не хранящих состояния, и синглтонов

При развертывании сеансового компонента, не хранящего состояния, контейнер создает несколько объектов и добавляет их в хранилище. Когда клиент вызывает метод сеансового компонента, не хранящего состояния, контейнер выбирает один экземпляр из хранилища, делегирует вызов метода этому экземпляру и возвращает его в хранилище. Если контейнер больше не нуждается в экземпляре (обычно такое происходит, когда контейнер хочет уменьшить количество экземпляров в хранилище), он разрушает его.

ПРИМЕЧАНИЕ

Контроль над EJB может производиться с помощью консоли управления GlassFish (GlassFish Administration Console). GlassFish затем позволяет настроить несколько параметров хранилища EJB. Вы можете установить размер хранилища (исходное, минимальное и максимальное количество компонентов в хранилище), количество компонентов, которые необходимо удалить при превышении времени простоя. Вы также можете задать количество миллисекунд, по прошествии которых будут удаляться объекты. Наличие этих настроек зависит от реализации; некоторые из них могут быть недоступны для других контейнеров EJB.

Создание сеансовых синглтонов зависит от того, как был создан экземпляр (с аннотацией `@StartUp` или без), а также от того, зависит ли он (`@DependsOn`) от синглтона, который был создан с этой аннотацией. Если ответ — «да», то контейнер создаст экземпляр во время развертывания. Если — «нет», то контейнер создаст экземпляр в тот момент, когда клиент вызовет бизнес-метод. Поскольку синглтоны существуют во время работы приложения, их экземпляры разрушаются при отключении контейнера.

Компоненты, хранящие состояние

Компоненты, хранящие состояние, программно незначительно отличаются от компонентов, не хранящих состояния, и синглтонов: меняются только метаданные

(`@Stateful` вместо `@Stateless` или `@Singleton`). Реальное отличие заключается в том, что компоненты, хранящие состояние, поддерживают диалог с клиентом и, следовательно, имеют немного иной жизненный цикл. Контейнер создает экземпляр и назначает его только одному клиенту. Затем каждый запрос от клиента передается этому экземпляру. Следуя данному принципу, в зависимости от вашего приложения, вы в итоге можете прийти к соотношению «один к одному» между клиентом и сеансовым компонентом (например, тысяча пользователей, работающих одновременно, может создать тысячу компонентов, сохраняющих состояние). Если один из клиентов не вызывает экземпляр своего компонента в течение длительного промежутка времени, контейнер должен удалить его, чтобы у виртуальной машины не закончилась память, и при этом сохранить состояние экземпляра в постоянном хранилище, а затем вернуть экземпляр с этим состоянием, когда это необходимо. Контейнер использует прием пассивизации и активизации.

Пассивизация — это процесс, когда контейнер сериализует экземпляр компонента на постоянный носитель (файл на диске, база данных и т. д.), вместо того чтобы держать его в памяти. Активизация, которая является противоположностью этому процессу, выполняется, когда экземпляр компонента снова необходим клиенту. Контейнер десериализует компонент из постоянного хранилища и помещает его обратно в память. Это означает, что атрибуты компонента должны быть сериализуемыми (они должны иметь примитивный тип или реализовывать интерфейс `java.io.Serializable`). На рис. 8.2 показан жизненный цикл компонентов, сохраняющих состояние, он описывается следующим образом.

1. Жизненный цикл компонента, сохраняющего состояние, начинается, когда клиент запрашивает ссылку на компонент (с использованием внедрения зависимостей или поиска JNDI). Контейнер создает новые экземпляры сеансовых компонентов и сохраняет их в памяти.
2. Если вновь созданный экземпляр применяет внедрение зависимостей через аннотации (`@Inject`, `@Resource`, `@EJB`, `@PersistenceContext` и т. д.) или дескрипторы развертывания, контейнер внедряет все необходимые ресурсы.
3. Если экземпляр имеет метод с аннотацией `@PostContruct`, контейнер вызывает его.
4. Компонент выполняет запрошенный вызов и остается в памяти, ожидая последующих запросов от клиентов.
5. Если клиент ничего не запрашивает в течение некоторого периода времени, то контейнер вызывает метод, аннотированный `@PrePassivate`, если таковой имеется, и пассивизирует экземпляр компонента в постоянное хранилище.
6. Если клиент вызывает пассивизированный компонент, контейнер помещает его обратно в память и вызывает метод с аннотацией `@PostActivate`, если таковой имеется.
7. Если клиент не вызывает пассивизированный экземпляр компонента в течение определенного периода времени, то контейнер разрушает его.
8. В качестве замены шагу 7, если клиент вызывает метод, аннотированный `@Remove`, контейнер вызывает метод с аннотацией `@PreDestroy`, если таковой имеется, и заканчивает жизненный цикл экземпляра компонента.

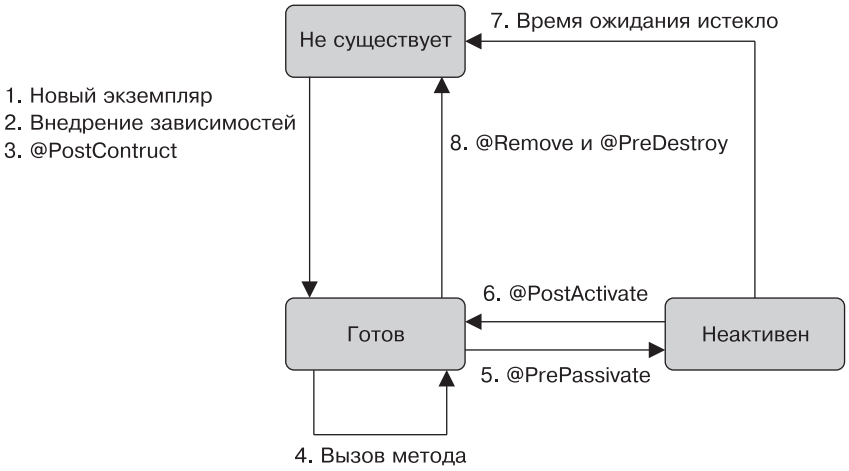


Рис. 8.2. Жизненный цикл компонента, сохраняющего состояние

В некоторых случаях компонент, сохраняющий состояние, содержит открытые ресурсы, такие как сетевые сокеты или соединения с базой данных. Поскольку контейнер не может держать эти ресурсы открытыми для каждого компонента, вам придется закрывать и снова открывать ресурсы до и после пассивизации, используя методы обратного вызова. Другая возможность состоит в отключении поведения активизации/пассивизации по умолчанию или компонента, сохраняющего состояние. Вы должны очень осторожно использовать эту особенность, но вы нуждаетесь именно в ней. Можете аннотировать ваши компоненты, сохраняющие состояние, с помощью атрибута `@Stateful(passivationCapable = False)`.

Методы обратного вызова

Как вы только что видели, каждый тип сессионных компонентов имеет свой собственный управляемый контейнером жизненный цикл. Контейнер позволяет вам предоставить бизнес-логику, которая будет использована при смене состояния компонента. Переход из одного состояния в другое может быть перехвачен контейнером для вызова методов, аннотированных одним из способов, показанных в табл. 8.1.

Таблица 8.1. Аннотации методов обратного вызова, предназначенных для работы с жизненным циклом

Аннотация	Описание
@PostConstruct	Метод должен быть вызван сразу после создания экземпляра компонента и выполнения контейнером внедрения зависимостей. В подобных методах часто выполняется инициализация
@PreDestroy	Метод должен быть вызван сразу после того, как контейнер уничтожит экземпляр компонента. Этот метод часто используется для

Продолжение ⇩

Таблица 8.1 *(продолжение)*

Аннотация	Описание
	освобождения ранее инициализированных ресурсов. У компонентов, сохраняющих состояние, это происходит, когда истекает время ожидания или когда отработает метод, аннотированный @Remove
@PrePassivate	Этот метод предназначен только для компонентов, сохраняющих состояние. Он должен быть вызван перед тем, как контейнер пассивизирует экземпляр. Обычно это дает компоненту время, чтобы подготовиться к сериализации и освободить ресурсы, которые не могут быть сериализованы (например, закрываются соединения с базой данных, менеджеры сообщений, сетевые сокеты и др.)
@PostActivate	Этот метод предназначен только для компонентов, сохраняющих состояние. Он должен быть вызван

ПРИМЕЧАНИЕ

В пакете Javax.ejb спецификации EJB 3.2 (JSR 345) определены аннотации @PrePassivate и @PostActivate. Они связаны с жизненным циклом EJB с сохранением состояния и могут произойти много раз во время существования EJB. Аннотации @PostConstruct и @PreDestroy являются частью спецификации CommonAnnotation 1.2 (JSR 250) и располагаются в пакете javax.annotation (как, например, @Resource или другие аннотации, связанные с безопасностью, которые вы увидите далее). Они связаны с жизненным циклом управляемых компонентов и срабатывают только один раз за время существования экземпляра класса (при его создании и разрушении).

Для методов обратного вызова применяются следующие правила.

- Метод не должен иметь никаких параметров и возвращает тип `void`.
- Метод не должен генерировать проверяемое исключение, но при этом может сгенерировать исключения времени выполнения. Генерация подобного исключения заставит откатить транзакцию, если таковая будет существовать (что описано в следующей главе).
- Метод может иметь типы доступа `public`, `private`, `protected` или быть доступным на уровне пакета, но при этом не должен иметь спецификаторов `static` или `final`.
- Метод может иметь несколько аннотаций (метод `init()`, который будет показан позже, в листинге 8.2, имеет аннотации `@PostConstruct` и `@PostActivate`). Однако может существовать только одна аннотация данного типа для каждого компонента (например, нельзя иметь два метода с аннотацией `@PostConstruct` для одного и того же сессионного компонента).
- Метод обратного вызова может получить доступ к данным среды окружения компонента (см. подраздел «Контекст именования среды» раздела «Внедрение зависимостей» главы 7).

Эти методы обратного вызова, как правило, используются для выделения и/или освобождения ресурсов компонента. В качестве примера в листинге 8.1 показан компонент `CacheEJB` типа «синглтон», использующий аннотацию `@PostConstruct` для инициализации кэша. Сразу после создания одного экземпляра компонента `CacheEJB` контейнер вызывает метод `initCache()`.

Листинг 8.1. Компонент-синглтон, инициализирующий свой кэш с помощью аннотации @PostConstruct

```
@Singleton
public class CacheEJB {
  private Map<Long, Object> cache = new HashMap<>();
  @PostConstruct
  private void initCache() {
    cache.put(1L, "Первый товар в кэше");
    cache.put(2L, "Второй товар в кэше");
  }

  public Object getFromCache(Long id) {
    if (cache.containsKey(id))
      return cache.get(id);
    else
      returnnull;
    }
}
```

В листинге 8.2 показан фрагмент кода для компонента, сохраняющего свое состояние. Контейнер поддерживает диалоговое состояние, которое может включать в себя тяжелые ресурсы, такие как подключения к базе данных. Поскольку открывать соединение с базой данных довольно затратно по времени, одно соединение должно использоваться во всех вызовах и закрываться, когда компонент простаивает (или пассивизирован).

Листинг 8.2. Инициализация и высвобождение ресурсов компонента, сохраняющего состояние

```
@Stateful
public class ShoppingCartEJB {
  @Resource(lookup = "java:comp/defaultDataSource")
  private DataSource ds;
  private Connection connection;

  private List<Item> cartItems = new ArrayList<>();
  @PostConstruct
  @PostActivate
  p rivate void init() {
    connection = ds.getConnection();
  }
  @PreDestroy
  @PrePassivate
  private void close() {
    connection.close();
  }
  //...
  @Remove
  public void checkout() {
    cartItems.clear();
  }
}
```

После создания экземпляра компонента, сохраняющего состояние, контейнер внедряет ссылку на источник данных, используемый по умолчанию (более подробно источники данных мы рассмотрим далее, а также в следующей главе) для атрибута `ds`. После того как внедрение будет выполнено, контейнер вызовет метод, аннотированный `@PostConstruct` (`init()`), который создает соединение с базой данных. Если контейнеру придется пассивизировать экземпляр, будет вызван метод `Close()` (`@PrePassivate`). Его цель — закрыть JDBC-соединение, которое имеет собственные ресурсы и никогда больше не понадобится, пока экземпляр пассивен. Когда клиент вызывает бизнес-метод компонента, контейнер активизирует его и снова вызывает метод `Init()` (`@PostActivate`). Когда клиент вызывает метод `checkout()` (с аннотацией `@Remove`), контейнер удаляет экземпляр, но сначала вновь будет вызван метод `Close()` (`@PreDestroy`). Обратите внимание, что для удобства чтения я опустил в этих методах обработку исключений, генерируемых SQL.

Служба таймера

Некоторые приложения Java EE должны планировать задачи, чтобы обеспечивать уведомления в определенное время. Так, например, приложение CD-Bookstore должно отправлять поздравительные письма своим покупателям каждый год в их день рождения, печатать ежемесячные статистические данные о продаваемых товарах, генерировать отчеты об уровнях инвентаризации каждую ночь и освежать технический кэш каждые 30 секунд.

В результате в EJB 2.1 было введено средство планирования — служба таймера, так как клиенты не могли использовать напрямую API `Thread`. По сравнению с другими проприетарными инструментами или фреймворками (UNIX-утилита cron, Quartz и др.) служба таймера имеет меньше возможностей. Только в версии 3.1 спецификации EJB можно заметить резкое улучшение службы таймера. Сегодня служба конкурирует с другими продуктами, поскольку выполняет большинство задач по планированию.

Служба таймера в EJB представляет собой службу контейнера, которая позволяет компонентам регистрироваться для использования методов обратного вызова. Уведомления EJB могут быть назначены в соответствии с графиком на основе календаря, в определенное время, по истечении определенного промежутка времени, или в конкретные повторяющиеся интервалы. Контейнер ведет учет всех таймеров и вызывает соответствующий метод экземпляра компонента, когда таймер отработал. На рис. 8.3 показаны два этапа, связанных с работой службы таймера. Во-первых, EJB необходимо создать таймер (автоматически или программно) и зарегистрироваться для использования метода обратного вызова, затем служба таймера вызывает зарегистрированный метод экземпляра EJB.

Таймеры предназначены для длительных бизнес-процессов и по умолчанию постоянны. Это означает, что они переживают отключения сервера и, как только сервер снова начинает работать, выполняются так, как если бы отключения не произошло. По желанию вы можете указать, что таймеры будут непостоянными.

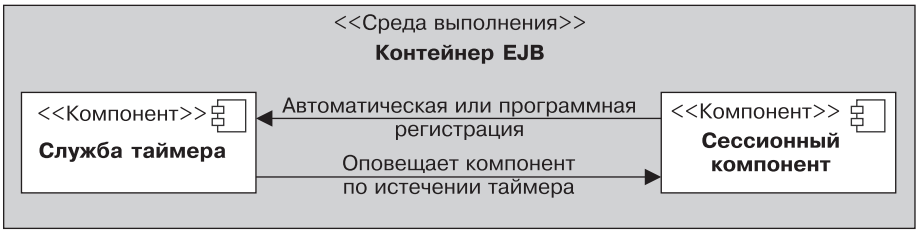


Рис. 8.3. Взаимодействие между службой таймера и сессионным компонентом

ПРИМЕЧАНИЕ

Компоненты, не сохраняющие состояния, синглтоны и MDB могут быть зарегистрированы службой таймера, но компоненты, сохраняющие состояние, такой возможности не имеют, а также не могут пользоваться API для планирования.

Таймеры могут быть созданы автоматически контейнером во время развертывания, если компонент имеет методы, аннотированные `@Schedule`. Но таймеры также могут быть созданы программно и в таком случае должны предоставить один метод обратного вызова с аннотацией `@Timeout`.

Выражения на основе календаря

Служба таймера использует синтаксис, основанный на календаре, который впервые появился в UNIX-утилите cron. Этот синтаксис применяется для программного создания таймера (с помощью класса `ScheduleExpression`) и для автоматического создания таймера (с помощью аннотации `@Schedule` или дескриптора развертывания). Выражение для планирования может выглядеть следующим образом:

```
year = "2008,2012,2016" dayOfWeek = "Sat,Sun" minute = "0-10,30,40"
```

В табл. 8.2 и 8.3 определены атрибуты создания выражений на основе календаря.

Таблица 8.2. Атрибуты выражений на основе календаря

Атрибут	Описание	Возможные значения	Значение по умолчанию
second	Одна или несколько секунд (не более минуты)	[0,59]	0
minute	Одна или несколько минут (не более часа)	[0,59]	0
hour	Один или несколько часов (не более суток)	[0,23]	0
dayOfMonth	Один или несколько дней (не более месяца)	[1,31] или {"1st", "2nd", "3rd", ..., "30th", "31st"} или {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"} или "Last" (последний день	*

Продолжение ⇩

Таблица 8.2 *(продолжение)*

Атрибут	Описание	Возможные значения	Значение по умолчанию
		месяца) или –x (означает x дней перед последним днем месяца)	
month	Один или несколько месяцев (не более года)	[1,12] или {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"}	
dayOfWeek	Один или несколько дней (не более недели)	[0,7] или {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"} — "0" и "7" относятся к воскресенью	*
year	Определенный календарный год	Четырехразрядный календарный год	*
timezone	Определенный часовой пояс	Список часовых поясов, предоставленный базой данных zoneinfo (или tz)	

Каждый атрибут выражения на основе календаря (секунды, минуты, часы и т. д.) поддерживает значения, выраженные в различных формах. Например, вы можете иметь список дней или диапазон лет. В табл. 8.3 определены формы, которые могут принимать атрибуты.

Таблица 8.3. Формы выражений

Форма	Описание	Пример
Одно значение	Атрибут может иметь только одно возможное значение	year = "2010" month= "May"
Шаблон поиска	Эта форма представляет все возможные значения заданного атрибута	second = "*" dayOfWeek = "*"
Список	Атрибут имеет два или более значения, разделенные запятой	year = "2008,2012,2016" dayOfWeek = "Sat,Sun" minute = "0–10,30,40"
Диапазон	Атрибут имеет диапазон значений, разделенный тире	second = "1–10" dayOfWeek = "Mon–Fri"
Инкремент	Атрибут имеет начальную точку и интервал, отделенный прямым слешем	minute = "*/15" second = "30/10"

Если вы использовали ранее синтаксис UNIX-утилиты cron, то этот синтаксис может показаться знакомым и гораздо более простым. С помощью такого богатого синтаксиса можно выразить практически любой вид календарного выражения, что показано в табл. 8.4.

Таблица 8.4. Примеры выражений

Пример	Выражение
Каждую среду в полночь	dayOfWeek = "Wed"
Каждую среду в полночь	second = "0", minute = "0", hour = "0", dayOfMonth = "*", month = "*", dayOfWeek = "Wed", year = "*"
Каждый будний день в 6:55	minute = "55", hour = "6", dayOfWeek = "Mon–Fri"
Каждый будний день в 6:55 по парижскому времени	minute = "55", hour = "6", dayOfWeek = "Mon–Fri", timezone = "Europe/Paris"
Каждую минуту каждого часа каждого дня	minute = "*", hour = "*"
Каждую секунду каждой минуты каждого часа каждого дня	second = "*", minute = "*", hour = "*"
Каждые понедельник и среду через 30 секунд после полудня	second = "30", hour = "12", dayOfWeek = "Mon, Fri"
Каждые пять минут в течение часа	minute = "*/5", hour = "*"
Каждые пять минут в течение часа	minute = "0,5,10,15,20,25,30,35,40,45,50,55", hour = "*"
Последний понедельник декабря в 3 часа дня	hour = "15", dayOfMonth = "Last Mon", month = "Dec"
За три дня до конца каждого месяца в 1 час дня	hour = "13", dayOfMonth = "–3"
Каждый второй час в течение дня, начиная с полудня каждый второй вторник каждого месяца	hour = "12/2", dayOfMonth = "2nd Tue"
Каждые 14 минут в рамках часа в 1 и 2 часа ночи	minute = "*/14", hour="1 , 2"
Каждые 14 минут в рамках часа в 1 и 2 часа ночи	minute = "0,14,28,42,56", our = "1,2"
Каждые 10 секунд в рамках минуты, начиная с 30 секунд	second = "30/10"
Каждые 10 секунд в рамках минуты, начиная с 30 секунд	second = "30,40,50"

Декларативное создание таймера

Таймеры могут быть созданы автоматически в контейнере во время развертывания на основе метаданных. Контейнер создает таймер для каждого метода, аннотированного `@javax.ejb.Schedule` или `@Schedule` (или XML-эквивалент в дескрипторе развертывания `ejb-jar.xml`). По умолчанию каждая аннотация `@Schedule` соответствует одному постоянному таймеру, но вы можете также определить непостоянные таймеры.

В листинге 8.3 показан компонент `StatisticsEJB`, в котором определены несколько методов. Метод `statisticsItemsSold()` создает таймер, который будет вызывать этот метод каждый первый день месяца в 5:30 утра. Метод `generateReport()` создает

два таймера (с помощью аннотации `@Schedule`): один из них срабатывает каждый день в 2 часа ночи, а другой — каждую среду в 2 часа дня. Метод `RefreshCache()` создает непостоянный таймер, который будет обновлять кэш каждые десять минут.

Листинг 8.3. Statistics EJB, регистрация четырех таймеров

```
@Stateless
public class StatisticsEJB {
  @Schedule(dayOfMonth = "1", hour = "5", minute = "30")
  public void statisticsItemsSold() {
    //...
  }
  @ Schedules({
    @ Schedule(hour = "2"),
    @ Schedule(hour = "14", dayOfWeek = "Wed")
  })
  public void generateReport() {
    //...
  }
  @Schedule(minute = "*/10", hour = "*", persistent = false)
  public void refreshCache() {
    // ...
  }
}
```

Программное создание таймера

Чтобы создать таймер программно, EJB необходимо получить доступ к интерфейсу `javax.ejb.TimerService` с использованием либо внедрения зависимостей, либо `SessionContext` (`SessionContext.getTimerService()`, см. раздел «Внедрение зависимостей» главы 7) или с помощью поиска JNDI. Как показано в табл. 8.5, API `TimerService` имеет несколько методов, которые позволяют создавать множество разнообразных таймеров и получать от них информацию.

Таблица 8.5. TimerService API

Аннотация	Описание
createTimer	Создает таймер на основе дат, промежутков времени или продолжительности. Эти методы не используют выражения на основе календаря
createSingle-ActionTimer	Создает таймер одного действия, который истекает в заданный момент времени или через заданный промежуток времени. Контейнер удаляет таймер после того, как метод обратного вызова был успешно вызван
createInterval-Timer	Создает таймер на основе промежутка времени, первое его срабатывание происходит в заданный момент, а все последующие — после заданного интервала
createCalendar-Timer	Создает таймер, использующий выражение на основе календаря, с помощью вспомогательного класса ScheduleExpression
getAllTimers	Возвращает список всех доступных таймеров (интерфейс javax.ejb.Timer)

Вспомогательный класс ScheduleExpression позволяет создать выражения на основе календаря программно. Вы найдете методы, относящиеся к атрибутам, определенным в табл. 8.2, и сможете запрограммировать все примеры, которые видели в табл. 8.4. Ниже приведено несколько строк кода, чтобы вы имели представление об этом:

```
new ScheduleExpression().dayOfMonth("Mon").month("Jan");
new ScheduleExpression().second("10,30,50").minute("*/5").hour("10-14");
new ScheduleExpression().dayOfWeek("1,5").timezone("Europe/Lisbon");
new ScheduleExpression().dayOfMonth(customer.getBirthDay())
```

Все методы службы TimerService (createSingleActionTimer, createCalendarTimer и т. д.) возвращают объект типа javax.ejb.Timer, который содержит информацию о созданном таймере (когда он был создан, является ли постоянным и т. д.). Объект класса Timer также позволяет EJB отменить таймер до его истечения. По истечении времени контейнер вызывает соответствующий метод @javax.ejb.Timeout компонента, передавая объект класса Timer. Компонент может иметь не более одного метода, аннотированного @Timeout.

Когда CustomerEJB (см. листинг 8.4) создает новый клиент в системе (метод createCustomer()), он создает таймер по календарю на основе даты рождения заказчика. Таким образом, с каждым годом контейнер вызовет компонент для создания и отправки заказчику электронного письма, содержащего поздравление с днем рождения. Для того чтобы сделать это, компонент, не сохраняющий состояние, сначала должен внедрить ссылку на службу таймера (с помощью @Resource). Метод CreateCustomer() сохраняет клиента в базе данных и использует день и месяц рождения для создания ScheduleExpression. Календарный таймер создается и, основываясь на объектах классов ScheduleExpression и customer, использует метод TimerConfig. Метод new TimerConfig(customer, true) настраивает устойчивый таймер (что указано параметром true), который передает объект типа customer.

Листинг 8.4. Класс CustomerEJB, создающий таймер программно

```
@Stateless
public class CustomerEJB {
  @Resource
  TimerService t imerService;
  @PersistenceContext(unitName = "chapter08PU")
  private EntityManager em;

  public void createCustomer(Customer customer) {
    em.persist(customer);
    ScheduleExpression birthDay = new ScheduleExpression(). ➥
      dayOfMonth(customer.getBirthDay()).month(customer.getBirthMonth());
    timerService.createCalendarTimer(birthDay, new TimerConfig(customer, true));
  }
  @Timeout
  public void sendBirthdayEmail(Timer timer) {
    Customer customer = (Customer) timer.getInfo();
    // ...
  }
}
```

После создания таймера контейнер будет вызывать метод `@Timeout` (`sendBirthdayEmail()`) каждый год. В него будет передаваться объект типа `Timer`. Поскольку таймер был сериализован вместе с объектом типа `customer`, метод может получить к нему доступ, вызвав метод `getInfo()`. Обратите внимание, что компонент может иметь не более одного метода, аннотированного `@Timeout`, для работы с таймерами, создаваемыми программно.

Авторизация

Основная цель модели безопасности EJB заключается в управлении доступом к бизнес-коду. В Java EE аутентификация обрабатывается на веб-уровне (или в клиентском приложении), принципал и его роль передаются на уровень EJB, а EJB проверяет, может ли аутентифицированный пользователь получить доступ к методу, основываясь на заданной роли. Авторизация может быть реализована либо декларативным, либо программным образом.

Если вы выберете декларативную авторизацию, то контроль доступа будет осуществляться контейнером EJB. При программной авторизации контроль доступа производится в коде с использованием JAAS API.

Декларативная авторизация

Декларативная авторизация может быть определена в компоненте с помощью аннотаций или дескриптора развертывания XML. Декларативная авторизация включает в себя объявление ролей, назначение уровня доступа методам (или всему компоненту) либо временное изменение сущности безопасности. Эти элементы управления созданы с помощью аннотаций, описанных в табл. 8.6. Каждая аннотация может быть использована как для компонента, так и для метода.

Таблица 8.6. Аннотации, связанные с безопасностью

Аннотация	Компонент	Метод	Описание
@PermitAll	X	X	Означает, что заданный метод (или целый компонент) доступен для всех (доступ для всех ролей)
@DenyAll	X	X	Означает, что ни одна роль не может выполнять заданный метод или все методы компонента (доступа нет ни у одной роли). Это может быть полезно, если вы хотите запретить доступ к методу в определенных условиях (например, метод launchNuclearWar() должен быть доступен только на производстве, но не при тестировании)
@RolesAllowed	X	X	Означает, что существует список ролей, позволяющих запускать заданный метод (или весь компонент)
@DeclareRoles	X		Определяет роли для проверки безопасности
@RunAs	X		Временно назначает новую роль принципала

Аннотация @RolesAllowed используется для того, чтобы указать, каким ролям будет доступен метод. Она может быть применена к конкретному методу или всему компоненту (в этом случае все бизнес-методы унаследуют эту роль компонента). Эта аннотация может принимать либо один экземпляр класса String (только одна роль может получить доступ к методу), либо массив экземпляров класса String (любая из ролей может получить доступ к методу). Аннотация @DeclareRoles, которую мы рассмотрим далее в этом разделе, может быть использована для объявления других ролей.

ПРИМЕЧАНИЕ

Аннотации безопасности (@RolesAllowed, @DenyAll и т. д.) являются частью спецификации Common Annotations 1.2 (JSR 250) и располагаются в пакете javax.annotation.security.

В листинге 8.5 показан класс ItemEJB, использующий аннотацию @RolesAllowed на уровне компонента и методов. Этот код указывает, что любой метод доступен принципалу, связанному с одной из следующих ролей: user, employee или admin. Метод deleteBook() переопределяет настройки уровня класса, открывая доступ только роли admin.

Листинг 8.5. Не сохраняющий состояние компонент, который открывает доступ определенным ролям

```
@Stateless
@RolesAllowed({"user", "employee", "admin"})
public class ItemEJB {
  @PersistenceContext(unitName = "chapter08PU")
  private EntityManager em;
  public Book findBookById(Long id) {
    return em.find(Book.class, id);
  }
  public Book createBook(Book book) {
    em.persist(book);
    return book;
  }
  @RolesAllowed("admin")
  public void deleteBook(Book book) {
    em.remove(em.merge(book));
  }
}
```

Аннотация @RolesAllowed определяет список ролей, которым разрешен доступ к методу. Аннотации @PermitAll и @DenyAll применяются для любой роли. Таким образом, вы можете использовать аннотацию @PermitAll, чтобы отметить EJB, или метод, чтобы он вызывался для любой роли. И наоборот, @DenyAll запрещает любым ролям доступ к методу.

Как вы можете видеть в листинге 8.6, поскольку метод findBookById() аннотирован @PermitAll, он теперь может быть доступен для любой роли, а не только user, employee или admin. С другой стороны, метод findConfidentialBook() не доступен никому (@DenyAll).

Листинг 8.6. Не сохраняющий состояние компонент, который использует аннотации @PermitAll и @DenyAll

```
@Stateless
@RolesAllowed({"user", "employee", "admin"})
public class ItemEJB {
  @PersistenceContext(unitName = "chapter08PU")
  private EntityManager em;
  @PermitAll
  public Book findBookById(Long id) {
    return em.find(Book.class, id);
  }
  public Book createBook(Book book) {
    em.persist(book);
    return book;
  }
  @ RolesAllowed("admin")
  public void deleteBook(Book book) {
    em.remove(em.merge(book));
  }
  @DenyAll
  public Book findConfidentialBook(Long secureId) {
    return em.find(Book.class, secureId);
  }
}
```

Аннотация `@DeclareRoles` немного отличается, поскольку она не управляет доступом. Она позволяет декларировать роли для всего приложения. При развертывании EJB в листинге 8.6 контейнер автоматически объявляет роли `user`, `employee` и `admin`, проверив аннотацию `@RolesAllowed`. Но вы можете объявить другие роли в домене безопасности для всего приложения (а не только для одного EJB) с помощью аннотации `@DeclareRoles`. Эта аннотация, которая применяется только на уровне класса, принимает массив ролей и объявляет их в домене безопасности. На самом деле роли безопасности объявляются с использованием одной из этих двух аннотаций или их комбинации. Если применяются обе аннотации, набор ролей объявляется с помощью обеих. Обычно мы объявляем роли для всего приложения, поэтому в данном случае имеет смысл объявить роли в дескрипторе развертывания, нежели с помощью аннотации `@DeclareRoles`.

При развертывании класса `ItemEJB` в листинге 8.7 объявляются пять ролей (`HR`, `salesDpt`, `user`, `employee` и `admin`). Затем с помощью аннотации `@RolesAllowed` некоторые из этих ролей получают доступ к определенным методам (как объяснялось ранее).

Листинг 8.7. Определение ролей для компонента, не сохраняющего состояние

```
@Stateless
@DeclareRoles({"HR", "salesDpt"})
@RolesAllowed({"user", "employee", "admin"})
public class ItemEJB {
  @PersistenceContext(unitName = "chapter08PU")
  private EntityManager em;
  public Book findBookById(Long id) {
```

```
    return em.find(Book.class, id);
  }
  public Book createBook(Book book) {
    em.persist(book);
    return book;
  }
  @RolesAllowed("admin")
  public void deleteBook(Book book) {
    em.remove(em.merge(book));
  }
}
```

Последняя аннотация, `@RunAs`, может пригодиться, если вам нужно временно назначить новую роль существующему принципалу. Возможно, вам понадобится сделать это, если вы, например, вызываете другой EJB внутри вашего метода, но этот EJB требует иной роли.

Например, `ItemEJB` в листинге 8.8 разрешает доступ ролям `role`, `employee` и `admin`. Когда одна из этих ролей обращается к методу, метод запускается с временной ролью `inventoryDpt` (`@RunAs("inventoryDpt")`). Это означает, что при вызове метода `createBook()` метод `InventoryEJB.addItem()` будет вызван с ролью `inventoryDpt`.

Листинг 8.8. Не сохраняющий состояние компонент, запущенный с другой ролью

```
@Stateless
@RolesAllowed({"user", "employee", "admin"})
@RunAs("inventoryDpt")
public class ItemEJB {
  @PersistenceContext(unitName = "chapter08PU")
  private EntityManager em;
  @EJB
  private InventoryEJB inventory;
  public List<Book> findBooks() {
    TypedQuery<Book> query = em.createNamedQuery("findAllBooks", Book.class);
    return query.getResultList();
  }
  public Book createBook(Book book) {
    em.persist(book);
    inventory.addItem(book);
    returnbook;
  }
}
```

Как вы можете видеть, декларативная авторизация дает вам легкий доступ к политике аутентификации. Но что, если необходимо предоставить параметры безопасности индивида или применить какую-либо бизнес-логику на основе текущей роли принципала? Здесь вам поможет программная авторизация.

Программная авторизация

Декларативная авторизация охватывает большинство вопросов безопасности, которые необходимо решить приложению. Но иногда нужно настроить более тонкую

авторизацию (позволив запустить блок кода, а не целый метод, разрешить или запретить доступ к индивиду вместо роли и т. д.). Вы можете использовать программную авторизацию, чтобы выборочно разрешать или блокировать доступ к роли или принципалу. Это становится возможно потому, что у вас есть прямой доступ к интерфейсу JAAS `java.security.Principal`, а также к контексту EJB, что позволяет проверить роль принципала в коде.

Интерфейс `SessionContext` определяет следующие методы, связанные с безопасностью:

- `isCallerInRole()` — возвращает переменную типа `boolean` и проверяет, имеет ли вызывающая сторона заданную роль безопасности;
- `getCallerPrincipal()` — возвращает `java.security.Principal`, который идентифицирует вызывающую сторону.

Чтобы понять, как применять эти методы, рассмотрим пример. Класс `ItemEJB` в листинге 8.9 не использует декларативные методы безопасности, но нуждается в выполнении какой-то проверки программно. Прежде всего, компоненту необходимо получить ссылку на его контекст (с помощью аннотации `@Resource`). В связи с этим метод `deleteBook()` может проверить, имеет ли вызывающая сторона роль `admin`. Если результат проверки отрицательный, генерируется исключение `java.lang.SecurityException`, уведомляющее пользователя о нарушении авторизации. Метод `createBook()` выполняет бизнес-логику с использованием ролей и принципалов. Обратите внимание, что метод `getCallerPrincipal()` возвращает объект типа `Principal`, который имеет имя. Метод проверяет, соответствует ли имя принципала значению `"paul"`, а затем устанавливает значение `"special user"` в сущности книги.

Листинг 8.9. Компонент, использующий программную безопасность

```
@Stateless
public class ItemEJB {
  @PersistenceContext(unitName = "chapter08PU")
  private EntityManager em;
  @Resource
  private SessionContext ctx;
  public void deleteBook(Book book) {
    if (!ctx.isCallerInRole("admin"))
      throw new SecurityException("Разрешено только администраторам");
    em.remove(em.merge(book));
  }

  public Book createBook(Book book) {
    if (ctx.isCallerInRole("employee") && !ctx.isCallerInRole("admin")) {
      book.setCreatedBy("employee only");
    } else if (ctx.getCallerPrincipal().getName().equals("paul")) {
      book.setCreatedBy("special user");
    }
    em.persist(book);
    returnbook;
  }
}
```

Все вместе

В разделе «Все вместе» главы 4 я продемонстрировал разработку сущности Book (показанную в листинге 4.3), которая отображается в базе данных Derby. Тогда я показал вам класс Main (см. листинг 4.4), который использует менеджер сущностей для сохранения книги и получения всех книг из базы данных (применяя явное разграничение транзакции: tx.begin() и tx.commit()).

В следующем примере описывается та же самая ситуация, но класс Main, использованный в главе 4, заменен сессионным компонентом, не сохраняющим состояние (BookEJB). EJB является транзакционным по своей природе (что вы увидите в следующей главе), так что наш сессионный компонент будет обрабатывать CRUD-операции для сущности Book с помощью транзакций, управляемых контейнером (CMT). Я также добавлю синглтон (DatabasePopulator), который будет заполнять при запуске базу данных, так что в ней всегда будут данные, а также CDI-производитель, способный внедрить устойчивый объект (с помощью аннотации @Inject). BookEJB, сущность Book, синглтон DatabasePopulator, CDI-производитель и все необходимые файлы конфигурации XML затем будут упакованы и развернуты в GlassFish. BookEJB будет вызываться удаленно классом Main (рис. 8.4), а также тестироваться встроенным контейнером (BookEJBIT).

Для того чтобы использовать транзакции, сессионный компонент должен получить доступ к базе данных через источник данных (jdbc/chapter08DS), который должен быть развернут в GlassFish и связан с базой данных chapter08DB (это делается с помощью аннотации @DataSourceDefinition, что вы увидите позже).

Структура каталога для проекта следует соглашениям Maven, так что вы должны поместить классы и файлы в следующие каталоги:

- src/main/java — для сущности Book, BookEJB, интерфейса BookEJBRemote, а также классов DatabasePopulator и DatabaseProducer;
- src/main/resources — для файла persistence.xml, содержащего устойчивый объект, используемый для базы данных Derby, а также для файла beans.xml, который заставляет срабатывать CDI;
- src/test/java — для класса BookEJBIT, предназначенного для тестирования;
- pom.xml — объектная модель Maven Project Object Model (POM), описывающая проект, ее зависимости от других внешних модулей и компоненты.

Написание сущности Entity

Листинг 8.10 ссылается на сущность Book, описанную в главе 4 (см. листинг 4.3), поэтому я не буду объяснять ее подробно (она использует аннотации JPA и Bean Validation). Одно отличие будет заключаться в том, что сущность Book в листинге 8.10 реализует интерфейс Serializable, поскольку она должна быть доступна удаленно из класса Main.

Листинг 8.10. Сущность Book, на которую наложены ограничения NamedQuery и BeanValidation

```
@Entity
@NamedQuery(name = FIND_ALL, query = "SELECT b FROM Book b")
```

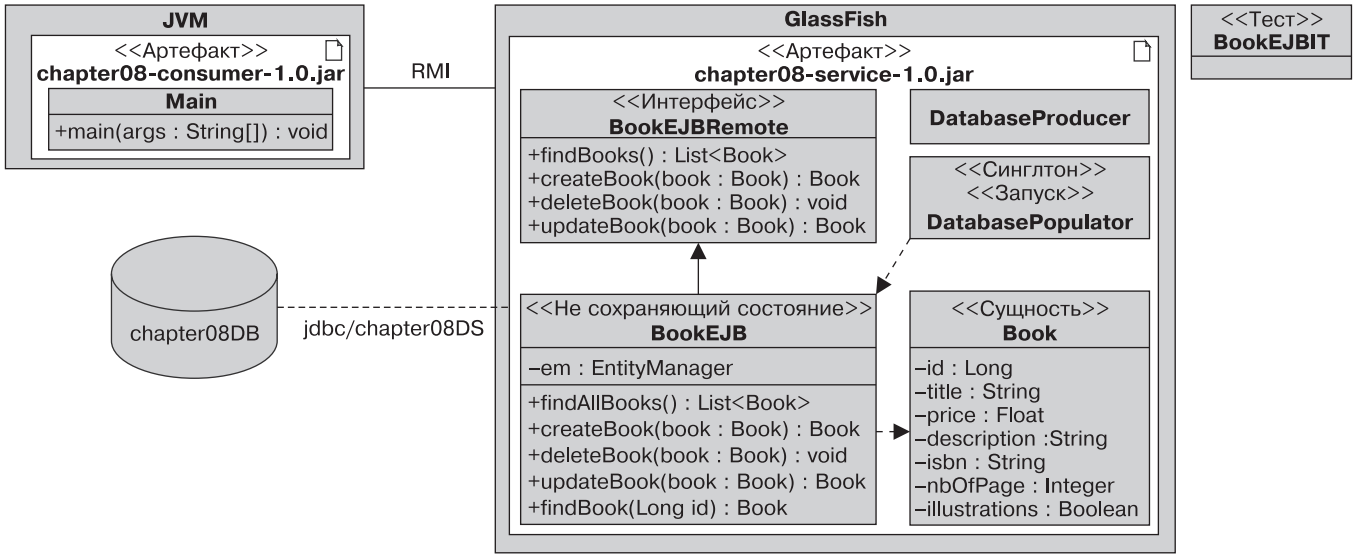


Рис. 8.4. Объединяем все вместе

```
public class Book implements Serializable {
  public static final String FIND_ALL = "Book.findAllBooks";
  @Id @GeneratedValue
  private Long id;
  @NotNull
  @Column(nullable = false)
  private String title;
  private Float price;
  @Size(max = 2000)
  @Column(length = 2000)
  private String description;
  private String isbn;
  private Integer nbOfPage;
  privateBooleanillustrations;
  // Конструкторы и методы доступа
}
```

Написание сеансового компонента BookEJB, не сохраняющего состояние

BookEJB является сессионным компонентом, не сохраняющим состояние, который действует как фасад и обрабатывает операции CRUD в сущности Book. В листинге 8.11 показан класс Java, который должен быть аннотирован @javax.ejb.Stateless. Он не предоставляет интерфейса для просмотра благодаря аннотации @LocalBean и реализует интерфейс BookEJBRemote (см. листинг 8.12). EJB получает ссылку на менеджер сущностей с использованием аннотации @Inject, поскольку объект класса EntityManager создается объектом класса DatabaseProducer (листинг 8.13). Приведу небольшие разъяснения по каждому бизнес-методу:

- findBooks — использует именованный запрос findAllBooks, определенный в сущности Book, чтобы получить все экземпляры книг из базы данных;
- createBook — принимает объект типа Book (которая не может иметь значение null) в качестве параметра, сохраняет его в базе данных и затем возвращает его;
- updateBook — принимает отдельные объекты типа Book (которые не могут иметь значение null) в качестве параметра и объединяет их. С помощью метода merge() объект прикрепляется к менеджеру сущностей и синхронизируется с базой данных;
- deleteBook — перед удалением объекта типа Book из базы данных этот метод должен повторно прикрепить объект к менеджеру сущностей (с помощью метода merge()), а затем удалить его.

Листинг 8.11. Сессионный компонент, не сохраняющий состояние, действует как фасад для операций CRUD

```
@Stateless
@LocalBean
public class BookEJB implements BookEJBRemote {
  @Inject
  private EntityManager em;
```

```
  public List<Book> findBooks() {
    TypedQuery<Book> query = em.createNamedQuery(FIND_ALL, Book.class);
    return query.getResultList();
  }
  public @NotNull Book createBook(@NotNull Book book) {
    em.persist(book);
    return book;
  }
  public @NotNull Book updateBook(@NotNull Book book) {
    return em.merge(book);
  }
  public void deleteBook(@NotNull Book book) {
    em.remove(em.merge(book));
  }
}
```

Основные различия между классом `Main`, определенным в главе 4 (см. листинг 4.4), и классом листинга 8.11 является то, что экземпляр `EntityManager` внедряется непосредственно в сессионный компонент вместо того, чтобы использовать для его создания `EntityManagerFactory`. Контейнер EJB работает с жизненным циклом `EntityManager`, поэтому внедряет его экземпляр и затем закрывает его при уничтожении EJB. Кроме того, вызовы JPA более не оборачиваются между вызовами `tx.begin()` и `tx.commit()`, так как методы сессионного компонента неявно считаются транзакционными. Это поведение по умолчанию известно как CMT, мы рассмотрим его в следующей главе.

Поскольку `BookEJB` вызывается удаленно классом `Main`, `BookEJB` должен реализовать удаленный интерфейс. Единственное различие между обычным интерфейсом `Java` и удаленным интерфейсом заключается в наличии аннотации `@Remote`, что показано в листинге 8.12.

Листинг 8.12. Удаленный интерфейс

```
@Remote
public interface BookEJBRemote {
  List<Book> findBooks();
  Book createBook(Book book);
  void deleteBook(Book book);
  Book updateBook(Book book);
}
```

Написание CDI DatabaseProducer

В листинге 8.11 в `BookEJB` внедрен объект типа `EntityManager` с помощью аннотации `@Inject`. Как вы теперь знаете, JPA позволяет приложению иметь несколько блоков хранения, которые можно различить по именам (в данном случае `"chapter08PU"`). Поскольку аннотация `@Inject` не может принять строку в качестве параметра (вы не можете написать `@Inject("chapter08PU")`), единственный способ внедрить объект типа `EntityManager` — написать следующий код: `@PersistenceContext(unitName = "chapter08PU")`. Благодаря CDI-производителям (представленным в главе 2) компонент в листин-

ге 8.13 производит объект типа EntityManager, который может быть внедрен с помощью аннотации @Inject.

Листинг 8.13. CDI компонент, производящий объект типа EntityManager

```
public class DatabaseProducer {
  @Produces
  @PersistenceContext(unitName = "chapter08PU")
  private EntityManager em;
}
```

Блок хранения для BookEJB

В главе 4 модуль хранения (см. листинг 4.5) должен был определить драйвер JDBC, URL для JDBC, пользователя и пароль для подключения к базе данных Derby, поскольку транзакциями управляло приложение (transaction-type = "RESOURCE_LOCAL"). В среде, управляемой контейнером, не приложение, а контейнер управляет EJB и транзакциями. Поэтому transaction-type для блока хранения (листинг 8.14) имеет значение "JTA". Другое отличие заключается в том, что в листинге 4.5 нам приходилось вручную указывать, какой поставщик JPA следует использовать (EclipseLink), а также список всех сущностей, которыми должен управлять поставщик. В среде, управляемой контейнером, мы работаем со стандартным поставщиком JPA (он идет в комплекте с контейнером EJB). Во время развертывания контейнер анализирует архив и находит все сущности, которыми он должен управлять (не нужно явно использовать элемент <class> в persistence.xml).

Листинг 8.14. Блок хранения, использующий источник данных chapter08DS

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" ➥
             xmlns:xsi=" http://www.w3.org/2001/XMLSchema-instance" ➥
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence ➥
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
             version="2.1">
  <persistence-unit name="chapter08PU" transaction-type="JTA">
    <jta-data-source>java:global/jdbc/chapter08DS</jta-data-source>
    <properties>
      <property name="eclipselink.target-database" value="DERBY"/>
      <property name="eclipselink.ddl-generation" value="drop-and-create-
tables"/>
      <property name="eclipselink.logging.level" value="INFO"/>
    </properties>
  </persistence-unit>
</persistence>
```

В листинге 8.11 в BookEJB внедряется ссылка на объект типа EntityManager, связанный с блоком хранения chapter08PU. Этот блок хранения (заданный в листинге 8.14) должен определить имя источника данных, к которому следует подключиться (jdbc/chapter08DS), не указывая при этом никакие свойства доступа (URL, JDBC-драйвер и т. д.).

Написание DatabasePopulator и определение источника данных

Источник данных jdbc/chapter08DS, необходимый для блока хранения, должен быть создан внутри контейнера EJB. Существует несколько способов сделать это. Самый простой — аннотировать с помощью @DataSourceDefinition любой управляемый компонент (листинг 8.15). Контейнер развернет компонент и создаст источник данных. Другой способ состоит в использовании интерфейса GlassFish, что вы увидите дальше.

Листинг 8.15. Синглтон, развертывающий источник данных и инициализирующий базу данных при запуске

```
@Singleton
@Startup
@DataSourceDefinition(
  className = "org.apache.derby.jdbc.EmbeddedDataSource",
  name = "java:global/jdbc/chapter08DS",
  user = "app",
  password = "app",
  databaseName = "chapter08DB",
  properties = {"connectionAttributes=;create=true"}
)
public class DatabasePopulator {
  @Inject
  private BookEJB bookEJB;
  private Book h2g2;
  private Book lord;
  @PostConstruct
  private void populateDB() {
    h2g2 = new Book("Изучаем Java EE 7", 35F, "Великая книга", ➥
"1-8763-9125-7", 605, true);
    lord = new Book("Властелин колец", 50.4f, "Фэнтези", "1-84023-742-2", ➥
1216, true);
    bookEJB.createBook(h2g2);
    bookEJB.createBook(lord);
  }
  @PreDestroy
  private void clearDB() {
    bookEJB.deleteBook(h2g2);
    bookEJB.deleteBook(lord);
  }
}
```

Синглтон DatabasePopulator (см. листинг 8.15) используется для инициализации некоторых данных при запуске (@Startup). Во время развертывания контейнер инициализирует синглтон и запустит метод populateDB(), поскольку тот имеет аннотацию @PostConstruct. Этот метод использует BookEJB для создания нескольких книг в базе данных. При выключении контейнер вызовет метод clearDB() (имеющий аннотацию @PostConstruct) для удаления книг из базы данных.

Вместо того чтобы использовать аннотацию `@DataSourceDefinition` для создания источника данных во время развертывания, вы также можете создать его с помощью консоли администрирования GlassFish или из командной строки. Использование командной строки — очень быстрый и легкий способ. Лишь убедитесь, что Derby и GlassFish запущены до введения следующих команд.

ПРИМЕЧАНИЕ

@DataSourceDefinition определяет источник данных и, как вы увидите в главе 13 (листинг 13.18), JMS 2.0 использует тот же механизм для определения ресурсов (ConnextionFactory и Destination) благодаря аннотациям @JMSConnectionFactoryDefinition и @JMSDestinationDefinition.

Перед созданием источника данных вам понадобится пул соединений. GlassFish поставляется с набором уже определенных пулов, которые вы можете использовать. Или же можете создать собственный пул с помощью следующей команды:

```
$ asadmin create-jdbc-connection-pool ➥
  --datasourceclassname=org.apache.derby.jdbc.ClientDataSource ➥
  --restype=javax.sql.DataSource ➥
  --property portNumber=1527:password=APP:user=APP:serverName=localhost: ➥
  databaseName=chapter08DB:connectionAttributes=;create\=true Chapter08Pool
```

Эта команда создает `Chapter08Pool` с использованием источника данных Derby и набора свойств для соединения с базой данных: название (`chapter08DB`), сервер (`localhost`) и порт (`1527`), который он слушает, пользователь (`APP`) и пароль (`APP`) для подключения. Если вы сейчас опросите этот источник данных, Derby создаст базу данных автоматически (потому что вы установили значения `connectionAttributes=;create\=TRUE`). Чтобы опросить источник данных, используйте следующую команду:

```
$ asadmin ping-connection-pool Chapter08Pool
```

После успешного выполнения этой команды вы должны увидеть каталог chapter08DB на жестком диске в том месте, где Derby хранит свои данные. Будут созданы база данных и пул соединений, и теперь вам нужно объявить источник данных `jdbc/chapter08DS` и связать его с только что созданным пулом следующим образом:

```
$ asadmin create-jdbc-resource --connectionpoolid ➥
  Chapter08Pool jdbc/chapter08DS
```

Для того чтобы получить список всех источников данных, хранящихся с помощью GlassFish, введите такую команду:

```
$ asadmin list-jdbc-resources
```

Написание интеграционного теста BookEJBIT

В версии 1.x и 2.x EJB интеграционное тестирование нашего `BookEJB` было бы затруднительно: вам пришлось бы использовать особенности определенных серверов приложений или внести изменения в код. С новым встроенным контейнером EJB становится проверяемым классом так же просто, как и любой другой класс, так как

он может работать в нормальной среде Java SE. Единственное, что необходимо сделать, — добавить конкретный файл с расширением JAR в каталог, где хранится ваш файл, как это делается в файле `pom.xml` (он показан ниже, в листинге 8.17) с помощью зависимости `glassfish-embedded-all`.

В главе 4 я объяснил, как можно провести интеграционное тестирование для сущности со встроенной базой данных, поэтому я не буду вдаваться в подробности по данному вопросу. Для того чтобы произвести интеграционное тестирование EJB, вам следует использовать встроенную в память базу данных Derby, различные блоки хранения и встроенный в память контейнер EJB. Все эти компоненты уже готовы и работают в том же процессе, требуется только создать тестовый класс JUnit (листинг 8.16). Этот тест инициализирует `EJBContainer(EJBContainer.createEJBContainer())`, получает контекст JNDI, ищет EJB и использует его для создания и удаления книг, а также их извлечения во встроенную в память базу данных и из нее. Благодаря конструкции `try-with-resources` встроенный контейнер закрывается автоматически в конце блока `try`.

Листинг 8.16. Интеграционное тестирование BookEJB с использованием встраиваемого контейнера

```
public class BookEJBIT {
  @Test
  public void shouldCreateABook() throws Exception {
    Map<String, Object> properties = new HashMap<>();
    properties.put(EJBContainer.MODULES, new File("target/classes"));
    try (EJBContainer ec = EJBContainer.createEJBContainer(properties)) {
      Contextctx = ec.getContext();
      // Проверяет зависимости JNDI (источники данных и EJB)
      assertNotNull(ctx.lookup("java:global/jdbc/chapter08DS"));
      assertNotNull(
ctx.lookup("java:global/classes/BookEJB!org.agoncal.book.javaee7.chapter08.
BookEJBRemote"));
      assertNotNull(
ctx.lookup("java:global/classes/BookEJB!org.agoncal.book.javaee7.chapter08.
BookEJB"));
      // Ищет EJB
      BookEJB bookEJB = (BookEJB)
ctx.lookup("java:global/classes/BookEJB!org.agoncal.book.javaee7.chapter08.
BookEJB");
      // Ищет все книги и убеждается, что их две (они добавлены
      // в базу с помощью DBPopulator)
      assertEquals(2, bookEJB.findBooks().size());
      // Создает книгу
      Book book = new Book("H2G2", 12.5F, "Научная фантастика", "1-24561-799-0",
354, false);
      // Сохраняет книгу в базе данных
      book = bookEJB.createBook(book);
      assertNotNull("ID не может быть пустым", book.getId());
      // Ищет все книги и убеждается, что их стало на одну больше
      assertEquals(3, bookEJB.findBooks().size());
      // Удаляет созданную книгу
```

```
        bookEJB.deleteBook(book);
        // Ищет все книги и убеждается, что их стало на одну меньше
        assertEquals(2, bookEJB.findBooks().size());
      }
    }
}
```

Метод `ShouldCreateABook()` является интеграционным тестом, который проверяет общее количество книг (оно должно быть равно двум, потому что класс `DatabasePopulator` инициализировал базу данных двумя книгами), создает новую книгу, убеждается, что общее количество книг увеличилось на единицу, удаляет одну и проверяет, стало ли общее количество книг снова равным двум. Для этого тест создает контейнер EJB, получает контекст JNDI и использует его для поиска объекта типа `BookEJB` (для того чтобы извлекать и создавать книги).

Компиляция, тестирование и упаковка с помощью Maven

Теперь вы можете использовать Maven для того, чтобы скомпилировать сущность `Book`, `BookEJB`, интерфейс `BookEJBRemote`, синглтон `DatabasePopulator` и класс CDI `DatabaseProducer`. Далее Maven упаковывает все это в один файл с расширением `.jar` с блоком хранения (`persistence.xml`), а также файл CDI `beans.xml`. В листинге 8.17 Maven использует файл `pom.xml` для описания проекта и внешних зависимостей. Этому примеру необходима зависимость `glassfish-embedded-all`, которая поставляется с Java EE 7 API, а также JUnit для проведения интеграционных тестов. Классы будут скомпилированы и упакованы в файл `chapter08-service-1.0.jar`. Следует также уведомить Maven о том, что вы будете использовать Java SE 7, сконфигурировав `maven-compiler-plugin` так, как это показано в листинге 8.17.

Листинг 8.17. Файл pom.xml для компиляции, тестирования и пакет EJB

```
<?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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <artifactId>chapter08</artifactId>
    <groupId>org.agoncal.book.javaee7</groupId>
    <version>1.0</version>
  </parent>
  <groupId>org.agoncal.book.javaee7.chapter08</groupId>
  <artifactId>chapter08-service</artifactId>
  <version>1.0</version>

  <dependencies>
    <dependency>
      <groupId>org.glassfish.main.extras</groupId>
      <artifactId>glassfish-embedded-all</artifactId>
```

```
      <version>4.0</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.5.1</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>2.12.4</version>
        <executions>
          <execution>
            <id>integration-test</id>
            <goals>
              <goal>integration-test</goal>
              <goal>verify</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Обратите внимание, что этот код включает в себя зависимость `glassfish-embedded-all` (`<scope>provided</scope>`), которая используется классом `test` (`<scope>test</scope>`) для вызова встроенного контейнера и запуска EJB.

Для компиляции и упаковки классов откройте интерпретатор командной строки и введите следующую команду Maven:

```
$ mvn package
```

Должно появиться сообщение `BUILD SUCCESSFUL`, которое сообщит о том, что компиляция и упаковка прошли успешно. Кроме того, если вы заглянете в подкаталог target, то увидите, что Maven создал файл `chapter08-services-1.0.jar`.

Вы можете выполнить интеграционное тестирование (см. листинг 8.16) с помощью надстройки Maven Failsafe, введя следующую команду Maven:

```
$ mvn integration-test
```

Развертывание на GlassFish

Теперь, когда все классы были упакованы в архив с расширением `.jar`, он может быть развернут на сервере приложений GlassFish. Прежде чем сделать это, убедитесь, что GlassFish и Derby запущены и работают.

Используйте утилиту `asadmin` для развертывания приложения на GlassFish. После выполнения этой команды будет выведено сообщение о результате выполнения операции развертывания.

```
$ asadmin deploy --force=true target\chapter08-service-1.0.jar
```

Теперь, когда EJB развернут вместе с сущностью, вспомогательным классом, а также блоком хранения в GlassFish; Derby запущен, создан источник данных, пришло время, чтобы написать и запустить класс `Main`.

Написание класса Main

Часто приложения Java EE состоят из веб-приложений, работающих в качестве клиентов для компонентов EJB (это описано в главе 10, где компоненты-подложки JSF будет вызывать EJB). А пока будем использовать обычный класс Java.

Класс `Main` (листинг 8.18) применяет JNDI для получения контекста `InitialContext`, чтобы он мог найти интерфейс `BookEJBRemote` (используя портативное имя JNDI). Помните, что этот класс `Main` выполняется вне контейнера EJB, поэтому внедрение нельзя осуществить. Метод `main()` начинает работу с поиска удаленного интерфейса EJB, отображая все книги из базы данных, создает новый экземпляр объекта `Book` и использует метод EJB `createBook()` для сохранения этой сущности. Затем он изменяет значение названия книги, обновляет книгу и удаляет ее. Поскольку код этого класса `Main` не имеет контекста хранения, сущность `Book` рассматривается как отдельный объект, который обрабатывается как обычный класс Java другим классом Java, без использования JPA. В EJB содержится контекст хранения. Кроме того, он использует менеджер сущностей для доступа к базе данных.

Листинг 8.18. Класс Main, вызывающий BookEJBRemote

```
public class Main {
  public static void main(String[] args) throws NamingException {
    // Выполняется поиск EJB
    Context ctx = new InitialContext();
    BookEJBRemote bookEJB = (BookEJBRemote) ctx.lookup("java:global/ ➥
    chapter08-service-1.0/BookEJB!org.agoncal.book.javaee7.chapter08. ➥
BookEJBRemote");
    // Получение и отображение всех книг из базы данных
    List<Book> books = bookEJB.findBooks();
    for (Book aBook : books) {
      System.out.println(aBook);
    }
```

```
        // Создается экземпляр книги
        Book book = new Book("H2G2", 12.5F, "Научная фантастика", "1-24561-799-0",
          354, false);
        book = bookEJB.createBook(book);
        System.out.println("Создана книга : " + book);
        book.setTitle("H2G2");
        book = bookEJB.updateBook(book);
        System.out.println("Создана книга : " + book);
        bookEJB.deleteBook(book);
        System.out.println("Книга удалена");
    }
}
```

Благодаря надстройке Maven Exec просто введите следующую команду, и Maven выполнит класс `Main` (см. листинг 8.18). Затем вы должны увидеть записи журнала, отображенные в консоли.

```
$ MVN Exec : Java
```

Резюме

В этой главе вы узнали, что компоненты, не сохраняющие состояния, и синглтоны имеют одинаковый жизненный цикл, а жизненный цикл компонентов, сохраняющих состояние, несколько отличается. Это потому, что компоненты, сохраняющие состояние, хранят состояние диалога с клиентом и нуждаются в его временной сериализации в надежное хранилище (пассивизации). Аннотации методов обратного вызова позволяют добавить бизнес-логику в ваш компонент до или после того, как произойдет какое-либо событие (`@PostConstruct`, `@PreDestroy` и т. д.).

Поскольку в версии EJB 3.1 возможности службы таймера были расширены, теперь она может эффективно конкурировать с другими инструментами планирования. Она происходит от утилиты cron, но более выразительна, что позволяет вам писать сложные, но читаемые выражения для планирования как декларативно, так и программно.

Что касается безопасности, вам следует иметь в виду, что на бизнес-уровне не выполняется авторизация пользователей — здесь происходит управление ролями. Декларативная безопасность поддерживается с помощью относительно небольшого количества аннотаций и позволяет охватить большинство вопросов, касающихся безопасности. Опять же вы можете переключиться на программную защиту и работать с JAAS API.

Глава 9
Транзакции

Управление транзакциями — важная задача для предприятий. Оно позволяет приложениям иметь согласованные данные и обрабатывать их надежным образом. Управление транзакциями осуществляется на низком уровне, разработчик бизнес-логики не должен заниматься этим. EJB предоставляет довольно простые службы для реализации этой особенности: можно делать это либо программно с высоким уровнем абстракции, либо декларативно, используя метаданные. С момента выхода Java EE 7 Managed Beans также имеет место подобный декларативный механизм.

Большая часть работы корпоративных приложений заключается в управлении данными: их хранении (как правило, в базе данных), получении, обработке и т. д. Часто это делается одновременно, несколько приложений пытаются получить доступ к одним и тем же данным. База данных имеет низкоуровневые механизмы сохранения параллельного доступа, такие как пессимистичные блокировки, и использует механизм транзакций для обеспечения сохранности данных. EJB также пользуется этими механизмами.

Я посвящаю первую часть этой главы описанию управления транзакциями в целом. Далее мы обсудим различные типы разграничения транзакций, поддерживаемых EJB: CMT и BTM. Закончится эта глава рассмотрением нового перехватчика транзакций, который может быть использован в Managed Beans.

Понимание транзакций

Данные имеют решающее значение для бизнеса, и они должны быть точными независимо от операций, которые вы выполняете, а также от количества приложений, одновременно получающих доступ к данным. Транзакция используется для обеспечения того, что данные хранятся в устойчивом состоянии. Она представляет собой логическую группу операций, которые должны быть выполнены в виде единого блока, также известную как единица работы. Эти операции могут включать сохранение данных в одну или несколько баз данных, отправку сообщений в MOM (промежуточное ПО, ориентированное на обработку сообщений) или вызов веб-служб. Компании полагаются на операции каждый день, работая с приложениями банкинга и электронной коммерции либо взаимодействуя с бизнесом партнеров.

Эти неделимые бизнес-операции выполняются либо последовательно, либо параллельно в течение относительно короткого промежутка времени. Каждая операция должна быть завершена успешно для того, чтобы вся транзакция была завершена успешно (мы говорим, что транзакция зафиксирована). Если одна из операций не выполняется успешно, то происходит сбой транзакции (это называется *откатом*

транзакции). Транзакции должны гарантировать определенную степень надежности и устойчивости, а также соответствовать правилам ACID.

Правила ACID

ACID — это аббревиатура, которая формируется из первых букв четырех основных свойств, определяющих надежность транзакций: атомарность (Atomicity), согласованность (Consistency), изолированность (Isolation) и надежность (Durability) (они описаны в табл. 9.1). Для того чтобы объяснить эти свойства, я воспользуюсь классическим примером банковского перевода: нужно снять сбережения с резервного счета, чтобы пополнить основной.

Таблица 9.1. Свойства ACID

Свойство	Описание
Атомарность	Транзакция состоит из одной или нескольких операций, сгруппированных в единицу работы. По завершении транзакции либо все операции выполняются успешно (фиксация), либо ни одна из них (откат) — в том случае, если произошло что-то неожиданное
Согласованность	По завершении транзакции данные остаются согласованными
Изолированность	Промежуточное состояние транзакции не демонстрируется внешним приложениям
Надежность	После того как транзакция успешно завершается, изменения должны стать видимыми другим приложениям

Вы можете себе представить последовательность операций в базе данных, выполняемых при переводе денег с одного счета на другой: с резервного счета деньги списываются с помощью оператора SQL `update`, на текущий счет деньги зачисляются с помощью другого оператора `update`, а также в другой таблице создается файл журнала, который используется для слежения за переводами. Эти операции должны выполняться над одной единицей работы (Атомарность), поскольку вы не хотите, чтобы деньги были списаны с одного счета и при этом не были зачислены на другой. С точки зрения внешних приложений, получающих доступ к счетам, результат операций должен быть виден только после выполнения всех операций (Изолированность). Если это условие выполняется, то внешнее приложение не может увидеть промежуточное состояние, когда с одного счета деньги уже списаны, а на другой еще не зачислены (если бы это произошло, приложение посчитало бы, что пользователь имеет меньше денег, чем на самом деле). Согласованность проявляется в том, что операции транзакции (как фиксация, так и откат) выполняются в рамках ограничений базы данных (например, первичных ключей, отношений или полей). После завершения перевода данные могут быть доступны из других приложений (надежность).

Условия считывания

Изолированность транзакции может быть определена с использованием различных условий чтения (грязное, повторяемое и фантомное чтение). Эти условия описы-

вают, что может произойти, когда две или более операции действуют с одними и теми же данными одновременно.

В зависимости от применяемого уровня изоляции можно полностью избежать или разрешить одновременный доступ к данным.

- *Грязное чтение* — происходит, когда транзакция считывает незафиксированные изменения, сделанные предыдущей транзакцией.
- *Повторное чтение* — происходит, когда считанные данные гарантированно будут выглядеть так же, если читать их снова в той же транзакции.
- *Фантомное чтение* — происходит, когда новые записи, добавленные в базу данных, могут быть обнаружены теми транзакциями, которые начались до операции вставки. Запросы будут включать в себя записи, добавленные другими транзакциями после того, как началась их текущая транзакция.

Уровни изоляции транзакций

Базы данных используют несколько различных методов блокировки для управления одновременным доступом к данным. Механизмы блокировки воздействуют на условия чтения, описанные ранее. Уровни изоляции обычно используются в базах данных и описывают, как блокировка применяется к данным в рамках транзакции. Рассмотрим четыре типа уровней изоляции.

- *Чтение незафиксированных данных* (менее ограничительный уровень изоляции) — транзакция может читать незафиксированные данные. Может произойти грязное, неповторяемое и фантомное чтение.
- *Чтение зафиксированных данных* — транзакция не может читать незафиксированные данные. Предотвращается грязное чтение, но не неповторяемое или фантомное.
- *Повторяемое чтение* — транзакция не может изменить данные, которые считываются другими транзакциями. Предотвращается грязное и неповторяемое чтение, но фантомное все еще может произойти.
- *Сериализуемое* (наиболее строгий уровень изоляции) — транзакция имеет эксклюзивное право на чтение. Другие транзакции не могут ни читать, ни записывать эти же данные.

Вообще говоря, если уровень изоляции становится более ограничительным, производительность системы снижается из-за того, что операции не могут получить доступ к одним и тем же данным. Тем не менее уровень изоляции обеспечивает согласованность данных. Обратите внимание, что не все СУБД реализуют все четыре уровня изоляции.

Локальные транзакции JTA

Для того чтобы транзакции работали как положено, должны присутствовать некоторые компоненты, а также должны выполняться свойства ACID. Сначала рассмотрим простейший пример приложения, выполняющего несколько изменений единого ресурса (скажем, базы данных). Когда существует только один транзакционный ресурс, все, что вам потребуется, — это локальные транзакции JTA. Локальная транзакция

ресурса — это транзакция, при осуществлении которой у вас имеется какой-то определенный ресурс, использующий свой специфический API. На рис. 9.1 показано приложение, взаимодействующее с ресурсом через менеджер транзакций и менеджер ресурсов.

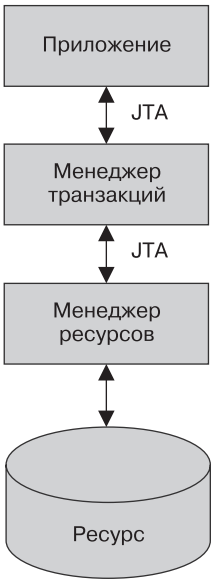


Рис. 9.1. Транзакция с участием одного ресурса

Компоненты, показанные на рис. 9.1, помогают отделить обработку транзакции от бизнес-логики самого приложения.

- *Менеджер транзакций* — основной компонент, отвечающий за управление операциями транзакций. Он создает транзакции от имени приложения, информирует менеджер ресурсов, что он участвует в сделке (операция, известная как задействование) и проводит фиксацию или откат в менеджере ресурсов.
- *Менеджер ресурсов* отвечает за управление ресурсами и их регистрацию в менеджере транзакций. Пример менеджера ресурса — драйвер для реляционной базы данных, ресурс JMS или коннектор Java.
- *Ресурс* — это постоянное хранилище, из которого вы можете читать или в который можете записывать данные (база данных, пункт назначения сообщения и т. п.).

Приложение не обязано следовать правилам ACID. Оно просто решает, фиксировать транзакцию или откатывать, а менеджер транзакций подготавливает все ресурсы для успешного осуществления этой операции.

Распределенные транзакции и XA

Как вы только что видели, операции с использованием одного ресурса (показанные на рис. 9.1) называют локальными транзакциями JTA. Однако многие корпоративные приложения используют несколько ресурсов. Возвращаясь к примеру перево-

да денежных средств, резервный и основной счета могут находиться в разных базах данных. В этом случае вам бы понадобилось управление транзакциями в нескольких ресурсах или же ресурсы должны были бы быть распределенными в вашей сети. Такие масштабные транзакции требуют специальных операций по координации при участии XA и службы транзакций Java (JTS).

На рис. 9.2 показано приложение, которое использует разграничение транзакций на нескольких ресурсах. Это означает, что в одной единице работы приложение может, например, сохранять данные в базу данных и отправить сообщение JMS.

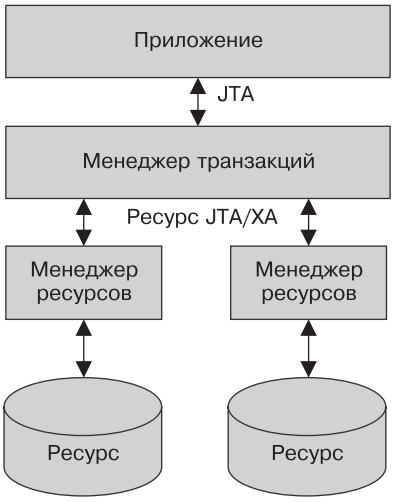


Рис. 9.2. Транзакция XA, включающая в себя два ресурса

Для того чтобы транзакции надежно выполнялись для нескольких ресурсов, менеджеру транзакций требуется использовать интерфейс управления ресурсами XA (eXtendedArchitecture — расширенная архитектура). XA — это стандарт, заданный группой Open Group (www.opengroup.org) для распределенной обработки транзакций (DTP) и обеспечивающий выполнение свойств ACID. Этот стандарт поддерживается JTA и позволяет гетерогенным менеджерам ресурсов разных производителей взаимодействовать через один интерфейс. XA использует механизм двухфазной фиксации (2pc), чтобы гарантировать, что все ресурсы либо зафиксированы, либо отменены для любой конкретной транзакции одновременно.

Для нашего примера перевода средств предположим, что деньги списываются с резервного счета из одной базы данных и транзакция завершается успешно. Далее на текущий счет, расположенный во второй базе, деньги зачисляются, но выполнить транзакцию не удается. Нам бы хотелось вернуться к первой базе данных и отменить зафиксированные изменения. Чтобы избежать этой проблемы несоответствия данных, двухфазная фиксация выполняет дополнительный подготовительный шаг перед окончательной фиксацией изменений, что показано на рис. 9.3. Во время фазы 1 каждый менеджер ресурсов получает уведомление с помощью команды «подготовки», которая говорит, что вот-вот будет произведена фиксация. Это позволяет менеджерам ресурсов объявить, могут ли они применить свои изменения. Если все они указывают,

что готовы, транзакция разрешается и выполняется, а все менеджеры ресурсов должны выполнить фиксацию во второй фазе.

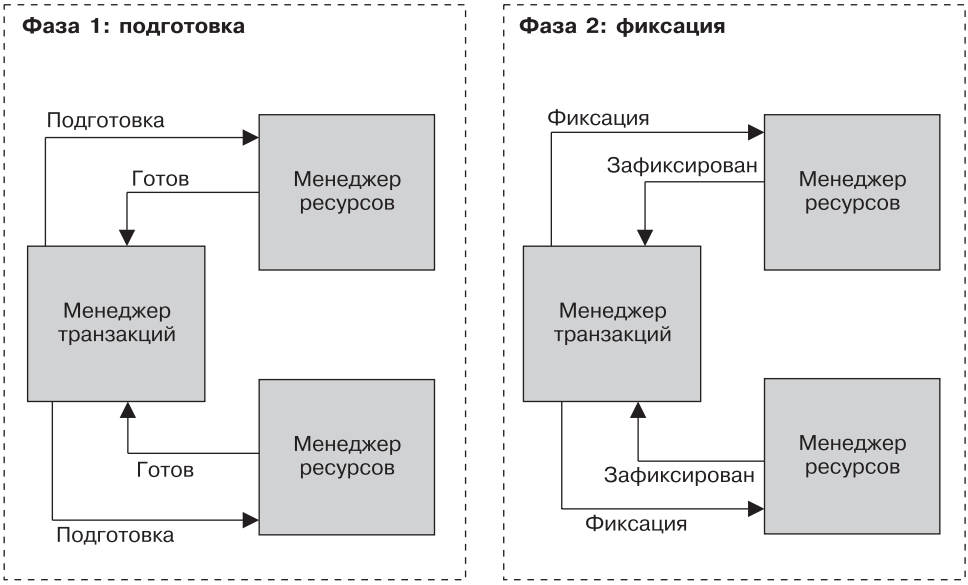


Рис. 9.3. Двухфазная фиксация

Большую часть времени ресурсы распределены по сети (рис. 9.4). Такая система использует службу JTS, которая реализует группу управления объектами (OMG) спецификации Object Transaction Service (OTS), что позволяет менеджеру транзакций участвовать в распределенных транзакциях с помощью протокола Internet Inter-ORB Protocol (IIOP).

По сравнению с рис. 9.2, где есть только один менеджер транзакций, на рис. 9.4 позволяется распространение распределенных транзакций с использованием IIOP. Текущая транзакция может быть распределена между различными компьютерами и различными базами данных от разных поставщиков. JTS предназначен для поставщиков, предоставляющих инфраструктуру систем транзакций. Как EJB-разработчику вам не придется беспокоиться об этом, просто используйте JTA, который взаимодействует с JTS на более высоком уровне.

Обзор спецификаций для работы с транзакциями

В Java EE 7, EJB и Managed Beans обрабатывают транзакции с помощью Java Transaction API (JTA), заданных в JSR 907. JTA определяет набор интерфейсов для приложений или контейнеров, предназначенных для транзакций с установленными границами. Кроме того, там определяется набор API, позволяющий работать с менеджером транзакций. Пакет `javax.transaction` задает эти интерфейсы.

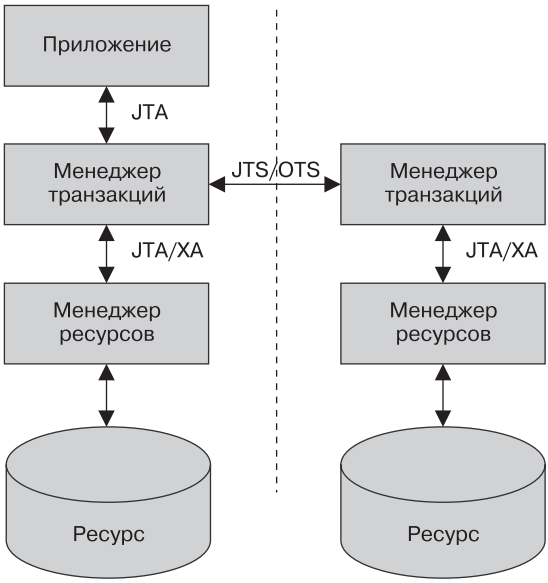


Рис. 9.4. Распределенная транзакция XA

JTS — это спецификация, предназначенная для создания менеджера транзакций, который поддерживает интерфейсы JTA на высоком уровне и стандартное отображение в Java спецификации службы транзакций объектов CORBA 1.1 на низком уровне. JTS предоставляет возможность совместимости транзакций с использованием стандартного протокола CORBA IIOP для распространения транзакций между серверами. JTS предназначен для производителей, которые обеспечивают инфраструктуру системы транзакций для промышленного промежуточного ПО.

Что касается управления транзакциями, то маловероятно, что вы захотите отдельно использовать JTS/JTA API в Java. Вместо этого вы делегируете транзакции в контейнер EJB, который содержит менеджер транзакций (он, в свою очередь, внутренне использует JTS и JTA).

Краткая история JTA

JTA является общим API для управления транзакциями в Java EE. Это позволяет начинать, фиксировать и откатывать транзакции независимо от ресурса. Если в транзакции участвует несколько ресурсов, JTA также позволяет вам выполнить транзакцию XA.

JTA 1.0 был введен в 1999 году и пережил несколько обновлений вплоть до версии 1.1 в 2002 году. Он оставался неизменным в течение десяти лет и наконец был обновлен до версии JTA 1.2 с Java EE 7.

Что нового в версии 1.2 JTA

Исторически в Java EE сделки были переданы EJB, так что разработчики не должны были использовать JTA API непосредственно в своем коде. Как JTA (JSR 907),

так и компоненты EJB (JSR 345) существовали с 1999 года. Спецификация EJB продолжала развиваться, но не JTA, которая уже была полной. С изменением Managed Beans, произошедшим в версии Java EE 7, одной из самых больших проблем была необходимость дать возможность использовать транзакции управляемым компонентам, а не только EJB. Для ее решения спецификацию JTA нужно было обновить. В версии JTA 1.2 добавлена поддержка управляемых контейнером транзакций независимо от EJB, а также аннотации `@TransactionScope` для области применения компонентов CDI.

JTA API состоит из классов и интерфейсов, сгруппированных в два пакета, описанных в табл. 9.2.

Таблица 9.2. Основные пакеты JTA

Пакет	Описание
javax.transaction	Содержит основные API JTA
javax.transaction.xa	Интерфейсы и классы, предназначенные для распределенных транзакций XA

Примеры реализации

В JTA примером реализации работы с транзакциями является менеджер транзакций GlassFish. Это модуль GlassFish, не предназначенный для использования вне сервера. Доступны и другие реализации, как коммерческие, так и с открытым исходным кодом, например JBoss Transaction Manager, Atomikos и Bitronix JTA.

Поддержка транзакций в EJB

При разработке бизнес-логики, использующей EJB, вам не придется беспокоиться о внутренней структуре менеджеров транзакций или менеджеров ресурсов, поскольку JTA абстрагируется от большинства скрытых сложностей. С помощью EJB вы очень легко сможете разработать приложение, использующее транзакции, предоставив контейнеру возможность реализовывать низкоуровневые протоколы транзакций, такие как двухфазная фиксация или распространение контекста транзакции. Контейнер EJB является менеджером транзакций, поддерживаемым JTA и JTS, который участвует в распределенных транзакциях, включающих другие контейнеры EJB и/или другие транзакционные ресурсы. В типичном приложении Java EE сессионные компоненты устанавливают границы транзакции, вызывают сущности, предназначенные для взаимодействия с базой данных или отправки JMS сообщений в контексте транзакции.

С момента своего создания модель EJB была разработана для управления транзакциями. На самом деле транзакции естественны для EJB, и по умолчанию каждый метод автоматически оборачивается в транзакцию. Это управляемые контейнером транзакции (CMT), ведь транзакции управляются контейнером EJB (эта особенность известна также как декларативное разграничение транзакций). Вы также можете решить управлять транзакциями самостоятельно с помощью транзакций, управляемых компонентом (BMT) (этот процесс называется программным раз-

граничением транзакций). Разграничение транзакций определяет, где транзакция начинается и заканчивается.

Управляемые контейнером транзакции. При декларативном управлении операциями вы делегируете проведение политики разграничения контейнеру. Вы не должны явно использовать JTA в коде (даже если используется JTA на нижнем уровне), но можете заставить контейнер разграничивать транзакции, автоматически начиная и фиксируя транзакции на основе метаданных. Контейнер EJB предоставляет службы управления транзакциями для сессионных компонентов и MDB (см. главу 13 для получения дополнительной информации об MDB). В корпоративном компоненте с транзакциями, управляемыми контейнером, сам контейнер EJB устанавливает границы этих операций.

В главе 7 вы видели несколько примеров сессионных компонентов, аннотаций и интерфейсов, но еще не встречали ничего относящегося к транзакциям. В листинге 9.1 показан код сессионного компонента, не сохраняющего состояние и использующего CMT. Как вы можете видеть, здесь не добавляются дополнительные аннотации и не реализуются особые интерфейсы. EJB является транзакционным по своей природе. При конфигурации с помощью исключений применяются все значения по умолчанию для управления транзакциями (`REQUIRED` — это атрибут, используемый по умолчанию для управления транзакциями, что объясняется далее в этом разделе).

Листинг 9.1. Компонент, не сохраняющий состояние и использующий CMT

```
@Stateless
public class ItemEJB {
  @PersistenceContext(unitName = "chapter09PU")
  private EntityManager em;
  @Inject
  private InventoryEJB inventory;
  public List<Book> findBooks() {
    TypedQuery<Book> query = em.createNamedQuery(FIND_ALL, Book.class);
    return query.getResultList();
  }
  public Book createBook(Book book) {
    em.persist(book);
    inventory.addItem(book);
    return book;
  }
}
```

Вы можете спросить, почему код в листинге 9.1 является транзакционным. Ответом станет контейнер. На рис. 9.5 показано, что происходит, когда клиент вызывает метод `createBook()`. Вызов клиента перехватывается контейнером, который проверяет непосредственно перед вызовом метода, связан ли контекст транзакции с вызовом. По умолчанию, если контекст транзакции недоступен, контейнер начинает новую транзакцию перед входом в метод, а затем вызывает метод `createBook()`. После выхода из метода контейнер автоматически фиксирует или откатывает транзакцию (если генерируется конкретный тип исключения, что вы увидите позже в разделе «Исключения и транзакции» данной главы).

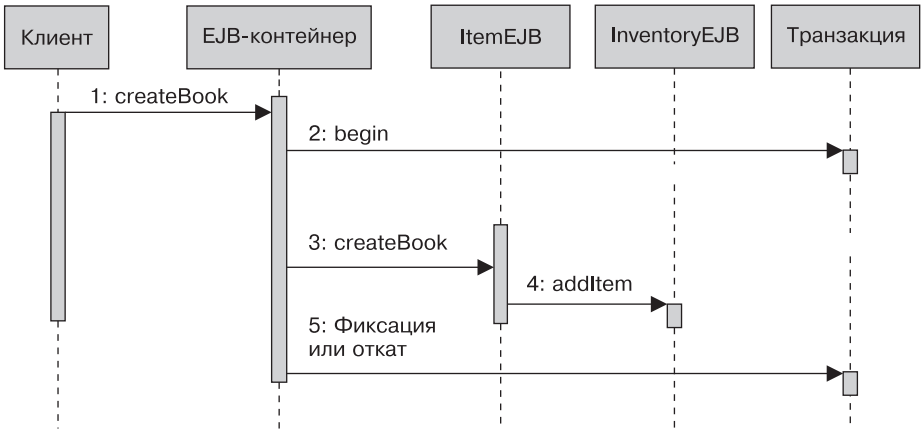


Рис. 9.5. Обработка транзакции контейнером

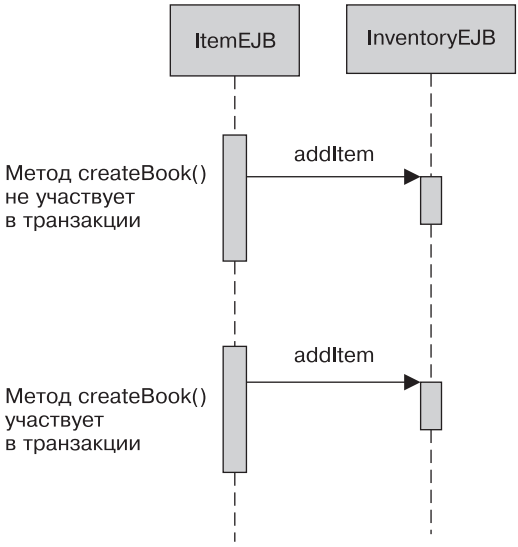
Использование на шаге `4:addItem()` контекста транзакции, применяемого на шаге `3:createBook()` (контекст может как принадлежать клиенту, так и создаваться контейнером), является поведением по умолчанию. Финальная фиксация происходит, если успешно отработали оба метода. Такое поведение может быть изменено с помощью метаданных (аннотации или дескриптора развертывания XML). В зависимости от выбранного вами атрибута транзакции (`REQUIRED`, `REQUIRES_NEW`, `SUPPORTS`, `MANDATORY`, `NOT_SUPPORTED` или `NEVER`) вы можете повлиять на то, как контейнер разграничивает операции: при запуске клиентом метода, использующего транзакции, контейнер применяет транзакцию клиента, запускает метод в новой транзакции, запускает метод без транзакции или генерирует исключение. В табл. 9.3 определены атрибуты транзакции.

Таблица 9.3. Атрибуты CMT

Атрибут	Описание
REQUIRED	Этот атрибут (задан по умолчанию) означает, что метод всегда должен вызываться внутри транзакции. Контейнер создает новую транзакцию, если метод вызывается из клиента, не использующего транзакции. Если клиент имеет контекст транзакций, бизнес-метод запускается внутри транзакции клиента. Вам следует указывать атрибут REQUIRED в том случае, когда вызываются методы, которые должны использовать транзакции, но при этом нельзя однозначно сказать, использует клиент транзакции или нет
REQUIRES_NEW	Контейнер всегда создает новую транзакцию перед тем, как начать выполнять метод. Независимо от того, имеет ли клиент контекст транзакции, контейнер временно приостанавливает эту транзакцию, создает новую, фиксирует или откатывает ее, а затем возобновляет первую транзакцию. Это означает, что успех или провал второй транзакции не имеет никакого эффекта для существующей транзакции клиента. Вам следует указывать атрибут REQUIRES_NEW, если вы не хотите выполнять откат транзакции, который мог бы затронуть клиент

Атрибут	Описание
SUPPORTS	Метод EJB наследует контекст транзакции клиента. Если контекст транзакции доступен, он используется этим методом, в противном случае контейнер вызывает метод без контекста транзакции. Вам следует указывать атрибут SUPPORTS, если у вас есть доступ к базе данных только для чтения
MANDATORY	Контейнер требует наличия транзакции перед вызовом бизнес-метода, но не станет создавать новый контекст. Если клиент имеет контекст транзакции, он используется, в противном случае генерируется исключение javax.ejb.EJBTransactionRequiredException
NOT_SUPPORTED	Метод EJB не может быть вызван при наличии контекста транзакции. Если клиент не имеет контекста транзакции, ничего не происходит, в противном случае контейнер приостанавливает транзакцию клиента, вызывает метод, а затем, как только метод отработает, возобновляет транзакцию
NEVER	Метод EJB не должен вызываться из клиента, использующего транзакции. Если клиент имеет контекст транзакции, контейнер генерирует исключение javax.ejb.EJBException

На рис. 9.6 проиллюстрированы все возможные варианты поведения, которые EJB может иметь в зависимости от наличия контекста транзакции у клиента. Например, если метод `createBook()` не имеет контекста транзакции (верхняя часть рисунка) и вызывает метод `addItem()`, имеющий атрибут `MANDATORY`, то генерируется исключение. В нижней части рис. 9.6 показаны те же комбинации, но для клиента, который имеет контекст транзакции.



Атрибут CMT	Что произойдет?
REQUIRED	Новая транзакция
REQUIRES_NEW	Новая транзакция
SUPPORTS	Нет транзакции
MANDATORY	Исключение
NOT_SUPPORTED	Нет транзакции
NEVER	Нет транзакции

Атрибут CMT	Что произойдет?
REQUIRED	Транзакция вызывающей стороны
REQUIRES_NEW	Новая транзакция
SUPPORTS	Транзакция вызывающей стороны
MANDATORY	Транзакция вызывающей стороны
NOT_SUPPORTED	Нет транзакции
NEVER	Исключение

Рис. 9.6. Два вызова InventoryEJB, осуществленные при различной политике транзакций

Чтобы применить один из этих шести атрибутов разграничения для вашего сессионного компонента, вы должны использовать аннотацию `@javax.ejb.TransactionAttribute` или дескриптор развертывания (установите элемент `<trans-attribute>` в файле `EJB-jar.xml`). Эти метаданные могут быть применены как к отдельным методам, так и ко всему компоненту. Если применять их на уровне компонента, все бизнес-методы будут наследовать это значение атрибута. В листинге 9.2 показано, как `ItemEJB` использует политику ограничения `SUPPORTS`. Кроме того, там переопределяется метод `createBook()`, который теперь будет иметь атрибут `REQUIRED`.

Листинг 9.2. Компонент, не сохраняющий своего состояния, использующий CMT

```
@Stateless
@TransactionAttribute(TransactionAttributeType.SUPPORTS)
public class ItemEJB {
  @PersistenceContext(unitName = "chapter09PU")
  private EntityManager em;
  @Inject
  private InventoryEJB inventory;
  public List<Book> findBooks() {
    TypedQuery<Book> query = em.createNamedQuery(FIND_ALL, Book.class);
    return query.getResultList();
  }
  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public Book createBook(Book book) {
    em.persist(book);
    inventory.addItem(book);
    returnbook;
  }
}
```

ПРИМЕЧАНИЕ

Контекст транзакций клиента не распространяется при асинхронном вызове метода (@Asynchronous). MDB поддерживает только атрибуты REQUIRES и NOT_SUPPORTED, что описано в главе 13.

Маркировка CMT для отката транзакции

Вы видели, что контейнер EJB разграничивает операции автоматически и вызывает операции начала, фиксации и отката от вашего имени. Но как разработчик вы, возможно, захотите предотвратить фиксируемую транзакцию, если произойдет какая-либо ошибка или сработает бизнес-правило. Важно подчеркнуть, что компонент CMT не допускает явного отката транзакций. Вместо этого вам нужно использовать контекст EJB (см. раздел «Внедрение зависимостей» главы 7) для того, чтобы проинформировать контейнер об откате.

Как вы можете видеть в листинге 9.3, `InventoryEJB` содержит метод `oneItemSold()`, который получает доступ к базе данных через менеджер хранения и посылает JMS-сообщение для того, чтобы сообщить отправляющей компании, что предмет продан и должен быть доставлен. Если количество доступных предметов равно нулю, методу необходимо явно откатить транзакцию. Чтобы сделать это, компонент, не

сохраняющий состояние, сначала должен получить контекст SessionContext с помощью внедрения зависимостей, а затем вызвать его метод setRollbackOnly(). Вызов этого метода не выполняет откат транзакции непосредственно, вместо этого в контейнере устанавливается флаг, который сигнализирует о том, что следует сделать откат транзакции в момент ее завершения. Только сессионные компоненты с разграничением с помощью CMT могут использовать этот метод (сессионные компоненты BMT откатывают транзакцию непосредственно, что показано в пункте «Транзакции, управляемые компонентом» далее).

Листинг 9.3. Компонент, не сохраняющий состояние, помечает транзакцию для отката

```
@Stateless
public class InventoryEJB {
  @PersistenceContext(unitName = "chapter09PU")
  private EntityManager em;
  @Resource
  private SessionContext ctx;
  public void oneItemSold(Item item) {
    item.decreaseAvailableStock();
    sendShippingMessage();
    if (inventoryLevel(item) == 0)
      ctx.setRollbackOnly();
  }
}
```

Аналогично компонент может вызвать метод SessionContext.getRollbackOnly(), который возвращает переменную типа boolean, чтобы определить, помечена ли текущая транзакция для отката.

Еще один способ программно информировать контейнер об откате — сгенерировать исключение определенного типа.

Исключения и транзакции

Обработка исключений в Java вводит в заблуждение с момента создания языка, поскольку она включает в себя как проверенные, так и непроверенные исключения. Связывание транзакций и исключений в EJB также весьма сложное дело. Прежде чем идти дальше, я хочу сказать, что генерация исключения в бизнес-метод не всегда пометит транзакцию как готовую для отката. Это зависит от типа исключения и метаданных, определяющих исключение. На самом деле в спецификации EJB 3.2 описаны два типа исключений.

- *Исключения приложения* — исключения связаны с бизнес-логикой, обрабатываемой EJB. Например, исключения приложения могут быть сгенерированы, если методу переданы неверные аргументы, количество предметов на складе невелико или же указан ошибочный номер кредитной карты. Генерация исключения приложения не гарантирует, что транзакция будет помечена для отката. Как будет подробно описано в табл. 9.4, контейнер не откатывает транзакции при генерации проверенных исключений (которые наследуют от класса java.lang.Exception), в случае непроверенных исключений (которые наследуют от класса RuntimeException) пометка добавляется.

- *Системные исключения* — исключения, вызванные неполадками системного уровня, такие как JNDI-ошибки, JVM-ошибки, неспособность получить соединение с базой данных и т. д. Системное исключение должно быть исключением подкласса `RuntimeException` или `java.rmi.RemoteException` (и, следовательно, подкласса `javax.ejb.EJBException`). Генерация системного исключения приводит к тому, что транзакция помечается для отката.

Исходя из этого определения, мы теперь знаем, что, если контейнер обнаружит системное исключение, такое как `ArithmeticException`, `ClassCastException`, `IllegalArgumentException` или `NullPointerException`, он откатит транзакцию. Действия при генерации исключения приложения зависят от многих факторов. В качестве примера изменим код листинга 9.3 и используем исключения приложения, как показано в листинге 9.4.

Листинг 9.4. Компонент, не сохраняющий состояние, который генерирует исключение приложения

```
@Stateless
public class InventoryEJB {
  @PersistenceContext(unitName = "chapter09PU")
  private EntityManager em;
  public void oneItemSold(Item item) throws InventoryLevelTooLowException{
    item.decreaseAvailableStock();
    sendShippingMessage();
    if (inventoryLevel(item) == 0)
      throw new InventoryLevelTooLowException();
  }
}
```

`InventoryLevelTooLowException` — это исключение приложения, потому что оно связано с бизнес-логикой метода `oneItemSold()`. Знать, хотите ли вы откатить транзакцию, — обязанность бизнес-логики. Исключение приложения наследует от проверенных и непроверенных исключений и имеет аннотацию `@javax.ejb.ApplicationException` (или эквивалент в XML в дескрипторе развертывания). Эта аннотация включает элемент `rollback`, которому может быть присвоено значение `true` для того, чтобы откатить транзакцию. В листинге 9.5 показано исключение `InventoryLevelTooLowException`, которое является аннотированным проверенным исключением.

Листинг 9.5. Исключение приложения, значение rollback = true

```
@ApplicationException(rollback = true)
public class InventoryLevelTooLowException extends Exception {
  public InventoryLevelTooLowException() { }
  public InventoryLevelTooLowException(String message) {
    super(message);
  }
}
```

Если `InventoryEJB` из листинга 9.4 сгенерирует исключение, определенное в листинге 9.5, транзакция будет помечена для отката и контейнер сделает фактический

откат, когда наступит время завершения транзакции. Это произойдет потому, что `InventoryLevelTooLowException` имеет аннотацию `@ApplicationException(rollback = true)`. В табл. 9.4 показаны все возможные комбинации с применением исключения. Первая строка таблицы может быть истолкована как «Если исключение приложения наследует от класса `Exception` и не имеет аннотации `@ApplicationException`, то генерация такого исключения будет означать, что транзакция не будет отмечена для отката.

Таблица 9.4. Комбинации исключений приложений

Наследует от	@ApplicationException	Транзакция помечена для отката
Exception	Нет аннотации	Нет
Exception	rollback = true	Да
Exception	rollback = false	Нет
RuntimeException	Нет аннотации	Да
RuntimeException	rollback = true	Да
RuntimeException	rollback = false	Нет

Транзакции, управляемые компонентом. С помощью CMT вы указываете контейнеру сделать разграничение транзакций, просто указав атрибут транзакции и используя контекст сеанса или исключения для пометки транзакции для отката. В некоторых случаях декларативный CMT не может обеспечить необходимую детализацию разграничения (например, метод не может генерировать более одной транзакции). Для того чтобы решить эту проблему, EJB предлагает программный способ управления разграничением транзакций с помощью BMT. BMT позволяет явно управлять границами транзакций (начало, фиксация, откат) с использованием JTA.

Для того чтобы отключить разграничение по умолчанию CMT и переключиться в режим BMT, компонент просто должен использовать аннотацию `@javax.ejb.TransactionManagement` (или XML-эквивалент в файле `ejb-jar.xml`) следующим образом:

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class ItemEJB {...}
```

С помощью разграничения BMT приложение запрашивает транзакции, контейнер EJB создает физическую транзакцию и заботится о низкоуровневых деталях. Кроме того, он не распространяет транзакции от одного BMT к другому.

Основной интерфейс, используемый для проведения BMT, — `javax.transaction.UserTransaction`. Он позволяет компоненту разграничивать транзакции, получает их состояние, задает тайм-аут и т. д. В контейнере EJB создается экземпляр типа `UserTransaction`, который становится доступным благодаря внедрению зависимостей, поиску JNDI или контексту `SessionContext` (с помощью метода `SessionContext.getUserTransaction()`). Этот API описывается в табл. 9.5.

Таблица 9.5. Методы интерфейса javax.transaction.UserTransaction

Метод	Описание
begin	Начинает новую транзакцию и связывает ее с текущим потоком
commit	Фиксирует транзакцию, связанную с текущим потоком
rollback	Откатывает транзакцию, связанную с текущим потоком
setRollbackOnly	Помечает транзакцию для отката
getStatus	Получает текущее состояние транзакции
setTransactionTimeout	Изменяет тайм-аут текущих транзакций

В листинге 9.6 показано, как разработать компонент BMT. Прежде всего мы получаем ссылку на объект типа `UserTransaction` с использованием внедрения через аннотацию `@Resource`. Метод `OneItemSold()` начинает транзакцию, выполняет обработку, а затем, в зависимости от бизнес-логики, совершает фиксацию или откат транзакции. Отмечу также, что транзакция помечается для отката в блоке `catch` (для удобства чтения я упростил обработку исключений).

Листинг 9.6. Компонент, не сохраняющий состояния и использующий BMT

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class InventoryEJB {
  @PersistenceContext(unitName = "chapter09PU")
  private EntityManager em;
  @Resource
  private UserTransaction ut;
  public void oneItemSold(Item item) {
    try {
      ut.begin();
      item.decreaseAvailableStock();
      sendShippingMessage();
      if (inventoryLevel(item) == 0)
        ut.rollback();
      else
        ut.commit();
      } catch (Exception e) {
      ut.rollback();
    }
    sendInventoryAlert();
  }
}
```

Разница с CMT, показанным в листинге 9.3, заключается в том, что с CMT контейнер запускает транзакцию до исполнения метода и фиксирует ее сразу после его отработки. С BMT, показанным в листинге 9.6, вы вручную определяете границы транзакции внутри самого метода.

Поддержка транзакций в Managed Beans

Как вы только что видели, CMT — одна из оригинальных и простых в применении функций EJB. При использовании декларативных транзакций для классов или методов корпоративного компонента контейнер может создавать новые транзакции (`REQUIRED`, `REQUIRES_NEW`), наследовать из уже существующих (`SUPPORTS`) или генерировать исключение, если транзакция не была создана заранее (`MANDATORY`). Это происходит потому, что контейнер перехватывает соответствующий вызов метода и добавляет операции, необходимые для инициализации, приостановки или завершения транзакций JTA.

Как часть лучшего согласования Managed Beans с платформой, одним из усовершенствований Java EE 7 является расширение CMT за рамки EJB. Это стало возможным благодаря перехватчикам и их связыванию (см. главу 2). Управление транзакциями в Managed Beans реализовано с использованием связывания перехватчиков CDI, как это показано в листинге 9.7.

Листинг 9.7. Связывание перехватчиков @javax.transaction.Transactional

```
@Inherited
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface Transactional {

  TxType value() default TxType.REQUIRED;
  Class[] rollbackOn() default {};
  Class[] dontRollbackOn() default {};
  public enum TxType {
    REQUIRED,
    REQUIRES_NEW,
    MANDATORY,
    SUPPORTS,
    NOT_SUPPORTED,
    NEVER
  }
}
```

Аннотация `javax.transaction.Transactional` (см. листинг 9.7) предоставляет приложению возможность декларативно контролировать границы транзакций в CDI Managed Beans, а также сервлетов, JAX-RS и конечных точек JAX-WS. Это обеспечивает семантику атрибутов транзакций EJB в CDI без зависимостей от других служб EJB, таких как RMI или службы таймера.

Например, в листинге 9.8 показана веб-служба JAX-RS (подробнее об этом читайте в главе 15), использующая аннотацию `@Transactional`, для того чтобы применять класс `EntityManager` для сохранения книг в базе данных. Такой код нельзя было написать до появления Java EE 7. Нужно было либо аннотировать веб-службу RESTful `@Stateless`, либо делегировать уровни хранения сессионного компонента.

Листинг 9.8. Веб-служба RESTful, предназначенная для создания книг с помощью транзакций

```
@Path("book")
@Transactional
public class BookRestService {
  @Context
  private UriInfo uriInfo;
  @PersistenceContext(unitName = "chapter09PU")
  private EntityManager em;
  @POST
  @Consumes(MediaType.APPLICATION_XML)
  public Response createBook(Book book) {
    em.persist(book);
   URI bookUri = uriInfo.getAbsolutePathBuilder().path(book.getId().toString()).
build();
    return Response.created(bookUri).build();
  }
  @GET
  @Produces(MediaType.APPLICATION_XML)
  @Transactional(Transactional.TxType.SUPPORTS)
  public Books getAllBooks() {
    TypedQuery<Book> query = em.createNamedQuery(Book.FIND_ALL, Book.class);
    Books books = new Books(query.getResultList());
    return books;
  }
}
```

Вы также можете изменить настройки транзакций политики по умолчанию (например, `SUPPORTS`) только с помощью атрибутов аннотации. Как показано в листинге 9.8, аннотацию `@Transactional` можно применить только для класса и/или метода.

Исключения и транзакции. Исключения и транзакции в Managed Beans немного отличаются от их аналогов в EJB. Как и в EJB, они используют такую же обработку исключений по умолчанию: исключения приложения (то есть проверяемые исключения) приводят к тому, что перехватчик не помечает транзакцию для отката, а для системных исключений (непроверяемых исключений) это выполняется. Но такое поведение по умолчанию может быть перегружено с помощью аннотации `@Transactional` с атрибутами `rollbackOn` и `dontRollbackOn`. Когда вы определяете класс для любого из этих атрибутов, разработанное поведение применяется и к подклассам этого класса. Если вы укажете оба атрибута, то `dontRollbackOn` будет иметь приоритет.

Чтобы переопределить поведение по умолчанию и заставить помечать транзакции для отката при генерации любых исключений приложения, вам следует написать следующую строку:

```
@Transactional(rollbackOn={Exception.class})
```

С другой стороны, если вы не хотите, чтобы транзакция не помечалась перехватчиком для отката при генерации исключения `IllegalStateException` (непрове-

ренное исключение) или любого из его подклассов, вы можете использовать атрибут `dontRollbackOn` следующим образом:

```
@Transactional(dontRollbackOn={IllegalStateException.class})
```

Вы можете указать оба атрибута для уточнения поведения транзакции. Каждый атрибут принимает массив классов и может использоваться следующим образом:

```
@Transactional(rollbackOn={SQLException.class},
    dontRollbackOn={SQLWarning.class, ArrayIndexOutOfBoundsException.class})
```

ПРИМЕЧАНИЕ

Не существует особой встроенной информации об исключениях приложения EJB (то есть исключениях с аннотацией @ApplicationException). Поскольку используется перехватчик, эти исключения должны рассматриваться как и любые другие исключения, если только иное не определено атрибутами rollbackOn и dontRollbackOn.

Резюме

В этой главе я показал вам, как управлять транзакциями в EJB и Managed Beans. Вы можете управлять транзакциями декларативно или программно.

Транзакции позволяют бизнес-уровню держать данные в согласованном состоянии, даже если к ним осуществляется одновременный доступ от нескольких приложений. Они следуют правилам ACID и могут быть распределены по нескольким ресурсам (базам данных, конечным точкам JMS, веб-службам и т. д.). CMT позволяет вам легко настроить поведение контейнера EJB с точки зрения разграничения транзакций. Вы можете влиять на это поведение, пометив транзакцию для отката с помощью контекста EJB или исключений. Вы всегда можете использовать BMT, если вам нужно более точное управление разграничением транзакций непосредственно с помощью JTA. Новинкой Java EE 7 является возможность использовать эти понятия для других компонентов, таких как сервлеты или веб-службы, благодаря связыванию перехватчиков CDI.

Глава 10
JavaServer Faces

Если вы хотите графически отобразить информацию, поступающую от сервера, то должны создать пользовательский интерфейс. Он может быть разным: настольное приложение, веб-приложение, работающее в браузере, или мобильное приложение, работающее в портативном устройстве, которое отображает графический интерфейс и взаимодействует с конечным пользователем.

Сегодня мы живем в интернет-зависимом мире. Поскольку наш транзакционный сервер обрабатывает тысячи запросов и общается с гетерогенными системами с помощью веб-служб, нам необходим уровень презентации для взаимодействия с конечными пользователями, предпочтительно работающий в браузере. Браузеры есть везде, пользовательские интерфейсы стали богаче, динамичнее и проще в применении, чем ранее. Многофункциональные веб-приложения становятся все популярнее, поскольку пользователи ожидают большего от работы в Интернете. Они нуждаются в использовании каталогов книг и компакт-дисков в Интернете, а также хотят иметь доступ к электронной почте и документам, получать уведомления по электронной почте. Они также иногда хотят, чтобы обновлялась лишь часть страницы по срабатыванию какого-либо события сервера. Добавьте к этому философию Web 2.0, согласно которой люди могут обмениваться любой информацией с группами друзей и взаимодействовать друг с другом, в результате веб-интерфейсы становятся все более и более сложными для разработки. JavaServer Faces (JSF, или просто Faces) был создан, чтобы облегчить создание графических интерфейсов.

Вдохновленный моделью компонентов Swing и другими фреймворками графических интерфейсов пользователя, JSF позволяет разработчикам думать в терминах компонентов, событий, компонентов-подложек и их взаимодействий, а не в терминах запросов, ответов и языка разметки. Его цель — сделать веб-разработку быстрее и проще, предоставив поддержку компонентов пользовательского интерфейса (таких как текстовые поля, списки, панели вкладок и сеток с данными) с точки зрения подхода быстрой разработки приложений (RAD).

В этой главе описаны веб-страницы, созданные с помощью HTML, CSS и JavaScript. Далее мы сфокусируемся на создании веб-интерфейсов с использованием компонентов JSF, а также научимся создавать собственные компоненты.

Концепция веб-страниц

Когда мы создаем веб-приложение, мы заинтересованы в отображении динамического содержимого, которое можно прочитать в браузере: список элементов каталога (например, компакт-диски и книги), информацию о клиенте под заданным именем, корзину товаров, содержащую предметы, которые клиент хочет купить, и т. д. С дру-

гой стороны, статический контент, такой как адрес издательства и часто задаваемые вопросы с информацией о том, как купить товары, изменяется редко или не изменяется никогда. Статическими также могут быть изображения, видео на странице.

Конечная цель создания веб-страницы — отображение в браузере. Страница должна использовать языки, которые браузер сможет понять, например HTML, XHTML, CSS и JavaScript.

HTML

Hypertext Markup Language (HTML) — основной язык для веб-страниц. Он основан на стандартном обобщенном языке разметки (SGML), который является метаязыком, определяющим языки разметки. HTML использует *разметку*, или *теги*, для того, чтобы структурировать текст на абзацы, списки, ссылки, кнопки, текстовые поля и т. д.

HTML-страница — это текстовый документ, используемый браузерами для представления текста и графики. Такие документы представляют собой текстовые файлы, которые часто имеют расширение `.html` или `.htm`. Веб-страница состоит из содержимого, тегов, позволяющих изменить некоторые свойства содержимого, и внешних объектов, таких как изображения и видео, а также из JavaScript- или CSS-файлов. Листинг 10.1 показывает, как с помощью HTML отобразить форму для создания новой книги.

Листинг 10.1. Страница newBook.html, имеющая некорректную структуру HTML

```
<H1>Создать новую книгу</h1>
<hr>
<TABLE border=0>
  <TR>
    <TD>ISBN :</TD>
    <TD><input type=text/></td>
  </tr>
  <tr>
    <td>Название :</td>
    <TD><input type=text/></td>
  </tr>
  <tr>
    <td>Цена :
    <TD><input type=text/>
  </tr>
  <tr>
    <td>Описание :
    <td><textarea name=textarea cols=20 rows=5></textarea>
  </tr>
  <TR>
    <TD>Количество страниц :
    <td><input type=text/>
  </tr>
  <tr>
    <td>Иллюстрации :
    <td><input type="checkbox"/>
```

```
  </tr>
</table>
<input type=submit value=Создать>
<hr>
<em>APress — Изучаем Java EE 7</em>
```

Как правило, корректная страница HTML начинается с тега <html>, который представляет собой контейнер для документа. За ним следуют теги <head> и <body>. В рамках тега <body> содержится видимое содержимое, например код HTML, где отображаются таблицы, метки, поля ввода и кнопка. Как вы можете видеть в листинге 10.1, разметка в файле newBook.html не соответствует этим правилам, но браузеры все равно отобразят эту некорректную страницу. В результате страница будет выглядеть так, как показано на рис. 10.1.

Рис. 10.1. Графическое представление страницы newBook.html

Страница, показанная на рис. 10.1, является вполне приемлемым результатом, несмотря на то что код листинга 10.1 не очень хорошо отформатирован в терминах XML.

- Страница не имеет тегов <html>, <head> или <body>.
- Теги <input type=submit value=Создать> и <hr> не закрыты.
- Значения атрибутов находятся не в кавычках (border=0 вместо border="0").
- В тегах используются прописные и строчные буквы (например, в листинге можно встретить <TR> и </tr>).

Большинство браузеров позволяют допускать такие ошибки и отобразят эту форму. Тем не менее, если вы захотите обработать этот элемент с помощью, например, XML-парсера, у вас ничего не выйдет. Чтобы понять, почему так получится, рассмотрим веб-страницу, которая отформатирована в строгом соответствии

с XML-структурой с использованием расширяемого языка разметки гипертекста (eXtensible HyperText Markup Language, XHTML).

ПРИМЕЧАНИЕ

На момент написания этой книги спецификация HTML5 еще не была завершена. HTML5 добавит много новых функций, таких как элементы <video>, <audio> и <canvas>, а также предоставит возможность работать с масштабируемой векторной графикой (SVG). Эти функции предназначены для того, чтобы упростить подключение и работу с мультимедийными и графическими материалами в Интернете без применения проприетарных надстроек и API. HTML5 также является потенциальным кандидатом для кросс-платформенных мобильных приложений, поскольку многие его функции были созданы с учетом возможности запуска на смартфонах или планшетах.

XHTML

XHTML был создан вскоре после появления HTML 4.01. Он происходит из HTML, но при этом переформулирован в строгом соответствии с XML. Это означает, что документ XHTML по сути является документом XML, который следует определенной схеме и имеет графическое представление в браузерах. XHTML-файл (с расширением `.xhtml`) может либо быть использован как XML, либо сразу отобразиться в браузере.

В отличие от HTML, такой подход имеет преимущество, которое заключается в обеспечении проверки документов с использованием стандартных инструментов XML (XSL, XSLT, XSL-преобразования и т. д.). Отсюда следует, что XHTML гораздо мощнее, чем HTML, потому что позволяет определить любой набор тегов. В листинге 10.2 показано, как будет выглядеть версия веб-страницы для создания книги, написанная на XHTML.

Листинг 10.2. Страница newBook.xhtml с корректной структурой XML

```
<?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" xml:lang="en" lang="en">
<head>
  <title>Создать новую книгу</title>
</head>
<body>
<h1>Создать новую книгу</h1>
<hr/>
<table border="0">
  <tr>
    <td>ISBN :</td>
    <td><input type="text"/></td>
  </tr>
  <tr>
    <td>Название :</td>
    <td><input type="text"/></td>
  </tr>
  <tr>
```

```
      <td>Цена :</td>
      <td><input type="text"/></td>
    </tr>
    <tr>
      <td>Описание :</td>
      <td><textarea cols="20" rows="5"></textarea></td>
    </tr>
    <tr>
      <td>Количество страниц :</td>
      <td><input type="text"/></td>
    </tr>
    <tr>
      <td>Иллюстрации :</td>
      <td><input type="checkbox"/></td>
    </tr>
  </table>
  <input type="submit" value="Создать"/>
  <hr/>
  <em>APress — Изучаем Java EE 7</em>
  </body>
  </html>
```

Между листингами 10.1 и 10.2 имеются некоторые различия: документ в листинге 10.2 следует строгой структуре и имеет теги <html>, <head> и <body>. Все теги закрыты, даже те, которые содержат пустые элементы (каждый тег <td> закрыт, а вместо <hr> используется <hr/>); атрибуты указываются в одинарных или двойных кавычках (<table border="0"> или <table border='0'>, но не <table border=0>) и все теги написаны строчными буквами (<tr> вместо <TR>). Сравните это с листингом 10.1: как отмечалось ранее, он содержит недействительный HTML-документ, который браузеры в любом случае смогут отобразить.

Строгая проверка соответствия правилам синтаксиса XML и ограничений схемы позволяет легко поддерживать и анализировать XHTML, и, как следствие, в настоящее время он стал более предпочтительным языком для создания веб-страниц. Документ, подтверждающий XHTML, является XML-документом, который соответствует спецификации HTML 4.01. Документ может использовать три различных формата проверки.

- *XHTML 1.0 Transitional* — очень мягкий вариант XHTML, позволяющий использовать презентационные элементы (такие как center, font и strike), которые исключены из строгой версии.
- *XHTML 1.0 Frameset* — переходный вариант, который также позволяет определить набор документов (общая черта Всемирной сети в конце 1990 года).
- *XHTML 1.0 Strict* — наиболее ограничительный вариант XHTML, который строго следует спецификации HTML 4.01.

CSS

Браузеры живут в мире языков клиентской стороны, таких как HTML, XHTML, CSS и JavaScript. Каскадные таблицы стилей (CSS) предназначены для описания

внешнего вида документа, созданного с помощью HTML или XHTML. CSS используется для определения цветов, шрифтов, макетов и других особенностей представления документа. Это позволяет разделить содержимое документа (написанное с помощью XHTML) от его представления (написанного с помощью CSS). Как и HTTP, HTML и XHTML, спецификация CSS поддерживается консорциумом World Wide Web (W3C).

Например, вы хотите изменить метки на странице `newBook.xhtml`, сделав их курсивными (`font-style: italic;`), изменив их цвет на синий (`color: # 000099;`) и увеличив размер шрифта (`font-size: 22px;`). Вам не придется повторять эти изменения для каждого тега. Вы можете определить стиль CSS (в теге `<style type="text/css">`) и дать ему псевдоним (например, `.row`) или переопределить существующие стили тегов (например, `body`, `table` или `h1`). Страница (листинг 10.3) будет использовать эти стили для элементов, для которых необходимо изменить их представление (`<td class="row">`).

Листинг 10.3. Страница NewBook.xhtml, для которой применены некоторые стили CSS

```
<!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" xml:lang="en" lang="en">
<head>
  <title>Создать новую книгу</title>
  <style type="text/css">
    body {
      font-family: Arial, Helvetica, sans-serif;
    }
    table {
      border: 0;
    }
    h1 {
      font-size: 22px;
      color: blue;
      font-style: italic;
    }
    .row {
      font-style: italic;
    }
  </style>
</head>
<body>
<h1>Создать новую книгу</h1>
<hr/>
<table>
  <tr>
    <td class="row">ISBN :</td>
    <td><input type="text"/></td>
  </tr>
  <tr>
    <td class="row">Название :</td>
    <td><input type="text"/></td>
```

```
    </tr>
    <tr>
      <td class="row">Цена :</td>
      <td><input type="text"/></td>
    </tr>
    <tr>
      <td class="row">Описание :</td>
      <td><textarea cols="20" rows="5"></textarea></td>
    </tr>
    <tr>
      <td class="row">Количество страниц :</td>
      <td><input type="text"/></td>
    </tr>
    <tr>
      <td class="row">Иллюстрации :</td>
      <td><input type="checkbox"/></td>
    </tr>
  </table>
  <input type="submit" value="Создать"/>
  <hr/>
  <em>APress — Изучаем Java EE 7</em>
  </body>
  </html>
```

В листинге 10.3 код CSS встроен в страницу XHTML. В реальном приложении все стили были бы размещены в отдельном файле, который бы импортировался на веб-страницу. Веб-дизайнер может составить один или несколько наборов стилей CSS для различных групп страниц, а затем составители содержимого смогут создавать или изменять страницы без необходимости заботиться об их внешнем виде.

По сравнению со страницей на рис. 10.1 конечный результат отличается тем, что все ярлыки выделены курсивом и название окрашено в синий цвет (рис. 10.2).

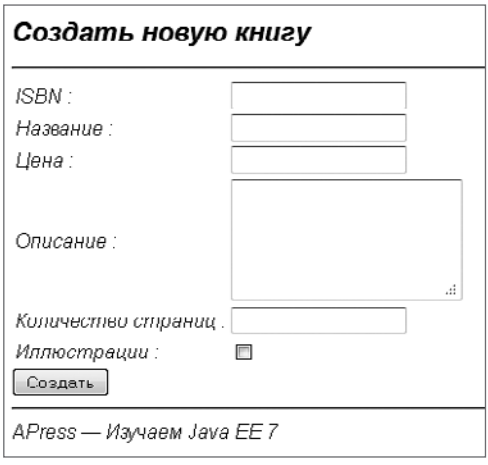


Рис. 10.2. Графическое представление страницы newBook.xhtml после применения стилей CSS

DOM

Страница XHTML является XML-документом и таким образом имеет представление объектной модели документа (DOM). DOM — это спецификация W3C, предназначенная для получения доступа и изменения содержимого и структуры XML-документов, а также абстрактный API для обработки запросов, перемещения и манипулирования такими документами. DOM можно рассматривать как древовидное представление структуры документа. Рисунок 10.3 показывает, как страница `newBook.xhtml` может выглядеть в качестве представления DOM. В корне находится тег `html`, на один уровень ниже — `head` и `body`, а под `body` — таблица со списком тегов `tr`.

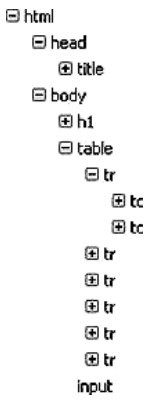


Рис. 10.3. Древовидное представление страницы newBook.xhtml

DOM обеспечивает стандартный способ взаимодействия с XML-документами. Вы можете обойти дерево и отредактировать содержимое узла (листа дерева). С помощью JavaScript вы можете сделать веб-страницы динамичными. Как вы увидите в главе 11, AJAX основан на JavaScript и взаимодействует с DOM-представлением веб-страницы.

JavaScript

До сих пор мы рассматривали разные языки, используемые для представления статического контента и графических деталей веб-страницы. Но зачастую веб-страницам необходимо взаимодействовать с конечным пользователем, показывая динамический контент. Как вы увидите, динамический контент может обрабатываться серверными технологиями, такими как JSF, но браузеры также могут обрабатывать его, выполнив JavaScript.

JavaScript — это язык сценариев, используемый на стороне клиента веб-разработки. Несмотря на свое название, он не имеет отношения к языку программирования Java, так как является интерпретируемым, слабо типизированным языком. JavaScript представляет собой мощный способ создания динамических веб-приложений путем написания функций, которые взаимодействуют с DOM на странице. W3C стандартизировал DOM, в то время как Европейская ассоциация производителей компьютеров (ECMA) стандартизировала JavaScript как спецификацию ECMAScript. Любая

страница, написанная с помощью этих стандартов (XHTML, CSS и JavaScript), должна выглядеть и вести себя более или менее одинаково в любом браузере, который придерживается этих принципов.

Примером взаимодействия с JavaScript DOM является страница `newBook.xhtml`, показанная в листинге 10.4. На ней отображается форма, где вы можете ввести информацию о книге. Цена книги должна быть заполнена пользователем на стороне клиента прежде, чем она отправится на сервер. Чтобы указать цену нужной записи, вы можете создать функцию JavaScript (`priceRequired()`), которая проверяет, является ли поле с ценой пустым.

Листинг 10.4. Страница newBook.xhtml, использующая JavaScript

```
<?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" xml:lang="en" lang="en">
<head>
  <title>Создать новую книгу</title>

  <script type="text/javascript">
    function priceRequired() {
      if (document.getElementById("price").value == "") {
          document.getElementById("priceError").innerHTML =
            "Пожалуйста, введите цену!";
      }
    }
  </script>

</head>
<body>
<h1>Создать новую книгу</h1>
<hr/>
<table border="0">
  <tr>
    <td>ISBN :</td>
    <td><input type="text"/></td>
  </tr>
  <tr>
    <td>Название :</td>
    <td><input type="text"/></td>
  </tr>
  <tr>
    <td>Цена :</td>
    <td><input id="price" type="text" onblur ="javascript:priceRequired()"/>
      <span id="priceError"/>
    </td>
  </tr>
  <tr>
    <td>Описание :</td>
    <td><textarea cols="20" rows="5"></textarea></td>
  </tr>
```

```
    <tr>
      <td>Количество страниц :</td>
      <td><input type="text"/></td>
    </tr>
    <tr>
      <td>Иллюстрации :</td>
      <td><input type="checkbox"/></td>
    </tr>
  </table>
  <input name="" type="submit" value="Создать"/>
  <hr/>
  <em>APress — Изучаем Java EE 7</em>
  </body>
  </html>
```

В листинге 10.4 функция JavaScript `priceRequired()` встроена в страницу с помощью тега `<script>` (однако она может быть вынесена и в отдельный файл). Эта функция вызывается, когда текстовое поле `price` теряет фокус (это делает событие `onblur`). Функция `priceRequired()` использует неявный объект `document`, который представляет модель DOM документа XHTML. Метод `GetElementById("price")` ищет элемент с идентификатором `price` (`<input id="price">`) и проверяет, является ли он пустым. Если это так, функция ищет другой элемент, который называется `priceError` (`getElementById("priceError")`), и присваивает ему значение `"Пожалуйста, введите цену!"`. Если цена не записана, клиент отобразит сообщение, показанное на рис. 10.4.

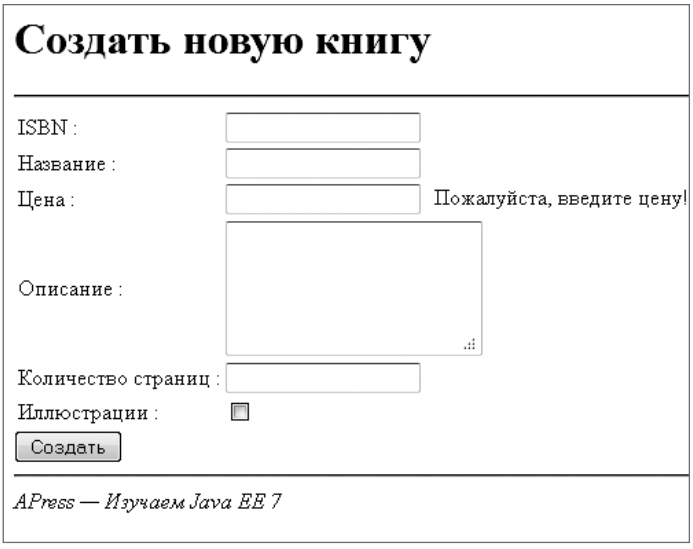


Рис. 10.4. Страница NewBook.html с сообщением об ошибке

JavaScript — это богатый язык. Предыдущий раздел охватывает лишь небольшую часть, которая демонстрирует взаимодействие JavaScript и DOM. Важно понимать, что функция JavaScript может получить объект на странице (по имени

или ID, например с помощью метода `GetElementById()`) и изменить его содержимое динамически на стороне клиента. Подробнее см. в разделе «AJAX» главы 11.

Концепция JSF

Вы только что видели технологии и языки, работающие на стороне клиента, такие как XHTML и CSS, которые представляют содержимое и визуальную составляющую статических веб-страниц. Чтобы добавить возможность взаимодействия и динамически изменять части веб-страницы, вы можете использовать функции JavaScript, которые работают в браузере. Но в большинстве случаев необходимо вызывать бизнес-слой EJB, чтобы отобразить данные из базы. Это динамическое содержимое может быть получено с помощью JSF на стороне сервера.

Архитектуру JSF легко понять (рис. 10.5), если вы знакомы с веб-платформами. Приложения JSF — это стандартные веб-приложения, которые перехватывают HTTP-запросы с помощью сервлета Faces и производят HTML. Архитектура позволяет подключить любой язык объявления страниц (или язык объявления видов), отрисовать его для различных устройств (браузер, мобильные устройства, планшеты и т. д.), а также создавать страницы с помощью событий, слушателей и компонентов, как Swing. Swing представляет собой набор инструментов Java для работы с виджетами и входит в состав Java SE, начиная с выпуска 1.2. Это GUI-фреймворк, предназначенный для создания настольных приложений (не веб-приложений) с помощью графических компонентов и модели слушателя событий для обработки данных, вводимых пользователем. JSF также имеет стандартный набор виджетов пользовательского интерфейса (UI) (кнопки, гиперссылки, флажки, текстовые поля и т. д.) и позволяет легко подключать сторонние компоненты. На рис. 10.5 показана архитектура JSF на очень высоком уровне.

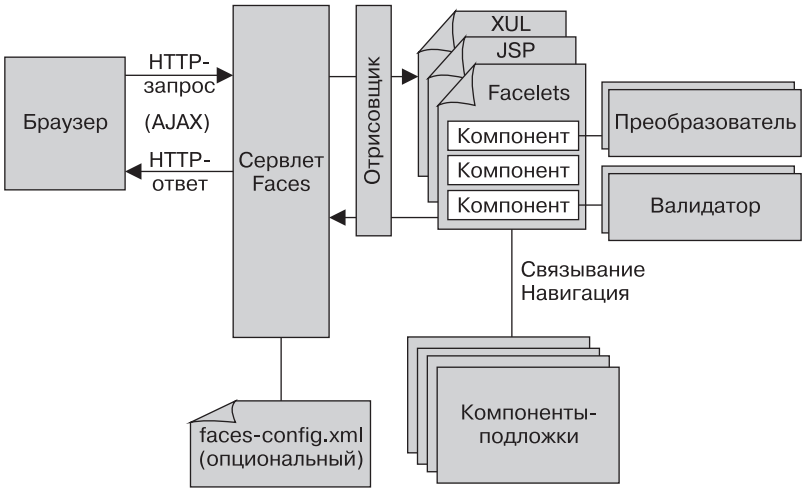


Рис. 10.5. Архитектура JSF

На рис. 10.5 представлены несколько важных компонентов JSF, которые делают его архитектуру богатой и гибкой.

- FacesServlet и `faces-config.xml`. FacesServlet является основным сервлетом приложения и опционально может быть настроен с помощью файла дескриптора `faces-config.xml`.
- Страницы и компоненты. JSF позволяет использовать несколько PDL, но Facelets рекомендован с версии JSF 2.0.
- Отрисовщики. Отвечают за отображение компонентов и перевод пользовательского ввода в значения свойств компонента.
- Преобразователи. Конвертируют значения компонента (`Date`, `Boolean` и т. д.), в значения разметки и из них (`String`).
- Валидаторы. Отвечают за то, чтобы значение, введенное пользователем, было корректным (большинство проверок может быть делегировано Bean Validation).
- Резервные компоненты и навигация. Бизнес-логика располагается в компонентах-подложках, которые также контролируют навигацию между страницами.
- Поддержка AJAX. JSF 2.2 поставляется со встроенной поддержкой AJAX, что описано в главе 11.
- Язык выражений (EL). EL используется в страницах JSF для привязки переменных и действий между компонентами и компонентами-подложками.

FacesServlet

Большинство веб-фреймворков используют шаблон проектирования Model — View — Controller (MVC — «модель — вид — представление»), и JSF не является исключением. Шаблон MVC используется для того, чтобы отделить представление (страницы) и модель (данные, которые будут отображаться в представлении). Контроллер обрабатывает действия пользователя, которые могут привести к изменениям в модели и обновлению представлений. В JSF контроллер является сервлетом и называется FacesServlet. Это сервлет, управляющий жизненным циклом обработки запросов для веб-приложений. Все пользовательские запросы проходят через FacesServlet, который рассматривает запрос и вызывает различные действия модели с использованием компонентов-подложек.

Этот сервлет является внутренней частью JSF. Он может быть настроен только с использованием внешних метаданных. Вплоть до версии JSF 1.2 единственным способом конфигурации был файл `faces-config.xml`. Сегодня, когда существует версия JSF 2.2, этот файл опционален и большая часть метаданных может быть определена с помощью аннотаций (для компонентов-подложек, преобразователей, компонентов, отрисовщиков и валидаторов).

Страницы и компоненты

Фреймворк JSF должен отправить страницу на устройство вывода клиента (например, браузер), поэтому ему требуется технология отображения. Этой технологией становится PDL. Приложение JSF может свободно использовать несколько

технологий для его PDL, но Facelets — предпочтительный вариант для версии JSF 2.2 (читайте подробнее в подразделе «Facelets» далее).

Страницы Facelets состоят из дерева компонентов (также называемых виджетами, или элементами управления), которые обеспечивают конкретную функциональность для взаимодействия с конечным пользователем (текстовое поле, кнопки, списки и т. д.). JSF имеет стандартный набор компонентов и позволяет легко создавать свои собственные. Страница проходит через насыщенный жизненный цикл, чтобы управлять этим деревом компонентов (инициализация, события, отрисовка и т. д.).

Код, приведенный в листинге 10.5, является XHTML-страницей Facelets, которая использует теги JSF (`xmlns:h="http://xmlns.jcp.org/jsf/html"`) для отображения формы с двумя полями ввода (ISBN и название книги) и кнопки. Эта страница состоит из нескольких JSF-компонентов. Некоторые из них не имеют внешнего вида, как и те, что используются для объявления заголовка (`<h:head>`), тела (`<h:body>`) или формы (`<h:form>`). Другие компоненты можно увидеть, они представляют собой метку (`<h:outputLabel>`), текстовое поле (`<h:inputText>`) или кнопки (`<h:commandButton>`). Обратите внимание, что чистые HTML-теги также могут быть использованы на странице (`<table>`, `<tr>`, `<hr/>` и т. д.).

Листинг 10.5. Фрагмент страницы JSF, использующей сочетание компонентов JSF и HTML

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
         "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://xmlns.jcp.org/jsf/
html">
<h:head>
  <title>Создать новую книгу</title>
</h:head>
<h:body>
  <h1>Создать новую книгу</h1>
  <hr/>
  <h:form>
    <table border="0">
      <tr>
        <td><h:outputLabel value="ISBN : "/></td>
        <td><h:inputText value="#{bookController.book.isbn}"/></td>
      </tr>
      <tr>
        <td><h:outputLabel value="Название :"/></td>
        <td><h:inputText value="#{bookController.book.title}"/></td>
      </tr>
    </table>
    <h:commandButton value="Создать новую книгу"
              action="#{bookController.doCreateBook}" styleClass="submit"/>
  </h:form>
  <hr/>
  <em>APress — Изучаем Java EE 7</em>
</h:body>
</html>
```

Facelets

Одной из причин создания JSF было намерение повторно использовать JSP (JavaServer Page) в качестве основного PDL, поскольку он уже был частью Java EE. JSP использовал JSTL и EL (JSP Standard Tag Library — стандартная библиотека тегов JSP), так что идея заключалась в том, чтобы задействовать все эти технологии в рамках JSF. JSP — это язык страниц, а JSF — это слой компонентов, располагающийся над ним. Тем не менее жизненные циклы JSP и JSF не совсем подходят друг другу. Теги в JSP обрабатываются один раз сверху вниз, чтобы сформировать ответ. JSF имеет более сложный жизненный цикл, в котором генерация дерева компонентов и отрисовка происходят в разных фазах. Именно здесь приходит на помощь Facelets, чтобы соответствовать жизненному циклу JSF.

Facelets начал свое существование как альтернатива JSP с открытым исходным кодом. В отличие от JSP, EL и JSTL он не имел JSR и не был частью Java EE. Facelets стал заменой JSP и предоставил альтернативу на основе XML (XHTML) для всех страниц JSF-приложений. Facelets был разработан на основе JSF, именно поэтому он представляет собой простую модель программирования. Из-за этого Facelets был указан в JSF 2.0 и сегодня предоставляется в Java EE 7 в качестве предпочтительного PDL для JSF.

Отрисовщики

JSF поддерживает две модели программирования для отображения компонента: непосредственную и делегированную реализации. При использовании прямой модели компоненты должны декодировать и закодировать себя в графическое представление. При использовании модели делегирования эти операции делегируются отрисовщику, который позволяет компоненту не зависеть от технологии отрисовки (браузер, переносное устройство и т. д.) и иметь несколько графических представлений.

Отрисовщик отвечает за отображение компонента и перевод пользовательского ввода в значение компонента. Рассматривайте отрисовщик как транслятор между клиентом и сервером: он декодирует запрос пользователя, чтобы установить значения компонентов, и кодирует ответ для создания представления компонента, который клиент понимает и может отобразить.

Отрисовщики располагаются в наборах отрисовки, которые сосредотачиваются на конкретном типе вывода. Для обеспечения переносимости приложения JSF включает поддержку стандартного комплекта отрисовщиков и связанных с ним отрисовщиков для HTML 4.01. Реализации JSF могут сформировать собственный комплект отрисовщиков для создания языка разметки для беспроводных устройств (WML), масштабируемой векторной графики (SVG) и т. д.

Преобразователи и валидаторы

Как только страница отрисована, пользователь может взаимодействовать с ней для ввода данных. Поскольку не существует никаких ограничений типов, отрисовщик не может знать заранее, как отобразить объект. Здесь в игру вступают преобразователи: они преобразуют объект (он может иметь тип `Integer`, `Date`, `Boolean`

и т. д.) в строку для отображения и из входной строки обратно в объект. JSF поставляется с набором преобразователей для общих типов (в пакете `javax.faces.convert`), но вы можете разработать свой собственный или включить сторонние преобразователи.

Иногда эти данные также должны быть подтверждены перед обработкой на сервере. Валидаторы отвечают за то, чтобы значение, введенное пользователем, было корректным. С одним компонентом могут быть связаны один или несколько валидаторов. Вместе с JSF поставляется несколько валидаторов (`LengthValidator`, `RegexValidator` и т. д.). Он также позволяет вам создать свой собственный валидатор с помощью аннотированных классов. Когда при преобразовании или проверке корректности появляется ошибка, сообщение для отображения отправляется ответу. С JSF 2.0 проверка может быть делегирована Bean Validation (это показано в главе 3).

Компоненты-подложки и навигация

Все рассмотренные концепции были связаны с одной страницей: что такое страница, что такое компонент, как они отрисовываются, преобразуются и проверяются? Веб-приложения состоят из нескольких страниц и выполняют бизнес-логику (например, путем вызова слоя EJB или RESTful-веб-служб). Обработка компонентов-подложек переходит от одной страницы к другой, вызывая EJB и синхронизируя данные с компонентами.

Компонент-подложка — это специализированный класс Java, который синхронизирует значения с компонентами, обрабатывает бизнес-логику и навигацию между страницами (подробнее о компонентах-подложках и навигации читайте в главе 11). Вы связываете компонент с определенным свойством компонентов-подложек или действия с помощью EL. Следующий код связывает атрибут `isbn` компонента `book` и вызывает действие компонента-подложки `doCreateBook`:

```
<h:inputText value="#{bookController.book.isbn}"/>
<h:commandButton value="Create" action="#{bookController.doCreateBook}"/>
```

Первая строка кода получает введенное текстовое значение непосредственно в свойство `book.isbn` компонента-подложки, которое называется `BookController`. Введенное текстовое значение синхронизировано со свойством `book.isbn` компонента-подложки. Компонент-подложка также обрабатывает события. Вторая строка кода показывает кнопку отправки, связанную с действием. Когда пользователь нажимает кнопку Submit (Отправить), срабатывает событие компонента-подложки, где выполняется слушатель событий (в этом примере — метод `doCreateBook()`).

В листинге 10.6 показан компонент-подложка `BookController`. Этот класс Java имеет CDI-аннотацию `@Named`, поэтому может быть использован в EL. У него есть свойство `book`, которое связано со значением компонента страницы (`value="#{bookController.book.isbn}"`). Метод `doCreateBook()` вызывает EJB, не сохраняющий состояние, а затем возвращает строку, которая позволяет перемещаться между страницами (`listBooks.xhtml`):

Листинг 10.6. Компонент-подложка BookController

```
@Named
@RequestScoped
public class BookController {
  @Inject
  private BookEJB bookEJB;
  private Book book = new Book();
  public String doCreateBook() {
    book = bookEJB.createBook(book);
    return "listBooks.xhtml";
  }
  // Методы работы со свойствами
}
```

Язык выражений

Связывание между страницей JSF и компонентом-подложкой Java выполняется через EL. Вы можете использовать утверждения EL для вывода значений переменных, атрибутов доступа объектов на странице или для вызова метода компонента-подложки. Базовый синтаксис утверждения EL выглядит так:

```
#{expr}
```

Утверждения `#{expr}` будут преобразовываться и оцениваться во время выполнения JSF. Выражения EL могут использовать большинство обычных операторов Java:

- *арифметические*: `+`, `-`, `*`, `/` (деление), `%` (целая часть);
- *операторы отношений*: `==` (равно), `!=` (не равно), `<` (меньше), `>` (больше), `<=` (меньше либо равно), `>=` (больше либо равно);
- *логические*: `&&` (и), `||` (или), `!` (не);
- *другие*: `()`, `empty`, `[]`.

Следует отметить, что некоторые операторы имеют как символьные, так и текстовые варианты (`>` может стать `gt`; `/` может стать `div` и т. д.). Эти эквиваленты позволят вам разрабатывать свои страницы JSF с XML-совместимым синтаксисом. Оператор `<` можно кодировать как `#{2 lt 3}`, а также `#{2 < 3}`.

Оператор `empty` проверяет, является ли объект пустым или же он ссылается на пустую строку, список, словарь или массив. Он возвращает значение `true`, если это верно, иначе возвращает `false`. Вы можете проверить, равен ли объект или один из его атрибутов значению `null`.

```
#{empty book}
#{empty book.isbn}
```

Оператор «точка» (`.`) используется для доступа к атрибуту `isbn` объекта `book`. Другой вариант синтаксиса — оператор `[]`. Доступ к атрибуту `isbn` может быть получен с помощью любой нотации:

```
#{book.isbn}
#{book[isbn]}
```

В версии EL 3.0 эта особенность была улучшена — появилась возможность вызывать методы. Следующий фрагмент кода показывает, как купить книгу, вызвав метод `book.buy()`, а также передачу параметра (используемая валюта):

```
# {book.buy()}
# {book.buy('EURO')}
```

Поддержка AJAX

Веб-приложение должно предоставлять богатый и отзывчивый интерфейс. Подобная реактивность может быть достигнута путем обновления только небольших фрагментов страницы в асинхронном режиме, что и позволяет делать AJAX. Предыдущие версии (1.x) JSF не предлагали собственного готового решения, пробел восполняли сторонние библиотеки, такие как a4jsf. Начиная с версии JSF 2.0, поддержка AJAX была добавлена в виде библиотеки JavaScript (`jsf.js`), определенной в спецификации и теге `<f:ajax>`. Например, вы можете использовать тег `<f:ajax>` для отправки формы в асинхронном режиме, что показано в следующем фрагменте кода:

```
<h:commandButton value="Создать книгу" action="#{bookController.
doCreateBook}">
  <f:ajax execute="@form" render=":booklist"/>
</h:commandButton>
```

Не волнуйтесь, если вы не понимаете этот код: целый раздел главы 11 посвящен AJAX.

Обзор спецификации JSF

Веб-разработка с помощью языка Java началась в 1996 году с появлением Servlet API, который представлял собой неразвитый способ создания динамических веб-страниц. Вы должны были манипулировать низкоуровневым HTTP API (`HttpServletRequest`, `HttpServletResponse`, `HttpSession` и т. д.), чтобы отобразить HTML-теги внутри вашего кода на Java. JSP появился в 1999 году и предоставил более высокий уровень абстракции, нежели сервлеты. В 2004 году вышла первая версия JSF. Начиная с версии 1.2, в 2006 году, JSF стал частью Java EE 5. JSF 2.2 входит в состав Java EE 7.

Краткая история веб-интерфейсов

Сначала веб-страницы были статическими. Пользователь запрашивает ресурс (веб-страницу, изображения, видео и т. д.), и сервер возвращает его — это довольно просто, но очень ограниченно. С ростом коммерческой деятельности в Интернете компании захотели предоставить клиентам динамическое содержимое. Первым решением для создания динамического контента был Common Gateway Interface (CGI). С помощью HTML-страниц и сценариев CGI, написанных на любых языках (от Perl до Visual Basic), приложение может обращаться к базам данных и предоставлять динамическое содержимое, но было ясно, что CGI слишком низкоуровневый (вы должны были обрабатывать HTTP-заголовки, вызывать команды HTTP и т. д.) и нуждается в улучшении.

В 1995 году появился новый язык — Java. Он имел независимый от платформы API пользовательского интерфейса под названием Annotation Window Toolkit (AWT). Позднее, в версии Java SE 1.2, AWT, опиравшийся на модуль пользовательского интерфейса операционной системы, был заменен Swing API (который рисует собственные виджеты, используя Java 2D). В эти первые дни Java браузер Netscape Navigator предложил поддержку этого нового языка, что положило начало эре апплетов. *Апплеты* — приложения, работающие на стороне клиента внутри браузера. Это позволило разработчикам писать приложения с помощью AWT или Swing и встраивать их в веб-страницу. Тем не менее апплеты не получили развития. Netscape также создал язык сценариев JavaScript, который выполняется непосредственно в браузере. Несмотря на некоторые несовместимости между браузерами, JavaScript сейчас активно используется и является мощным средством для создания динамических веб-приложений.

После того как апплеты не получили широкого распространения, компания Sun предоставила сервлеты как способ создания динамичных веб-клиентов. Сервлеты были альтернативой сценариям CGI, потому что предоставляли библиотеку более высокого уровня для обработки HTTP, имели полный доступ к Java API (получение доступа к базе данных, удаленные вызовы и т. д.) и могли создавать HTML как ответ, который будет виден пользователю.

Компания Sun выпустила JSP в 1999 году как расширение модели сервлетов. Но, поскольку JSP представлял собой смесь кода на Java и HTML, в 2001 году появился фреймворк с открытым исходным кодом и предоставил новый подход — Struts. Он расширил Servlet API и вдохновил разработчиков использовать архитектуру MVC. Новейшая история полна других фреймворков, каждый из которых пытался заполнить пробелы предыдущего (Tapestry, Wicket, WebWork, DWR, Spring MVC и т. д.). Сегодня JSF 2.2 является стандартным веб-фреймворком Java EE 7. Он конкурирует со Struts и Tapestry в пространстве Java. Google Web Toolkit (GWT), Flex и JavaFX могут дополнять JSF.

ПРИМЕЧАНИЕ

JSP 1.2 и Servlet 2.3 были указаны вместе в JSR 53. В то же время JSTL был указан в JSR 52. Начиная с 2002 года спецификация JSP 2.0 развивалась отдельно от сервлетов в JSR 152. В 2006 году JSP 2.1 был частью Java EE 5 и способствовал интеграции JSF и JSP путем введения единого EL. В версии Java EE 7 как JSP, так и EL стали развиваться отдельно (JSP 2.3 и Expression Language 3.0). В этой книге не рассматривается JSP, поскольку его можно считать устаревшим, а JSF предпочтителен для создания веб-интерфейсов в Java EE.

Краткая история JSF

JSF 1.0 был создан в 2001 году как JSR 127, а в 2004 году была выпущена версия 1.1. Версия 1.2 появилась в Java EE как JSR 252 только в 2006 году. Самой большой проблемой этой версии было сохранение обратной совместимости, а также интеграция JSP с единым EL. Несмотря на эти усилия, JSF и JSP не подошли идеально, так что в качестве альтернативы JSP были введены другие структуры, такие как Facelets.

JSF 2.0 — это крупный релиз, развившийся в JSR 314 и ставший частью Java EE 6. Он упростил навигацию между страницами, разработку с помощью аннотаций (для компонентов-подложек, отрисовщиков, преобразователей и валидаторов), разработку графических компонентов, поставил на первое место запросы GET, в отличие от POST (что позволило пользователям добавлять страницы в закладки), новый ресурс механизма обработки (для изображений, CSS, файлов JavaScript...). Кроме того, была добавлена поддержка AJAX. Сегодня JSF 2.2 (JSR 344) следит за этой простотой разработки и привносит много новых возможностей.

Что нового в JSF 2.2

JSF 2.2 (JSR 344) — это более развитая версия 2.1, которую можно считать шагом вперед, поскольку она вносит следующие новые возможности.

- *Совместимая с HTML разметка* — возможность создавать представление в чистом HTML-коде и добавлять с помощью JSF дополнительные особенности сервера.
- *Разрешение использования атрибута id для всего содержимого HTML5* — в отличие от предыдущих стандартов HTML, HTML5 позволяет использовать атрибут `id` для любых элементов.
- *Faces Flow* — значительное улучшение JSF. Понятие «поток» было добавлено в JSF 2.2. Потоки проводят пользователя через несколько экранов, таких как мастера, многоэкранные подписки, заказы и т. д.
- *Управление очередью запросов AJAX* — JSF добавляет поддержку для управления очередью AJAX-запросов. Если в течение этого периода задержки поступает несколько запросов, только самый последний отправляется на сервер.
- *Инъекции во все артефакты JSF* — в JSF 2.1 относительно немногие артефакты JSF могли использовать внедрение. В JSF 2.2 внедрение можно применять везде (для преобразователей, валидаторов, компонентов и т. д.).
- *Переход к CDI* — JSF 2.0 привнес собственный механизм определения области действия и компоненты-подложки. Сегодня CDI стал заменой для старых компонентов-подложек JSF, а JSF 2.2 поддерживает области действия непосредственно для CDI (это означает, что весь пакет `javax.faces.bean` становится устаревшим и может быть удален из новых версий JSF).

В табл. 10.1 перечислены основные пакеты, определенные сегодня в JavaServer Faces 2.2.

Таблица 10.1. Основные пакеты JavaServer Faces

Пакет	Описание
javax.faces	Основные API JavaServer Faces
javax.faces.application	API, используемый для связи бизнес-логики приложения с JavaServer Faces
javax.faces.component	Основные API для работы с компонентами пользовательского интерфейса
javax.faces.context	Классы и интерфейсы, определяющие информацию о состоянии по запросу

Пакет	Описание
javax.faces.convert	Классы и интерфейсы, определяющие преобразователи
javax.faces.event	Интерфейсы, описывающие события и слушатели событий, а также конкретная реализация классов событий
javax.faces.flow	Классы и API среды выполнения для Faces Flow
javax.faces.lifecycle	Классы и интерфейсы, определяющие управление жизненным циклом для реализации JSF
javax.faces.render	Классы и интерфейсы, определяющие модель отрисовки
javax.faces.validator	Интерфейс, определяющий модель валидатора и конкретные реализации классов валидатора
javax.faces.webapp	Классы, требуемые для интеграции JSF в веб-приложения, включая стандартные сервлеты Faces

Эталонные реализации

Mojarra, названный в честь рыбы, обитающей в Южной Америке и на Карибском побережье, является эталонной реализацией JSF 2.2 с открытым исходным кодом от компании Oracle. Mojarra доступен в GlassFish v4 и используется в следующем примере.

MyFaces — это проект Apache Software Foundation, который является организатором нескольких подпроектов, связанных с JavaServer Faces, в том числе MyFaces Core — реализацией среды выполнения JSF.

Создание JSF-страниц и компонентов

Создание страницы JSF отличается от написания HTML-страницы тем, что JSF является серверной технологией. Если вы напишете JSF-страницу и запустите ее прямо в браузере, то не увидите ожидаемое графическое представление. Страница JSF должна быть отрисована в HTML на сервере перед отправкой в браузер. Разработчик может объединить JSF-компоненты с HTML и CSS для задания необходимых стилей, если захочет, но все это будет отрисовано сервером.

Страницу JSF можно рассматривать как дерево компонентов, которое использует несколько библиотек тегов (как определенных спецификацией JSF, так и написанных самостоятельно или сторонними разработчиками). Тело страницы компилируется в дерево графических объектов Java, и их значения привязываются к компоненту-подложке. Таким образом, страница имеет богатый жизненный цикл, который обрабатывает много различных фаз.

Структура страницы JSF

Страница JSF — это практически файл XHTML, определяющий список библиотек тегов в заголовке. Ее тело содержит графическое представление. В качестве примера снова используем страницу, представляющую форму для создания новой книги (вернитесь к рис. 10.1). В листинге 10.7 показаны эти два фрагмента. Код `xmlns:h="http://xmlns.jcp.org/jsf/html"` импортирует библиотеку тегов, которая

называется `http://xmlns.jcp.org/jsf/html` и задает ей псевдоним `h`. Затем этот псевдоним используется в теле страницы, когда нам необходимо отобразить определенный компонент, который принадлежит библиотеке тегов (`<h:panelGrid>` или `<h:inputText>`). В этом примере задействована только одна библиотека тегов, но вы можете иметь столько библиотек, сколько вам нужно.

Листинг 10.7. Страница JSF с использованием библиотеки тегов и компонентов

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

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

<h:head>
  <title>Создать новую книгу</title>
</h:head>
<h:body>
  <h1>Создать новую книгу</h1>
  <hr/>
  <h:form>
    <h:panelGrid columns="2">
      <h:outputLabel value="ISBN : "/>
      <h:inputText value="#{bookController.book.isbn}"/>

      <h:outputLabel value="Название :"/>
      <h:inputText value="#{bookController.book.title}"/>

      <h:outputLabel value="Цена : "/>
      <h:inputText value="#{bookController.book.price}"/>

      <h:outputLabel value="Описание : "/>
      <h:inputTextarea value="#{bookController.book.description}" cols="20"
rows="5"/>

      <h:outputLabel value="Количество страниц : "/>
      <h:inputText value="#{bookController.book.nbOfPage}"/>

      <h:outputLabel value="Иллюстрации : "/>
      <h:selectBooleanCheckbox value="#{bookController.book.illustrations}"/>

    </h:panelGrid>
    <h:commandButton value="Создать книгу" action="#{bookController.
doCreateBook}"/>

  </h:form>
  <hr/>
  <h:outputText value="APress — Изучаем Java EE 7" style="font-style:
italic"/>
</h:body>
</html>
```

Заголовок

Заголовок страницы JSF можно рассматривать как механизм импорта Java: это место, где вы объявляете набор библиотек компонентов, которые будет использовать страница. В следующем фрагменте кода после пролога на XML следует объявление типа документа (DTD) `xhtml1-transitional.dtd`. Корневой элемент страницы — `html` из пространства имен `http://www.w3.org/1999/xhtml`. Далее идет набор пространств имен XML, объявляющий библиотеки тегов, используемых на страницах JSF с определенным префиксом (`h` и `f`):

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

В табл. 10.2 перечислены все библиотеки тегов, которые определены в спецификациях JSF и JSTL и могут быть использованы на странице Facelets. Там есть основные компоненты JSF Core и графические компоненты HTML, а также библиотеки шаблонов и композитные библиотеки (позволяющие создавать пользовательские компоненты). Есть также библиотеки ядра и библиотеки функций, которые являются частью JSTL.

Таблица 10.2. Библиотеки тегов, допускаемые к использованию в Facelets PDL

URI	Префикс	Описание
http://xmlns.jcp.org/jsf/html	h	Библиотека классов содержит компоненты и их HTML-отрисовки (h:commandButton, h:comandLink, h:inputText и т. д.)
http://xmlns.jcp.org/jsf/core	f	Библиотека содержит пользовательские действия, которые независимы от созданных ранее отрисовок (f:selectItem, f:validateLength, f:convertNumber и т. д.)
http://xmlns.jcp.org/jsf/facelets	ui	Теги этой библиотеки добавляют поддержку шаблонов
http://xmlns.jcp.org/jsf/composite	composite	Библиотека тегов применяется для объявления и определения составных компонентов
http://xmlns.jcp.org/jsp/jstl/core	c	Facelets-страницы могут использовать некоторые основные библиотеки тегов JSP (<c:if/>,<c:forEach/> или <c:catch/>)
http://xmlns.jcp.org/jsp/jstl/functions	fn	Facelets-страницы могут использовать все функции библиотек тегов JSP

Body

Как показано в листинге 10.7, тело страницы JSF описывает набор хорошо организованных графических (и неграфических) компонентов, которые, после того

как оказались на сервере, предоставят вам требуемое HTML-представление. Страница JSF — это дерево компонентов, таких как `<h:outputText>`, которые были преобразованы в дерево компонентов пользовательского интерфейса. Компонент пользовательского интерфейса — это класс Java, который явно или неявно наследует от абстрактного класса `javax.faces.component.UIComponent`. Этот класс определяет методы навигации по дереву компонентов, взаимодействуя с компонентами-подложками, а также проверяя и преобразуя данные так же, как и механизм отрисовки.

Спецификация JSF предоставляет несколько встроенных HTML-компонентов и компонентов ядра, необходимых для создания веб-приложения. Все эти компоненты наследуют от `UIComponent` и перечислены в табл. 10.3. Обратите внимание, что все классы определены в пакете `javax.faces.component`. Далее в разделе «HTML-компоненты JSF» вы увидите HTML-компоненты, которые располагаются в `javax.faces.component.html`.

Таблица 10.3. Стандартные компоненты пользовательского интерфейса JSF

Компонент	Описание
UIColumn	Представляет колонку в родительском компоненте UIData
UICommand	Представляет графические компоненты, такие как кнопки, гиперссылки или меню
UIComponent	Базовый класс для всех компонентов пользовательского интерфейса в JSF
UIComponentBase	Класс, созданный для удобства, реализующий конкретное поведение по умолчанию для всех методов, определенных в UIComponent
UIData	Поддерживает связывание данных с коллекциями объектов, часто применяется для отрисовки таблиц, списков и деревьев
UIForm	Представляет форму пользовательского ввода и является контейнером других компонентов
UIGraphic	Выводит изображения
UIInput	Представляет компоненты, предназначенные для ввода данных, такие как поля ввода, текстовые области и т. д.
UIMessage, UIMessages	Отвечает за вывод одного или нескольких сообщений для определенного UIComponent
UIOutcomeTarget	Представляет графические кнопки и гиперссылки, позволяющие добавлять страницу в закладки
UIOutput	Представляет компоненты, предназначенные для вывода, такие как метки или другие формы вывода текста
UIPanel	Представляет компоненты пользовательского интерфейса, которые служат контейнерами для других компонентов, не требуя при этом отправки формы
UIParameter	Представляет информацию, которая не требует отрисовки
UISelectBoolean	Представляет флажки
UISelectItem, UISelectItems	Представляет один или несколько элементов в списке

Компонент	Описание
UISelectOne, UISelectMany	Представляет такие компоненты, как раскрывающиеся списки или группы флажков, и позволяют выбирать один или несколько элементов
UIViewAction	Представляет вызов метода, который происходит при запросе обработки жизненного цикла
UIViewParameter	Представляет двунаправленное связывание между параметром запроса и свойством компонента-подложки
UIViewRoot	Представляет корень дерева компонентов и не имеет графической отрисовки

Для того чтобы иметь HTML-представление страницы, описанной в листинге 10.7, среда выполнения JSF фактически использует HTML-представление каждого компонента, определенного на странице. Эти компоненты HTML наследуют от одного из компонентов, перечисленных в табл. 10.3. Например, `<h:outputLabel>` определяется классом `javax.faces.component.html.HtmlOutputLabel`, который наследует от `UIOutput`. Компонент `<h:panelGrid>` определяется в классе `javax.faces.component.html.HtmlPanelGrid`, который наследует от `UIPanel`, и т. д. XML в листинге 10.8 является представлением в виде дерева страницы, описанной в листинге 10.7.

Листинг 10.8. Упрощенный вид дерева компонентов XML newBook

```
<UIViewRoot>
  <html xmlns="http://www.w3.org/1999/xhtml">
    <UIOutput><title>Создать новую книгу</title></UIOutput>
    <UIOutput>
      <h1>Создать новую книгу</h1>
      <hr/>

      <HtmlForm>
        <HtmlPanelGrid>

          <HtmlOutputLabel value="ISBN : "/>
          <HtmlInputText/>

          <HtmlOutputLabel value="Название :"/>
          <HtmlInputText/>

          <HtmlOutputLabel value="Цена : "/>
          <HtmlInputText/>

          <HtmlOutputLabel value="Описание : "/>
          <HtmlInputTextarea/>

          <HtmlOutputLabel value="Количество страниц : "/>
          <HtmlInputText/>

          <HtmlOutputLabel value="Иллюстрации : "/>
          <HtmlSelectBooleanCheckbox/>
```

```
        </HtmlPanelGrid>

        <HtmlCommandButton value="Создать книгу"/>
      </HtmlForm>

      <hr/>
      <HtmlOutputText value="APress — Изучаем Java EE 7"
style="font-style: italic"/>

    </UIOutput>
  </html>
</UIViewRoot>
```

СОВЕТ

Вы легко можете получить представление дерева компонентов страницы JSF, просто добавив везде тег <ui:debug> и нажав сочетание клавиш Ctrl+Shift+D.

Жизненный цикл

Страница JSF — это дерево компонентов со специфичным жизненным циклом. Вы должны понимать этот цикл, чтобы знать, когда проверяются компоненты или обновляется модель. Нажатие кнопки вызывает отправку запроса от вашего браузера на сервер. Этот запрос переводится в событие, которое может быть обработано логикой вашего приложения на сервере. Все входные данные пользователя проходят фазу проверки перед обновлением модели и вызовом бизнес-логики. JSF ответственен за то, чтобы каждый графический компонент (как предки, так и потомки) был правильно отображен в браузере. На рис. 10.6 показаны различные этапы жизненного цикла страницы JSF.

Жизненный цикл JSF делится на шесть отдельных этапов.

1. *Восстановление вида.* JSF находит целевое представление и применяет к нему пользовательский ввод. Если это первое посещение страницы клиентом, JSF создает представление как компонент `UIViewRoot` (корень дерева компонентов, которые составляют определенную страницу, что показано в листинге 10.8). Если это не первый запрос, то ранее сохраненный компонент `UIViewRoot` извлекается для обработки текущего запроса HTTP.
2. *Применение параметров запроса.* Параметры, которые появились из запроса (из полей ввода в веб-форме, из cookies или из заголовков запроса), применяются к различным компонентам страницы. Обратите внимание, что обновляют свое состояние только компоненты пользовательского интерфейса, но не компоненты-подложки, составляющие модель.
3. *Проверка процесса.* После предыдущих шагов компоненты пользовательского интерфейса имеют значения. При проверке обработки JSF проходит по дереву компонентов и опрашивает каждый из них, чтобы убедиться, что его значение является приемлемым. Если преобразование и проверка проходят успешно для всех компонентов, то жизненный цикл продолжается на следующем этапе. В противном случае жизненный цикл переходит на шаг *отображения ответа*, выводя при этом соответствующие сообщения об ошибке проверки и преобразования.

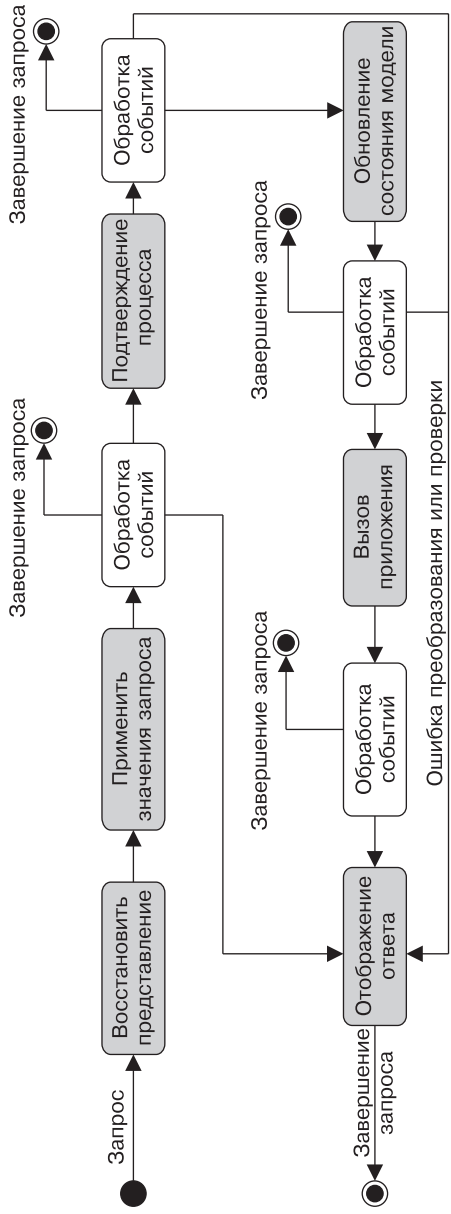


Рис. 10.6. Жизненный цикл JSF

4. *Обновление значений модели*. Все проверенные значения компонентов связываются с соответствующими компонентами-подложками.
5. *Вызов приложения*. Теперь вы можете выполнить бизнес-логику. Любое выбранное действие будет выполнено компонентом-подложкой, здесь вступает в действие навигация, поскольку она возвращает ответ отрисовщика.
6. *Отрисовка ответа*. Ответ будет отображен клиенту. Вторичная цель этой фазы заключается в сохранении состояния представления, так что он может быть восстановлен на этапе восстановления представления, если пользователь запрашивает его снова.

Поток исполнения цикла «запрос/ответ» может как протекать через каждую фазу, так и опускать некоторые из них, в зависимости от запроса и того, что происходит во время обработки. Если случается ошибка, поток выполнения моментально переходит к фазе отрисовки ответа. Обратите внимание, что четыре фазы могут генерировать сообщения: применение параметров запроса, проверка процесса, обновление значений модели и вызов приложения. Независимо от того, есть сообщения или нет, выходные данные отправляются пользователю в фазе отрисовки ответа.

Анатомия компонентов JSF

Страница JSF представляет собой дерево компонентов. Некоторые из этих компонентов имеют HTML-представления, другие — нет, они позволяют вам проверять данные, преобразовывать их или разрешить вызов AJAX. Вы можете использовать несколько компонентов для разработки веб-страниц:

- встроенные компоненты JSF: HTML, Core и Templating;
- теги JSTL;
- ваши собственные компоненты;
- компоненты сторонних производителей (с открытым исходным кодом или коммерческие).

Иногда этим компонентам нужны внешние ресурсы, такие как изображения, CSS или JavaScript. JSF очень грамотно управляет ими, позволяя связать ваши ресурсы в зависимости от локали, библиотеки или ее номера версии (подробнее об этом читайте в разделе «Управление ресурсами» данной главы). Как вы увидите, страницы JSF также могут иметь доступ к неявным объектам, позволяющим компонентам использовать параметры запроса HTTP, информацию из cookies или заголовков HTTP.

HTML-компоненты JSF

Архитектура JSF была разработана так, чтобы не зависеть от любого протокола или языка разметки и позволить создавать приложения для HTML-клиентов, которые работают по протоколу HTTP. Пользовательский интерфейс конкретной веб-страницы создается при объединении компонентов. Компоненты обеспечивают конкретные функции для взаимодействия с конечным пользователем (метки, таблицы, флажки и т. д.). JSF предоставляет несколько встроенных компонентов HTML, которые покрывают большую часть общих требований.

Страница — это дерево компонентов. Каждый компонент представлен классом, который наследует от `javax.faces.component.UIComponent` и имеет свойства, методы и события. Компоненты в дереве связаны отношениями «родитель — потомок» с другими компонентами, начиная с корневого элемента дерева, который является экземпляром типа `UIViewRoot` (см. листинг 10.8). Сосредоточимся на использовании этих компонентов на веб-страницах. Пакет `javax.faces.component.html` описывает HTML-компоненты.

Команды

Команды (`UICommand`) — это элементы управления, на которые пользователь может нажать, чтобы вызвать действие. Такие компоненты, как правило, отрисовываются в виде кнопки или гиперссылки. В табл. 10.4 перечислены два тега команд, которые могут быть использованы.

Таблица 10.4. Теги команд

Тег	Класс	Описание
<h:command-Button>	HtmlCommand-Button	Представляет HTML-элемент, предназначенный для ввода данных кнопки Отправить или Очистить
<h:command-Link>	HtmlCommand-Link	Представляет HTML-элемент для гиперссылки, который действует как кнопка Отправить. Этот компонент должен быть помещен внутрь формы

Если на вашу страницу необходимо добавить кнопку Отправить или Очистить, изображения, на которых можно щелкнуть, или гиперссылки, вызывающие событие, то можно сделать это следующим образом:

```
<h:commandButton value="Отправить"/>
<h:commandButton type="reset" value="Очистить"/>
<h:commandButton image="book.gif" title="Изображение"/>
<h:commandLink>Гиперссылка</h:commandLink>
```

По умолчанию `commandButton` имеет тип `submit`, но он может быть изменен на `reset` (`type="reset"`). Если вы хотите превратить кнопку в изображение, не используя атрибут `value` (это название кнопки), вместо этого следует задать атрибут `image`, чтобы указать путь к рисунку, который вы хотите отобразить. Ниже приведен графический результат предыдущего кода.

Обе кнопки и ссылка имеют атрибут action для вызова метода компонента-подложки. Например, чтобы вызвать метод doNew() класса BookController, используйте следующее утверждение EL для того, чтобы указать его в атрибуте action:

```
<h:commandLink action="#{bookController.doNew}">
  Создать новую книгу
</h:commandLink>
```

Цели

Предыдущие компоненты команд вызывают действие на компонентах-подложках путем генерации HTTP-запроса POST. Если вам нужна кнопка или ссылка для перехода на другую страницу путем создания HTTP-запроса GET, используйте компоненты целей (UIOutcomeTarget), определенные в табл. 10.5. Отмечу, что эти компоненты позволяют добавлять страницы в закладки в приложениях JSF.

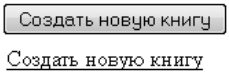
Таблица 10.5. Теги целей

Тег	Класс	Описание
<h:button>	HtmlOutcomeTarget-Button	Отрисовывает элемент ввода данных HTML для кнопки (при нажатии создается HTTP-запрос GET)
<h:link>	HtmlOutcomeTarget-Link	Отрисовывает элемент привязки HTML <a> (при нажатии создается HTTP-запрос GET)

Если вам нужно перейти с одной страницы на другую, не вызывая компоненты-подложки, можно использовать эти компоненты следующим образом

```
<h:button outcome="newBook.xhtml" value="A bookmarkable button link"/>
<h:link outcome="newBook.xhtml" value="A bookmarkable link"/>
```

Графическое представление выглядит так, как и ожидалось.



Компоненты ввода

Компоненты ввода (UIInput) — это те компоненты, которые отображают текущее значение пользователю и позволяют ему вводить различные виды текстовой информации. Это могут быть текстовые поля, текстовые области или компоненты, предназначенные для ввода пароля и скрытых данных. Теги этих компонентов перечислены в табл. 10.6.

Таблица 10.6. Теги компонентов ввода

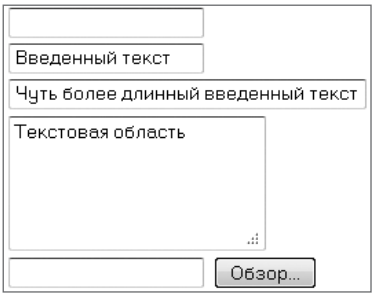
Тег	Класс	Описание
<h:input-Hidden>	HtmlInputHidden	Представляет HTML-элемент ввода для скрытых данных (полезен для переноса значений от страницы к странице вне сессии)

Тег	Класс	Описание
<h:input-Secret>	HtmlInputSecret	Представляет HTML-элемент ввода для паролей. При повторной загрузке страницы любое ранее введенное значение не будет отображено (из соображений безопасности), если только свойство redisplay не имеет значения true
<h:input-Text>	HtmlInputText	Представляет HTML-элемент ввода для текста
<h:input-Textarea>	HtmlInputTextarea	Представляет текстовую область в HTML
<h:input-File>	HtmlInputFile	Позволяет просматривать каталог, выбирать и загружать файл

Многие веб-страницы содержат формы, в которых пользователь должен ввести данные или войти в систему с помощью пароля. Компоненты ввода имеют несколько атрибутов, позволяющих изменять их длину, содержимое и внешний вид, а именно:

```
<h:inputHidden value="Скрытые данные"/>
<h:inputSecret maxlength="8"/>
<h:inputText value="Введенный текст"/>
<h:inputText size="40" value="Чуть более длинный введенный текст"/>
<h:inputTextarea rows="4" cols="20" value="Текстовая область"/>
<h:inputFile/>
```

Все компоненты имеют значение `value`, предназначенное для установки значения по умолчанию. Вы можете использовать атрибут `maxlength` для проверки того, что длина введенного текста не превосходит заданное значение, или атрибут `size` для изменения стандартного размера компонента. Предыдущий код будет иметь такое графическое представление.



Компоненты вывода

Компоненты вывода (классы, которые наследуют от `UIOutput`) отображают значение, при необходимости извлекаемое из компонентов-подложек, выражений или фиксированного текста. Пользователь не может непосредственно изменить значение, потому что оно предназначено исключительно для демонстрационных целей. В табл. 10.7 показаны все доступные теги вывода.

Таблица 10.7. Теги вывода

Тег	Класс	Описание
<h:outputLabel>	HtmlOutputLabel	Отображает элемент HTML <label>
<h:outputLink>	HtmlOutputLink	Отображает элемент привязки HTML <a>
<h:outputText>	HtmlOutputText	Выводит текст
<h:outputFormat>	HtmlOutputFormat	Отрисовывает параметризованный текст

Большинство веб-страниц должны отображать текст. Вы можете сделать это с помощью обычных HTML-элементов, но теги вывода JSF позволяют отображать содержимое переменной, связанной с компонентами-подложками через EL. Вы можете отображать текст с помощью тега `<h:outputText>` и гиперссылки — с помощью `<h:outputLink>`. Обратите внимание, что разница между тегами `<h:commandLink>` и `<h:outputLink>` заключается в том, что последний отображает ссылку, но не вызывает никаких методов компонентов-подложек при нажатии. Он просто создает внешнюю ссылку или привязку.

```
<h:outputLabel value="#{bookController.book.title}"/>
<h:outputText value="Текст"/>
<h:outputLink value=" http://www.apress.com/">Ссылка</h:outputLink>
<h:outputFormat value="Добро пожаловать {0}. Вы можете купить {1} товаров">
  <f:param value="#{user.firstName}" />
  <f:param value="#{user.itemsBought}" />
</h:outputFormat>
```

Приведенный выше код не имеет специального графического представления, это просто текст. Ниже приводится HTML-вариант:

```
<label>Название книги</label>
Текст
<a href=" http://www.apress.com/">Ссылка</a>
Добро пожаловать, Пол. Вы можете купить 5 товаров
```

Компоненты выбора

Флажки, переключатели и раскрывающиеся списки представляют собой компоненты (табл. 10.8), которые используются для выбора одного или нескольких значений из списка. Компоненты выбора позволяют пользователю выбирать между определенным набором доступных вариантов.

Таблица 10.8. Теги выбора

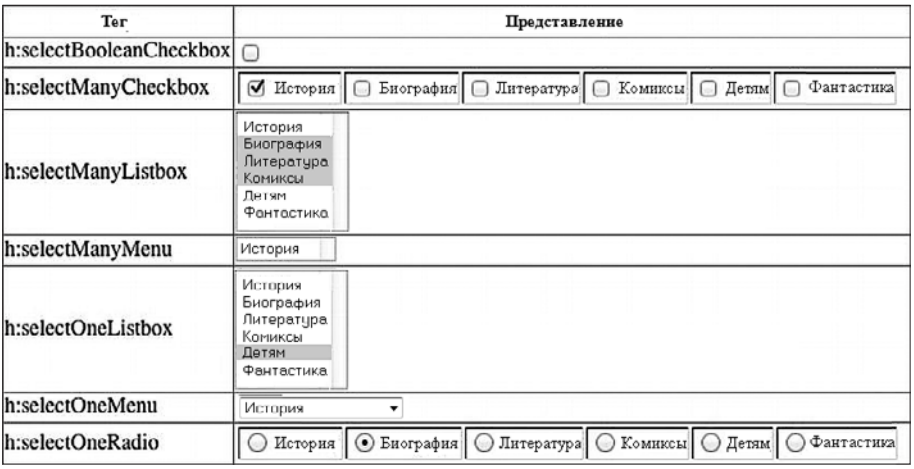
Тег	Класс	Описание
<h:selectBoolean-Checkbox>	HtmlSelectBoolean-Checkbox	Отображается один флажок, представляющий двоичное значение. Флажок будет установлен в зависимости от значения свойства
<h:selectMany-Checkbox>	HtmlSelectMany-Checkbox	Отрисовывается список флажков
<h:selectMany-Listbox>	HtmlSelectMany-Listbox	Отрисовывается компонент множественного выбора, где могут быть выбраны один или несколько вариантов

Тег	Класс	Описание
<h:selectMany-Menu>	HtmlSelectMany-Menu	Отрисовывается HTML-элемент <select>
<h:selectOne-Listbox>	HtmlSelectOne-Listbox	Отрисовывается компонент одиночного выбора, где может быть выбран только один вариант
<h:selectOneMenu>	HtmlSelectOneMenu	Отрисовывается компонент одиночного выбора, где может быть выбран только один вариант. Показывает только один доступный вариант в любой момент времени
<h:selectOneRadio>	HtmlSelectOneRadio	Отрисовывает список переключателей

Теги, перечисленные в этой таблице, имеют графическое представление, но требуют наличия других тегов для хранения доступных вариантов: `<f:selectItem>` или `<f:selectItems>`. Эти теги вложены в графические компоненты и создают дополнительные доступные параметры. Чтобы представить список жанров книг, вы должны вложить набор тегов `<f:selectItem>` в тег `<h:selectOneMenu>`.

```
<h:selectOneMenu>
  <f:selectItem itemLabel="История"/>
  <f:selectItem itemLabel="Биография"/>
  <f:selectItem itemLabel="Литература"/>
  <f:selectItem itemLabel="Комиксы"/>
  <f:selectItem itemLabel="Детям"/>
  <f:selectItem itemLabel="Фантастика"/>
</h:selectOneMenu>
```

Следующий рисунок демонстрирует все возможные представления. Некоторые списки позволяют выбрать несколько вариантов, другие — только один, и, как для всех других компонентов, вы можете привязать значение компонента-подложки (`List`, `Set` и т. д.) непосредственно к одному из списков.



Графика

Существует только один компонент для отображения изображений: <h:graphicImage> (класс HtmlGraphicImage). Он показывает пользователю графику, на которой тот может щелкнуть кнопкой мыши. Этот тег отображается как HTML-элемент <img>. Он имеет несколько атрибутов, позволяющих изменить размер изображения, использовать рисунок в качестве карты гиперсвязей и т. д. Вы можете вывести изображение, которое привязано к свойству в компоненте-подложке и располагается в файловой системе или базе данных. Следующий код отображает изображение и изменяет его размер:

```
<h:graphicImage value="book.gif" height="200" width="320"/>
```

Сетка и таблицы

Очень часто данные должны быть отображены в виде таблицы (табл. 10.9). JSF содержит очень мощный тег <h:dataTable>, который выполняет перебор списка элементов и создает таблицу. Таблицы также могут быть использованы для создания макета пользовательского интерфейса, основанного на сетке. В этом случае вы можете применить теги <h:panelGrid> и <h:panelGroup> для демонстрации компонентов. Тег <h:dataTable> предназначен для отображения таблицы, содержащей данные некоторых моделей вашего приложения, в то время как <h:panelGrid> применяется для макетов и размещения элементов.

Таблица 10.9. Теги сетки и таблиц

Тег	Класс	Описание
<h:dataTable>	HtmlDataTable	Представляет набор повторяющихся данных, которые будут отрисованы в HTML-элементе <table>
<h:column>	HtmlColumn	Отрисовывает одну колонку данных внутри компонента <h:dataTable>
<h:panelGrid>	HtmlPanelGrid	Отрисовывает HTML-элемент <table>
<h:panelGroup>	HtmlPanelGroup	Является контейнером для компонентов, которые могут быть встроены в <h:panelGrid>

В отличие от <h:dataTable> тег <h:panelGrid> не использует базовую модель данных для отображения строк данных. Скорее, этот компонент является контейнером макета, который отрисовывает другие компоненты JSF в сетке из строк и столбцов. Вы можете указать, сколько столбцов используется для отображения компонентов, и <h:panelGrid> определит, сколько понадобится строк. Атрибут columns хранит количество столбцов, которое необходимо отрисовать, прежде чем отрисовывать новую строку. Например, в результате действия следующего кода будет отрисовано три столбца и две строки (отобразятся сначала строки, а затем столбцы):

```
<h:panelGrid columns="3" border="1">
  <h:outputLabel value="Один"/>
  <h:outputLabel value="Два"/>
  <h:outputLabel value="Три"/>
  <h:outputLabel value="Четыре"/>
```

```
  <h:outputLabel value="Пять"/>
  <h:outputLabel value="Шесть"/>
</h:panelGrid>
```

Если вы хотите объединить несколько компонентов в один столбец, то можете использовать тег <h:panelGroup>, что отрисует всех его потомков как один компонент. Вы также можете определить верхний и нижний колонтитулы с помощью специального тега <f:facet>:

```
<h:panelGrid columns="3" border="1">
  <f:facet name="header">
    <h:outputText value="Заголовок"/>
  </f:facet>
  <h:outputLabel value="Один"/>
  <h:outputLabel value="Два"/>
  <h:outputLabel value="Три"/>
  <h:outputLabel value="Четыре"/>
  <h:outputLabel value="Пять"/>
  <h:outputLabel value="Шесть"/>
  <f:facet name="footer">
    h:outputText value="Footer"/>
  </f:facet>
</h:panelGrid>
```

Две таблицы, описанные ранее, будут иметь следующее графическое представление, одна из них будет содержать верхний и нижний колонтитулы.

Один	Два	Три
Четыре	Пять	Шесть

Верхний колонтитул		
Один	Два	Три
Четыре	Пять	Шесть
Нижний колонтитул		

Сообщения об ошибках

Приложения иногда могут генерировать исключения из-за неверного формата данных или по техническим причинам. Но поскольку для пользователей это важно, следует отображать в интерфейсе достаточное количество информации, чтобы обратить их внимание на проблему и помочь исправить ее (но, конечно, не весь стек исключения). Механизм управления сообщениями об ошибке будет включать на страницу теги <h:message> и <h:messages> (табл. 10.10). Тег <h:message> привязан к конкретным компонентам, в то время как <h:messages> может обеспечить глобальный механизм оповещения для всех компонентов на странице.

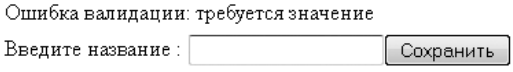
Таблица 10.10. Теги сообщений

Тег	Класс	Описание
<h:message>	HtmlMessage	Отрисовывает одно сообщение об ошибке
<h:messages>	HtmlMessages	Отрисовывает все ожидающие сообщения об ошибке

Сообщения могут иметь различную степень важности (`INFO`, `WARN`, `ERROR` и `FATAL`), каждая из которых соответствует стилю CSS (`infoStyle`, `warnStyle`, `errorStyle` и `fatalStyle`). Таким образом, каждый тип сообщения может иметь разный тип отрисовки. Например, следующий код будет отображать все сообщения в красном цвете:

```
<h:messages style="color:red"/>
<h:form>
   Введите название:
   <h:inputText value="#{bookController.title}" required="true"/>
   <h:commandButton action="#{bookController.save}" value="Сохранить"/>
</h:form>
```

На экране появится поле ввода, привязанное к свойству компонента-подложки. Это свойство обязательно для заполнения, поэтому, если пользователь нажимает кнопку Сохранить, не заполняя его, будет отображено сообщение об ошибке.



Разное

Существуют и другие теги, которые не имеют никакого графического представления, но имеют эквиваленты в HTML. Родной HTML-тег может быть использован и будет работать. Для тегов JSF доступны дополнительные атрибуты, упрощающие разработку. Так, например, вы можете добавить на страницу библиотеку JavaScript с помощью стандартного HTML тега `<script type="text/javascript">`. Но с помощью тега JSF `<h:outputScript>` можно использовать новые ресурсы, что вы увидите в разделе «Управление ресурсами» данной главы. В табл. 10.11 показаны эти теги.

Таблица 10.11. Прочие теги

Тег	Класс	Описание
<h:body>	HtmlBody	Отрисовывает HTML-элемент <body>
<h:head>	HtmlHead	Отрисовывает HTML-элемент <head>
<h:form>	HtmlForm	Отрисовывает HTML-элемент <form>
<h:doctype>	HtmlDoctype	Отрисовывает HTML-элемент <doctype>
<h:outputScript>	Output	Отрисовывает разметку для тега <script>
<h:outputStylesheet>	Output	Отрисовывает разметку для элемента <link>

Базовые атрибуты

Каждый тег компонентов HTML, представленный вам, может отличаться сложностью и количеством возможных атрибутов. Например, тег <h:message> имеет всего несколько атрибутов, включая стиль CSS (например, <h:messages style="color:red"/>), но тег <h:inputFile>, будучи более сложным компонентом, имеет больше атрибутов (местоположение загружаемого файла, просматриваемый каталог и т. д.). Однако все компоненты HTML имеют несколько общих атрибутов, описанных в табл. 10.12.

Таблица 10.12. Некоторые базовые атрибуты

Атрибут	Описание
id	Идентификатор компонента
rendered	Двоичное значение, указывающее, следует ли отрисовывать компонент
value	Значение компонента (текст или утверждение EL)
converter	Имя класса преобразователя
validator	Имя класса валидатора
required	Если имеет значение true, требует введения значения в соответствующее поле

Основные теги JSF

HTML-теги компонентов JSF могут быть использованы только для отрисовки HTML (помните, что JSF безразличен к конечному устройству отображения и может использовать несколько средств визуализации). Но некоторые основные действия JavaServer Faces не зависят от конкретной визуализации. Они не имеют никакого визуального представления и в основном используются для передачи атрибутов или параметров других компонентов, а также для преобразования или валидации. Эти теги содержатся в табл. 10.13.

Таблица 10.13. Основные теги JSF

Тег	Описание
<f:facet>	Добавляет фасет к компоненту
<f:attribute>	Добавляет атрибут компоненту
<f:param>	Создает компонент-потомок с параметром
<f:actionListener>	Добавляет компоненту слушатель действий, изменений значений или действий над свойствами
<f:valueChangeListener>	
<f:propertyActionListener>	
<f:phaseListener>	Добавляет на страницу слушатель фаз
<f:converter>	Добавляет компоненту произвольный преобразователь или использует особый преобразователь даты/времени и чисел
<f:convertDateTime>	
<f:convertNumber>	

Продолжение ⇩

Таблица 10.13 *(продолжение)*

Тег	Описание
<f:validator>	Добавляет компоненту произвольный валидатор или использует определенный валидатор длины или регулярных выражений
<f:validatorDoubleRange>	
<f:validatorLongRange>	
<f:validateLength>	
<f:validateRegex>	
<f:validateBean>	Определяет группы проверки для Bean Validation (см. главу 3)
<f:loadBundle>	Загружает ресурсы
<f:selectItem>	Определяет один или несколько элементов для выбора одного или нескольких компонентов
<f:selectItems>	
<f:ajax>	Включает поведение AJAX

Теги шаблонов JSF

Типичное веб-приложение содержит несколько страниц, которые имеют одинаковый внешний вид, верхний и нижний колонтитулы и т. д. Facelets позволяет определить макет в файле шаблона, который может быть использован для всех страниц в веб-приложении. Шаблон определяет области (с помощью тега `<ui:insert>`), где содержимое может быть заменено. Клиентская страница затем использует теги `<ui:component>`, `<ui:composition>`, `<ui:fragment>` или `<ui:decorate>` для встраивания фрагментов в шаблон. В табл. 10.14 перечислены различные теги шаблонов.

Таблица 10.14. Теги шаблонов

Тег	Описание
<ui:composition>	Определяет композицию, которая может использовать шаблон. Несколько композиций могут применять один и тот же шаблон
<ui:component>	Создает компонент
<ui:debug>	Собирает информацию об отладке
<ui:define>	Определяет содержимое, которое вставлено на страницу с помощью шаблона
<ui:decorate>	Позволяет декорировать содержимое на странице
<ui:fragment>	Добавляет фрагмент страницы
<ui:include>	Инкапсулирует и повторно использует содержимое среди нескольких страниц XHTML
<ui:insert>	Вставляет содержимое в шаблон
<ui:param>	Передает параметры во включенный файл (с использованием тега <ui:include> или шаблона)
<ui:repeat>	Проходит по всему списку объектов
<ui:remove>	Удаляет содержимое страницы

В качестве примера повторно используем страницу, представляющую форму для создания новой книги (вернитесь к рис. 10.1). Вы можете сказать, что название — это верхний колонтитул (Создать новую книгу), а нижний — это текст Apress — Изучаем Java EE 7. Таким образом, шаблон, который называется `layout.xhtml`, будет выглядеть как код в листинге 10.9.

Листинг 10.9. Файл layout.xhtml, выступающий в качестве шаблона

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets">

<h:head>
  <title><ui:insert name="title">Название по умолчанию</ui:insert></title>
</h:head>
<h:body>
  <h1><ui:insert name="title">Название по умолчанию</ui:insert></h1>
  <hr/>

  <ui:insert name="content">Содержимое по умолчанию</ui:insert>

  <hr/>
  <h:outputText value="APress – Изучаем Java EE 7" style="font-style:
italic"/>

</h:body>
</html>
```

Шаблон в листинге 10.9 сначала должен определить необходимую библиотеку тегов (`xmlns:ui="http://xmlns.jcp.org/jsf/Facelets"`). Затем он использует тег `<ui:insert>` для того, чтобы вставить атрибут `title` в теги `<title>` и `<h1>`. Тело страницы будет вставлено в атрибут `content`.

Чтобы использовать этот шаблон, страница `newBook.xhtml`, показанная в листинге 10.10, должна объявить, какой шаблон ей нужен (`<ui:composition Template="layout.xhtml">`). Далее следует связать атрибуты, определенные в теге `<ui:define>`, с тегом `<ui:insert>` в шаблоне. Например, заголовок одной страницы — Создать новую книгу. Этот текст хранится в переменной `title` (`<ui:define name="title">`), который связан с соответствующим тегом `<ui:insert name="title">`. То же самое можно определить для остальной части страницы, которая вставляется в переменную `content` (`<ui:define name="content">`).

Листинг 10.10. Страница newBook.xhtml, использующая в качестве шаблона страницу layout.xhtml

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets">

<ui:composition template="layout.xhtml">

  <ui:define name="title">Создать новую книгу</ui:define>
```

```
    <ui:define name="content">

      <h:form>
        <h:panelGrid columns="2">
          <h:outputLabel value="ISBN : "/>
          <h:inputText value="#{bookController.book.isbn}"/>

          <h:outputLabel value="Название :"/>
          <h:inputText value="#{bookController.book.title}"/>

          <h:outputLabel value="Цена : "/>
          <h:inputText value="#{bookController.book.price}"/>

          <h:outputLabel value="Описание : "/>
          <h:inputTextarea value="#{bookController.book.description}" cols="20"
rows="5"/>

          <h:outputLabel value="Количество страниц : "/>
          <h:inputText value="#{bookController.book.nbOfPage}"/>

          <h:outputLabel value="Иллюстрации : "/>
          <h:selectBooleanCheckbox value="#{bookController.book.
illustrations}"/>

        </h:panelGrid>
        <h:commandButton value="Создать книгу" action="#{bookController.
doCreateBook}"/>
      </h:form>

    </ui:define>

</ui:composition>
</html>
```

Теги JSTL

Стандартная библиотека тегов JSP стандартизирует несколько общих действий с помощью тегов. Эти действия варьируются от настройки значения объекта до перехвата исключений, управления структурой потока с условиями и итераторами, анализа XML и доступа к базам данных. Таблица 10.15 содержит эти теги и URI (унифицированный идентификатор ресурса), используемые для ссылок на библиотеки и типичные префиксы.

Таблица 10.15. Библиотеки тегов JSTL

Область функциональсти	URI	Типичный префикс
Основные функции	http://xmlns.jcp.org/jsp/jstl/core	c
Обработка XML	http://xmlns.jcp.org/jsp/jstl/xml	x
I18N и форматирование	http://xmlns.jcp.org/jsp/jstl/fmt	fmt

Область функциональсти	URI	Типичный префикс
Доступ к базам данных	http://xmlns.jcp.org/jsp/jstl/sql	sql
Функции	http://xmlns.jcp.org/jsp/jstl/functions	fn

Но Facelets предоставляет только подмножество JSTL-библиотек. На самом деле он предоставляет только подмножество библиотек тегов основных функций и полную версию библиотек тегов i18n и форматирования (не включая библиотеки по обработке XML, библиотеки функций или библиотеки доступа к базе данных).

Основные действия

Основные действия предоставляют теги для работы с переменными, исправления ошибок, проведения испытаний и выполнения циклов и итераций. В табл. 10.16 отображены действия, поддерживаемые Facelets.

Таблица 10.16. Основные действия, поддерживаемые Facelets

Действие	Описание
<c:set>	Устанавливает значение переменной внутри области действия
<c:catch>	Ловит исключение, наследующее от класса java.lang.Throwable для любых встроенных действий
<c:if>	Оценивает, является ли выражение правдивым
<c:choose>	Предоставляет взаимно эксклюзивные условия
<c:when>	Представляет собой альтернативу действию <c:choose>
<c:otherwise>	Представляет собой последнюю альтернативу действию <c:choose>
<c:forEach>	Повторяет вложенное тело в коллекции объектов

Для того чтобы увидеть эти теги в действии, напишем страницу JSF, которая проходит по числам от 3 до 15, проверяет, является число четным или нечетным, и отображает эту информацию перед каждым номером (листинг 10.11).

Листинг 10.11. Страница JSF, показывающая список четных и нечетных чисел

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:c="http://xmlns.jcp.org/jsp/jstl/core">
<h:head>
  <title>Проверка номера</title>
</h:head>
<h:body>
  <h1>Проверка номера</h1>
  <hr/>

  <c:set var="upperLimit" value="20"/>
  < c:forEach var="i" begin="3" end="#{upperLimit - 5}">

    <c:choose>
```

```
        <c:when test="#{i%2 == 0}">
          <h:outputText value="#{i} – четное"/><br/>
        </c:when>
        <c:otherwise>
          <h:outputText value="#{i} – нечетное"/><br/>
        </c:otherwise>
      </c:choose>

    </c:forEach>

    <hr/>
    <h:outputText value="APress – Изучаем Java EE 7" style="font-style:
  italic"/>
  </h:body>
  </html>
```

Для использования библиотеки основных тегов страница должна импортировать ее URI с префиксом (то есть пространство имен XML). В коде листинга 10.11 значение `20` присваивается переменной `upperlimit` с помощью тега `<c:set>`, а затем выполняются итерации от числа 3 до 15 (20 – 5). Вы можете увидеть использование EL в арифметическом выражении `#{upperlimit-5}`. Внутри цикла значение индекса (переменная `i`) проверяется на четность (`<c:when test="#{i%2 == 0}">`). Тег `<h:outputText>` отображает значение индекса с текстом (`"#{i} – четное"`).

Логика создается только с помощью тегов, и эта страница полностью совместима с XML, поскольку использует язык разметки, который программисты, работающие не с Java, также смогут прочесть и понять.

Действия форматирования

Действия форматирования обеспечивают поддержку форматирования дат, чисел, валют и процентов, а также интернационализации (i18n). Вы можете получить или установить локаль и часовой пояс, а также кодировку веб-страницы. В табл. 10.17 перечислены действия, содержащиеся в библиотеке форматирования.

Таблица 10.17. Действия форматирования

Действия	Описание
<fmt:message>	Интернационализирует сообщение, получая его из набора ресурсов
<fmt:param>	Передает параметр для тега <fmt:message>
<fmt:bundle>	Определяет набор ресурсов
<fmt:setLocale>	Устанавливает локаль
<fmt:requestEncoding>	Устанавливает кодировку символов запроса
<fmt:timeZone>	Определяет часовой пояс, согласно которому будет форматироваться информация о времени
<fmt:setTimeZone>	Хранит указанный часовой пояс в переменной
<fmt:formatNumber>	Форматирует числовое значение (число, валюту, процент) с точки зрения локали

Действия	Описание
<fmt:parseNumber>	Преобразует строковое представление чисел, валют и процентов
<fmt:formatDate>	Форматирует даты и время с точки зрения локали
<fmt:parseDate>	Преобразует строковое представление в дату и время

Для понимания этих тегов взгляните на страницу JSF, показанную в листинге 10.12, который использует некоторые из этих действий для форматирования дат и чисел, а также для интернационализации валют.

Листинг 10.12. Форматирование дат и чисел на странице JSF

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:c="http://xmlns.jcp.org/jsp/jstl/core"
      xmlns:fmt="http://xmlns.jcp.org/jsp/jstl/fmt">
<h:head><title>Форматирование дат</title></h:head>
<h:body>

  Даты<br/>
  <c:set var="now" value="#{bookController.currentDate}"/>
  <fmt:formatDate type="time" value="#{now}"/><br/>
  <fmt:formatDate type="date" value="#{now}"/><br/>
  <fmt:formatDate type="both" dateStyle="short" timeStyle="short" ➥
    value="#{now}"/><br/>
  <fmt:formatDate type="both" dateStyle="long" timeStyle="long" ➥
    value="#{now}"/><br/>

  Валюта<br/>
  <fmt:setLocale value="en_us"/>
  <fmt:formatNumber value="20.50" type="currency"/><br/>
  <fmt:setLocale value="en_gb"/>
  <fmt:formatNumber value="20.50" type="currency"/><br/>

</h:body>
</html>
```

Страница импортирует библиотеки core и format и использует префиксы c и fmt. Строка <c:ser var="now" value="#{}bookController.currentDate}"/> устанавливает текущую дату переменной с именем now. Тег <fmt:formatDate> форматирует эту дату с использованием различных шаблонов: показывает только время, только дату, время и дату. Строка <fmt:setlocale value="en_US"/> задает локаль Соединенных Штатов и форматирует валюту так, что результат становится равным $20,50. Затем локаль изменяется на Великобританию, и в результате устанавливается валюта — фунты стерлингов. Эта страница JSF выдает следующий результат:

```
Даты
11:31:12
14 may 2013
14/05/1311:31
```

```
14 may 201311:31:12 CET

Валюта
$20.50
£20.50
```

Управление ресурсами

Большинству компонентов могут потребоваться внешние ресурсы для того, чтобы быть отрисованными надлежащим образом. Тег `<h:graphicImage>` нуждается во внешнем рисунке для отображения, тег `<h:commandButton>` также может выводить изображения в виде кнопки, тег `<h:outputScript>` ссылается на внешний файл JavaScript. Кроме того, компоненты могут применять стили CSS (с помощью тега `<h:outputStylesheet>`). В JSF ресурс — это статический элемент, который можно передать компоненту так, что он может быть отображен (рисунки) или обработан (JavaScript, CSS) в браузере.

Предыдущие версии JSF не имели средств для работы с ресурсами. Если вы хотели передать ресурс, нужно было положить его в веб-каталог, чтобы браузер клиента смог получить к нему доступ. Проблема состояла в том, что при необходимости обновить файл вы должны были бы заменить файл в каталоге, а также иметь разные каталоги для локализованных ресурсов (например, изображений с английским или португальским текстом). Версия JSF 2.0 принесла поддержку этой функциональности, так что теперь вы можете поместить ваши ресурсы непосредственно в каталог `resources` или упаковать их в отдельный JAR-файл (с опциональным указанием версии и/или локали). Ресурс может быть помещен в корне веб-приложения по следующему пути:

```
resources/<resourceIdentifier>
```

или в JAR-файл в каталоге `WEB-INF/lib`:

```
WEB-INF/lib/{*.jar}/META-INF/resources/<resourceIdentifier>
```

Тег `<resourceIdentifier>` содержит несколько вложенных каталогов, определенных следующим образом:

```
[localePrefix/][libraryName/][libVersion/]resourceName[/resourceVersion]
```

Элементы этого идентификатора ресурса, заключенные в квадратные скобки, необязательны. Локальный префикс состоит из кода языка и следующего за ним опционального кода страны (`EN`, `en_US`, `Pt`, `pt_BR`). Как показывает эта строка, вы можете добавить номер версии к имени библиотеки или непосредственно к ресурсу. Ниже приведены структуры ресурсов, которые вы можете создать:

```
book.gif
en/book.gif
en_us/book.gif
en/myLibrary/book.gif
myLibrary/book.gif
myLibrary/1_0/book.gif
myLibrary/1_0/book.gif/2_3.gif
```

Далее вы можете использовать ресурс, например изображение `book.gif`, непосредственно в компоненте `<h:graphicImage>` (`value="book.gif"`) либо указав имя библиотеки (`library="MyLibrary"`). Ресурс с правильной локалью клиент подгрузит автоматически. Обратите внимание, что `resource` — это неявный объект (подробнее об этом поговорим позже) и что вы можете также использовать синтаксис `resource['book.gif']` или `resource['MyLibrary:book.gif']` для доступа к ресурсу.

```
<h:graphicImage value="book.gif" />
<h:graphicImage value="book.gif" library="myLibrary" />
<h:graphicImage value="#{resource['book.gif']}" />
<h:graphicImage value="#{resource['myLibrary:book.gif']}" />
```

ПРИМЕЧАНИЕ

По умолчанию ресурсы хранятся в каталоге resource. Начиная с версии JSF 2.2, вы можете изменить это значение по умолчанию, установив значение переменной javax.faces.WEBAPP_RESOURCES_DIRECTORY в дескрипторе развертывания web.xml. Если этот параметр установлен, то среда выполнения JSF интерпретирует его значение как путь относительно корня веб-приложения, по которому и следует искать ресурсы.

Неявные объекты

Чтобы получить доступ к ресурсу, можно использовать неявный объект, который называется `resources` (например, `#{resources['book.gif']}`). Этот объект не должен быть объявлен, он доступен на всех страницах JSF. Такие объекты называются *неявными объектами* (или *неявными переменными*). Неявные объекты — это специальные идентификаторы, которые сопоставляются с конкретными часто используемыми объектами. Они неявные, поскольку страница имеет к ним доступ и может использовать их без необходимости их явного объявления или инициализации. Неявные объекты используются в выражениях EL. В табл. 10.18 перечислены все неявные объекты, к которым может иметь доступ страница.

Таблица 10.18. Неявные объекты

Неявный объект	Описание	Возвращаемый тип
application	Представляет среду веб-приложения. Используется для получения конфигурационных параметров приложения	Object
applicationScope	Преобразует имена глобальных атрибутов приложения в их значения	Map
component	Указывает на текущий компонент	UIComponent
cc	Указывает на текущий составной компонент	UIComponent
cookie	Определяет объект типа Map, содержащий имена cookies (являющиеся ключами) и объекты типа Cookie	Map
facesContext	Указывает на объект типа FacesContext для текущего запроса	FacesContext

Продолжение ⇩

Таблица 10.18 *(продолжение)*

Неявный объект	Описание	Возвращаемый тип
flash	Представляет объект, использующий флеш (подробнее об областях видимости читайте в главе 11)	Object
header	Преобразует имя HTTP-заголовка в значение заголовка типа String	Map
headerValues	Преобразует имя HTTP-заголовка в набор всех значений заголовка типа String[]	Map
initParam	Преобразует имена параметров инициализации контекста к значениям типа String	Map
param	Преобразует имена параметров запроса к одному значению параметра типа String	Map
paramValues	Преобразует имена параметров запроса к набору всех значений параметра типа String[]	Map
request	Представляет объект запроса HTTP	Object
requestScope	Преобразует имена атрибутов запроса к их значениям	Map
resource	Представляет объект ресурса	Object
session	Представляет объект сеанса HTTP	Object
sessionScope	Преобразует имена атрибутов сеанса к их значениям	Map
view	Представляет текущее представление	UIViewRoot
viewScope	Преобразует имена атрибутов представления к их значениям	Map

Все эти неявные объекты — реальные объекты с интерфейсами, и, как только вы узнаете их API (смотрите технические характеристики или документацию), вы сможете получить доступ к их атрибутам с помощью EL. Например, строка `#{}view.locale` позволит получить локаль текущего представления (`en_US`, `pt_PT` и т. д.). Если вы храните объект `book` в области действия сеанса, вы можете получить доступ к нему так: `#{sessionScope.book}`. Вы даже можете использовать более богатый алгоритм для отображения всех HTTP-заголовков и их значений, что показано в листинге 10.13.

Листинг 10.13. Страницы, где с использованием неявных объектов отображаются локаль и HTTP-заголовки

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:c="http://xmlns.jcp.org/jsp/jstl/core">
<h:body>
  <h1>Неявные объекты</h1>
  <hr/>

  <h3>Локаль</h3>
```

```
    <h:outputText value="#{view.locale}"/>

    <h3>headerValues</h3>
    <c:forEach var="parameter" items="#{headerValues}">
      <h:outputText value="#{parameter.key}"/> =
      <c:forEach var="value" items="#{parameter.value}">
        <h:outputText value="#{value}" escape="false"/><br/>
      </c:forEach>
    </c:forEach>

    <hr/>
    <h:outputText value="APress — Изучаем Java EE 7" style="font-style:
  italic"/>
  </h:body>
  </html>
```

Если выполнить код листинга 10.13, вы получите страницу, показанную на рис. 10.7.

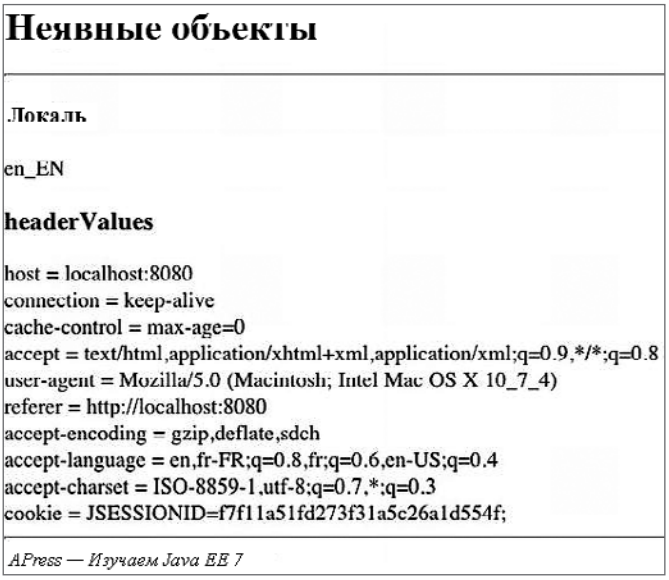


Рис. 10.7. Страница JSF, отображающая значения неявных объектов

Составные компоненты

Все рассмотренные выше компоненты и библиотеки тегов являются частью JSF и поставляются с любой реализацией, которая следует спецификации. Поскольку JSF основан на многократно используемых компонентах, он предоставляет дизайн, позволяющий легко создавать и интегрировать собственные или сторонние компоненты в приложения.

Ранее я упоминал, что все названные выше компоненты явно или неявно наследуют от класса `javax.faces.component.UIComponent`. До появления версии JSF 2.0, если вы хотели создать свой собственный компонент, вы должны были унаследовать класс компонента, который наиболее близок к вашему (`UICommand`, `UIGraphic`, `UIOutput` и др.), объявить его в файле `faces-config.xml`, а также обеспечить обработчик тегов и отрисовщик. Эти шаги были сложными, и другие веб-фреймворки, такие как Facelets, показали, что можно гораздо проще создать мощные компоненты. Здесь следует рассмотреть составные компоненты: они дают разработчикам возможность писать настоящие, многоразовые составляющие пользовательского интерфейса JSF без кода Java или конфигурации XML.

Новый подход предполагает создание страницы XHTML, которая содержит компоненты, а затем использование ее в качестве компонента на других страницах. Эта страница XHTML затем рассматривается как реальный компонент, который может поддерживать работу с валидаторами, преобразователями и слушателями. Внутри составного компонента разрешено использовать любую допустимую разметку, в том числе шаблоны. Составные компоненты обрабатываются как ресурсы и, следовательно, должны находиться в пределах стандартных каталогов resources. В табл. 10.19 перечислены все теги, участвующие в создании и определении составного компонента.

Таблица 10.19. Теги, используемые для объявления и определения составных компонентов

Тег	Описание
<composite:interface>	Описывает контракт компонента
<composite:implementation>	Определяет реализацию компонента
<composite:attribute>	Определяет атрибут, который может быть задан экземпляру компонента. Их может не быть вообще либо быть сразу несколько внутри раздела <composite:interface>
<composite:facet>	Указывает, что этот компонент поддерживает фасет
<composite:insertFacet>	Используется в разделе <composite:interface>. Вставленный фасет будет отрисован в компоненте
<composite:insertChildren>	Применяется в разделе <composite:interface>. Любые компоненты-потомки или шаблоны внутри компонента будут вставлены в отрисовываемый результат
<composite:valueHolder>	Указывает, что компонент, чей контракт определен в разделе <composite:interface>, в который этот компонент вложен, предоставляет реализацию типа ValueHolder
<composite:renderFacet>	Используется в разделе <composite:interface> для отрисовки фасета
<composite:extension>	Применяется в разделе <composite:interface> для того, чтобы включить XML-содержимое, не определенное в спецификации JSF
<composite:editableValue-Holder>	Указывает, что компонент, чей контракт определен в разделе <composite:interface>, в который этот компонент вложен, предоставляет реализацию типа EditableValueHolder

Тег	Описание
<composite:clientBehavior>	Определяет контракт для поведений, которые могут изменить отрисованное содержимое компонента
<composite:actionSource>	Указывает, что компонент, чей контракт определен в разделе <composite:interface>, в который этот компонент вложен, предоставляет реализацию типа ActionSource

Рассмотрим пример, который показывает, как легко можно создать графический компонент и использовать его на других страницах. Вы, возможно, помните из предыдущих глав, что приложение CD-BookStore продает два разных типа товара: книги и компакт-диски. В главе 4 я представлял их как три различных объекта: `Book` и `CD`, наследующие от класса `Item`. Класс `Item` содержит общие атрибуты (название, цену и описание), а затем для классов `Book` и `CD` указываются специализированные свойства (`isbn`, `publisher`, `nbOfPage` и `illustrations` для `Book`; `musicCompany`, `numberOfCDs`, `totalDuration` и `genre` для `CD`). Если вы хотите, чтобы ваше веб-приложение имело возможность создавать новые книги и компакт-диски, понадобятся две разные формы. Но общие атрибуты класса `Item` могут располагаться на отдельной странице, которая будет выступать в качестве компонента. На рис. 10.8 показаны эти две страницы.

Рис. 10.8. Две формы, одна из которых предназначена для создания компакт-диска, а вторая — для создания книги

Создадим составной компонент с двумя полями для ввода текста (название и цена) и одной текстовой областью (для описания). Подход к написанию компонентов JSF 2.2 относительно близок к такому подходу в Java. Вы должны сначала написать интерфейс — `<composite:interface>` (листинг 10.14), который выступает в качестве отправной точки для компонента. В нем описываются имена и параметры, используемые компонентом (в данном случае это компонент `item` и CSS-свойство `style`). Затем идет

реализация — <composite:implementation>. Это тело, графическое представление компонента, написанное на XHTML, и любые теги JSF, которые вы видели ранее. Интерфейс и реализация находятся на одной и той же XHTML-странице.

Листинг 10.14. Составной компонент newItem.xhtml

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

<composite:interface>
  <composite:attribute name="item" required="true"/>
  <composite:attribute name="style" required="false"/>
</composite:interface>

<composite:implementation>
  <h:panelGrid columns="2" style="#{cc.attrs.style}">
    <h:outputLabel value="Название : "/>
    <h:inputText value="#{cc.attrs.item.title}"/>

    <h:outputLabel value="Цена : "/>
    <h:inputText value="#{cc.attrs.item.price}"/>

    <h:outputLabel value="Описание : "/>
    <h:inputTextarea value="#{cc.attrs.item.description}" cols="20" rows="5"/>

  </h:panelGrid>
</composite:implementation>
</html>
```

Компонент в листинге 10.14 объявляет интерфейс с двумя атрибутами: item, представляющий компонент Item (и подклассы Book и CD), и style, который является стилем CSS, используемым для отрисовки. Эти атрибуты затем используются для реализации составного компонента с помощью следующего синтаксиса:

```
#{cc.attrs.[attributeName]}
```

Этот код указывает, что метод GetAttributes() будет вызван для текущего составного компонента (из табл. 10.18 вы помните, что cc — неявный объект, который указывает на текущий компонент). Внутри возвращенного объекта типа Map код будет искать значение с ключом attributeName. Так компонент использует атрибуты, которые определены в интерфейсе.

Прежде чем я объясню, как работать с этим компонентом, вспомните вопрос управления ресурсами, рассмотренный ранее в этой главе, и понятие конфигурации с помощью исключения. Компонент должен быть сохранен в файл, который располагается внутри библиотеки ресурсов. Например, файл для этого примера называется newItem.xhtml и сохраняется по адресу /resources/apress. Если вы оставите все значения по умолчанию, то для использования этого компонента вам нужно объявить библиотеку apress и указать для нее пространство имен XML (ago в следующем коде):

```
<html xmlns:ago="http://xmlns.jcp.org/jsf/composite/apress">
```

Затем вызовите компонент newItem (название страницы), передав все необходимые параметры: параметр item, который относится к сущности Item, и style, который является необязательным параметром, относящимся к стилю CSS.

```
<ago:newItem item="#{itemController.book} "style="myCssStyle"/>
<ago:newItem item="#{itemController.cd} "/>
```

Чтобы дать вам общее представление о том, как включить компонент, в листинге 10.15 показана страница newBook.xhtml, представляющая форму, которая позволяет ввести данные о книге. Она включает в себя составной компонент newItem и добавляет поля ввода для ISBN, количества страниц, а также флажок, позволяющий указать, есть ли в книге иллюстрации.

Листинг 10.15. Страница newBook.xhtml, использующая компонент newItem

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:ago="http://xmlns.jcp.org/jsf/composite/apress">
<h:head>
  <title>Создать новую книгу</title>
</h:head>
<h:body>
  <h1>Создать новую книгу</h1>
  <hr/>
  <h:form>
    <ago:newItem item="#{itemController.book}"/>
    <h:panelGrid columns="2">
      <h:outputLabel value="ISBN : "/>
      <h:inputText value="#{itemController.book.isbn}"/>
      <h:outputLabel value="Количество страниц : "/>
      <h:inputText value="#{itemController.book.nbOfPage}"/>
      <h:outputLabel value="Иллюстрации : "/>
      <h:selectBooleanCheckbox value="#{itemController.book.illustrations}"/>
    </h:panelGrid>
    <h:commandButton value="Создать книгу" action="#{itemController.
doCreateBook}"/>
  </h:form>
  <hr/>
  <h:outputText value="APress - Изучаем Java EE 7" style="font-style:
italic"/>
</h:body>
</html>
```

ПРИМЕЧАНИЕ

Создание компонентов — непростая задача, так что разработчиков нужно вдохновлять писать собственные бизнес-компоненты, чтобы использовать их между страницами. Но некоторые компоненты настолько распространены (календарь, палитра, панель прокрутки, «хлебные крошки»...), что вы не должны развивать их дальше, нужно просто загрузить их из сторонних библиотек компонентов. Сегодня существует три основных проекта с открытым кодом, которые включают сотни уже готовых графических компонентов: PrimeFaces, RichFaces и IceFaces. Проверьте свои библиотеки компонентов, и вы можете получить необходимый компонент, не прибегая к разработке.

Резюме

В данной главе рассматривались различные способы создания веб-страниц с использованием клиентских языков, таких как HTML, XHTML или CSS, а также создание динамического содержимого с помощью серверных страниц JSF. Сегодня гонка между пользовательскими интерфейсами продолжается, распространение насыщенных приложений для Рабочего стола (RDA), насыщенных интернет-приложений (RIA), приложений для мобильных устройств (смартфонов или планшетов) и т. д. Несколько лет назад JSF вошел в эту гонку и сегодня, с момента выпуска версии 2.2, проводит свою собственную.

Архитектура JSF основана на компонентах и насыщенном API для разработки отрисовщиков, преобразователей, валидаторов и т. д. Он может отрисовать и преобразовать несколько языков, несмотря на то что предпочтительным PDL для веб-страниц является Facelets. JSF предоставляет набор стандартных виджетов (кнопки, гиперссылки, флажки и т. д.) и гибкую модель для создания ваших собственных виджетов (составные компоненты). Кроме того, сообщество разработало несколько сторонних библиотек компонентов с открытым исходным кодом.

Но JSF может взаимодействовать с компонентами-подложками для обработки данных и навигации по страницам. Из главы 11 вы узнаете, как работает навигация, как компоненты связаны с компонентами-подложками, а также научитесь писать собственный преобразователь и валидатор.

Глава 11
Обработка и навигация

В главе 10 я показал вам, как создать веб-страницы с помощью JSF-компонентов. Тем не менее создавать страницы с применением графических компонентов недостаточно, эти страницы должны взаимодействовать с системой серверов, осуществлять навигацию по другим страницам, а также проверять и преобразовывать данные. Спецификация JSF очень насыщенна — компоненты-подложки позволяют вызывать бизнес-уровень и осуществлять навигацию по вашему приложению. Набор классов дает возможность преобразовывать значения компонентов в заданный тип и из него, а также проверять их соответствие бизнес-правилам. Интеграция с Bean Validation также облегчает проверку в JSF 2.2. С помощью аннотаций можно легко разрабатывать компоненты-подложки, собственные преобразователи и валидаторы.

JSF 2.2 упрощает и насыщает работу с динамическими пользовательскими интерфейсами. Он имеет встроенную поддержку вызовов AJAX. Спецификация поставляется с библиотекой JavaScript, что позволяет осуществлять асинхронные вызовы сервера и обновлять небольшую часть страницы.

Создание пользовательских интерфейсов, управление навигацией по всему приложению и синхронный либо асинхронный вызов бизнес-логики стали возможны потому, что JSF построен по шаблону «модель — представление — контроллер» (Model — View — Controller, MVC). Каждая часть существует отдельно от других, что позволяет изменять пользовательский интерфейс без ущерба для бизнес-логики и наоборот. В этой главе представлены все эти понятия, и она дополняет главу 10.

Шаблон MVC

JSF, как и большинство веб-платформ поощряет разделение обязанностей с использованием вариаций шаблона проектирования MVC. MVC — это архитектурный шаблон, предназначенный для изоляции бизнес-логики от пользовательского интерфейса. Бизнес-логика не должна перемешиваться с кодом пользовательского интерфейса. Если они смешаются, приложения станет намного труднее поддерживать и расширять. Применение MVC приводит к созданию слабосвязанного приложения; в приложениях такого типа проще изменить визуальный внешний вид приложения или основные правила бизнес-логики, не затрагивая другие компоненты.

Модель в MVC представляет собой данные приложения, отображение соответствует пользовательскому интерфейсу, а контроллер управляет связью между ними (рис. 11.1).

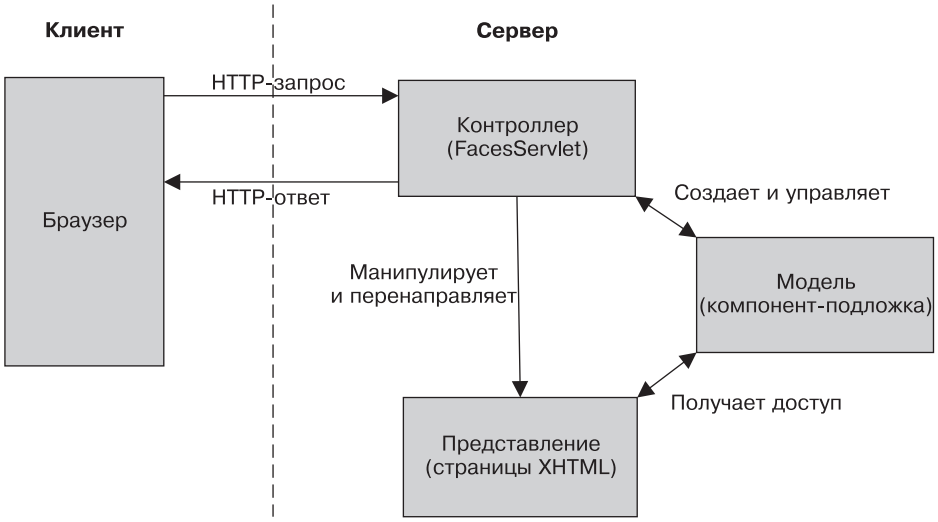


Рис. 11.1. Шаблон проектирования MVC

Модель представлена содержимым, которое часто хранится в базе данных и отображается в представлении. Модель строится, не заботясь о внешнем виде, в котором она предстанет пользователю. В JSF она может состоять из компонентов-подложек, вызовов EJB, сущностей JPA и т. д.

Представление в JSF — это фактически страница XHTML, когда мы имеем дело с веб-интерфейсами. Как объяснялось в главе 10, представление обеспечивает графическое отображение модели. Модель может также иметь несколько представлений, показывая книгу, например, в виде формы или списка.

Когда пользователь манипулирует представлением, тот информирует контроллер о желаемых изменениях. Затем контроллер собирает данные, преобразует и проверяет их, вызывает бизнес-логику и генерирует содержимое XHTML. В JSF контроллером является `FacesServlet`.

FacesServlet

`FacesServlet` — это реализация типа `javax.servlet.Servlet`. Она действует как центральный элемент контроллера, через который проходят все запросы пользователей. Как показано на рис. 11.2, при возникновении события (например, когда пользователь нажимает кнопку) уведомление о нем отправляется через HTTP на сервер и перехватывается объектом типа `javax.faces.webapp.FacesServlet`. Он изучает запрос и вызывает различные действия для модели с использованием компонентов-подложек.

`FacesServlet` принимает входящие запросы и передает управление объекту типа `javax.faces.lifecycle.Lifecycle`. Используя фабрику, он создает объект типа `javax.faces.context.FacesContext`, который содержит и обрабатывает всю информацию о состоянии для каждого запроса. Объект типа `Lifecycle` использует объект типа `FacesContext` в течение шести различных фаз жизненного цикла страницы (описан в главе 10) до отрисовки ответа.

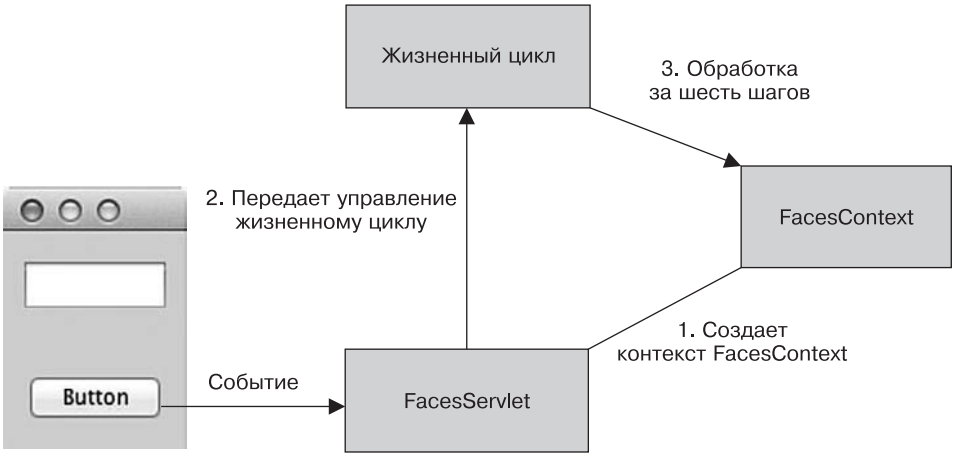


Рис. 11.2. Взаимодействия FacesServlet

Запросы пользователей обрабатываются средой выполнения JSF, которая объявляет объект типа `FacesServlet`, используя общее поведение, благодаря конфигурации с помощью исключения. Если параметры по умолчанию вас чем-то не устраивают, вы можете настроить `FacesServlet` в файле `web.xml`. Пример этого файла приведен в листинге 11.1.

Листинг 11.1. Определение FacesServlet в файле web.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee ➥
         http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">

  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>

  <context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
  </context-param>
  <context-param>
    <param-name>javax.faces.FACELETS_SKIP_COMMENTS</param-name>
    <param-value>true</param-value>
```

```
    </context-param>
</web-app>
```

В этом файле определяется объект типа `javax.faces.webapp.FacesServlet`: задаются его имя (`Faces Servlet`) и способ отображения. В данном примере все запросы, которые имеют расширение `.jsf`, управляются сервлетом, а любые запросы вида `http://localhost:8080/chapter11-1.0/newBook.jsf` будут обрабатываться JSF.

ПРИМЕЧАНИЕ

Стандартное отображение сервлетов для JSF-страниц — *.faces или /faces/*. В листинге 11.1 это значение по умолчанию было изменено на .jsf, но, если .faces вам подходит, можно опустить конфигурацию отображения сервлета.

Вы также можете настроить несколько параметров, касающихся JSF, внутри `<context-param>`. Некоторые из них перечислены в табл. 11.1.

Таблица 11.1. Некоторые специфические для JSF параметры

Параметр	Описание
javax.faces.CONFIG_FILES	Определяет список разделенных запятой зависящих от контекста путей, по которым реализация JSF будет искать ресурсы
javax.faces.DEFAULT_SUFFIX	Позволяет веб-приложению определить список альтернативных суффиксов для страниц, имеющих содержимое JSF (то есть с расширением .jsf)
javax.faces.FACELETS_BUFFER_SIZE	Размер буфера ответа. По умолчанию равно –1
javax.faces.FACELETS_REFRESH_PERIOD	Когда запрашивается страница, этот параметр будет использоваться как интервал, по истечении которого компилятор будет проверять наличие изменений. Значение –1 отключает эту проверку
javax.faces.FACELETS_SKIP_COMMENTS	Если установлено значение true, среда выполнения гарантирует, что XML-комментарии на странице Facelets не будут доставлены клиенту
javax.faces.LIFECYCLE_ID	Идентифицирует объект типа Lifecycle как использованный при обработке запросов JSF
javax.faces.STATE_SAVING_METHOD	Определяет местоположение сохранения состояния. Корректными значениями являются server, которое задается по умолчанию (обычно сохраняется в HttpSession), и client (сохраняется в скрытом поле при последующей отправке формы)
javax.faces.PROJECT_STAGE	Описывает, на каком этапе жизненного цикла находится это конкретное приложение JSF (Development, UnitTest, SystemTest или Production). Этот параметр может быть использован реализацией JSF для кэширования ресурсов, например, чтобы улучшить производительность на производстве

Параметр	Описание
javax.faces.DISABLE_FACELET_JSF_VIEWHANDLER	Значение true отключает Facelets как стандартный язык объявления страниц (PDL)
javax.faces.WEBAPP_RESOURCES_DIRECTORY	Если установлено значение этого параметра, среда выполнения JSF интерпретирует его как путь относительно корня веб-приложения, где должны располагаться ресурсы
javax.faces.LIBRARIES	Интерпретирует каждый файл, найденный в списке путей, разделенных двоеточием, как библиотеку тегов Facelets

FacesContext

JSF определяет абстрактный класс `javax.faces.context.FacesContext` для представления контекстной информации, связанной с обработкой входящих запросов и создания соответствующего ответа. Этот класс позволяет взаимодействовать с пользовательским интерфейсом и остальной частью среды JSF.

Чтобы получить к нему доступ, вы можете использовать на ваших страницах неявный объект `FacesContext` (см. главу 10 для обсуждения неявных объектов) или получить ссылку в ваш компонент-подложку с помощью статического метода `getCurrentInstance()`. Он вернет экземпляр класса `FacesContext` для текущего потока, а затем вы можете вызвать методы, перечисленные в табл. 11.2.

Таблица 11.2. Некоторые методы FacesContext

Метод	Описание
AddMessage	Присоединяет сообщение (информационное, предупреждающее, сообщение об ошибке либо сообщение о фатальной ошибке)
GetApplication	Возвращает объект типа Application, связанный с этим веб-приложением
GetAttributes	Возвращает объект типа Map, представляющий атрибуты, связанные с объектом типа FacesContext
getCurrentInstance	Возвращает объект типа FacesContext для запроса, который обрабатывается в текущем потоке
getELContext	Возвращает объект типа ELContext для текущего объекта типа FacesContext
getMaximumSeverity	Возвращает максимальную степень тяжести, записанную в любом FacesMessage, внесенном в очередь
GetMessages	Возвращает коллекцию объектов типа FacesMessage
getPartialViewContext	Возвращает объект типа PartialViewContext для заданного запроса. Он используется для внедрения логики в цикл управления обработкой/отрисовкой (например, для обработки AJAX)
getViewRoot	Возвращает корневой компонент, связанный с запросом

Продолжение ⇩

Таблица 11.2 *(продолжение)*

Метод	Описание
release	Высвобождает любые ресурсы, связанные с объектом типа FacesContext
renderResponse	Сигнализирует реализации JSF о том, что текущая фаза обрабатывающего запросы жизненного цикла была закончена, управление должно быть передано фазе «Отрисовать ответ», минуя любые фазы, которые еще не были выполнены
responseComplete	Сигнализирует реализации JSF о том, что HTTP-ответ для этого запроса уже был сгенерирован (например, переадресация HTTP), а также о том, что жизненный цикл обработки запросов должен прекратить свою работу, как только завершится текущая фаза

Faces-config.xml

Класс `FacesServlet` является внутренним в реализациях JSF, и, хотя у вас нет доступа к его коду, вам требуются метаданные, чтобы настроить его или некоторые свойства в файле `web.xml`, как показано в листинге 11.1. К настоящему времени вы привыкли к двум возможным вариантам работы с метаданными в Java EE 7: аннотации и дескрипторам развертывания XML (`/WEB-INF/faces-config.xml`).

До появления JSF 2.0 единственный выход заключался в использовании XML, сегодня же компоненты-подложки, преобразователи, слушатели событий, отрисовщики и валидаторы могут иметь аннотации, поэтому использование XML-файлов для конфигурации стало необязательным. Даже большая часть навигации между страницами может быть организована с помощью либо XML, либо Java-кода. В листинге 11.2 показан фрагмент файла `faces-config.xml`. В этом примере определяется локаль по умолчанию (`fr`), набор сообщения для интернационализации, а также некоторые правила навигации. Скоро вы увидите, как можно наладить систему навигации и без помощи `faces-config.xml`.

Листинг 11.2. Фрагмент файла faces-config.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config xmlns="http://xmlns.jcp.org/xml/ns/javaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee ➥
              http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd"
              version="2.2">
  <application>
    <locale-config>
      <default-locale>fr</default-locale>
    </locale-config>
    <resource-bundle>
      <base-name>messages</base-name>
      <var>msg</var>
    </resource-bundle>
  </application>
```

```
  <navigation-rule>
    <from-view-id>*</from-view-id>
    <navigation-case>
      <from-outcome>doCreateBook-success</from-outcome>
      <to-view-id>/listBooks.htm</to-view-id>
    </navigation-case>
  </navigation-rule>
</faces-config>
```

Создание компонентов-подложек

Как отмечалось ранее в этой главе, шаблон MVC поощряет разделение между моделью, представлением и контроллером: JSF-страницы являются представлением, а `FacesServlet` — контроллером. Компоненты-подложки — это шлюз к модели.

Компоненты-подложки — это аннотированные классы Java, они занимают центральное место в веб-приложениях. Они могут выполнять бизнес-логику (или делегировать ее выполнение компонентам EJB), обрабатывать навигацию между страницами и хранить данные. Типичное JSF-приложение включает в себя один или несколько компонентов-подложек, которые могут быть разделены между несколькими страницами. Эти данные хранятся в атрибутах компонентов-подложек, и действия, выполняемые на странице, вызывают метод компонента. Для связывания компонентов с компонентами-подложками следует использовать язык выражений (например, `#{bookController.book.title}`).

ПРИМЕЧАНИЕ

Я предпочитаю использовать термин «компонент-подложка», а не «управляемый компонент», но исторически они имеют одинаковое значение в JSF. «Управляемый компонент» относится к более общей модели компонентов в Java EE, которую я представил в главе 2 вместе с CDI. В Java EE любой POJO (Plain Old Java Object — «простой Java-объект в старом стиле»), управляемый контейнером или провайдером, называется управляемым компонентом. Поэтому компонент-подложка на самом деле является управляемым компонентом, но для JSF.

Создать компонент-подложку так же просто, как сущность EJB или JPA: это всего лишь класс Java с CDI-аннотацией `@Named` (листинг 11.3) и областью действия CDI (в данном случае это `@RequestScoped`). Они не используют записи файла `faces-config.xml`, вспомогательные классы и не наследуют от других классов. JSF 2.2 также задействует механизм конфигурации с помощью исключения, поэтому, указав только две аннотации, вы можете использовать значения по умолчанию и развертывать веб-приложение, имеющее компонент-подложку.

Листинг 11.3. Простой компонент-подложка BookController

```
@Named
@RequestScoped
public class BookController {
  private Book book = new Book();
  public String doCreateBook() {
    createBook(book);
```

```
    return "newBook.xhtml";
  }
  // Конструкторы и методы работы со свойствами
}
```

В листинге 11.3 показана программная модель компонентов-подложек: он обладает состоянием (атрибут `book`) в течение конкретного периода времени (область действия), определяет методы действий (`doCreateBook()`) и обрабатывает навигацию (возвращает страницу `newBook.xhtml`).

С помощью языка выражений компонент может связать его значения со свойствами (например, `<h:outputText value="#{bookController.book.isbn}"/>`) или вызовом метода (`<h:commandLink action="#{bookController.doCreateBook}">`).

Структура компонентов-подложек

Компоненты-подложки — это классы Java, которые управляются `FacesServlet`. Компоненты пользовательского интерфейса связаны со свойствами компонентов-подложек и могут вызывать методы действий. Компоненты-подложки должны следовать следующим требованиям.

- Класс должен иметь аннотацию `@javax.inject.Named` или XML-эквивалент в дескрипторе развертывания `faces-config.xml`.
- Класс должен иметь область действия (например, `@RequestScoped`).
- Класс должен быть определен как общедоступный, но не окончательный или абстрактный.
- Класс должен иметь общедоступный конструктор без аргументов, который будет использован контейнером для создания экземпляров этого класса.
- Класс не должен определять метод `finalize()`.
- Атрибуты должны включать общедоступные методы работы со свойствами для того, чтобы быть связанными с компонентом.

Следуя модели легкого использования Java EE 7, компоненты-подложки могут быть просто аннотированными POJO, устраняя необходимость конфигурации. Как вы помните, в главе 2 я представил стереотипы CDI. `@javax.enterprise.inject.Model` — идеальный стереотип для использования в компонентах-подложках, поскольку он имеет аннотации `@Named` и `@RequestScoped` (как показано в листинге 11.4). Таким образом, вы можете переписать `BookController`, продемонстрированный в листинге 11.3, заменив аннотации на `@Model` (листинг 11.5), и получить такое же поведение.

Листинг 11.4. Встроенный в CDI стереотип @Model

```
@Named
@RequestScoped
@Documented
@Stereotype
@Target({ TYPE, METHOD, FIELD })
@Retention(RUNTIME)
public @interface Model {
}
```

Листинг 11.5. Класс BookController, использующий стереотип @Model

```
@Model
public class BookController {
  private Book book = new Book();

  public String doCreateBook() {
    createBook(book);
    return "listBooks.xhtml";
  }
  // Конструкторы, методы работы со свойствами
}
```

Наличие у класса аннотации `@javax.inject.Named` позволяет языку выражений использовать эту страницу. С помощью аннотации `@Named` также можно указать другое имя для компонента-подложки (которое по умолчанию является именем класса и начинается со строчной буквы). Компоненты пользовательского интерфейса связаны со свойствами компонента-подложки, изменение имени по умолчанию влияет на способ вызова свойства или метода. В коде, представленном в листинге 11.6, компонент-подложка переименовывается в `MyBean`.

Листинг 11.6. Переименование компонента-подложки

```
@Named("myBean")
@RequestScoped
public class BookController06 {
   private Book book = new Book();
   public String doCreateBook() {
     createBook(book);
     return "listBooks.xhtml";
   }
   // Конструкторы, методы работы со свойствами
}
```

Чтобы вызвать атрибуты или методы этого компонента-подложки на ваших страницах, вам следует использовать переопределенное имя таким образом:

```
<h:outputText value="#{myBean.book.isbn}"/>
<h:form>
  <h:commandLink action="#{myBean.doCreateBook}">Создать новую
     книгу</h:commandLink>
</h:form>
```

ПРИМЕЧАНИЕ

До появления версии JSF 2.2 вы должны были аннотировать компонент-подложку с помощью аннотации @javax.faces.bean.ManagedBean. Это вводит в заблуждение, потому что данная аннотация имеет имя Managed Bean (Управляемый компонент), определяющее более общую компонентную модель Java EE, которая является частью JSR-250 Common Annotations (@javax.annotation.ManagedBean). Но что более важно, в версии JSF 2.2 была улучшена интеграция с CDI и теперь рекомендуется использовать аннотацию CDI @Named, а также области действия CDI. Пакет javax.faces.bean будет объявлен устаревшим в следующей версии JSF.

Области действия

Объекты, которые создаются как часть компонента-подложки, имеют определенный срок службы и могут или не могут быть доступными для компонентов пользовательского интерфейса или объектов в веб-приложении. Срок службы и доступность объекта известны как *область действия*. В веб-приложении (в компонентах-подложках и страницах) вы можете указать область действия объекта, используя разную продолжительность.

- *Приложение* — это наименее ограничительный вариант (используйте аннотацию `@ApplicationScoped` для вашего компонента-подложки) с самой большой продолжительностью жизни. Созданные объекты доступны во всех циклах запросов/ответов для всех клиентов, использующих веб-приложение, до тех пор, пока приложение активно. Эти объекты можно вызывать одновременно из нескольких источников, они должны быть потокобезопасными (устанавливается с помощью ключевого слова `syncronized`). Объекты с такой областью действия могут использовать другие объекты либо без области действия, либо с такой же областью действия.
- *Сессия* — объекты доступны для любых циклов запросов/ответов, которые принадлежат сессии клиента (`@SessionScoped`). Эти объекты имеют состояние, которое сохраняется между запросами и хранится до тех пор, пока сессия не будет завершена. Объекты с такой областью действия могут использовать и другие объекты либо без области действия, либо с такой же областью действия, либо с областью действия на уровне приложения.
- *Представление* — объекты доступны в пределах заданного представления, пока оно не изменится, и их состояние сохраняется до тех пор, пока пользователь не перейдет к новому представлению (в этот момент состояние стирается). Объекты с такой областью действия могут использовать и другие объекты либо без области действия, либо с такой же областью действия, либо с областью действия на уровне приложения или сеанса. Компоненты-подложки могут быть аннотированы с помощью `@ViewScoped`.
- *Запрос* — состояние доступно в начале запроса и до тех пор, пока клиенту не был отправлен ответ (аннотация `@RequestScoped`). Клиент может выполнять несколько запросов, но остаться на одном и том же представлении. Именно поэтому продолжительность `@ViewScoped` больше, чем `@RequestScoped`. Объекты с такой областью действия могут использовать и другие объекты либо без области действия, либо с такой же областью действия, либо с областью действия на уровне приложения или сеанса.
- *Момент* — эта область действия была введена в версии JSF 2.0 и представляет собой короткую коммуникацию. Это способ передачи временных объектов, которые распространяются на одном представлении и очищаются перед переходом к другому. Такая область действия может быть использована только программно, поскольку аннотации для нее не существует.
- *Поток* — объекты в этой области действия создаются, когда пользователь входит в указанный поток, и освобождаются, когда он выходит из потока (с помощью аннотации `@FlowScoped`).

ПРИМЕЧАНИЕ

Все аннотации областей видимости, используемые в JSF 2.2, теперь являются областями действия CDI (то есть все они аннотированы @javax.enterprise.context.NormalScope). Но они находятся в разных спецификациях: @ApplicationScoped, @RequestScoped и @SessionScoped являются частью CDI, а @ViewScoped и @FlowScoped определяются в спецификации JSF.

Вам следует быть осторожными при выборе области действия для ваших компонентов-подложек. Указывайте ровно такую область, которая им необходима. Компоненты-подложки с большой областью действия (например, `@ApplicationScoped`) увеличивают потребление памяти, и потенциальная необходимость сохранять их может привести к увеличению частоты использования диска и сети. Не имеет смысла задавать большую область действия для объекта, который используется только в одном компоненте. И наоборот, объект со слишком большим ограничением будет недоступен различным частям вашего приложения. Вы также должны иметь в виду одновременный доступ. Несколько сеансов, получающих доступ к одному и тому же компоненту с областью действия на уровне приложения, могут создать проблемы, связанные с потокобезопасностью.

Код в листинге 11.7 определяет компонент-подложку с областью действия на уровне приложения. Кроме того, здесь по созданию объекта (`@PostConstruct`) инициализируется атрибут `defaultBook`.

Листинг 11.7. Компонент-подложка с областью действия на уровне приложения

```
@Named
@ApplicationScoped
public class InitController {
  private Book defaultBook;
  @PostConstruct
  private void init() {
    defaultBook = new Book("default title", 0F, "default description");
  }
  // Конструкторы, методы работы со свойствами
}
```

Жизненный цикл и аннотации функций обратного вызова

В главе 10 объяснялся жизненный цикл страницы (он имеет шесть фаз, от момента получения запроса до отрисовки ответа). Компоненты-подложки также имеют жизненный цикл (рис. 11.3), который полностью отличается от жизненного цикла страницы. На самом деле жизненный цикл компонентов-подложек сходен с жизненным циклом управляемых компонентов, за исключением того, что если они и существуют, то только на протяжении срока, определенного областью действия.

Компоненты-подложки, работающие в контейнере-сервлете, могут использовать аннотации `@PostConstruct` и `@PreDestroy`. Когда контейнер создаст экземпляр компонента-подложки, он вызовет метод обратного вызова `@PostConstruct`, если таковой имеется. После этого этапа компонент-подложка будет привязан к области

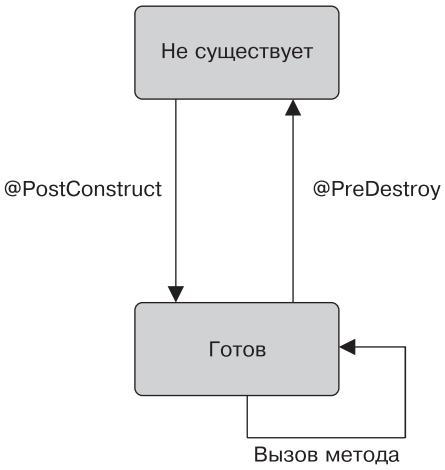


Рис. 11.3. Жизненный цикл компонента-подложки

действия и станет отвечать на запрос любого пользователя. Перед удалением компонента-подложки контейнер вызывает метод `@PreDestroy`. Эти методы могут быть использованы для инициализации атрибутов или для создания и высвобождения любого внешнего ресурса (см. листинг 11.7).

Обработка исключений и сообщений

Компоненты-подложки обрабатывают бизнес-логику, вызывают EJB, сохраняют и извлекают данные из баз данных и т. д., и иногда все может пойти не так, как планировалось. В этом случае пользователь должен быть проинформирован, чтобы он мог принять меры. Сообщения, предназначенные для этого, могут быть разделены на две категории: ошибки приложений (с участием бизнес-логики, базы данных или подключения к сети) и ошибки ввода данных пользователем (неправильный ISBN или пустые поля). Ошибки приложения могут создать совершенно другую страницу, попросив пользователя повторить попытку через несколько секунд или, например, позвонить по телефону горячей линии. Ошибки ввода данных могут отображаться на той же странице, при этом появится текст с описанием ошибки. Сообщения могут быть информационными, допустим указывать, что книга была успешно добавлена в базу данных.

В главе 10 вы видели теги, которые применяются для отображения сообщений на страницах (`<h:message>` и `<h:messages>`). Для создания таких сообщений JSF позволяет вам помещать события в очередь, вызвав метод компонента-подложки `facesContext.addMessage()`. Сигнатура метода выглядит так:

```
void addMessage(String clientId, FacesMessage message)
```

Этот метод добавляет объект типа `FacesMessage` в набор сообщений, которые будут отображаться. Первый параметр метода указывает идентификатор клиента. Он относится к расположению компонента пользовательского интерфейса, зарегистрировавшего сообщение, в объектной модели документа (DOM) (например, `bookForm:isbn`

относится к компоненту пользовательского интерфейса, который имеет идентификатор isbn внутри формы bookForm). Если ClientId содержит значение null, то сообщение не относится ни к одному компоненту и считается глобальным на всей странице. Сообщение состоит из обобщенного текста, подробного текста и степени тяжести (фатальная ошибка, ошибка, предупреждение и информация). Сообщения могут быть интернационализированы с использованием наборов сообщений.

```
FacesMessage(Severity severity, String summary, String detail)
```

Код в листинге 11.8 — это фрагмент компонента-подложки, создающего книгу. Если создание книги проходит успешно, в очередь помещается глобальное информационное сообщение (FacesMessage со степенью тяжести SEVERITY_INFO). Если перехватывается исключение, то в очередь сообщений на отображение добавляется сообщение об ошибке. Обратите внимание, что оба сообщения являются глобальными, потому что ClientId имеет значение null. С другой стороны, когда мы будем проверять значение isbn и название книги, введенные пользователем, мы захотим отобразить предупреждающее сообщение (SEVERITY_WARN), связанное с компонентом пользовательского интерфейса (например, bookForm:title является идентификатором DOM текстового поля title, расположенного в форме bookForm).

Листинг 11.8. Добавление сообщений различной степени тяжести

```
@Named
@RequestScoped
public class BookController {
  @Inject
  private BookEJB bookEJB;
  private Book book = new Book();

  public String doCreateBook() {
    FacesContext ctx = FacesContext.getCurrentInstance();

    if (book.getIsbn() == null || "".equals(book.getIsbn())) {
      ctx.addMessage("bookForm:isbn", ➥
      new FacesMessage(FacesMessage.SEVERITY_WARN, ➥
        "Неверный ISBN", "Вы должны ввести ISBN"));
    }
    if (book.getTitle() == null || "".equals(book.getTitle())) {
      ctx.addMessage("bookForm:title", ➥
      new FacesMessage(FacesMessage.SEVERITY_WARN, ➥
       "Неверное название", "Вы должны ввести название книги"));
    }

    if (ctx.getMessageList().size() != 0)
      return null;

    try {
      book = bookEJB.createBook(book);
      ctx.addMessage(null, new FacesMessage(FacesMessage.SEVERITY_INFO, ➥
        "Книга создана", "Книга" + book.getTitle() + " была создана ➥
```

```
            with id=" + book.getId()));

        } catch (Exception e) {
          ctx.addMessage(null, new FacesMessage(FacesMessage.SEVERITY_ERROR,
            "Книга не может быть создана", e.getMessage()));
        }
        return null;
      }
      // Конструкторы, методы работы со свойствами
    }
```

На этой странице доступны `FacesContext` и `FacesMessage`. Страница может затем отобразить глобальные сообщения (чье значение `clientId` равно `null`) с помощью одного тега `<h:messages>`. Когда мы хотим отобразить сообщение в определенном месте на странице для конкретного компонента (как это обычно бывает в случае проверки или ошибки преобразования), мы используем `<h:message>`. На рис. 11.4 показана страница с сообщениями, предназначенными непосредственно для полей `ISBN` и `title`.

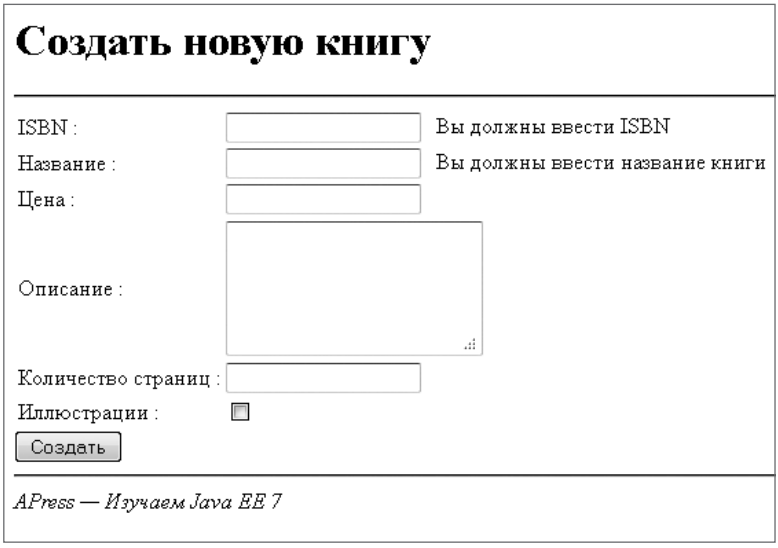


Рис. 11.4. Страница, отображающая сообщение для компонентов пользовательского интерфейса

Страница будет иметь текстовое поле ввода с идентификатором (`id="isbn"`), а тег `<h:message>` будет ссылаться на этот компонент (`for="ISBN"`). В результате конкретное сообщение будет отображаться только для этого компонента.

```
<h:messages infoStyle="color:blue" warnStyle="color:orange"
  errorStyle="color:red"/>
<h:form id="bookForm">
  <h:inputText id="isbn" value="#{bookController.book.isbn}"/>
  <h:message for="isbn"/>
</h:form>
```

Если поле ISBN не заполнено, рядом с текстовым полем ввода ISBN появится предупреждающее сообщение. JSF использует этот механизм обмена сообщениями для преобразователей, валидаторов и Bean Validation. Обратите внимание, что тег <h:messages> может иметь разные стили в зависимости от степени тяжести (здесь ошибки выделяются красным цветом, а предупреждения — оранжевым).

Объединение JSF и EJB

До этого момента все связывание и вызов бизнес-методов выполнялись с помощью компонентов-подложек. Как вы вскоре увидите, компоненты-подложки также отвечают и за навигацию. Так что, если вы хотите обработать какие-либо данные, а затем перенаправить пользователя на другую страницу, имеет смысл воспользоваться компонентом-подложкой. Но иногда нужно просто получить доступ к EJB, чтобы, например, увидеть список книг для заполнения таблицы в базе данных. Вы все еще можете воспользоваться компонентом-подложкой, но он будет в основном делегировать вызов EJB. Так почему бы не вызывать напрямую EJB со страницы?

Одна из целей CDI — сведение компонентов JSF и EJB. Что вам нужно делать? Не так много: просто добавьте аннотацию @Named к вашему EJB, и вы сможете вызвать его благодаря языку выражений. В листинге 11.9 показан EJB, не сохраняющий состояния, который использует JPA, чтобы получить список всех книг из базы данных. Он имеет аннотацию CDI @Named и, следовательно, может быть вызван на странице JSF.

Листинг 11.9. EJB, не сохраняющий состояния, с CDI-аннотацией @Named

```
@Named
@Stateless
public class BookEJB {
  @Inject
  private EntityManager em;
  public List<Book> findAllBooks() {
    return em.createNamedQuery("findAllBooks", Book.class).getResultList();
  }
}
```

Метод findAllBooks() в листинге 11.9 возвращает список книг. Чтобы заполнить таблицу базы данных этим списком, не нужно использовать компонент-подложку. Страницы могут непосредственно вызывать метод bookEJB.findAllBooks() в компоненте <h:dataTable> с помощью EL.

```
<h:dataTable value="#{bookEJB.findAllBooks()}" var="book">
  <h:column>
    <h:outputText value="#{book.title}"/>
  </h:column>
</h:dataTable>
```

Навигация

Веб-приложения состоят из нескольких страниц, по которым необходимо переходить. В зависимости от приложения вы можете иметь различные уровни навигации

потоков страниц, которые являются более или менее сложными. Вы можете думать о мастерах, где реально вернуться к предыдущей или начальной странице, бизнес-кейсах, где вы переходите на определенную страницу в зависимости от правила, и т. д. JSF имеет несколько способов навигации и позволяет контролировать поток, основываясь на одной странице, а также глобально для всего приложения.

Явная навигация

В JSF вы можете легко перейти на страницу, если знаете ее имя. Вам просто нужно явно задать имя страницы либо в методе компонента-подложки, либо на странице JSF. FacesServlet, выступая в качестве контроллера, сделает все остальное: перехватит вызов, получит страницу по ее названию, привяжет значения к компонентам и отрисует страницу для пользователя. При явной навигации вы можете выбрать между мгновенным переходом на страницу и выполнением некоторых действий перед переходом.

Когда вы просто хотите перейти со страницы на страницу, выбрав ссылку или нажав кнопку, не выполняя обработки, можно использовать компоненты пользовательского интерфейса `<h:button>`, `<h:link>` и `<h:outputLink>`:

```
<h:link value="Вернуться к созданию книги" outcome="newBook.xhtml"/>
```

Но иногда этого недостаточно, поскольку вы должны получить доступ к бизнес-уровню или базе данных для получения или обработки данных. В этом случае необходимо использовать `<h:commandButton>` и `<h:commandLink>` и их атрибут `action`, который позволяет выбирать методы компонента-подложки, в отличие от выбора страницы с помощью компонентов `<h:link>` или `<h:button>`:

```
<h:commandButton value="Создать книгу" action="#{bookController.doCreateBook}">
```

Представьте, что существует первая страница (`newBook.xhtml`), отображающая форму для создания книги. Как только вы нажмете кнопку Создать книгу, будет вызван метод компонента-подложки, создана книга и пользователь перейдет на страницу `listBooks.xhtml`, на которой перечислены все книги. Как только страница загрузится в браузер, ссылка Вернуться к созданию книги позволит вернуться непосредственно к предыдущей странице без какой-либо обработки (рис. 11.5).

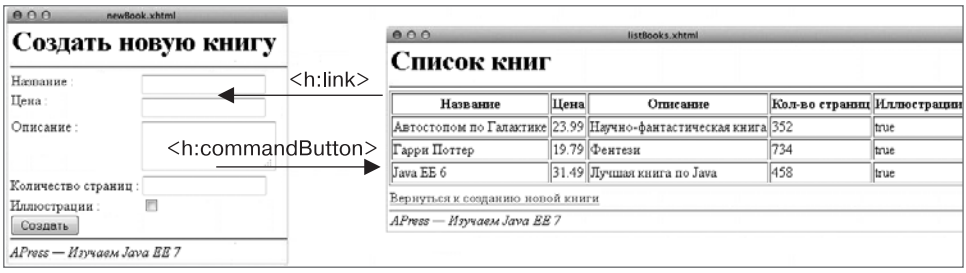


Рис. 11.5. Навигация между страницами newBook.xhtml и listBooks.xhtml

Переходить между страницами просто, но это действие по-прежнему нуждается в компоненте-подложке (`BookController`), чтобы выполнить некоторую бизнес-логику и навигацию. Обе страницы применяют компоненты (кнопки и ссылки)

для навигации и взаимодействия с компонентом-подложкой. Страница newBook.xhtml использует кнопку для вызова метода компонента-подложки doCreateBook() (<h:commandButton action="#{bookController.doCreateBook}"/>). Метод doCreateBook() затем вызовет EJB для сохранения сущности Book в базе данных и возвращения строки, содержащей имя следующей страницы. После этого страница listBooks.xhtml использует обычную ссылку для того, чтобы вернуться к странице newBook.xhtml без какой-либо обработки (<h:link outcome="newBook.xhtml"/>).

Компонент <h:commandButton> не вызывает непосредственно ту страницу, на которую нужно перейти. Он вызывает метод компонента-подложки, отвечающий за навигацию и решающий, какую страницу загрузить далее. Навигация работает согласно набору правил, которые определяют все возможные пути навигации в приложении. В коде компонента-подложки из листинга 11.10 используются простейшие правила навигации — возвращается та страница, на которую нужно перейти.

Листинг 11.10. Компонент-подложка, явно определяющий навигацию

```
@Named
@RequestScoped
public class BookController {
  @Inject
  private BookEJB bookEJB;
  private Book book = new Book();
  private List<Book> bookList = new ArrayList<>();

  public String doCreateBook() {
    book = bookEJB.createBook(book);
    bookList = bookEJB.findBooks();
    return "listBooks.xhtml";
  }
  //Конструкторы, методы работы со свойствами
}
```

Правила навигации

Когда компонент-подложка возвращает строку, эта строка может иметь несколько форм. В листинге 11.10 вы могли увидеть самый простой вариант — имя страницы. По умолчанию расширение файла этой страницы — .xhtml, но вы можете изменить его, если сконфигурируете FacesServlet с помощью файла web.xml, как было показано ранее.

С помощью JSF навигация по страницам может быть определена внешне в файле faces-config.xml. В нем навигация конфигурируется с помощью элемента <navigation-rule>, определяющего начальную страницу, условие и целевую страницу, на которую нужно перейти при срабатывании условия. Условие основывается на логическом имени, а не на имени страницы. Код компонента-подложки из листинга 11.10 может иметь логическое имя success, что показано в листинге 11.11.

Листинг 11.11. Фрагмент компонента-подложки, использующего логическое имя

```
@Named
@RequestScoped
public class BookController {
```

```
  // ...
  public String doCreateBook() {
    book = bookEJB.createBook(book);
    bookList = bookEJB.findBooks();
    return "success";
  }
  // Конструкторы, методы работы со свойствами
}
```

В листинге 11.12 показана структура файла `faces-config.xml`. Тег `<from-view-id>` определяет страницу, на которой происходит запрос действия. В этом случае вы начинаете со страницы `newBook.xhtml` перед тем, как сделать вызов компонента-подложки. Если было возвращено логическое имя `success` (`<from-outcome>`), `FacesServlet` перейдет к вызову страницы `listBooks.xhtml` (`<to-view-id>`).

Листинг 11.12. Файл faces-config.xml, определяющий правила навигации

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config xmlns="http://xmlns.jcp.org/xml/ns/javaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee ➥
              http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd"
              version="2.2">
  <navigation-rule>
    <from-view-id>newBook.xhtml</from-view-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>listBooks.xhtml</to-view-id>
    </navigation-case>
  </navigation-rule>
</faces-config>
```

Навигация может быть выполнена непосредственно в компонентах-подложках или с помощью файла `faces-congfig.xml`, но как понять, когда следует использовать тот или иной вариант? Первая причина возвращать имя страницы в компоненте-подложке — это простота. Код Java явный, и нет никаких внешних файлов XML, с которыми надо работать. Если же веб-приложению требуется активно сменять отображаемые страницы, лучше хранить правила навигации в одном месте, поскольку при внесении изменений придется поменять только один файл, а не несколько страниц. Можно также использовать комбинированный подход, храня одну часть правил навигации в компонентах-подложках, а другую — в файле `faces-config.xml`.

Есть один случай, когда использование XML-конфигурации очень полезно. Когда существуют глобальные ссылки на некоторые страницы (например, авторизация или выход могут быть выполнены во всем приложении), вам не захочется объявлять их на каждой странице. Глобальные правила навигации могут быть указаны в XML (эта же особенность невозможна внутри компонентов-подложек).

```
<navigation-rule>
  <from-view-id>*</from-view-id>
  <navigation-case>
    <from-outcome>logout</from-outcome>
```

```
      <to-view-id>logout.xhtml</to-view-id>
    </navigation-case>
  </navigation-rule>
```

Если у вас есть действие, которое выполняется на каждой странице приложения, вы можете использовать элемент navigation-rule без элемента <from-view-id> или же обратиться к шаблону поиска (*). В предыдущем коде указывается, что для любой страницы, на которой находится пользователь, в том случае, если метод компонента-подложки возвращает логическое имя logout, следует перенаправить пользователя на страницу logout.xhtml.

Предыдущие примеры показали простой вариант навигации, когда каждая страница имеет только одно правило навигации и только одну страницу, на которую следует перейти. Это не самый распространенный случай, и, в зависимости от определенных правил и исключений, пользователи могут быть перенаправлены на разные страницы. Этого можно достичь как с помощью компонентов-подложек, так и с помощью файла faces-config.xml. В следующем коде показан оператор switch, который перенаправляет пользователя на разные страницы. Обратите внимание, что если возвращено значение null, пользователь возвращается обратно на текущую страницу:

```
public String doCreateBook() {
  book = bookEJB.createBook(book);
  bookList = bookEJB.findBooks();
  switch (value) {
    case 1: return "page1.xhtml"; break;
    case 2: return "page2.xhtml"; break;
    default: return null; break;
  }
}
```

В версии JSF 2.0 внесено улучшение, касающееся условных переходов по страницам. Оно позволяет указывать предварительное условие, которое должно быть удовлетворено для перехода на другую страницу. Это предварительное условие определяется как выражение с новым конфигурационным элементом <fr>. В следующем примере, если пользователь имеет роль admin, процедура выхода из учетной записи отличается и переадресовывает его на страницу logout_admin.xhtml:

```
<navigation-rule>
  <from-view-id>*</from-view-id>
    <navigation-case>
      <from-outcome>logout</from-outcome>
      <to-view-id>logout_admin.xhtml</to-view-id>
      <if>#{userController.isAdmin}</if>
    </navigation-case>
</navigation-rule>
```

Добавление страниц в закладки

До выхода версии JSF 2.0 каждое взаимодействие клиента и сервера происходило на основе HTTP POST (<h:commandButton> и <h:commandLink>). Хотя это было удобно для большинства ситуаций, возникали проблемы при добавлении страницы

веб-приложения в закладки. В JSF 2.0 была представлена возможность добавлять страницу в закладки с использованием двух новых тегов — <h:button>, <h:link>, а также тега ViewParam. Это позволило осуществлять поддержку HTTP-запросов GET. Но модель все еще не была готова, поэтому приходилось использовать для этого фазу preRenderView. В JSF 2.2 был представлен ViewAction, позволяющий упростить процедуру добавления в закладки.

ViewParam и ViewAction предоставляют механизм обработки GET-запросов и связывания параметров, переданных с HTTP-запросом, со свойствами модели, используя EL. Страница с тегом <f:viewParam> может получать параметры из GET-запроса непосредственно в связанные свойства. Следующий код дает возможность получить доступ к странице viewBook.xhtml с помощью параметра id в запросе (например, viewBook.xhtml?id=123) и связать его с атрибутом bookController.book.id:

```
<f:metadata>
  < f:viewParam name="id" value="#{bookController.book.id}"/>
</f:metadata>
```

Как только извлекается параметр id из запроса HTTP и связывается со свойством, вам следует вызвать действие компонента-подложки. В JSF 2.2 определен новый тег <f:viewAction>, который определяет действие, характерное для приложения. Для того чтобы получить книгу из базы данных, основываясь на параметре id, вам следует вызвать метод doFindBookById таким образом:

```
<f:metadata>
  <f:viewParam name="id" value="#{bookController.book.id}"/>
  <f:viewAction action="#{bookController.doFindBookById}"/>
</f:metadata>
```

Этот прием позволяет вам непосредственно менять URL в браузере, чтобы указать на желаемую книгу (например, viewBook.xhtml?id=123 или viewBook.xhtml?id=456) и добавить страницу в закладки.

Преобразование и проверка

Вы уже видели, как обрабатывать сообщения для того, чтобы проинформировать конечного пользователя о действиях, которые необходимо предпринять. Одним из возможных действий может быть исправление неверно введенного значения (например, неправильного ISBN). JSF предоставляет стандартный механизм преобразования и проверки, который может обрабатывать данные, введенные пользователем для того, чтобы обеспечить их целостность. Благодаря ему вы можете быть уверены в корректности данных при вызове бизнес-методов для их обработки. Преобразование и проверка позволяют разработчику сфокусироваться на бизнес-логике, а не на проверке введенных данных на равенство значению null, вхождении в диапазон значений и т. д.

Преобразование происходит в тот момент, когда введенные конечным пользователем данные должны быть преобразованы из строки в объект и наоборот. Оно позволяет убедиться, что все данные имеют правильный тип — например, можно преобразовать строку в объект типа java.util.Date или Integer либо цену в долларах

в цену в евро. Что касается проверки, она гарантирует, что данные имеют необходимое содержимое (дата представлена в формате dd/MM/yyyy, число с плавающей точкой находится в диапазоне между 3,14 и 3,15 и т. д.).

Преобразование и проверка происходят во время различных фаз жизненного цикла страницы (которые вы видели в предыдущей главе), как показано на рис. 11.6.

Во время фазы «Применить значения запроса» на рис. 11.6 значение компонента пользовательского интерфейса преобразуется в целевой объект (например, из строки в дату) и затем проверяется на этапе «Проверка процесса». Логично предположить, что преобразование и проверка происходят до связывания данных компонента с компонентом-подложкой (которое выполняется во время фазы «Обновление значений модели»). Если обнаружена ошибка, это приведет к добавлениям сообщений об ошибке и сокращению жизненного цикла, поскольку он переходит сразу к фазе «Отрисовать ответ» (сообщения будут отображаться в пользовательском интерфейсе с помощью тега <h:messages/>). Во время этой фазы свойства компонента-подложки преобразуются обратно в строку, которая будет отображена.

JSF имеет набор стандартных преобразователей и валидаторов и позволяет разработчикам легко создавать свои собственные.

Преобразователи

Когда форма отображается в браузере, конечный пользователь заполняет поля ввода и нажимает кнопки, что приводит к транспортировке данных на сервер в запросе HTTP, который использует формат строки. Перед обновлением модели в компоненте-подложке эти данные должны быть преобразованы из строки в целевые объекты (Float, Integer, BigDecimal и т. д.). Обратное действие будет иметь место, когда данные должны будут отправиться обратно клиенту в ответе и отобразиться в браузере.

JSF поставляется с преобразователями для общих типов, таких как даты и числа. Если свойство компонента-подложки имеет примитивный тип (Integer, int, Float, float и т. д.), JSF автоматически преобразует значение компонента пользовательского интерфейса к правильному типу и обратно. Если это свойство имеет другой тип данных, вы должны предоставить свой преобразователь. В табл. 11.3 перечислены все стандартные преобразователи, которые находятся в пакете javax.faces.convert.

Таблица 11.3. Стандартные преобразователи

Преобразователь	Описание
BigDecimalConverter	Преобразует строку к типу java.math.BigDecimal и наоборот
BigIntegerConverter	Преобразует строку к типу java.math.BigInteger и наоборот
BooleanConverter	Преобразует строку к типу Boolean (и примитиву boolean) и наоборот
ByteConverter	Преобразует строку к типу Byte (и примитиву byte) и наоборот
CharacterConverter	Преобразует строку к типу Character (и примитиву char) и наоборот
DateTimeConverter	Преобразует строку к типу java.util.Date и наоборот

Продолжение ⇩

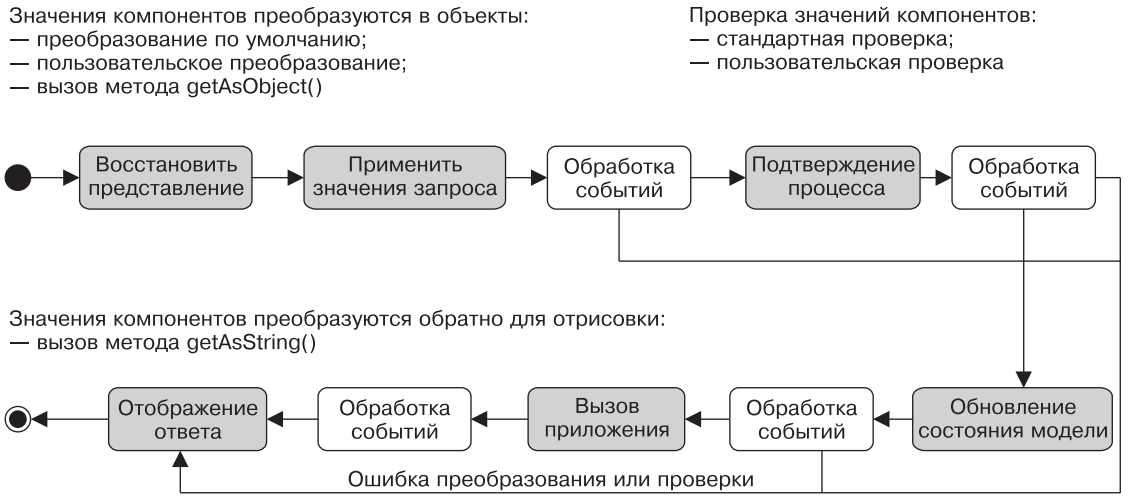


Рис. 11.6. Преобразование и проверка во время жизненного цикла страницы

Таблица 11.3 *(продолжение)*

Преобразователь	Описание
DoubleConverter	Преобразует строку к типу Double (и примитиву double) и наоборот
EnumConverter	Преобразует строку к типу Enum (и примитиву enum) и наоборот
FloatConverter	Преобразует строку к типу Float (и примитиву float) и наоборот
IntegerConverter	Преобразует строку к типу Integer (и примитиву int) и наоборот
LongConverter	Преобразует строку к типу Long (и примитиву long) и наоборот
NumberConverter	Преобразует строку к абстрактному классу java.lang.Number и наоборот
ShortConverter	Преобразует строку к типу Short (и примитиву short) и наоборот

JSF будет автоматически преобразовывать вводимые значения к числам, когда свойство компонента-подложки будет иметь примитивный числовой тип, и к дате или времени, когда свойство будет иметь тип даты. Если автоматическое преобразование вам не подходит, вы можете явно управлять им с помощью стандартных тегов `convertNumber` и `convertDateTime`. Чтобы использовать эти теги, необходимо вложить преобразователь внутрь любого из тегов ввода или вывода. Преобразователь будет вызываться JSF в течение жизненного цикла.

Тег `convertNumber` имеет атрибуты, которые позволяют преобразовывать входное значение к числу (по умолчанию), валюте или процентам. Вы можете задать символ валюты или количество десятичных разрядов, а также шаблон форматирования, определяя, как число должно быть отформатировано и разобрано:

```
<h:inputText value="#{bookController.book.price}">
  <f:convertNumber currencySymbol="$" type="currency"/>
</h:inputText>
```

Тег `convertDateTime` может конвертировать даты в различных форматах (дата, время или и то и другое). Он имеет несколько атрибутов, которые управляют преобразованием даты и часового пояса. Атрибут `pattern` позволяет идентифицировать шаблон строки даты, которая будет преобразована:

```
<h:inputText value="#{bookController.book.publishedDate}">
  <f:convertDateTime pattern="MM/dd/yy"/>
</h:inputText>
```

Пользовательские преобразователи

Иногда преобразования чисел, дат, перечислений и т. д. недостаточно и вам может потребоваться собственное преобразование. Свой преобразователь легко разработать и использовать его на странице с помощью JSF. Вы просто должны написать класс, реализующий интерфейс `javax.faces.convert.Converter`, и зарегистрировать его с помощью метаданных. Этот интерфейс имеет два метода:

```
Object getAsObject(FacesContext ctx, UIComponent component, String value)
String getAsString(FacesContext ctx, UIComponent component, Object value)
```

Метод getAsObject() преобразует строковое значение компонента пользовательского интерфейса в соответствующий поддерживаемый тип и возвращает новый экземпляр. Этот метод генерирует исключение ConverterException, если преобразование не удалось. И наоборот, метод getAsString() преобразует объект предоставленного типа в строку, которая будет отрисована с помощью языка разметки (например, XHTML).

Как только вы разработаете собственный преобразователь, он должен быть зарегистрирован, чтобы можно было использовать его в веб-приложении. Один из методов регистрации — объявление преобразователя в файле faces-config.xml, а другой — использование аннотации @FacesConverter.

В листинге 11.13 показано, как создать собственный преобразователь, который преобразует цены из долларов в евро. Он начинается со связывания этого преобразователя с именем euroConverter с помощью аннотации @FacesConverter("euroConverter") и реализации интерфейса Converter. В этом примере только переопределяется метод getAsString(), который возвращает строковое представление данной цены в евро.

Листинг 11.13. Преобразователь евро

```
@FacesConverter("euroConverter")
public class EuroConverter implements Converter {
  @Override
  public Object getAsObject(FacesContext context, UIComponent component, String
value) {
    return value;
  }
  @Override
  public String getAsString(FacesContext ctx, UIComponent component, Object
value) {
    float amountInDollars = Float.parseFloat(value.toString());
    double ammountInEuros = amountInDollars * 0.8;
    DecimalFormat df = new DecimalFormat("###,##0.##");
    return df.format(ammountInEuros)
  }
}
```

Для того чтобы использовать этот преобразователь, укажите либо атрибут converter тега <h:outputText>, либо тег <f:converter>. В обоих случаях вам следует передать имя своего преобразователя, определенное в аннотации @FacesConverter (euroConverter). Следующий код отображает два выходных текста, один из которых представляет цену в долларах, а другой — преобразование этой цены в евро:

```
<h:outputText value="#{book.price}"/> // в долларах
<h:outputText value="#{book.price}">  // конвертируем в евро
  <f:converter converterId="euroConverter"/>
</h:outputText>
```

Или же вы можете использовать атрибут converter тега outputText:

```
<h:outputText value="#{book.price}" converter="euroConverter"/>
```

Валидаторы

При работе с веб-приложениями должна быть обеспечена точность вводимых пользователем данных. Точный ввод данных может быть навязан на стороне клиента с применением JavaScript или на стороне сервера с помощью валидаторов и Bean Validation. JSF упрощает проверку данных благодаря стандартным и пользовательским серверным валидаторам. Валидаторы действуют как первый уровень контроля, проверяя значения компонентов пользовательского интерфейса перед тем, как они будут обработаны компонентом-подложкой.

Компоненты пользовательского интерфейса, как правило, выполняют простую проверку, например проверяют, является ли значение обязательным. Так, например, следующий тег требует, чтобы было введено значение в текстовом поле ввода:

```
<h:inputText value="#{bookController.book.title}" required="true"/>
```

Если вы не введете значение, то JSF вернет страницу с сообщением о том, что должно быть введено значение (страница должна иметь тег `<h:messages>`). При этом используется тот же механизм сообщений, который я описал ранее. Но JSF поставляется с набором валидаторов (перечисленных в табл. 11.4). Они определены в пакете `javax.faces.validator`.

Таблица 11.4. Стандартные валидаторы

Валидатор	Описание
DoubleRangeValidator	Проверяет значение соответствующего компонента на вхождение в заданный интервал, границы которого имеют тип double
LengthValidator	Проверяет количество символов в строке, связанной с компонентом
LongRangeValidator	Проверяет значение соответствующего компонента на вхождение в заданный интервал, границы которого имеют тип long
MethodExpression-Validator	Выполняет проверку, выполнив метод объекта
RequiredValidator	Эквивалентен присвоению значения true атрибуту required входного компонента
RegexValidator	Проверяет значение определенного компонента на соответствие регулярному выражению

Эти валидаторы полезны для общих задач, например таких, как определение длины поля или диапазона номеров. Они могут быть легко связаны с компонентом так же, как преобразователи (они оба могут быть использованы на одном компоненте). Следующий код обеспечивает, что название книги составляет от 2 до 20 символов в длину, а ее цена — от $1 до 500:

```
<h:inputText value="#{bookController.book.title}" required="true">
  <f:validateLength minimum="2" maximum="20"/>
</h:inputText>
<h:inputText value="#{bookController.book.price}">
  <f:validateLongRange minimum="1" maximum="500"/>
</h:inputText>
```

Пользовательские валидаторы

Возможно, стандартные валидаторы JSF не могут удовлетворить ваши потребности — вам, вероятно, потребуется иметь данные, которые должны следовать определенному формату, такому как почтовый индекс, область или адрес электронной почты. Вы должны создать собственный валидатор для обработки этих случаев. Как и преобразователи, валидатор — это класс, который должен реализовывать интерфейс и переопределить отдельные методы. В случае валидаторов таким интерфейсом является `javax.faces.validator.Validator`, который имеет один метод `validate()`:

```
void validate(FacesContext context, UIComponent component, Object value)
```

В этом методе аргумент `value` содержит проверяемое на основе какой-либо бизнес-логики значение. Если процесс проверки проходит нормально, вы можете просто вернуться из метода и жизненный цикл страницы будет продолжаться. Если нет, то может быть сгенерировано исключение `ValidatorException` и включено сообщение `FacesMessage`, содержащее краткое и полное описание возникшей ошибки. Вы должны зарегистрировать валидатор либо в файле `faces-config.xml`, либо с помощью аннотации `@FacesValidator`.

В качестве примера создадим проверку ISBN, которая обеспечивает, что введенный пользователем ISBN для книги следует определенному формату (с помощью регулярных выражений). В листинге 11.14 показан код валидатора `IsbnValidator`.

Листинг 11.14. Валидатор ISBN

```
@FacesValidator("isbnValidator")
public class IsbnValidator implements Validator {
  private Pattern pattern;
  private Matcher matcher;

  @Override
  public void validate(FacesContext context, UIComponent component, ➥
                       Object value) throws ValidatorException {
    String componentValue = value.toString();

    pattern = Pattern.compile("(?=[-0-9xX]{13}$)");
    matcher = pattern.matcher(componentValue);

    if (!matcher.find()) {
      String message = MessageFormat.format("{0} неверный формат isbn", ➥
        componentValue);
      FacesMessage facesMessage = ➥
        new FacesMessage(FacesMessage.SEVERITY_ERROR, msg, msg);
      throw new ValidatorException(facesMessage);
    }
  }
}
```

Код в листинге 11.14 начинается со связывания валидатора с именем `isbnValidator`, чтобы он мог быть использован на странице. Он реализует интерфейс `Validator` и добавляет логику проверки в метод `validate()`. С помощью регулярных выраже-

ний валидатор проверяет, что ISBN имеет правильный формат. Если нет — сообщение добавляется в контекст и генерируется исключение. JSF автоматически возобновит жизненный цикл страницы, повторно ее вызовет, а также отобразит сообщение об ошибке. Вы можете использовать этот валидатор на ваших страницах с помощью атрибута `validator` или тега `<f:validator>`.

```
<h:inputText value="#{bookController.book.isbn}" validator="isbnValidator"/>
// или
<h:inputText value="#{bookController.book.isbn}">
  <f:validator validatorId="isbnValidator" />
</h:inputText>
```

Интеграция с Bean Validation

Начиная с версии JSF 2.0, поддержка Bean Validation стала обязательной (то есть реализация JSF должна поддерживать данный компонент). Это работает просто. Если страница JSF имеет несколько текстовых полей и каждое текстовое поле связано со свойством компонента-подложки, имеющим хотя бы одну ограничивающую аннотацию Bean Validation, то во время фазы проверки вызывается метод `javax.validation.Validator.validate()`. Поэтому Bean Validation вызывается автоматически, разработчик ничего не должен делать на странице JSF или вызывать любой метод Bean Validation в компоненте-подложке. Среда выполнения JSF также гарантирует, что каждый `ConstraintViolation`, который привел к попытке проверки данных модели, будет обернут в `FacesMessage` и добавлен к `FacesContext`, как это обычно происходит для любого другого вида валидатора. Так, простой тег `<h:messages>` на странице отобразит список сообщений, вызванных `ConstraintViolation`.

В главе 3 я представил проверочные группы. Если вам необходимо проверить на вашей странице некоторые компоненты из определенной группы, можете использовать тег `<f:validateBean>`. Приведенный ниже код проверяет название и цену книги только для группы `Delivery`:

```
<f:validateBean validationGroups="org.agoncal.book.javaee7.Delivery">
  <h:inputText value="#{bookController.book.title}"/>
  <h:inputText value="#{bookController.book.price}"/>
</f:validateBean>
```

AJAX

Протокол HTTP основывается на механизме «запрос/ответ»: клиент нуждается в чем-то, посылает запрос и получает ответ от сервера, как правило, целую веб-страницу. Коммуникация следует этому направлению: клиент запрашивает что-то с сервера, а не наоборот. Однако веб-приложения должны предоставлять богатый и отзывчивый интерфейс, а также реагировать на события сервера, обновлять часть страницы, хранить виджеты и т. д. В нормальной ситуации «запроса/ответа» сервер должен послать обратно целую веб-страницу, даже если должна измениться лишь небольшая ее часть. Если страница имеет значительный размер, вы будете перегружать сеть, поскольку браузеру необходимо будет загрузить всю страницу. Если вы хотите увеличить отзывчивость браузера и повысить удобство работы

со страницей, вам нужно обновлять лишь небольшие ее участки. Это можно сделать с помощью AJAX.

AJAX (Asynchronous JavaScript and XML — асинхронный JavaScript и XML) представляет собой набор приемов веб-разработки, используемых для создания интерактивных веб-приложений. AJAX позволяет веб-приложению получать порции данных с сервера асинхронно, не пересекаясь с отображением и поведением существующей страницы. Как только браузер получает данные, ему нужны лишь фрагменты, требующие обновления с использованием DOM и JavaScript.

Общие понятия

Термин AJAX был придуман в 2005 году. Еще в 1999 году компания Microsoft создала объект `XMLHttpRequest` как элемент управления ActiveX в Internet Explorer 5. В 2006 году W3C выпустил первый проект спецификации для объекта `XMLHttpRequest`, который сейчас поддерживается большинством браузеров. В то же время несколько компаний придумали способ сделать AJAX промышленным стандартом для насыщенной платформы приложений на основе открытых технологий. Результатом этой работы стало создание OpenAjax Alliance, который состоит из продавцов, проектов с открытым исходным кодом и компаний, использующих технологии на основе AJAX.

Как показано на рис. 11.7, в традиционных веб-приложенях браузер должен запросить полные HTML-документы с сервера. Пользователь нажимает кнопку, чтобы отправить или получить информацию, ждет ответа сервера, а затем получает всю страницу, которая потом загружается в браузер. AJAX, с другой стороны, использует асинхронную передачу данных (HTTP-запросы) между браузером и сервером, что позволяет веб-страницам запрашивать с сервера небольшие фрагменты информации (JSON- или XML-данные), а не целые страницы. Пользователь остается на той же странице, в то время как часть JavaScript запрашивает или отправляет данные на сервер асинхронно и обновляются только части страницы, что делает веб-приложения быстрее и удобнее для пользователя.

В принципе, AJAX основан на следующих концепциях:

- XHTML и CSS для представления данных;
- DOM для динамического отображения и взаимодействия с данными;
- XML и XSLT для обмена, обработки и отображения XML-данных;
- объект `XMLHttpRequest` для асинхронной связи;
- JavaScript, чтобы свести эти технологии вместе в браузере.

`XMLHttpRequest` — важная часть AJAX, так как он является DOM API, используемым JavaScript для передачи XML от браузера на сервер. Возвращаемые данные должны быть получены на стороне клиента для обновления части страницы динамически с помощью JavaScript. Данные могут иметь несколько форматов, такие как XHTML, JSON или даже простой текст.

AJAX поддерживается с версии JSF 2.0, так что вам не придется разрабатывать JavaScript для обработки `XMLHttpRequest`. Вы должны использовать библиотеку JavaScript, которая указана и поставляется с реализацией JSF.

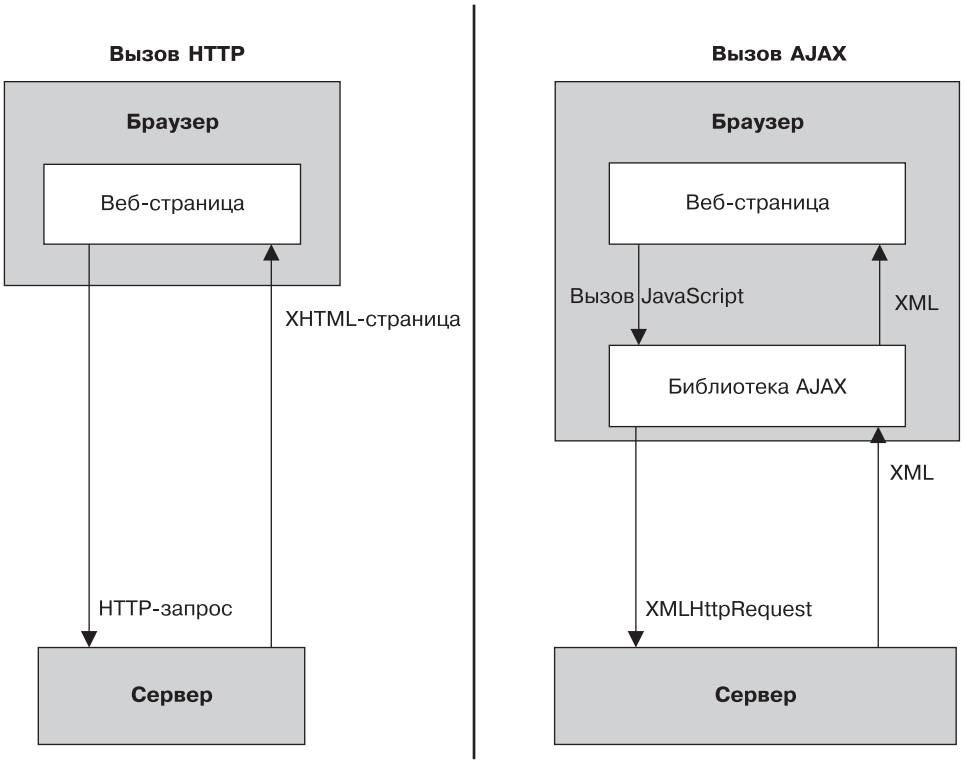


Рис. 11.7. Простые вызовы HTTP против HTTP-вызовов AJAX

Поддержка в JSF

Предыдущие версии JSF не предлагали встроенного решения AJAX, так что, чтобы заполнить этот пробел, должны были использоваться сторонние библиотеки. Иногда это усложняло код за счет производительности. Начиная с версии JSF 2.0, все стало гораздо проще, так как поддержка AJAX встроена в любую реализацию JSF.

Прежде всего, существует указанная библиотека JavaScript (`jsf.js`) для выполнения взаимодействия с AJAX, которое означает, что вы не должны разрабатывать собственные сценарии или манипулировать объектами `XMLHttpRequest` напрямую. Вместо разработки кода на JavaScript вы можете использовать набор стандартных функций для отправки асинхронных запросов и приема данных. Для того чтобы применять эту библиотеку на вашей странице, вам нужно добавить ресурс `jsf.js` следующей строкой кода:

```
<h:outputScript name="jsf.js" library="javax.faces" target="head"/>
```

Тег `<h:outputScript>` отрисовывает элемент разметки `<script>`, ссылаясь на JavaScript-файл `jsf.js` из библиотеки `javax.faces` (что соответствует новому способу управления ресурсами JSF, а это означает, что файл `jsf.js` находится в каталоге `META-INF/resources/javax/faces`). Обратите внимание, что имя пространства

имен высокого уровня javax регистрируется внутри OpenAjax Alliance. Этот JavaScript API используется для инициирования взаимодействий с JSF на стороне клиента, включая частичный обход дерева и частичное обновление страницы. Функция, которую мы будем напрямую применять на наших страницах, называется request, так как она отвечает за передачу запроса AJAX на сервер. Ее сигнатура выглядит следующим образом:

```
jsf.ajax.request(ELEMENT, |event|, { |OPTIONS| });
```

ELEMENT – это любой компонент JSF или элемент XHTML, из которого вы вызываете событие. Как правило, для отправки формы этот элемент будет кнопкой. EVENT — это любое событие JavaScript, поддерживаемое этим элементом, например onmousedown, onclick, onblur и т. д. Аргумент OPTIONS — это массив, который может содержать следующие пары «имя/значение»:

- execute: '<разделенный пробелами список компонентов пользовательского интерфейса>' — отправляет список ID компонентов на сервер для того, чтобы они были обработаны в течение фазы выполнения запроса;
- render: '<разделенный пробелами список компонентов пользовательского интерфейса>' — отрисовывает список ID компонентов, которые были обработаны на этапе отрисовки запроса.

В качестве примера рассмотрим следующий код, который показывает кнопку, вызывающую асинхронный метод doCreateBook, передавая все параметры с формы книги. Кнопка вызывает функцию jsf.ajax.request, когда пользователь нажимает кнопку (событие onclick). Аргумент this относится к самому элементу (кнопка), а параметры — к ID компонента (isbn, title, price...):

```
<h:commandButton value="Создать книгу" onclick="jsf.ajax.request(this, event,
    {execute:'isbn title price description nbOfPage illustrations',
     render:'booklist'}); return false;"
    actionListener="#{bookController.doCreateBook}" />
```

Когда клиент делает AJAX запрос, жизненный цикл страницы на стороне сервера остается неизменным (он проходит через те же шесть фаз). Основное преимущество состоит в том, что ответ представляет собой небольшой фрагмент данных, а не большую HTML-страницу в браузере. Фаза «Применить запрос» определяет, является ли текущий запрос «частичным запросом», и объект PartialViewContext используется на протяжении жизненного цикла страницы. Он содержит методы и свойства, которые относятся к частичной обработке запроса и частичной отрисовке ответа. В конце жизненного цикла ответ AJAX (или, строго говоря, частичный ответ) отправляется клиенту на этапе «Отрисовать ответ». Он обычно состоит из XHTML-, XML- или JSON-кода, который будет преобразован с помощью JavaScript на клиентской стороне.

JSF 2 также включает в себя декларативный подход, который удобнее и проще в использовании. В этом подходе задействуется новый тег <f:ajax>. Вместо самостоятельного кодирования JavaScript для вызова запроса AJAX вы можете декларативно указать такое же поведение, не используя никакого кода JavaScript.

```
<h:commandButton value="Создать книгу" action="#{bookController.
doCreateBook}">
    <f:ajax execute="@form" render=":booklist"/>
</h:commandButton>
```

В этом примере `render` указывает на идентификатор компонента, который мы хотим отрисовать (компонент `Booklist`). Атрибут `execute` относится ко всем компонентам, которые принадлежат форме. В табл. 11.5 показаны все возможные значения, которые могут принимать атрибуты `render` и `execute`.

Таблица 11.5. Возможные значения атрибутов render и execute

Значение	Описание
@all	Отрисовывает или выполняет все компоненты представления
@none	Не отрисовывает и не выполняет ни одного компонента представления (применяется по умолчанию, если не задано другое значение)
@this	Отрисовывает или выполняет только этот компонент (компонент, сделавший запрос AJAX)
@form	Отрисовывает или выполняет все компоненты внутри этой формы (из которой был сделан запрос AJAX)
Разделенный пробелами список идентификаторов	Один или несколько идентификаторов компонентов, которые должны быть отрисованы или выполнены
Язык выражений	Выражение, которое преобразуется в коллекцию строк

Все вместе

Теперь объединим все эти понятия и напишем веб-приложение, позволяющее пользователю создавать новую книгу, выводить все книги из базы данных и наглядно представлять характеристики книг. Это веб-приложение состоит из двух веб-страниц:

- `newBook.xhtml` отображает в верхней части форму, позволяя вам создавать книги, а также выводит список всех книг в нижней части. Когда вы создаете книгу и нажимаете кнопку Создать книгу, внизу обновится только список книг с помощью AJAX. При щелчке на ссылке-названии книги пользователь переходит на другую страницу и отображаются характеристики книги;
- `viewBook.xhtml` показывает детали этой книги (название, цену, описание и т. д.). Можно добавлять страницы в закладки, а также отображать содержание любой книги, просто изменив URL (например, `viewBook.xhtml?id=123` или `viewBook.xhtml?id=456`).

Эти две страницы имеют один и тот же шаблон, показывающий ссылку в верхней части (чтобы вернуться к странице `newBook.xhtml`) и метку в нижней части. На рис. 11.8 приведены эти две страницы и различные пути навигации (либо с помощью `<h:link>`, либо посредством `<h:commandButton>`).

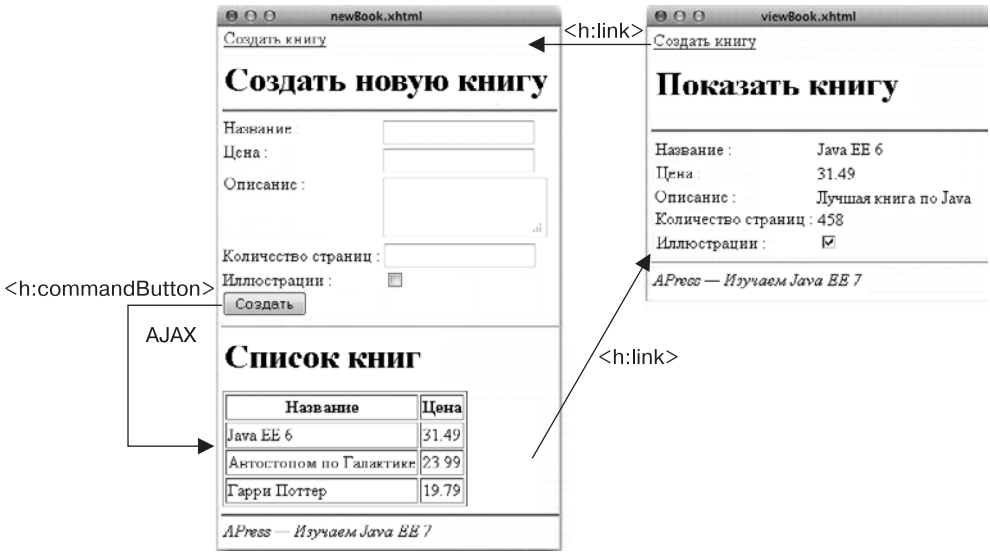


Рис. 11.8. Пути навигации между страницами newBook.xhtml и viewBook.xhtml

Страницы используют компоненты-подложки `BookController` для хранения необходимых свойств и для навигации. Воспользовавшись возможностью сохранения с помощью JPA и бизнес-логикой посредством EJB, можно подключить вместе все части (JSF, EJB, JPA, Bean Validation и CDI). Компонент-подложка делегирует всю бизнес-логику `BookEJB`, он содержит три метода: один сохраняет книгу в базу данных (`createBook()`), второй позволяет получить все книги (`findAllBooks()`), а третий — получить книгу по ее идентификатору (`findBookById()`). Этот сессионный компонент, не сохраняющий состояние, использует API менеджера сущностей для манипулирования сущностью `Book` (с помощью аннотаций Bean Validation).

На рис. 11.9 показаны взаимодействующие компоненты этого веб-приложения. Они упакованы в файл с расширением `.war` и развернуты в запущенном экземпляре GlassFish и в памяти базы данных Derby.

Это веб-приложение соответствует структуре каталогов Maven, поэтому классы, файлы и веб-страницы должны быть помещены в следующие каталоги:

- `src/main/java` — содержит сущность `Book`, `BookEJB` и компонент-подложку `BookController`;
- `src/main/resources` — включает файл `META-INF/persistence.xml`, используемый для отображения сущности в базе данных;
- `src/main/webapp` — содержит две веб-страницы — `newBook.xhtml` и `viewBook.xhtml`;
- `src/main/webapp/WEB-INF` — содержит файлы `faces-config.xml` и `beans.xml`, используемые для вызова JSF и CDI;
- `pom.xml` — представляет модель объектов проекта Maven (POM) с описанием проекта, его зависимостей и надстроек.

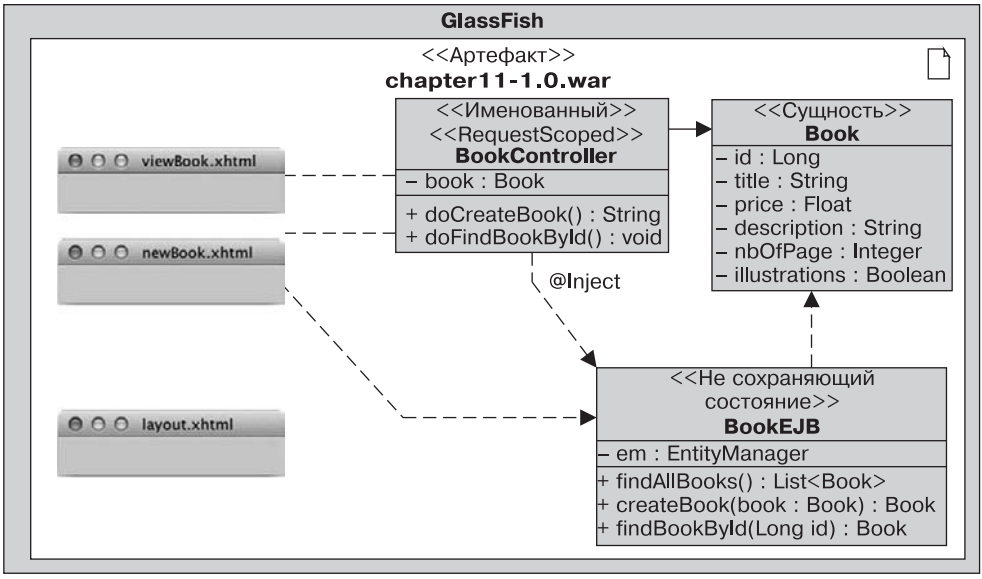


Рис. 11.9. Страницы и классы веб-приложения

Написание сущности Book

Я не буду вдаваться в излишние подробности, говоря о листинге 11.15, поскольку вы сразу поймете код сущности `Book`. Помимо преобразования JPA и аннотаций Bean Validation, обратите внимание на именованный запрос `findAllBooks`, который извлекает из базы данные, отсортированные по названию.

Листинг 11.15. Сущность Book, содержащая именованный запрос и аннотацию Bean Validation

```
@Entity
@NamedQuery(name = "findAllBooks", query = "SELECT b FROM Book b
            ORDER BY b.title DESC")
public class Book {
  @Id @GeneratedValue
  private Long id;
  @NotNull @Size(min = 4, max = 50)
  @Column(nullable = false)
  private String title;
  private Float price;
  @Column(length = 2000)
  private String description;
  private Integer nbOfPage;
  private Boolean illustrations;
  // Конструкторы, методы работы со свойствами
}
```

Как вы знаете, эта сущность также должна содержать файл `persistence.xml`, но ради простоты я не буду приводить его здесь (см. главу 4).

Написание BookEJB

В листинге 11.16 содержится сессионный компонент, который создает представление без интерфейса. Это означает, что клиент (то есть компонент-подложка) не нуждается в интерфейсе (локальном или удаленном) и может непосредственно ссылаться на EJB. В этот EJB будет внедрена ссылка на менеджер сущностей с помощью аннотации `@Inject`. Для этого ей нужен CDI `DatabaseProducer`, похожий на тот, что был показан в листинге 8.13 (не приводится здесь, см. главу 8). С помощью этого менеджера сущностей сохраняется сущность `Book` (метод `createBook()`), запрашиваются все книги из базы данных (по именованному запросу `findAllBooks`) и выполняется поиск книги по ее идентификатору. Обратите внимание, что EJB имеет аннотацию `@Named`, поэтому он может быть вызван со страницы JSF (подробнее об этом — далее) и определяет источник данных с помощью аннотации `@DataSourceDefinition`. Этот EJB не нуждается в дескрипторе развертывания.

Листинг 11.16. EJB, не сохраняющий свое состояние, который создает и получает книги

```
@Named
@Stateless
@DataSourceDefinition(name = "java:global/jdbc/lab11DS",
        className = "org.apache.derby.jdbc.EmbeddedDriver",
        url = "jdbc:derby:memory:lab11DB;create=true;user=app;password=app"
)
public class BookEJB {
  @Inject
  private EntityManager em;

  public Book createBook(Book book) {
    em.persist(book);
    return book;
  }

  public List<Book> findAllBooks() {
    return em.createNamedQuery("findAllBooks", Book.class).getResultList();
  }

  public Book findBookById(Long id) {
    return em.find(Book.class, id);
  }
}
```

Написание компонента-подложки BookController

Одна из обязанностей компонента-подложки — взаимодействие с другими уровнями приложения (например, слоем EJB) или выполнение проверки. `BookController` в листинге 11.17 содержит аннотацию `@Named`, поэтому может быть использован на страницах JSF. Вторая аннотация — `@RequestScoped` — определяет продолжительность жизни компонента. Он существует в течение всего срока существования запроса.

Компонент-подложка содержит один атрибут book, который станет использоваться на страницах. Это объект, который будет преобразован на представлении (странице newBook.xhtml), отображен в viewBook.xhtml и сохранен в базе данных.

Листинг 11.17. Компонент-подложка BookController, вызывающий EJB

```
@Named
@RequestScoped
public class BookController {
  @Inject
  private BookEJB bookEJB;
  private Book book = new Book();

  public String doCreateBook() {
    bookEJB.createBook(book);
    FacesContext.getCurrentInstance().addMessage(null,
      new FacesMessage(FacesMessage.SEVERITY_INFO, "Book created",
      "Книга" + book.getTitle() + " была создана с id=" + book.getId()));
    return "newBook.xhtml";
  }

  public void doFindBookById() {
    book = bookEJB.findBookById(book.getId());
  }
  // Методы работы со свойствами
}
```

Вся бизнес-логика (создание и получение книг) осуществляется через компонент-подложку BookEJB. В него внедряется ссылка на EJB с помощью аннотации @Inject. Он имеет два метода, которые будут вызываться страницами.

- doCreateBook() — позволяет создавать книги благодаря вызову EJB, не сохраняющего состояние, и передаче атрибута Book. Если сохранение прошло успешно, то на странице будут отображены информационные сообщения. Далее метод возвращает имя страницы, на которую нужно перейти.
- doFindBookById() — используется страницей viewBook.xhtml для извлечения книги по ее идентификатору с применением EJB, не сохраняющего состояние.

В листинге 11.17 показан компонент-подложка BookController. Все методы работы со свойствами в этом фрагменте кода опущены для удобства чтения, но необходимы для атрибута book.

Написание шаблона layout.xhtml

Как newBook.xhtml, так и viewBook.xhtml используют один и тот же шаблон (layout.xhtml, определенный в листинге 11.18). Как вы можете увидеть на рис. 11.8, шаблон имеет заголовок, верхний колонтитул с тегом <h:link>, отображающим ссылку Создать книгу, которая позволяет перейти к странице newBook.xhtml, и тегом <h:messages>, предназначенным для вывода ошибок и информационных сообщений, а также нижний колонтитул.

Листинг 11.18. Шаблон Layout.xhtml, используемый обеими страницами

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:head>
  <title><ui:insert name="title">Название по умолчанию</ui:insert></title>
</h:head>
<h:body>
  <h:link value="Create a book" outcome="newBook.xhtml"/>

  <h1><ui:insert name="title">Заголовок по умолчанию</ui:insert></h1>
  <hr/>

  <h:messages id="errors" infoStyle="color:blue" warnStyle="color:orange" ➥
                          errorStyle="color:red"/>

  <ui:insert name="content">Содержимое по умолчанию</ui:insert>

  <hr/>
  <h:outputText value="APress — Изучаем Java EE 7" style="font-style:
italic"/>
</h:body>
</html>
```

Написание страницы newBook.xhtml

newBook.xhtml — отдельная страница, которая имеет форму в верхней части, предназначенную для ввода данных о книге (названия, цены, описания, количества страниц и иллюстраций), а также списка книг в нижней части (см. рис. 11.8). Каждый раз, когда нажатием кнопки создается новая книга, список обновляется, показывая созданную книгу. После отправки формы в обновлении нуждается только часть страницы, содержащая список. Для этого используется AJAX.

Код в листинге 11.19 показывает верхнюю часть страницы, которая представляет собой форму. Переменная BookController относится к компоненту-подложке BookController, который ответственнен за всю бизнес-логику (см. листинг 11.17). Book — это сущность, и ее доступ к ее атрибутам выполняется с помощью языка выражений (#{bookController.book.title} связан с названием).

Листинг 11.19. Верхняя часть страницы newBook.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
<ui:composition template="layout.xhtml">
  <ui:define name="title">Создать новую книгу</ui:define>
  <ui:define name="content">
```

```
    <h:form id="bookForm">
      <h:panelGrid columns="2">
        <h:outputLabel value="Название : "/>
        <h:inputText value="#{bookController.book.title}"/>

        <h:outputLabel value="Цена : "/>
        <h:inputText value="#{bookController.book.price}"/>

        <h:outputLabel value="Описание : "/>
        <h:inputTextarea value="#{bookController.book.description}" cols="16"
rows="3"/>

        <h:outputLabel value="Количество страниц : "/>
        <h:inputText value="#{bookController.book.nbOfPage}"/>

        <h:outputLabel value="Иллюстрации : "/>
        <h:selectBooleanCheckbox value="#{bookController.book.
illustrations}"/>
      </h:panelGrid>

      <h:commandButton value="Создать книгу" action="#{bookController.
doCreateBook}">
        <f:ajax execute="@form" render=":booklist :errors"/>
      </h:commandButton>
    </h:form>
    ...
```

Тег `<h:commandButton>` описывает кнопку, из которой делается вызов AJAX. Когда пользователь нажимает ее, тег `<f:ajax>` передает все параметры формы с помощью `@form` (вместо этого он мог бы передать список всех идентификаторов компонентов). Значения всех компонентов пользовательского интерфейса, которые находятся в форме, затем передаются на сервер.

Метод `doCreateBook()` компонента-подложки вызывается в тот момент, когда сохраняется новая книга, а также когда возвращается список книг. Если исключение не сгенерировалось, будет возвращено название страницы, к которой нужно перейти: `newBook.xhtml`. В этом случае пользователь остается на той же странице. Отрисовка этого списка на стороне клиента производится асинхронно благодаря AJAX. Элемент `render` ссылается на ID `booklist` как на идентификатор таблицы данных, отображающей все книги (листинг 11.20), а также на ID `errors`, который определен на странице `layout.xhtml` (см. листинг 11.18), если сообщение должно быть отображено.

Листинг 11.20. Нижняя часть страницы newBook.xhtml

```
    ...
    <hr/>
    <h1>Список книг</h1>

    <h:dataTable id="booklist" value="#{bookEJB.findAllBooks()}" var="bk"
border="1">
        <h:column>
      <f:facet name="header">
```

```
            <h:outputText value="Название"/>
          </f:facet>
          <h:link outcome="viewBook.xhtml?id=#{bk.id}" value="#{bk.title}"/>
        </h:column>

        <h:column>
          <f:facet name="header">
            <h:outputText value="Цена"/>
          </f:facet>
          <h:outputText value="#{bk.price}"/>
        </h:column>
      </h:dataTable>
    </ui:define>
  </ui:composition>
  </html>
```

В листинге 11.20 показана нижняя часть страницы. Для отображения списка книг используется тег `<h:dataTable>`, чье значение извлекается прямо из `BookEJB` (вместо `BookController`) благодаря аннотации `@Named` (см. листинг 11.16). Тег `<h:dataTable>` связывается с методом `bookEJB.findAllBooks()`, который возвращает объект типа `List<Book>` и объявляет переменную `bk` для перебора списка. Затем внутри тега `<h:dataTable>` вы можете указывать такие выражения, как `#{bk.price}`, чтобы получить атрибут книги `price` и отобразить его. Тег `<h:link>` создает HTML-ссылку, при выборе которой происходит переход к странице `viewBook.xhtml` и передается идентификатор книги (`viewBook.xhtml?ID=#{bk.id}`).

Написание страницы viewBook.xhtml

Страница `viewBook.xhtml` позволяет просмотреть сведения о книге (название, цену, описание, количество страниц и наличие иллюстраций). Она использует возможности добавления в закладки JSF 2, а это означает, что вы можете получить доступ к деталям любой книги, просто изменив URL благодаря `ViewAction`.

Как видно в листинге 11.21, страница привязывается к параметру `id` (например, `viewBook.xhtml?id=123`) атрибута `#{bookController.book.id}`. Как только идентификатор привязывается, вызывается действие `#{bookController.doFindBookById}`. Метод `BookController.doFindBookById` (см. листинг 11.17) вызывает EJB и получает книгу из базы данных, а затем страница может связать атрибуты книги (например, `#{bookController.book.title}`) с компонентами для вывода текста.

Листинг 11.21. Страница ViewBook.xhtml

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

  <f:metadata>
    <f:viewParam name="id" value="#{bookController.book.id}"/>
```

```
      <f:viewAction action="#{bookController.doFindBookById}"/>
    </f:metadata>

    <ui:define name="title">Показать книгу</ui:define>

    <ui:define name="content">
      <h:panelGrid columns="2">
        <h:outputLabel value="Название : "/>
        <h:outputText value="#{bookController.book.title}"/>

        <h:outputLabel value="Цена : "/>
        <h:outputText value="#{bookController.book.price}"/>

        <h:outputLabel value="Описание : "/>
        <h:outputText value="#{bookController.book.description}" cols="16"
rows="3"/>

        <h:outputLabel value="Количество страниц : "/>
        <h:outputText value="#{bookController.book.nbOfPage}"/>

        <h:outputLabel value="Иллюстрации : "/>
        <h:selectBooleanCheckbox value="#{bookController.book.illustrations}"/>
      </h:panelGrid>
    </ui:define>
  </ui:composition>
  </html>
```

Компиляция и упаковка с помощью Maven

Веб-приложения должны быть скомпилированы и упакованы в архив с расширением `.war` (`<packaging>war</packaging>`). Файл `pom.xml`, показанный в листинге 11.22, объявляет все необходимые зависимости для компиляции кода (`glassfish-embedded-all`) и использует JDK версии 1.7.

Листинг 11.22. Файл pom.xml для компиляции и упаковки веб-приложений

```
<?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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <artifactId>chapter11</artifactId>
    <groupId>org.agoncal.book.javaee7</groupId>
    <version>1.0</version>
  </parent>
  <groupId>org.agoncal.book.javaee7.chapter11</groupId>
  <artifactId>chapter11</artifactId>
  <version>1.0</version>
```

```
    <packaging>war</packaging>

    <dependencies>
      <dependency>
        <groupId>org.glassfish.main.extras</groupId>
        <artifactId>glassfish-embedded-all</artifactId>
        <version>4.0</version>
        <scope>provided</scope>
      </dependency>
    </dependencies>

    <build>
      <plugins>
        <plugin>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>2.5.1</version>
          <configuration>
            <source>1.7</source>
            <target>1.7</target>
          </configuration>
        </plugin>
        <plugin>
          <artifactId>maven-war-plugin</artifactId>
          <version>2.2</version>
          <configuration>
            <failOnMissingWebXml>false</failOnMissingWebXml>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </project>
```

Для компиляции и упаковки классов откройте интерпретатор командной строки в каталоге, содержащем файл `pom.xml`, и введите следующую команду Maven:

```
$ mvn package
```

Перейдите к каталогу target и найдите файл `chapter11-1.0.war`. Откройте его, и вы увидите, что он содержит сущность `Book`, `BookEJB`, компонент-подложку `BookController`, три дескриптора развертывания (`persistence.xml`, `faces-config.xml` и `beans.xml`) и две веб-страницы (`newBook.xhtml` и `viewBook.xhtml`).

Развертывание на GlassFish

После того как вы упакуете веб-приложение, оно должно быть развернуто в GlassFish. Убедитесь, что GlassFish запущен и работает. Откройте окно командной строки, перейдите в каталог target, где находится файл `chapter11-1.0.war`, и введите следующую команду:

```
$ asadmin deploy chapter11-1.0.war
```

Если развертывание прошло успешно, следующая команда должна возвращать имя развернутого приложения и его тип. Будет выведено два типа: web, поскольку это веб-приложение, и EJB, потому что приложение содержит EJB.

```
$ asadmin list components
chapter11-1.0 <ejb, web>
```

Запуск примера

Теперь, когда приложение развернуто, откройте браузер и перейдите по следующему URL: http://localhost:8080/chapter11-1.0/newBook.faces.

Браузер указывает на newBook.faces, а не на newBook.xhtml из-за стандартного преобразования FacesServlet. По расширению .faces JSF узнает, что должен обработать страницу перед ее отрисовкой. Как только появится страница newBook, введите данные и нажмите кнопку Отправить. Книга сохранится в базе данных и появится в списке в нижней части страницы.

Если вы используете инструмент веб-разработчика для проверки того, что происходит в сети между браузером и GlassFish, то увидите AJAX в действии. Отрисовывается частичный XML-ответ, а не вся HTML-страница. Частичный ответ сервера содержит часть страницы XHTML, которая будет обновляться. JavaScript ищет элементы страницы errors и booklist и применяет необходимые изменения (обновляя DOM). Частичный ответ в листинге 11.23 говорит сам за себя. Он указывает, что обновление должно быть сделано для компонентов, определенных в элементах errors и booklist (<update id="booklist">). Тело элемента <update> является фрагментом XHTML, который должен переопределить фактические данные таблицы. Конечно, этот XML автоматически обрабатывается с помощью обработчиков AJAX, встроенных в JSF, — разработчик не должен ничего делать с ним вручную.

Листинг 11.23. Частичный ответ, полученный браузером

```
<?xml version='1.0' encoding='UTF-8'?>
<partial-response>
  <changes>
    <update id="errors">
      <![CDATA[
      <ul id="errors"><li style="color:blue">Книга создана</li></ul>
      ]]>
    </update>
    <update id="booklist">
      <![CDATA[
      <table id="booklist" border="1">
        <thead>
          <tr>
            <th scope="col">Название</th>
            <th scope="col">Цена</th>
          </tr>
        </thead>
        <tbody>
          <tr>
            <td><a href="/chapter11-1.0/viewBook.faces?id=54">Java EE 6</a></td>
```

```
            <td>31.49</td>
          </tr>
          <tr>
            <td><a href="/chapter11-1.0/viewBook.faces?id=1">Гарри Поттер</a></td>
            <td>19.79</td>
          </tr>
          <tr>
            <td><a href="/chapter11-1.0/viewBook.faces?id=51">H2G2</a></td>
            <td>23.99</td>
          </tr>
        </tbody>
      </table>
      ]]>
    </update>
  </changes>
</partial-response>
```

Резюме

В главе 10 рассматривался графический аспект JSF, а эта глава была посвящена его динамической стороне. JSF следует шаблону проектирования MVC, и его спецификация охватывает все: от создания пользовательских интерфейсов с компонентами до обработки данных с помощью компонентов-подложек.

Компоненты-подложки находятся в основе JSF, поскольку используются для обработки бизнес-логики, вызовов EJB и баз данных, а также для навигации между страницами. Они имеют область действия и жизненный цикл (этим они напоминают сеансовые компоненты, не сохраняющие состояние), а также объявляют методы и свойства, которые привязаны к компонентам пользовательского интерфейса, с помощью языка выражений. Аннотации и конфигурация с применением исключений значительно упрощены начиная с версии JSF 2.2, большая часть XML-конфигурации теперь необязательна.

В этой главе показано, как обрабатываются преобразование и валидация данных для любых компонентов, предназначенных для ввода данных. JSF определяет набор преобразователей и валидаторов для наиболее распространенных случаев, но он также позволяет вам легко создавать и регистрировать собственные. Интеграция с Bean Validation естественна, и вам не нужно писать код для интеграции с JSF.

Поскольку AJAX существовал уже в течение нескольких лет, JSF имеет его встроенную поддержку, что позволяет веб-страницам вызывать базовые компоненты асинхронно. JSF определяет стандартную библиотеку JavaScript, и разработчику не нужно писать сценарии. Вместо этого следует использовать функции для обновления части страницы.

Последующие четыре главы будут сосредоточены на взаимодействии с системами благодаря обмену сообщениями, веб-службам SOAP и RESTful с использованием XML или JSON.

Глава 12
Обработка XML и JSON

XML использовался в платформе Java EE с появления дескрипторов развертывания и метаданных. Мы имеем дело с XML, когда приходится работать с файлами `persistence.xml`, `beans.xml` или `ejb-jar.xml`. Часто работа с Java позволяет разработчикам впервые войти в богатый мир XML. Однако мы быстро обнаружили, что работа с XML — это больше чем простое развертывание веб-приложения или EJB.

XML — это промышленный стандарт, определенный W3C. Хотя он не привязан ни к какому языку программирования или поставщику программного обеспечения, этот язык решил проблему независимости данных и совместимости. XML является расширяемым, не зависит от платформы и поддерживает интернационализацию, поэтому он стал предпочтительным языком для обмена данными между программными компонентами, системами и предприятиями (например, за счет использования веб-служб SOAP, которые будут описаны в главе 14).

С другой стороны, JSON появился вместе с JavaScript и служит для представления простых структур данных менее избыточным, нежели XML, способом. Если быть точным, формат JSON часто используется для сериализации и передачи структурированных данных по сети. Он быстро стал настолько популярным, что сегодня последние версии браузеров имеют встроенную поддержку кодирования/декодирования JSON. В дополнение к его отношениям с браузерами и JavaScript, JSON может служить форматом обмена данными (например, широко используется в веб-службах RESTful, которые будут рассмотрены в главе 15).

В данной главе я опишу оба формата, XML и JSON, сосредоточив внимание на структуре документа и API-интерфейсах, позволяющих манипулировать этими структурами. Мир XML богаче, так что вы увидите несколько спецификаций, которые могут помочь в анализе, проверке или связывании XML с объектами Java. Они настолько укоренились в нашей экосистеме, что большинство этих XML-спецификаций относятся к Java SE. JSON является относительным новичком в платформе Java и, следовательно, имеет меньше поддержки в Java SE/EE.

Основные сведения об XML

Расширяемый язык разметки (XML), унаследовавший многое от SGML (Standard Generalized Markup Language — стандартный обобщенный язык разметки), разрабатывался как язык для определения новых форматов документов во Всемирной

паутине. XML может считаться метаязыком, так как он используется для построения других языков. Сегодня он обеспечивает основу для многих языков, специфичных для конкретных областей, таких как MathML (Mathematical Markup Language — математический язык разметки), VXML (Voice Markup Language — голосовой язык разметки) или OpenOffice и LibreOffice (OpenDocument).

XML используется для создания структурированных данных, доступных для чтения человеком, и самостоятельного описания документов, которые соответствуют набору правил. XML-анализаторы могут затем проверить структуру любого документа XML, учитывая правила его языка. XML-документы — это текстовые структуры, описываемые с помощью тегов разметки (слова, окруженные символами '<' и '>').

XML-документ

В листинге 12.1 показан XML-документ, представляющий заказ клиента в приложении CD-BookStore (см. главу 1). Следует отметить, что этот документ легко читаем, а также структурирован и, следовательно, может распознаваться внешней системой. В этом случае он описывает информацию о заказе, клиенте, купленных товарах и кредитной карте, использованной для оплаты.

Листинг 12.1. XML-документ, представляющий собой заказ

```
<?xml version="1.0" encoding="UTF-8" ?>
<order id="1234" date="05/06/2013">
  <customer first_name="Джеймс" last_name="Роррисон">
    <email>j.rorri@me.com</email>
    <phoneNumber>+44 1234 1234</phoneNumber>
</customer>
<content>
    <order_line item="H2G2" quantity="1">
      <unit_price>23.5</unit_price>
    </order_line>
    <order_line item="Гарри Поттер" quantity="2">
      <unit_price>34.99</unit_price>
    </order_line>
  </content>
  <credit_card number="1357" expiry_date="10/13" control_number="234"
type="Visa"/>
</order>
```

Документ начинается с необязательного объявления XML (с указанием, какая версия XML и кодировка символов используются в документе), далее идет разметка и содержимое. Разметка, также называемая тегами, описывает структуру документа, что позволяет легко отправлять и получать данные или преобразовывать их из одного формата в другой.

Как вы можете видеть в табл. 12.1, терминология XML несложная. Несмотря на эту простоту и читаемость, XML можно использовать для описания любого вида документов, структур данных или дескрипторов развертывания, когда дело доходит до Java EE.

Таблица 12.1. Терминология XML

Термин	Определение
Символ Unicode	XML-документ — это символьная строка, которая может состоять из практически всех доступных символов Unicode
Разметка и содержимое	Символы Unicode подразделяются на разметку и содержимое. Разметка начинается с символов < и > (например, <email>), а все, что не является разметкой, считается содержимым (например, j.rorri@me.com)
Тег	Теги могут быть трех видов — открывающий (<email>), закрывающий (</email>) и пустой (email/)
Элемент	Элемент начинается с открывающего тега и заканчивается соответствующим закрывающим (или состоит только из пустого тега). Он также может содержать прочие элементы, которые будут называться потомками. Пример элемента: <email>j.rorri@me.com</email>
Атрибут	Атрибут представляет собой пару «имя/значение», которая располагается в открывающем или закрывающем теге. В следующем примере item — это атрибут тега order_line: <order_line item="H2G2">
Объявление XML	XML-документы могут начинаться с объявления информации о них, что показано в следующем примере: <?xml version="1.0" encoding="UTF-8" ?>

Проверка схемы XML

Терминология XML настолько широка, что позволяет написать с помощью XML все что угодно и объявить собственный язык. На самом деле можно написать так много всего, что ваша структура XML станет бессмысленной, если вы не определите грамматику. Эта грамматика может быть установлена с помощью определения схемы XML. Подключив грамматику к XML-документу, вы можете заставить любой анализатор XML проверять документ на соответствие правилам определенного диалекта XML. Это позволит разгрузить приложение, поскольку анализатор автоматически проверяет XML-документы.

ПРИМЕЧАНИЕ

Первый и самый ранний механизм определения языка — это определение типа документа (DTD). Хотя он используется в некоторых старых фреймворках, механизм DTD был заменен на XSD в связи с многочисленными ограничениями DTD. Одним из основных ограничений является тот факт, что DTD не представляет собой корректный XML-документ, поэтому он не может обрабатываться XML-анализаторами как обычный XML. DTD также не может полноценно ограничивать структуру и содержимое XML-документов.

Определение схемы XML (XSD) представляет собой основанное на XML определение грамматики, которое используется для описания структуры и содержимого XML-документа. Например, схема в листинге 12.2 может использоваться для XML-документа, представленного выше (см. листинг 12.1), придавая ему дополнительный смысл: «Это не просто текстовый файл, это структурированный документ, представляющий заказ, содержащий товары и информацию о клиенте». Во время обмена документами XSD описывает соглашение между производителем и потребителем, потому что описывает корректное XML-сообщение между двумя сторонами.

Листинг 12.2. XSD, описывающее XML-документ, который содержит информацию о заказе

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema version="1.0" xmlns:xs= "http://www.w3.org/2001/XMLSchema">
  <xs:element name="order" type="order"/>

  <xs:complexType name="creditCard">
    <xs:sequence/>
    <xs:attribute name="number" type="xs:string"/>
    <xs:attribute name="expiry_date" type="xs:string"/>
    <xs:attribute name="control_number" type="xs:int"/>
    <xs:attribute name="type" type="xs:string"/>
  </xs:complexType>

  <xs:complexType name="customer">
    <xs:sequence>
      <xs:element name="email" type="xs:string" minOccurs="0"/>
      <xs:element name="phoneNumber" type="xs:string" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="first_name" type="xs:string"/>
    <xs:attribute name="last_name" type="xs:string"/>
  </xs:complexType>

  <xs:complexType name="order">
    <xs:sequence>
      <xs:element name="customer" type="customer" minOccurs="0"/>
      <xs:element name="content" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="order_line" type="orderLine" minOccurs= "0" ➥
                        maxOccurs= "unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="credit_card" type="creditCard" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:long"/>
    <xs:attribute name="date" type="xs:dateTime"/>
  </xs:complexType>

  <xs:complexType name="orderLine">
    <xs:sequence>
      <xs:element name="unit_price" type="xs:double" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="item" type="xs:string"/>
    <xs:attribute name="quantity" type="xs:int"/>
  </xs:complexType>
</xs:schema>
```

Как видите, XSD позволяет очень точно определить как простые (`<xs:attribute name="expiry_date" type="xs:string"/>`), так и сложные типы данных (`<xs:complexType`

name="creditCard">) и даже позволяет типам наследовать свойства других типов. Схема заказа состоит из различных дочерних элементов, наиболее значимых элементов, сложных типов и атрибутов, которые определяют внешний вид содержимого. Благодаря XSD элементы и атрибуты становятся строго типизированными и имеют информацию о типе данных, связанную с ними. Данный строго типизированный XML теперь можно преобразовать в объект с использованием таких технологий, как JAXB, о которой вы узнаете далее в этой главе.

В табл. 12.2 перечислено только подмножество XSD-элементов и атрибутов. Язык XSD гораздо богаче, но эта книга не является энциклопедией по нему. Если вы хотите узнать больше о XSD, его структуре и типах данных, посетите сайт, связанный с W3C.

Таблица 12.2. Элементы и атрибуты XSD

Элемент	Описание
schema	Корневой элемент любой схемы XML. Может содержать несколько атрибутов, например версию схемы
xmlns	Каждый элемент схемы имеет стандартный префикс xs: (или xsd:, может быть использован любой префикс), который связан с пространством имен схемы XML (xmlns) путем объявления: xmlns:xsd="http://www.w3.org/2001/XMLSchema"
element	Элементы объявляются с помощью элемента element. Например, order — это элемент, который в документе выглядит как <order id="1234" date="11/08/2013" total_amount="93.48">
type	Элемент может иметь простой тип, например string, decimal, long или double (type="xs:long") или сложный (type="customer")
minOccurs, maxOccurs	Определяет минимальное или максимальное количество появлений типа. Может быть положительным целым числом или иметь значение unbounded, которое говорит о том, что верхней границы нет
complexType	Определяет комплексный тип, имеющий собственные элементы, подэлементы и атрибуты. Элемент типа complexType может содержать другие элементы типа complexType. Например, комплексный тип order содержит другой комплексный элемент
Sequence	Элемент может содержать другие элементы, которые называются элементами-потомками. Элемент такого типа указывает, что элементы-потомки должны следовать в цепочке. Каждый потомок может встречаться от 0 до неограниченного количества раз
attribute	Комплексный тип может иметь один или несколько атрибутов, которые определены с помощью элементов типа attribute. Тип orderLine имеет два атрибута: item и quantity
choice	Используется для указания, что может присутствовать только один набор элементов
complexContent	Комплексный тип может расширять или ограничивать другой комплексный тип с помощью элемента типа complexContent
extension	Элемент такого типа расширяет существующий элемент типа simpleType или complexType

Если у вас есть XML-документ и связанное с ним определение схемы XML, вы можете использовать анализатор, чтобы он проверил документ за вас. Анализаторы могут быть одного из двух основных типов: DOM и SAX.

Анализ с помощью SAX и DOM

Перед тем как документ может быть использован, он должен быть проанализирован и проверен анализатором XML. Анализатор, также называемый процессором, выполняет анализ разметки и делает данные, содержащиеся в XML, доступными для нуждающихся в них приложений. Большинство XML-анализаторов можно использовать в двух различных режимах. Один режим — это объектная модель документа (DOM), который считывает весь источник данных XML и строит в памяти его древовидное представление. Другой режим — модель на основе событий под названием «простой API для XML» (SAX), который считывает источник данных XML и выполняет функции обратного вызова приложения всякий раз, когда он сталкивается с определенным разделом (концом элемента).

DOM

API объектной модели документа (Document Object Model, DOM), как правило, легко использовать. Он обеспечивает привычную древовидную структуру объектов, что позволяет приложению переставлять узлы, а также добавлять или удалять содержимое по мере необходимости.

Модель DOM, как правило, проста в реализации, но ее построение требует чтения всей XML-структуры и хранения всего дерева объектов в памяти. Таким образом, DOM лучше использовать для небольших структур данных XML в случаях, когда скорость не имеет первостепенного значения для приложения или требуется произвольный доступ ко всему содержимому документа. Есть и другие технологии, такие как JDOM и DOM4J, которые дают возможность применять принципы объектно-ориентированного программирования при работе с XML для несложных приложений.

SAX

Потоковая модель анализаторов используется для локальной обработки ресурсов, где произвольный доступ к другим частям данных ресурса не требуется. Простой API для XML (SAX) основан на частичном анализе потоковой модели, в которой данные пошагово поставляются в приложение для чтения клиентом.

SAX основан на событиях и имеет последовательный механизм доступа, что позволяет обрабатывать один элемент за другим. При использовании анализатора SAX событие SAX срабатывает всякий раз, когда выявляется конец элемента XML. Событие включает в себя имя элемента, который только что закончился. Обработчик SAX — это конечный автомат, который может работать только с частью XML-документа, который уже был проанализирован.

SAX является самым быстрым способом анализа XML и подходит для работы с большими документами, которые не могут быть считаны в память целиком. Это, как правило, предпочтительно для высокопроизводительных серверных приложе-

ний и фильтров данных, которые не требуют представления данных в памяти. Однако этот способ очень требователен к навыкам разработчика.

Выполнение запросов с помощью XPath

XPath — это язык запросов, предназначенный для опроса XML-структур, которые используют другие XML-стандарты, в том числе XSLT, XPointer и XQuery. Он определяет синтаксис для создания выражений, значение которых оценивается на базе XML-документа.

Выражения XPath могут представлять собой узел, двоичное значение, число или строку. Наиболее распространенный тип выражений XPath — это путь, который представляет собой адрес узла. Например, выражение XPath `/` — это выражение, которое представляет все узлы XML-документа, начиная с корневого. Ниже приводится выражение XPath, представляющее все узлы `unit_price`, стоимость которых составляет более 20 (сам XML-документ вы видели выше (см. листинг 12.1)).

```
//content/order_line[unit_price>=20]/unit_price
```

Кроме того, XPath имеет набор встроенных функций, которые позволяют разрабатывать очень сложные выражения. Следующее выражение возвращает текстовый узел-потомок элементов `unit_price`:

```
//content/order_line[unit_price>=20]/unit_price/text()
```

XQuery — это другой язык запросов, предназначенный для опроса коллекций данных XML с помощью выражений XPath. XQuery синтаксически похож на SQL, его набор ключевых слов включает в себя `FOR`, `LET`, `WHERE`, `ORDER BY` или `RETURN`. Ниже представлено простое выражение XQuery, которое использует функцию `doc()` (она читает документ `Order.xml`), чтобы вернуть все дочерние узлы `order_line`, у которых значение атрибута `quantity` больше `1`, а `unit_price` — менее `50`;

```
for $orderLine in doc("order.xml")//content/order_line[@quantity>1]
where $orderLine/unit_price < 50
return $orderLine/unit_price/text()
```

С помощью XQuery можно выполнить и более сложные запросы, создавая сложные условия или упорядочивая результаты.

В некоторых случаях извлечение информации из XML-документа с использованием API может быть слишком громоздким, в основном из-за того, что критерий для поиска данных сложен и необходимо писать много кода для перебора узлов. Языки запросов XML, такие как XPath 1.0 и XQuery, предоставляют многофункциональные механизмы извлечения информации из XML-документов.

Преобразование с помощью XSLT

Ключевым преимуществом XML, по сравнению с другими форматами данных, является способность к преобразованию XML-документа из одного словаря в другой в общем виде. Например, вы можете трансформировать XML-документ в форму, подготовленную к печати, или веб-страницу. Технология, которая позволяет такое преобразование, называется расширяемым языком стилей для преобразований (XSLT).

Проще говоря, XSLT представляет собой фреймворк для трансформации структуры XML-документа, объединив его с таблицей стилей XSL для получения выходного документа. Все, что вам нужно сделать, — это создать таблицу стилей XSL, которая содержит набор инструкций по преобразованию для трансформации исходного дерева в конечное. Затем процессор XSLT преобразует исходный документ, связывая шаблоны исходного дерева XML с шаблонами применяемой таблицы стилей.

Шаблон — это выражение XPath, которое сравнивается с элементами исходного дерева. При совпадении создается часть результирующего дерева. При конструировании результирующего дерева элементы источника могут быть отфильтрованы и переупорядочены, также может быть добавлена произвольная структура.

С помощью таблицы стилей XSLT, представленной в листинге 12.3, документ `order.xml`, показанный выше (см. листинг 12.1), преобразуется в веб-страницу XHTML, показывая таблицу проданных товаров, цена которых — больше 30.

Листинг 12.3. Таблица стилей XSLT для документа, содержащего информацию о продажах

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/
Transform">
  <xsl:template match="/">
    <html>
      <body>
        <h2>Проданные товары</h2>
        <table border="1">
          <tr>
            <th>Название</th>
            <th>Количество</th>
            <th>Общая стоимость</th>
          </tr>

          <xsl:for-each select="order/content/order_line">
            <tr>
              <td>
                <xsl:value-of select="@item"/>
              </td>
              <td>
                <xsl:value-of select="@quantity"/>
              </td>

              <xsl:choose>
                <xsl:when test="unit_price> 30">
                  <td bgcolor="#FF0000">
                    <xsl:value-of select="unit_price"/>
                  </td>
                </xsl:when>
                <xsl:otherwise>
                  <td>
                    <xsl:value-of select="unit_price"/>
                  </td>
```

```
                </xsl:otherwise>
              </xsl:choose>
            </tr>
          </xsl:for-each>

        </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Общий API для XSLT располагается в пакете `javax.xml.transform` и используется для составления инструкций стилей и преобразования источника XML в результирующий XML-документ. XSLT также можно использовать вместе с SAX API, чтобы преобразовать данные в XML.

Обзор спецификаций XML

XML-спецификации стали рекомендацией W3C в 1998 году, после чего были представлены несколько XML-спецификаций, таких как XSLT, XPath, схемы XML и XQuery, которые стали стандартными в W3C. Не зависящий от платформы код Java и платформонезависимые данные XML являются двумя взаимодополняющими аспектами, которые привели к стандартизации и упрощению различных Java API для XML. Это сделало разработку приложений Java, связанных с XML, гораздо проще.

Краткая история XML-спецификаций

W3C — это консорциум, который известен развитием и поддержанием веб-технологий, таких как HTML, XHTML, RDF или CSS. W3C также является центральным органом XML и всех связанных с XML технологий — схем XML, XSLT, XPATH и DOM.

Развитие XML началось в 1996 году усилиями рабочей группы XML W3C и привело к появлению рекомендации W3C в феврале 1998 года. Тем не менее технология не была совершенно новой. Она была основана на SGML, который был разработан в начале 1980-х и стал стандартом ISO в 1986 году.

XSD (схема XML) предлагает средства для описания структуры XML-документов в файле с расширением `.xsd`. Эта структура ограничивает содержимое XML-документов и, следовательно, может быть использована для их проверки. XSD является одним из нескольких языков схем XML. Он был первым самостоятельным языком схем для XML, опубликованным в качестве рекомендации W3C в 2001 году.

XSLT — это одна из первых спецификаций XML, которая была создана под влиянием функциональных языков и языков текстовых шаблонов. Она используется для преобразования XML-документов. Самым прямым предшественником является DSSSL (Document Style Semantics and Specification Language — язык описания семантики и стиля документа), который выполнял для SGML те же задачи, которые сейчас XSLT выполняет для XML. XSLT 1.0 стал частью W3C в 1999 году,

и этот проект привел к созданию XPath. Xalan, Saxon и AltovaXML — лишь некоторые из процессоров XSLT, предназначенных для преобразования XML.

XPath 1.0 — это язык запросов для адресации узлов в документе XML. Его ввели и приняли в качестве рекомендации W3C в 1999 году. Первоначально он мотивировался желанием обеспечить общий синтаксис для XPointer и XSLT. XPath 1.0 может быть использован непосредственно внутри Java или встроен в такие языки, как XSLT, XQuery или схема XML.

Миссия проекта XML Query заключается в предоставлении гибкой системы запросов для извлечения данных из документов. Развитие XQuery 1.0 усилиями рабочей группы XML Query было тесно скоординировано с развитием XSLT 2.0 рабочей группой XSL. Эти две группы несут общую ответственность за XPath 2.0, которая представляет собой подмножество XQuery 1.0. XQuery 1.0 стал рекомендацией W3C 23 января 2007 года.

DOM — это древоподобный интерфейс для представления и взаимодействия с содержимым, структурами и стилями в HTML-, XHTML- и XML-документах. В начале существования DOM был попыткой разработать стандарт для языков сценариев, используемый в браузерах. Текущая версия спецификации DOM, DOM Level 3, поддерживает XPath, а также интерфейс для сериализации документов в формате XML.

SAX является первым широко принятым API для XML в Java. Это потоковый, основанный на событиях интерфейс для анализа XML-данных. SAX был первоначально реализован в Java, но теперь поддерживается почти во всех основных языках программирования.

С самого начала разработка спецификаций XML была связана с повышением удобства использования XML.

В табл. 12.3 приведены некоторые спецификации для технологий XML.

Таблица 12.3. Спецификации W3C для XML

Спецификация	Версия	URL
Extensible Markup Language (XML)	1.1	http://www.w3.org/TR/xml11/
XML Schema	1.0	http://www.w3.org/TR/xmlschema-1
Extensible Stylesheet (XSLT)	1.0	http://www.w3.org/TR/xslt
XML Path (XPath)	1.0	http://www.w3.org/TR/xpath
Document Object Model (DOM)	level 3	http://www.w3.org/TR/DOM-Level-3-Core/
Simple API for XML (SAX)	2.0.2	http://sax.sourceforge.net/

Спецификации XML в Java

Экосистема XML была создана W3C. Однако поскольку она работает рука об руку с Java, есть несколько связанных с XML спецификаций, которые были созданы в рамках JCP. Примеров множество — от обработки XML до связывания документов с объектами Java.

JAXP (Java Architecture for XML Processing — архитектура Java для обработки XML) — это низкоуровневая спецификация (JSR 206), которая дает возможность

очень гибко обрабатывать XML, а также позволяет использовать SAX, DOM или XSLT. Этот API также применяется JAXB и StAX.

Спецификация JAXB обеспечивает набор API-интерфейсов и аннотаций для представления XML-документов как артефактов Java, что позволяет работать с соответствующими объектами Java. JAXB (JSR 222) обрабатывает демаршалинг документов XML в объекты и наоборот. Даже если JAXB можно использовать для работы с XML, он тесно интегрирован с JAX-WS (смотрите главу 14).

StAX (Streaming API for XML — потоковый API для XML) версии 1.0 (JSR 173) представляет собой API для чтения и записи XML-документов. Основным направлением его деятельности является использование преимуществ древоподобных API (анализаторов DOM) и API, основанных на событиях (анализаторов SAX). Первые позволяют получить произвольный, неограниченный доступ к документу, в то время как вторые занимают меньше памяти и предъявляют меньше требований к процессору.

В табл. 12.4 перечислены все спецификации Java, связанные с XML.

Таблица 12.4. Спецификации, связанные с XML

Спецификация	Версия	JSR	URL
JAXP	1.3	206	http://jcp.org/en/jsr/detail?id=206
JAXB	2.2	222	http://jcp.org/en/jsr/detail?id=222
StAX	1.0	173	http://jcp.org/en/jsr/detail?id=173

Примеры реализаций

Основная цель примеров реализаций (Reference Implementation, RI) — это поддержка разработки спецификаций и их проверка. StAX RI является примером реализации для спецификации JSR-173, которая основана на стандартном анализаторе потоковой модели. StAX был включен в JDK, начиная с версии 1.6, и может быть загружен отдельно для JDK версий 1.4 и 1.5. RI для спецификации JAXP интегрирован в Java SE, как и Metro, который является примером реализации JAXB. Metro — это высококачественная реализация JAXB, которая используется в некоторых продуктах компании Oracle.

Архитектура Java для обработки XML

Архитектура Java для обработки XML (Java Architecture for XML Processing, JAXP) представляет собой API , который обеспечивает общий, независимый от реализации интерфейс для создания и использования SAX, DOM и XSLT API в Java.

До появления JAXP существовали разные несовместимые версии анализаторов и преобразователей XML от различных поставщиков. JAXP предоставил уровень абстракции поверх этих реализаций конкретных производителей API для анализа и преобразования XML-ресурсов.

Обратите внимание, что JAXP не использует другой механизм анализа и преобразования XML-документов. Вместо этого приложения могут применять его

для доступа к базовым XML API. Приложения могут заменить реализацию одного поставщика другой.

Используя JAXP, можно анализировать XML-документы с помощью SAX или DOM или преобразовать их в новый формат с помощью XSLT. Архитектура JAXP API изображена на рис. 12.1.

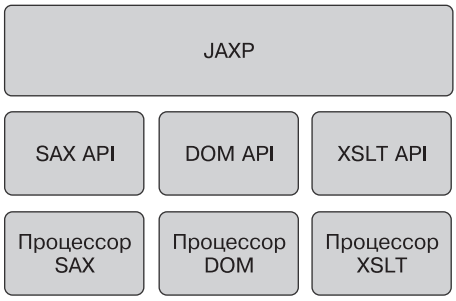


Рис. 12.1. Архитектура JAXP

JAXP состоит из четырех пакетов, перечисленных в табл. 12.5. В них вы найдете интерфейсы и классы, предназначенные для анализа и преобразования XML-данных.

Таблица 12.5. Пакеты JAXP

Пакет	Описание
javax.xml.parsers	Общий интерфейс для анализаторов DOM и SAX
org.w3c.dom	Общий API для работы с DOM в Java
org.xml.sax	Определяет интерфейсы, использованные для анализаторов SAX
javax.xml.transform	API XLST для преобразования XML в другие типы документов

Конфигурирование JAXP

Ввиду гибкости JAXP его можно настроить на использование любой реализации обработки. Однако вы всегда можете применить стандартные значения в качестве примера реализации (JAXP использует Xerces как стандартный синтаксический анализатор XML и Xalan — как процессор XSLT для преобразования XML-документов).

Представьте себе сценарий, в котором имеется более одной совместимой с JAXP реализации в пути к вашим классам. В этом случае вы должны указать JAXP, какой API следует использовать. В зависимости от того, работает ваше приложение в автономном режиме клиента или режиме сервера приложений, вы можете предоставить своему приложению анализатор или преобразователь XML с помощью файла свойств. Например, поместив файл `jaxp.properties` (стандартный файл, следующий формату `java.util.Properties`) в подкаталоге lib каталога JRE указывает реализации JAXP использовать заданные фабрики. Ниже представлено содержимое файла `jaxp.properties`, который определяет, какой строитель DOM, анализатор SAX и преобразователь XSLT будет применяться:

```
javax.xml.parsers.DocumentBuilderFactory=org.apache.xerces.jaxp.
DocumentBuilderFactoryImpl
javax.xml.parsers.SAXParserFactory=org.apache.xerces.jaxp.SAXParserFactoryImpl
javax.xml.transform.TransformerFactory=org.apache.xalan.processor.
TransformerFactoryImpl
```

Другой способ заключается в настройке системных свойств Java перед запуском приложения. Например, следующее системное свойство указывает JVM использовать Xerces как анализатор XML:

```
-Djavax.xml.parsers.DocumentBuilderFactory=org.apache.xerces.jaxp.
DocumentBuilderFactoryImpl
```

Основные свойства системы, которые можно изменить, чтобы ввести новые анализаторы или преобразователи, приведены в табл. 12.6.

Таблица 12.6. Системные свойства для конфигурации анализаторов/преобразователей XML

Системное свойство	Описание
javax.xml.parsers.DocumentBuilderFactory	Устанавливает строитель DOM
javax.xml.parsers.SAXParserFactory	Конфигурирует анализатор SAX
javax.xml.transform.TransformerFactory	Определяет, какую реализацию XSLT следует использовать

JAXP и SAX

SAX известен своими низкими требованиями к памяти и быстрой обработкой данных. SAX — это управляемый событиями механизм с последовательным доступом, предназначенный для анализа XML-документов. Вы должны предоставить анализатор с методами обратного вызова, вызываемыми анализатором по мере чтения XML-документа. Например, анализатор SAX вызывает один метод приложения всякий раз, когда достигает конца элемента, и вызывает другой метод, если встречается текстовый узел.

Способ обработки текущего элемента XML-документа без сохранения состояния ранее проанализированных элементов называется обработкой, не зависящей от состояния. Это наиболее подходящая модель обработки XML-ресурсов с помощью анализаторов SAX. Другой вариант — обработка, зависящая от состояния, которая обрабатывается такими анализаторами, как StAX.

В листинге 12.4 показан класс, который анализирует документ `order.xml` с помощью модели SAX, основанной на событиях. Класс `SaxParsing` наследуется от класса `DefaultHandler`, который требуется для решения различных задач анализа и реализует четыре различных обработчика (`ContentHandler`, `ErrorHandler`, `DTDHandler` и `EntityResolver`). `SAXParserFactory` настраивает и создает объект анализатора SAX. Как упоминалось ранее, можно вручную настроить системное свойство `javax.xml.parsers.SAXParserFactory` на использование сторонних анализаторов SAX. `SAXParser` является оберткой объекта `SAXReader`, на который может ссылаться `getXMLReader()`, поэтому, когда вызывается метод `parse()` анализатора SAX, читатель вызывает один

из нескольких методов обработчика, реализованных приложением (например, метод `StartElement`).

Листинг 12.4. Анализатор SAX, обрабатывающий документ, который хранит информацию о заказе

```
public class SaxParsing extends DefaultHandler {

  private List<OrderLine> orderLines = new ArrayList<>();
  private OrderLine orderLine;
  private Boolean dealingWithUnitPrice = false;
  private StringBuffer unitPriceBuffer;

  public List<OrderLine> parseOrderLines() {

    try {
      File xmlDocument = Paths.get("src/main/resources/order.xml").toFile();

      // Фабрика SAX
      SAXParserFactory factory = SAXParserFactory.newInstance();
      SAXParser saxParser = factory.newSAXParser();

      // Анализ документа
      saxParser.parse(xmlDocument, this);
    }  catch (SAXException | IOException | ParserConfigurationException e) {
      e.printStackTrace();
    }
    return orderLines;
  }

  @Override
  public void startElement(String namespaceURI, String localName, String
qualifiedName,
                           Attributes attrs) throws SAXException {

    switch (qualifiedName) {
      // Получение узла order_line
      case "order_line":
        orderLine = new OrderLine();
        for (int i = 0; i < attrs.getLength(); i++) {
          switch (attrs.getLocalName(i)) {
            case "item":
              orderLine.setItem(attrs.getValue(i));
              break;
            case "quantity":
              orderLine.setQuantity(Integer.valueOf(attrs.getValue(i)));
              break;
          }
        }
        break;
      case "unit_price":
```

```
          dealingWithUnitPrice = true;
          unitPriceBuffer = new StringBuffer();
          break;
      }
    }
    @Override
    public void characters(char[] ch, int start, int length) throws SAXException
{
        if (dealingWithUnitPrice)
          unitPriceBuffer.append(ch, start, length);
    }

    @Override
    public void endElement(String namespaceURI, String localName, String
qualifiedName)
                          throws SAXException {

      switch (qualifiedName) {
        case "order_line":
          orderLines.add(orderLine);
          break;
        case "unit_price":
          orderLine.setUnitPrice(Double.valueOf(unitPriceBuffer.toString()));
          dealingWithUnitPrice = false;
          break;
      }
    }
}
```

Интерфейс `ContentHandler` обрабатывает основные связанные с документами события, такие как начало и конец элемента, с помощью методов `startDocument`, `endDocument`, `startElement` и `endElement`. Эти методы вызываются, когда встречаются открывающий или закрывающий теги XML. В примере, приведенном выше (см. листинг 12.4), метод `startElement` проверяет, является ли элемент `order_Line` либо `unit_price`, чтобы или создать объект типа `OrderLine`, или получить значение цены за единицу товара. Интерфейс `ContentHandler` также имеет метод `characters()`, который вызывается, когда анализатор встречает группу символов в элементе XML (в примере выше (см. листинг 12.4) метод `characters()` помещает в буфер цену за каждый товар).

Для обеспечения обработки ошибок во время анализа XML документа `ErrorHandler` может быть зарегистрирован в `SAXReader`. Методы интерфейса `ErrorHandler` (`warning`, `error` и `fatalError`) вызываются в ответ на различные типы ошибок анализа.

Интерфейс `DTDHandler` определяет методы для обработки событий, связанных с DTD. Анализатор использует `DTDHandler`, чтобы передать нотацию и непроанализированные объявления сущности приложению. При обработке DTD полезно распознать и действовать соответственно объявлениям непроанализированных сущностей.

JAXP и DOM

JAXP предоставляет интерфейсы для анализа и изменения XML-данных с использованием DOM API. Точкой входа является класс `javax.xml.parsers.DocumentBuilderFactory`. Он используется для производства объекта типа `DocumentBuilder`, как показано в листинге 12.5. Используя один из методов `parse` класса `DocumentBuilder`, вы можете создать древовидную структуру XML-данных в объекте типа `org.w3c.dom.Document`. Дерево содержит узлы дерева (например, элементы и текстовые узлы), которые являются реализациями интерфейса `org.w3c.dom.Node`. Кроме того, чтобы создать объект документа, вы можете использовать метод `newDocument()` объекта типа `DocumentBuilder`.

Листинг 12.5. Анализатор DOM, обрабатывающий документ, который хранит информацию о заказе

```
public class DomParsing {

  public List<OrderLine> parseOrderLines() {

    List<OrderLine> orderLines = new ArrayList<>();

    try {
      File xmlDocument = Paths.get("src/main/resources/order.xml").toFile();

      // Фабрика DOM
      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

      // Анализ документа
      DocumentBuilder documentBuilder = factory.newDocumentBuilder();
      Document document = documentBuilder.parse(xmlDocument);

      // Получение узла order_line
      NodeList orderLinesNode = document.getElementsByTagName("order_line");
      for (int i = 0; i < orderLinesNode.getLength(); i++) {
        Element orderLineNode = (Element) orderLinesNode.item(i);
        OrderLine orderLine = new OrderLine();
        orderLine.setItem(orderLineNode.getAttribute("item"));
        orderLine.setQuantity(Integer.valueOf(orderLineNode.
getAttribute("quantity")));

        Node unitPriceNode = orderLineNode.getChildNodes().item(1);
        orderLine.setUnitPrice(Double.valueOf(unitPriceNode.getFirstChild().
getNodeValue()));

          orderLines.add(orderLine);
      }

    }  catch (SAXException | IOException | ParserConfigurationException e) {
      e.printStackTrace();
    }
    return orderLines;
  }
}
```

В данном примере анализируется документ order.xml, в результате чего в памяти создается его древовидное представление. Благодаря многочисленным методам Document вы можете получить список узлов order_Line или атрибут quantity (например, GetAttribute("quantity")), чтобы создать объект OrderLine.

Как упоминалось ранее, можно переопределить стандартный для платформы анализатор DOM, установив соответствующим образом системное свойство javax.xml.parsers.DocumentBuilderFactory.

JAXP и XSLT

JAXP также используется для преобразования XML-документов с помощью XSLT API. XSLT взаимодействует с XML-ресурсами для трансформации источника XML в результирующий XML-документ с помощью преобразования стилей (см. листинг 12.3).

В листинге 12.6 принимается документ order.xml (см. листинг 12.1) и преобразуется в HTML-страницу с помощью XSLT, определенной выше (см. листинг 12.3). Сначала код использует метод newInstance() класса javax.xml.transform.TransformerFactory для создания экземпляра класса TransformerFactory. Затем он вызывается метод newTransformer() для создания нового преобразователя XSLT. Тогда он трансформирует документ order.xml в результирующую HTML-страницу.

Листинг 12.6. Преобразование XML-документа с помощью XSLT

```
public class XsltTransforming {
  public String transformOrder() {

    StringWriter writer = new StringWriter();

    try {
      File xmlDocument = Paths.get("src/main/resources/order.xml").toFile();
      File stylesheet = Paths.get("src/main/resources/order.xsl").toFile();

      TransformerFactory factory =
        TransformerFactory.newInstance("net.sf.saxon.TransformerFactoryImpl",
null);

      // Преобразование документа
      Transformer transformer = factory.newTransformer(new
StreamSource(stylesheet));
      transformer.transform(new StreamSource(xmlDocument), new
StreamResult(writer));

    }  catch (TransformerException e) {
      e.printStackTrace();
    }

    return writer.toString();
  }
}
```

Архитектура Java для связывания XML

Java предлагает различные способы манипулирования XML: от общих API, которые поставляются с JDK, таких как `javax.xml.stream.XMLStreamWriter` и `java.beans.XMLEncoder`, до более сложных и низкоуровневых моделей, таких как SAX, DOM или JAXP.

Спецификация (JSR 222) архитектуры Java для связывания XML (JAXB) обеспечивает более высокий уровень абстракции, чем SAX или DOM, и основана на аннотациях. JAXB определяет стандарт связывания представлений Java с XML и наоборот. Это позволяет работать с объектами Java, которые представляют XML-документы.

В листинге 12.7 представлен простой класс `CreditCard` с аннотацией JAXB `@javax.xml.bind.annotation.XmlRootElement`. JAXB свяжет объект типа `CreditCard` с XML.

Листинг 12.7. Класс CreditCard с аннотацией JAXB

```
@XmlRootElement
public class CreditCard {

   private String number;
   private String expiryDate;
   private Integer controlNumber;
   private String type;

   // Конструкторы, методы работы со свойствами
}
```

За исключением аннотации `@XmlRootElement` выше приведен код простого класса Java. С этим механизмом аннотаций и маршалинга JAXB способен создать XML-представление объекта типа `CreditCard`, которое выглядит как XML-документ, показанный в листинге 12.8.

Листинг 12.8. XML-документ, представляющий данные кредитных карт

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<creditCard>
  <controlNumber>566</controlNumber>
  <expiryDate>10/14</expiryDate>
  <number>12345678</number>
  <type>Visa</type>
</creditCard>
```

Маршалинг — это процесс преобразования объекта в XML (рис. 12.2). Обратное преобразование также возможно с помощью JAXB. При демаршалинге в качестве входных данных был бы принят XML-документ из листинга 12.8 и создан экземпляр типа `CreditCard` со значениями, определенными в документе.

JAXB управляет XML-документами и определениями XML-схем (XSD) прозрачным, объектно-ориентированным способом, что скрывает сложность языка XSD. JAXB может автоматически сгенерировать схему, которая проверила бы XML-структуру кредитной карты, чтобы убедиться, что она имеет правильную структуру и типы данных (благодаря утилите `schemagen`, поставляющейся с JDK). В листинге 12.9 показана XML-схема (XSD) класса `CreditCard`.

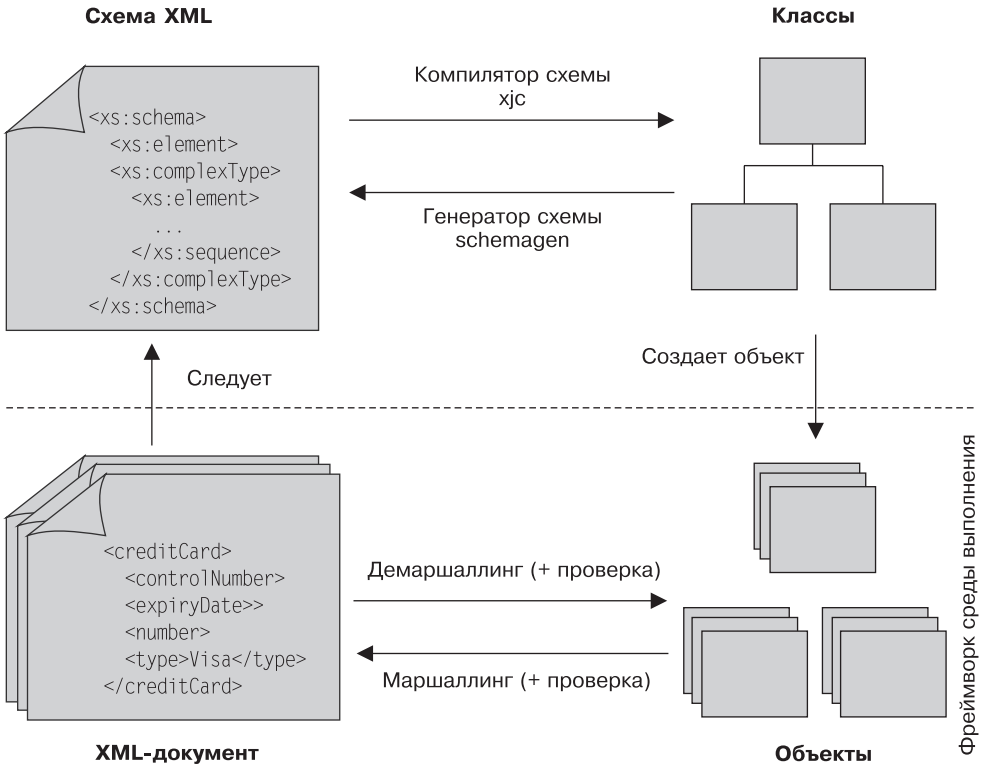


Рис. 12.2. Архитектура JAXB

Листинг 12.9. XML-схема для проверки предыдущего документа

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="creditCard" type="creditCard"/>
  <xs:complexType name="creditCard">
    <xs:sequence>
      <xs:element name="controlNumber" type="xs:int" minOccurs="0"/>
      <xs:element name="expiryDate" type="xs:string" minOccurs="0"/>
      <xs:element name="number" type="xs:string" minOccurs="0"/>
      <xs:element name="type" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Схема, приведенная выше, состоит из простых элементов (`controlNumber`, `ExpiryDate` и т. д.) и сложного типа (`creditCard`). Обратите внимание, что все метки используют префикс `xs` (`xs:element`, `xs:string` и т. д.). Он называется пространством имен и определяется в заголовочном теге документа `xmlns` (пространство имен XML):

```
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema" >
```

Пространства имен создают уникальные префиксы для элементов в виде отдельных документов или приложений, которые используются вместе. В основном они применяются, чтобы избежать конфликтов, которые могут возникнуть, если одинаковое имя элемента появляется в нескольких документах (например, тег `<element>` может встречаться в нескольких документах и иметь разные значения).

JAXB обеспечивает легкое двустороннее преобразование между объектами Java и XML-структурами. Это позволяет трансформировать объекты Java в XML-данные без необходимости создания сложного кода, который трудно поддерживать и отлаживать. Например, JAXB позволяет легко переносить состояние объекта в XML-данные или сериализовать его сетевой поток. С другой стороны, JAXB позволяет работать с XML-документами, как будто они являются объектами Java без необходимости явно выполнять анализ SAX или DOM в коде приложения.

Связывание

JAXB API, определенный в пакете `javax.xml.bind`, предоставляет набор интерфейсов и классов для создания XML-документов и генерации классов Java. Другими словами, он связывает две модели. Фреймворк среды выполнения JAXB реализует операции маршалинга и демаршалинга. В табл. 12.7 перечислены основные используемые для них пакеты JAXB.

Таблица 12.7. Пакеты JAXB

Пакет	Описание
javax.xml.bind	Фреймворк связывания среды выполнения, имеющий возможность выполнять операции маршалинга, демаршалинга и проверки
javax.xml.bind.annotation	Аннотации для настройки преобразований между программой Java и XML-данными
javax.xml.bind.annotation.adapters	Классы-адаптеры JAXB
javax.xml.bind.attachment	Позволяет выполнять маршалинг для оптимизации хранения двоичных данных и демаршалинг корня документа, содержащего форматы двоичных данных
javax.xml.bind.helpers	Содержит частичные стандартные реализации некоторых интерфейсов javax.xml.binding
javax.xml.bind.util	Предоставляет полезные вспомогательные классы

Как было показано выше (см. рис. 12.2), маршалинг — это процесс преобразования экземпляров аннотированных с помощью JAXB-классов в XML-представления. Кроме того, демаршалинг — это процесс преобразования XML-представлений в дерево объектов. В процессе маршалинга/демаршалинга JAXB также может проверить XML на соответствие схеме XSD (см. листинг 12.9). JAXB может также работать на уровне классов и способен автоматически генерировать схемы из набора классов и наоборот.

В центре JAXB API находится класс `javax.xml.bind.JAXBContext`. Этот абстрактный класс управляет связыванием между XML-документами и объектами Java, поскольку он предоставляет:

- класс `Unmarshaller`, который преобразует XML-документ в граф объектов и, возможно, проверяет XML;
- класс `Marshaller`, который принимает граф объектов и трансформирует его в документ XML.

Например, чтобы преобразовать наш объект `CreditCard` в документ XML (листинг 12.10), следует использовать метод `Marshaller.marshal()`. Этот метод принимает в качестве параметра объект и выполняет операцию маршалинга; конечный тип объекта может быть разным (`StringWriter` будет хранить строковое представление документа XML, а `FileOutputStream` сохранит его в файл).

Листинг 12.10. Класс Main, выполняющий маршалинг объекта CreditCard

```
public class Main {

  public static void main(String[] args) throws JAXBException {

    CreditCard creditCard = new CreditCard("1234", "12/09", 6398, "Visa");
    StringWriter writer = new StringWriter();

    JAXBContext context = JAXBContext.newInstance(CreditCard.class);
    Marshaller m = context.createMarshaller();
    m.marshal(creditCard, writer);

    System.out.println(writer.toString());
  }
}
```

Данный код создает экземпляр типа `JAXBContext` с помощью статического метода `newInstance()`, которому передается корневой класс, для которого необходимо выполнить маршалинг (`CreditCard.class`). Из созданного объекта `Marshaller` затем вызывается метод `marshal()`, который генерирует XML-представление (см. листинг 12.8) объекта, представляющего информацию о кредитной карте, в объект типа `StringWriter` и отображает его. Такой же подход можно применить для демаршалинга XML-документа в объект с помощью метода `Unmarshaller.unmarshal()`.

Metro, пример реализации JAXB, имеет и другие инструменты, в частности компилятор схем (`xjc`) и генератор схем (`schemaGen`). Хотя маршалинг/демаршалинг работает с объектами и XML-документами, компилятор и генератор схем работают с классами и XML-схемами. Эти инструменты можно использовать из командной строки (поставляются в комплекте с Java SE 7) или как цель Maven.

Аннотации

JAXB во многом похож на JPA. Тем не менее вместо преобразования объектов в форму, пригодную для записи в базу данных, JAXB выполняет преобразование в XML-документ. Кроме того, как и JPA, JAXB определяет набор аннотаций (в пакете `javax.xml.bind.annotation`), позволяющих настроить такое преобразование,

и зависит от конфигурации с помощью исключений для минимизации работы разработчика. Если постоянные объекты должны иметь аннотацию `@Entity`, в JAXB ее аналогом является `@XmlRootElement` (листинг 12.11).

Листинг 12.11. Пользовательский класс CreditCard

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class CreditCard {

  @XmlAttribute(required = true)
  private String number;
  @XmlElement(name = "expiry-date", defaultValue = "01/10")
  private String expiryDate;
  private String type;
  @XmlElement(name = "control-number")
  private Integer controlNumber;

  // Конструкторы, методы работы со свойствами
}
```

Здесь аннотация `@XmlRootElement` уведомляет JAXB, что класс `CreditCard` является корневым элементом XML-документа. Если эта аннотация отсутствует, JAXB сгенерирует исключение при попытке выполнить его маршалинг. Этот класс затем преображается в схему, показанную в листинге 12.12 с использованием всех стандартных правил преобразования JAXB (каждый атрибут сопоставляется с элементом и имеет такое же имя).

Листинг 12.12. Схема кредитной карточки с атрибутами и значениями по умолчанию

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="creditCard" type="creditCard"/>

  <xs:complexType name="creditCard">
    <xs:sequence>
      <xs:element name="expiry-date" type="xs:string" default="01/10"
minOccurs="0"/>
      <xs:element name="type" type="xs:string" minOccurs="0"/>
      <xs:element name="control-number" type="xs:int" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="number" type="xs:string" use="required"/>
  </xs:complexType>
</xs:schema>
```

С помощью объекта `Marshaller` вы можете легко получить XML-представление объекта `CreditCard` (см. листинг 12.10). Корневой элемент `<creditCard>` представляет объект `CreditCard` и включает в себя значение каждого атрибута.

JAXB дает возможность настраивать и управлять этой структурой XML. Документ XML состоит из элементов (`<element>Значение</element>`) и атрибутов (`<element attribute="значение"/>`). JAXB использует две аннотации, чтобы их раз-

личать: `@XmlAttribute` и `@XmlElement`. Каждая аннотация имеет набор параметров, который позволяет вам переименовать атрибут, разрешить или запретить указывать пустые значения, назначить значения по умолчанию и т. д. Свойства класса преобразуются в XML-элементы по умолчанию, если они не имеют аннотации `@XmlAttribute`.

В примере, приведенном выше (см. листинг 12.11), эти аннотации используются, чтобы включить номер кредитной карты в атрибут (а не в элемент) и изменить срок истечения карточки и контрольное число. Этот класс будет преобразован в другую схему, в которой номер кредитной карты представлен в виде обязательного атрибута `<xs:attribute>`, а срок истечения карточки изменяется на значение по умолчанию `01/10` (см. листинг 12.12).

XML-представление типа `CreditCard` также изменится, как вы можете видеть в листинге 12.13 (по сравнению с приведенным ранее (см. листинг 12.8)).

Листинг 12.13. XML-документ, представляющий пользовательский объект CreditCard

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<creditCard number="1234">
  <expiry-date>12/09</expiry-date>
  <type>Visa</type>
  <control-number> 6398</control-number>
</creditCard>
```

В табл. 12.8 перечислены основные аннотации JAXB. Некоторые из них могут аннотировать атрибуты (или методы получения значений свойств), другие классы, а некоторые — использоваться для всего пакета (например, `@XmlSchema`).

Таблица 12.8. Аннотации JAXB

Аннотация	Описание
@XmlAccessorType	Управляет необходимостью преобразования атрибутов или геттеров (FIELD, NONE, PROPERTY, PUBLIC_MEMBER)
@XmlAttribute	Преобразует атрибут или геттер к XML-атрибуту примитивного типа (String, Boolean, Integer и т. д.)
@XmlElement	Преобразует нестатический атрибут или геттер без модификатора transient в XML-элемент
@XmlElements	Действует как контейнер для нескольких аннотаций @XmlElement
@XmlEnum	Преобразует перечисление к представлению XML
@XmlEnumValue	Идентифицирует константу-перечисление
@XmlID	Идентифицирует ключевое поле элемента XML (или типа String), которое может быть использовано для ссылки на элемент с использованием аннотации @XmlIDREF (концепции XML Schema ID и IDREF)
@XmlIDREF	Преобразует свойство в схеме в XML IDREF
@XMLList	Преобразует свойство в список
@XmlMimeType	Идентифицирует текстовое представление типа MIME

Продолжение ⇩

Таблица 12.8 *(продолжение)*

Аннотация	Описание
@Xmlns	Идентифицирует пространство имен XML
@XmlRootElement	Представляет аннотацию, необходимую любому классу для связывания в качестве корневого элемента XML
@XmlSchema	Преобразует имя пакета в пространство имен XML
@XmlTransient	Информирует JXB, что не нужно связывать атрибут (аналогичен ключевому слову transient в Java или аннотации @Transient в JPA)
@XmlType	Аннотирует класс как комплексный тип в схеме XML
@XMLValue	Позволяет преобразовать класс в простое содержимое или тип схемы

При использовании этих аннотаций можно преобразовать данные объекты в конкретные XML-схемы. Иногда вам нужна такая гибкость для работы со старыми веб-службами, как вы увидите в следующих главах. Ссылаясь на JPA, когда необходимо преобразовать сущности в ранее созданную базу данных, существует набор аннотаций, который позволяет удовлетворить потребности настройки отображения любого фрагмента таблицы (столбцы, таблицы, внешние ключи и т. д.). Для веб-служб принцип схожий: они описаны в файле WSDL, который указан в XML. Если служба устарела, ее файл WSDL нельзя изменить. Вместо этого следует использовать механизм преобразования его в объекты, именно поэтому JAXB часто применяется при работе с веб-службами.

ПРИМЕЧАНИЕ

В этом разделе я упоминал JPA несколько раз, потому что технологии JPA и JAXB в значительной степени полагаются на аннотации и используются для преобразования объектов в форму, пригодную для различных источников (базы данных или XML). С точки зрения архитектуры сущности должны использоваться только для преобразования данных в форму для базы данных, а JAXB классы — для преобразования данных в XML. Однако иногда вам может понадобиться иметь оба представления. Как вы увидите в главе 14, один класс можно аннотировать и @Entity, и @XmlRootElement.

Основные сведения о JSON

Нотация объектов JavaScript (JSON) представляет собой легкий формат обмена данными, то есть менее подробный и более удобочитаемый, чем XML. Он часто используется для сериализации и передачи структурированных данных через сетевое соединение между сервером и веб-приложением.

В качестве альтернативы XML JSON непосредственно используется в коде JavaScript на веб-страницах. Это основная причина прибегать к JSON, а не к другому представлению данных. В листинге 12.14 показано JSON-представление документа XML, содержащего информацию о заказе и приведенного раннее (см. листинг 12.1).

Листинг 12.14. Представление информации о заказе в формате JSON

```
{
  "order": {
    "id": "1234",
    "date": "05/06/2013",
    "customer": {
      "first_name": "Джеймс",
      "last_name": "Роррисон",
      "email": "j.rorri@me.com",
      "phoneNumber": "+44 1234 1234"
    },
    "content": {
      "order_line": [
        {
          "item": "H2G2",
          "quantity": "1",
          "unit_price": "23.5"
        },
        {
          "item": "Гарри Поттер",
          "quantity": "2",
          "unit_price": "34.99"
        }
      ]
    },
    "credit_card": {
      "number": "1357",
      "expiry_date": "10/13",
      "control_number": "234",
      "type": "Visa"
    }
  }
}
```

JSON-объекты можно сериализовать в JSON, который в конечном итоге будет иметь менее сложную структуру, чем XML. Заключив значение переменной в фигурные скобки, вы указываете, что ее значение является объектом. Внутри объекта мы можем объявить любое количество свойств, используя пары `"имя":"значение"`, разделенные запятыми. Чтобы получить доступ к информации, хранящейся в JSON, можно просто сослаться на объект и имя свойства.

Документы JSON. JSON является текстовым, независимым от языка форматом, который использует набор соглашений для представления простых структур данных. Многие языки реализовали API для анализа документов JSON. Структуры данных в JSON просты для человеческого понимания и очень похожи на структуры данных в Java. JSON может представлять четыре примитивных типа (число, строка, двоичное значение и `null`) и два структурированных (объекты и массивы). В табл. 12.9 перечислены соглашения JSON для представления данных.

Таблица 12.9. Терминология JSON

Терминология	Описание
Число	Число в JSON очень похоже на таковое в Java, за исключением того, что в JSON не могут применяться восьмеричные и шестнадцатеричные значения
Строка	Строка — это последовательность из нуля или более символов Unicode, ограниченная двойными кавычками, использующая для экранирования символ обратного слеша
Значение	Значение в JSON может быть представлено в одном из следующих форматов: строка в двойных кавычках, число, двоичное значение, объект или массив
Массив	Массив — это упорядоченный набор значений. Квадратные скобки ([,]) обозначают начало и конец массива. Значения массива разделены запятыми (,) и могут быть объектами
Объект	JSON и Java имеют одинаковое определение объекта. В JSON объект — это неупорядоченный набор пар «имя/значение». Фигурные скобки ({,}) обозначают начало и конец объекта. Пары «имя/значение» в JSON разделяются запятыми (,) и представляют атрибуты POJO в Java

Двоеточие в JSON используется в качестве разделителя имен, а запятая — в качестве разделителя значений. Корректные данные JSON представляют собой сериализованный объект или массив структур данных, которые могут быть вложенными. Например, объект JSON может содержать массив JSON. В примере, приведенном выше (см. листинг 12.14), `content` — это объект, который содержит массив `order_line`.

Обзор спецификаций JSON

JSON наследуется от объектных литералов JavaScript. JSON был представлен в качестве RFC 4627 в IETF (Internet Engineering Task Force — Инженерный совет Интернета) в 2006 году. IETF является открытым международным сообществом проектировщиков сетей, исследователей, операторов и продавцов, продвигающих эволюцию архитектуры Интернета.

Официальный тип содержимого JSON — `application/json` (см. главу 15 о вебслужбах RESTful), а расширение имен файлов JSON — `.json`. Хотя JSON в настоящее время не является рекомендацией W3C, многие спецификации W3C и API прямо или косвенно основаны на JSON или используют этот формат, например JSON-LD, JSONPath, JSONT и JSONiq. Кроме того, большое количество языков программирования внедрили API для анализа и генерации данных в формате JSON.

Java имеет несколько реализаций для обработки, преобразования, сериализации/десериализации или генерации данных JSON, например JSON-Lib, fastjson, Flexjson, Jettison, Jackson и т. д. (посетите сайт http://json.org, на котором перечислены несколько Java API для работы с JSON); каждый из них полезен для разных сценариев.

Чтобы предоставить стандартный Java API для обработки JSON, JSR 353 (Java API для обработки JSON) был отправлен в JCP в 2011 году и был выпущен с Java EE 7.

JSON-P

Java API для обработки JSON (JSR 353), известный как JSON-P, — это спецификация, которая позволяет обрабатывать JSON в Java. Обработка включает в себя механизмы для анализа, создания, преобразования и запроса данных JSON. JSON-P представляет собой стандарт построения объектов Java с JSON с использованием API, похожего на DOM для XML. В то же время он обеспечивает механизм производства и потребления JSON с помощью потоков, аналогично StAX для XML.

Хотя это не является строгим требованием в оригинальном JSR, некоторые реализации JSON-P могут также обеспечить связывание данных JSON с объектами Java и наоборот (но это будет указано в будущем JSR, которое можно было бы назвать JSON-B, где B означает binding («связывание»)). В табл. 12.10 перечислены важные пакеты JSON-P.

Таблица 12.10. Пакеты JSON-P

Пакет	Описание
javax.json	Предоставляет API для описания структуры данных JSON (например, класс JsonArray для массива JSON и класс JsonObject для объекта JSON), предоставляет точку входа для анализа, построения, чтения и записи объектов и массивов JSON с помощью потоков
javax.json.spi	Интерфейс провайдера служб (Service Provider Interface) служит для подключения реализаций JsonParser и JsonGenerator
javax.json.stream	Предоставляет потоковый API для анализа и генерации JSON

Пример реализации

Несколько обработчиков JSON реализовано в Java, открытый исходный код для реализации JSON-P (JSR 353) имеет JSON RI.

Обработка JSON

JSON-P предоставляет две различные модели программирования для обработки JSON-документов: API объектной модели, а также потоковый API. Как и в DOM API для XML, API объектной модели обеспечивает классы для модели JSON-объектов и массивов в древовидной структуре, которые представляют JSON-данные в памяти. Как и в случае DOM API, API объектной модели обеспечивает гибкую навигацию и запрашивает все дерево содержимого.

Потоковый API — это низкоуровневый API, предназначенный для эффективной обработки больших объемов данных JSON. Потоковый API очень похож на StAX API для XML. Это дает возможность потоку JSON не сохранять весь документ в памяти. Потоковый API предоставляет основанный на событиях анализатор потоковой модели, что позволяет пользователю обрабатывать или отменять события

анализатора, а также запрашивать следующее событие (вытягивать его). Генератор JSON также поможет вам записать JSON с помощью потоков.

JSR 353 имеет основной API `javax.json.Json`, который представляет собой класс для создания объектов, обрабатывающих JSON. Этот центральный API имеет методы создания объектов типа `JsonParser`, `JsonGenerator`, `JsonWriter`, `JsonReader`, `JsonArrayBuilder` и `JsonObjectBuilder`.

Построение JSON

Структуры объекта и массива в JSON представлены классами `javax.json.JsonObject` и `javax.json.JsonArray`. API позволяет перемещаться и запрашивать древовидную структуру данных.

Класс `JsonObject` имеет словарный вид для доступа к неупорядоченному набору из нуля или более пар «имя/значение». Аналогично `JSONArray` предоставляет списковый вид для доступа к упорядоченной последовательности из нуля или более значений. API использует строительные шаблоны для создания древовидного представления `JsonObject` и `JSONArray` с помощью интерфейсов `javax.json.JsonObjectBuilder` и `javax.json.JsonArrayBuilder`.

В листинге 12.15 показано, как построить заказ в JSON, приведенный выше (см. листинг 12.14). Как видите, класс `Json` используется для создания объектов типов `JsonObjectBuilder` и `JsonArrayBuilder`, которые в конечном итоге создадут объект типа `JsonObject` (с применением финального метода `build()`). `JsonObject` предоставляет словарный вид для преобразований пар «имя/значение» объекта JSON.

Листинг 12.15. Класс OrderJsonBuilder создает объект JSON, содержащий информацию о заказе

```
public class OrderJsonBuilder {
  public JsonObject buildPurchaseOrder() {

    return  Json.createObjectBuilder().add("order", Json.createObjectBuilder()
            .add("id", "1234")
            .add("date", "05/06/2013")
            .add("customer", Json.createObjectBuilder()
                .add("first_name", "Джеймс")
                .add("last_name", "Роррисон")
                .add("email", "j.rorri@me.com")
                .add("phoneNumber", "+44 1234 1234"))
            .add("content", Json.createObjectBuilder()
                .add("order_line", Json.createArrayBuilder()
                            .add(Json.createObjectBuilder()
                                    .add("item", "H2G2")
                                    .add("quantity", "1")
                                    .add("unit_price", "23.5"))
                            .add(Json.createObjectBuilder( )
                                    .add("item", "Гарри Поттер")
                                    .add("quantity", "2")
                                    .add("unit_price", "34.99"))))
            .add("credit_card", Json.createObjectBuilder( )
                    .add("number", "1357")
```

```
                    .add("expiry_date", "10/13")
                    .add("control_number", "234")
                    .add("type", "Visa"))).build();
    }
}
```

Объект типа JsonObject также можно создать из источника входных данных (например, InputStream или Reader), используя интерфейс javax.json.JsonReader. В следующем примере показано, как считывать и создавать объект типа JsonObject с помощью интерфейса JsonReader. JsonReader создается из файла order.json (см. листинг 12.14). Затем, чтобы получить доступ к объекту order, вызывается метод getJsonObject(), который возвращает JsonObject. Если объект не найден, возвращается значение null:

```
JsonReader reader = Json.createReader(new FileReader("order.json"));
JsonObject jsonObject = reader.readObject();
j onObject = jsonObject.getJsonObject("order");
```

JsonReader также предоставляет общий метод read() для чтения любого подтипа javax.json.JsonStructure (JsonObject и JSONArray). Использование метода JsonStructure.getValueType() возвращает объект типа ValueType (ARRAY, OBJECT, STRING, NUMBER, TRUE, FALSE, NULL), а затем вы можете считать значение. Метод toString() типа JsonStructure возвращает JSON представление объектной модели.

Аналогично JsonObject и JSONArray можно записать в выходной источник (например, OutputStream или Writer) с помощью класса javax.json.JsonWriter. Метод-строитель Json.createWriter() может создать объект типа JsonWriter для различных типов вывода.

Анализ JSON

Потоковый API (пакет javax.json.stream) облегчает анализ JSON с помощью потоков, обеспечивая последовательный доступ только для чтения. Он предоставляет интерфейс javax.json.stream.JsonParser для анализа документов JSON. Точкой входа является класс-фабрика javax.json.Json, который имеет метод createParser(), возвращающий объект типа javax.json.stream.JsonParser из указанного источника входного сигнала (например, Reader или InputStream). К примеру, анализатор JSON, предназначенный для анализа пустого объекта JSON, можно создать следующим образом:

```
StringReader reader = new StringReader("{}");
JsonParser parser = Json.createParser(reader);
```

Вы можете настроить анализатор, передав в его метод createParserFactory() свойство Map. Эта фабрика создает объект типа JsonParser, сконфигурированный специально для анализа ваших данных JSON:

```
StringReader reader = new StringReader("{}");
JsonParserFactory factory = Json.createParserFactory(properties);
JsonParser parser = factory.createParser(reader);
```

JsonParser основан на потоковой модели «активный синтаксический анализ». Это означает, что анализатор генерирует событие, когда достигнуто имя/значение

JSON или считаны начало/конец массива/объекта. В табл. 12.11 перечислены все события, вызываемые анализатором.

Таблица 12.11. События анализатора JSon

Пакет	Описание
START_OBJECT	Событие для начала объекта JSON (генерируется, когда считан символ «{»)
END_OBJECT	Событие для конца объекта JSON (генерируется, когда считан символ «}»)
START_ARRAY	Событие для начала массива JSON (генерируется, когда считан символ «[»)
END_ARRAY	Событие для конца массива JSON (генерируется, когда считан символ «]»)
KEY_NAME	Событие для имени в паре «имя(ключ)/значение» объекта JSON
VALUE_STRING	Событие для значения типа string
VALUE_NUMBER	Событие для значения типа number
VALUE_TRUE	Событие для значения true
VALUE_FALSE	Событие для значения false
VALUE_NULL	Событие для значения null

Класс в листинге 12.16 анализирует JSON, показанный выше (см. листинг 12.14) (сохраненный в файле `order.json`), для извлечения электронной почты клиента. Анализатор движется вперед, пока не встретит имя свойства `email`. Метод `next()` заставляет анализатор перейти в другое состояние. При этом будет возвращено следующее перечисление `javax.json.stream.JsonParser.Event` (см. табл. 12.11) для следующего состояния анализа. Когда анализатор достигает значения свойства `email`, он возвращает его.

Листинг 12.16. Класс OrderJsonParser, анализирующий представление информации о заказе в JSON

```
public class OrderJsonParser {

  public String parsePurchaseOrderAndReturnEmail() throws
FileNotFoundException {
    String email = null;

    JsonParser parser = Json.createParser(new FileReader("src/main/resources/
order.json"));
    while (parser.hasNext()) {
      JsonParser.Event event = parser.next();
      while (parser.hasNext() && !(event.equals(JsonParser.Event.KEY_NAME) &&
                                   parser.getString().matches("email"))) {
        event = parser.next();
      }

      if (event.equals(JsonParser.Event.KEY_NAME) && ➥
```

```
                        parser.getString().matches("email")) {
                parser.next();
                email = parser.getString();
              }
            }
            return email;
          }
        }
```

Используя метод `Event.equals()`, можно определить тип события и обработать JSON в зависимости от него. В то время как `JsonParser` создает поток JSON, вы можете использовать метод `getString()`, чтобы получить строку, представляющую каждое имя (ключ) и значение в зависимости от состояния анализатора. Имя возвращается, если событие имеет тип `KEY_NAME`, строковое значение возвращается, если тип события — `VALUE_STRING`, а числовое значение будет возвращено, когда событие имеет тип `VALUE_NUMBER`. В дополнение к методу `getString()`, который возвращает строковое значение, можно использовать и другие методы, такие как `getIntValue()`, `getLongValue()` и `getBigDecimalValue()` в зависимости от типа.

Если во время анализа встретился неправильный формат данных, анализатор сгенерирует исключение времени выполнения, например `javax.json.stream.JsonParsingException` или `java.lang.IllegalStateException`, в зависимости от источника проблемы.

Генерация JSON

API строителя JSON позволяет создать структуру дерева JSON в памяти. `JsonParser` анализирует объект JSON с помощью потоков, в то время как объект типа `javax.json.stream.JsonGenerator` позволяет записывать JSON в поток, используя одно событие за раз.

Класс в листинге 12.17 использует метод `createGenerator()` основной фабрики `javax.json.Json` для получения объекта типа `JsonGenerator` и генерирует JSON-документ, приведенный ранее (см. листинг 12.14). Генератор пишет пары «имя/значение» в объекты JSON и массивы JSON.

Листинг 12.17. Создание объектов, представляющих информацию о заказе, в JSON

```
public class OrderJsonGenerator {

  public StringWriter generatePurchaseOrder() throws IOException {
    StringWriter writer = new StringWriter();
    JsonGenerator generator = Json.createGenerator(writer);
    generator.writeStartObject()
             .write("id", "1234")
             .write("date", "05/06/2013")
             .writeStartObject("customer")
               .write("first_name", "Джеймс")
               .write("last_name", "Роррисон")
               .write("email", "j.rorri@me.com")
               .write("phoneNumber", "+44 1234 1234")
             .writeEnd()
```

```
            .writeStartArray("content")
              .writeStartObject()
                .write("item", "H2G2")
                .write("unit_price", "23.5")
                .write("quantity", "1")
              .writeEnd()
              .writeStartObject()
                .write("item", "Гарри Поттер")
                .write("unit_price", "34.99")
                .write("quantity", "2")
              .writeEnd()
            .writeEnd()
            .writeStartObject("credit_card")
              .write("number", "123412341234")
              .write("expiry_date", "10/13")
              .write("control_number", "234")
              .write("type", "Visa")
            .writeEnd()
          .writeEnd()
          .close();
    return writer;
  }
}
```

При генерации JSON необходимо быть знакомым с контекстом объектов и массивов. JSON-пары «имя/значение» могут быть записаны в объект, тогда как значения JSON — в виде массива. В то время как метод `writeStartObject()` записывает стартовый символ объекта JSON ({), метод `writeStartArray()` используется для записи стартового символа массива JSON ([). Каждый открытый контекст должен закрываться с помощью метода `writeEnd()`. После записи конца текущего контекста родительский контекст становится новым текущим.

Метод `writeStartObject()` используется, чтобы начать новый контекст объекта-потомка, а метод `writeStartArray()` начинает новый контекст массива-потомка. Оба метода можно использовать только в контексте массива или если контекст еще не начался, и оба метода возможно вызвать, только когда контекста нет. Контекст начинается, когда вызван один из этих методов.

Класс `JsonGenerator` предоставляет и другие методы, такие как `write()`, чтобы записать пару «имя/значение» JSON в текущий контекст объекта или значение в текущий контекст массива.

Хотя метод `flush()` можно использовать для записи любых буферизованных выходных данных, метод `close()` закрывает генератор и освобождает все связанные с ним ресурсы.

Все вместе

Собрав все понятия этой главы, напишем POJO `CreditCard` с использованием JAXB и JSON-P, чтобы получить XML- и JSON-представление кредитной карты. Для проверки обоих форматов мы будем писать тесты. Класс `CreditCardXMLTest` выпол-

няет маршалинг и демаршалинг объекта типа CreditCard, чтобы убедиться, что представление XML правильное. Класс CreditCardJSonTest проверяет, хорошо ли отформатирован также созданный JSON.

Написание класса CreditCard

Класс CreditCard в листинге 12.18 имеет JAXB-аннотацию @XmlRootElement, что позволяет выполнить его маршалинг в XML. Другая аннотация JAXB @XmlAccessorType с параметром XmlAccessType.FIELD указывает JAXB связывать атрибуты, а не геттеры.

Листинг 12.18. Класс CreditCard с аннотацией JAXB

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class CreditCard {
  @XmlAttribute
  private String number;
  @XmlElement(name = "expiry_date")
  private String expiryDate;
  @XmlElement(name = "control_number")
  private Integer controlNumber;
  private String type;
  //Конструкторы, методы работы со свойствами
}
```

У объекта CreditCard есть такие базовые поля, как номер кредитной карты, дата истечения срока (в формате MM/YY), тип кредитной карты (Visa, Master Card, American Express и т. д.), а также контрольное число. Некоторые из этих атрибутов имеют аннотацию @XmlAttribute для преобразования в атрибут XML. Аннотация @XmlElement используется для замены имени элемента XML.

Написание теста CreditCardXMLTest

Класс CreditCardXMLTest, показанный в листинге 12.19, выполняет маршалинг и демаршалинг объекта типа CreditCard с использованием механизма маршалинга JAXB. Метод ShouldMarshallACreditCard создает экземпляр класса CreditCard и проверяет, имеет ли он правильное представление XML. Метод shouldUnmarshallACreditCard делает все наоборот — он выполняет демаршалинг XML-документа в объект типа CreditCard и убеждается, что значения объекта установлены правильно.

Листинг 12.19. Тест CreditCardXMLTest выполняет маршалинг и демаршалинг XML

```
public class CreditCardXMLTest {
  public static final String creditCardXML =
         "<?xml version=\"1.0\" encoding=\"UTF-8\" standalone=\"yes\"?>\n" +
         "<creditCard number=\"12345678\">\n" +
         "<expiry_date>10/14</expiry_date>\n" +
         "<control_number>566</control_number>\n" +
         "<type>Visa</type>\n" +
         "</creditCard>";

  @Test
```

```
  public void shouldMarshallACreditCard() throws JAXBException {

    CreditCard creditCard = new CreditCard("12345678", "10/14", 566, "Visa");

    StringWriter writer = new StringWriter();
    JAXBContext context = JAXBContext.newInstance(CreditCard.class);
    Marshaller m = context.createMarshaller();
    m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
    m.marshal(creditCard, writer);

    System.out.println(writer);

    assertEquals(creditCardXML, writer.toString().trim());
  }

  @Test
  public void shouldUnmarshallACreditCard() throws JAXBException {
    StringReader reader = new StringReader(creditCardXML);
    JAXBContext context = JAXBContext.newInstance(CreditCard.class);
    Unmarshaller u = context.createUnmarshaller();
    CreditCard creditCard = (CreditCard) u.unmarshal(reader);

    assertEquals("12345678", creditCard.getNumber());
    assertEquals("10/14", creditCard.getExpiryDate());
    assertEquals((Object) 566, creditCard.getControlNumber());
    assertEquals("Visa", creditCard.getType());
  }
}
```

Написание теста CreditCardJSonTest

Класс `CreditCardJSonTest`, показанный в листинге 12.20, использует API `JsonGenerator` для написания представления объекта `CreditCard` в JSON. Затем он проверяет, имеет ли `JsonObject` корректный синтаксис, сравнивая его с константой `creditCardJSon`.

Листинг 12.20. Тест CreditCardJSonTest генерирует JSon

```
public class CreditCardJSonTest {
  public static final String creditCardJSon =
          "{\"creditCard\":" +
                  "{\"number\":\"12345678\"," +
                  "\"expiryDate\":\"10/14\"," +
                  "\"controlNumber\":566," +
                  "\"type\":\"Visa\"}" +
          "}";

  @Test
  public void shouldGenerateACreditCard(){
    CreditCard creditCard = new CreditCard("12345678", "10/14", 566, "Visa");

    StringWriter writer = new StringWriter();
    JsonGenerator generator = Json.createGenerator(writer);
```

```
        generator.writeStartObject()
                 .writeStartObject("creditCard")
                   .write("number", creditCard.getNumber())
                   .write("expiryDate", creditCard.getExpiryDate())
                   .write("controlNumber", creditCard.getControlNumber())
                   .write("type", creditCard.getType())
                 .writeEnd()
               .writeEnd()
               .close();

        assertEquals(creditCardJSon, writer.toString().trim());

    }
}
```

Резюме

XML — нечто большее, чем просто текстовый формат для описания документов. Это механизм для описания независимых от платформы комплексных структурированных данных. Java предоставляет набор мощных, легких API для анализа, проверки и генерации XML-данных. В Java поддерживаются различные модели анализа, такие как DOM, SAX и StAX. Хотя вы можете использовать низкоуровневые Java API для работы с XML на основе модели DOM или SAX, JAXP API предоставляет классы-оболочки для анализа ваших XML-ресурсов на основе DOM- или SAX-модели и передает XML-документ с помощью XSLT и XPath.

Архитектура Java для связывания XML (JAXB) определяет стандарт связывания Java-представлений с XML и наоборот. Это обеспечивает высокий уровень абстракции, так как архитектура основана на аннотациях. Притом что JAXB можно использовать в любых приложениях Java, он хорошо вписывается в пространство веб-служб, потому что любая информация, которой можно обмениваться, записана в XML, что вы увидите в главе 14.

JSON представляет собой легкий формат обмена данными. Он является альтернативой XML, и его рекомендуется использовать для более простых структур данных. JSON-P выполняет анализ и генерацию данных в формате JSON через потоки. Даже если эквивалента JAXB для JSON пока не существует, спецификации, такие как JAX-RS, используют JSON-P для возвращения JSON-объектов веб-служб RESTful (см. главу 15).

Глава 13
Обмен сообщениями

Большая часть коммуникаций между компонентами, которую вы видели до сих пор, является синхронной: один класс вызывает другой, управляемый компонент вызывает EJB, который вызывает сущность, и т. д. В таких случаях вызывающая сторона и ее адресат должны быть запущены и работать для обеспечения успешной коммуникации и вызывающей стороне нужно ждать, пока адресат завершит свою работу, чтобы продолжить свою. За исключением асинхронных вызовов в EJB (благодаря аннотации `@Asynchronous`), большинство компонентов Java EE используют синхронные вызовы (локально или удаленно). Когда мы говорим об обмене сообщениями, то имеем в виду слабосвязанную асинхронную коммуникацию между компонентами.

Промежуточное ПО, ориентированное на обработку сообщений (MOM), является программным обеспечением (поставщиком), предоставляющим возможность асинхронного обмена сообщениями между разнородными системами. Оно может быть буфером между системами, которые производят и потребляют сообщения в своем собственном темпе (например, одна система работает постоянно, а другая — только ночью). Они изначально слабо связаны, а производители не знают, кто находится на другом конце канала, потребляет сообщение и выполняет действия. Производитель и потребитель могут быть недоступны в одно и то же время, чтобы общаться друг с другом. На самом деле они даже не знают о существовании друг друга, поскольку используют промежуточный буфер. В связи с этим MOM полностью отличается от многих технологий, например RMI, которые требуют от приложения знать сигнатуру удаленного метода приложения.

Сегодня типичная организация имеет множество приложений, часто написанных на разных языках, которые выполняют четко определенные задачи. MOM основано на асинхронной модели взаимодействия, поэтому оно позволяет этим приложениям работать самостоятельно и в то же время являться частью информационного рабочего процесса. Обмен сообщениями — это хорошее решение для слабо связанной асинхронной интеграции существующих и новых приложений, при условии, что производитель и потребитель согласуют формат сообщений и промежуточный пункт назначения. Эта связь может быть локальной, внутри организации, или распределенной между несколькими внешними службами.

Основные сведения об обмене сообщениями

Промежуточное ПО, ориентированное на обработку сообщений (MOM), использует специальную лексику. Когда сообщение отправляется, программное обеспе-

чение, которое сохраняет сообщение и отправляет его, называется *поставщиком* (или иногда *брокером*). Отправитель сообщения — это *производитель*, а местоположение, где хранится сообщение, — *место назначения*. Компонент, принимающий сообщение, называется *потребителем*. Любой компонент, заинтересованный в сообщении, расположенном в заданном месте назначения, может потребить его. Эти понятия проиллюстрированы на рис. 13.1.

Рис. 13.1. Архитектура MOM

В Java EE API, который работает с этими понятиями, называется JMS (Java Message Service — служба сообщений Java). Он имеет набор интерфейсов и классы, которые позволяют подключаться к поставщику, создавать сообщение, отправлять и получать его. JMS не передает сообщения физически, это просто API, который требует наличия поставщика, отвечающего за обработку сообщений. При работе в контейнере EJB могут быть использованы MDB (Message-Driven Beans — управляемые сообщениями компоненты) для получения сообщений, чей обмен управляется контейнером.

На высоком уровне архитектура обмена сообщениями состоит из следующих компонентов (рис. 13.2).

- *Поставщик*. JMS — это только API, поэтому он нуждается в реализации способа обмена сообщениями, то есть в поставщике (также известном как брокер сообщений). Поставщик обрабатывает буферизацию и доставку сообщений.
- *Клиенты*. Клиентом является любое приложение Java или компонент, который производит или потребляет сообщение с помощью поставщика. «Клиент» — это общий термин для производителя, отправителя, издателя, потребителя, приемника и подписчика.
- *Сообщения*. Это объект, которые клиенты отправляют или получают от поставщика.
- *Администрируемые объекты*. Брокер сообщений должен предоставить клиенту администрируемые объекты (фабрики подключений и места назначения) с помощью поиска JNDI или внедрения (как вы увидите далее).

Поставщик сообщений позволяет выполнять асинхронную коммуникацию, предоставляя место назначения, где сообщения могут храниться, пока не будут доставлены клиенту (см. рис. 13.1). Существуют два типа мест назначения, каждый из которых применяется к конкретной модели обмена сообщениями.

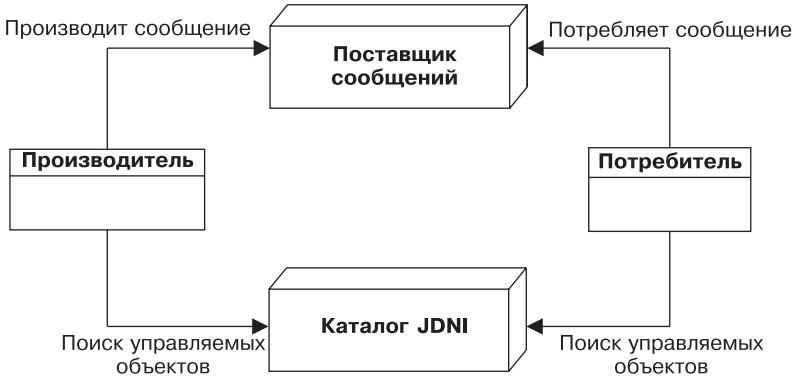


Рис. 13.2. Архитектура обмена сообщениями

- *От точки к точке (point-to-point, P2P)*. В этой модели место назначения, используемое для хранения сообщений, называется очередью. В этой модели один клиент помещает сообщение в очередь, а другой получает сообщение. Как только получение сообщения подтверждено, поставщик сообщений удаляет его из очереди.
- *Публикация-подписка (publish-subscribe, pub-sub)*. Место назначения называется темой. При использовании данной модели клиент публикует сообщение в теме и все абоненты этой темы получают сообщение.

От точки к точке

В модели P2P сообщение проходит от одного производителя к одному потребителю. Модель построена вокруг концепции очередей сообщений, отправителей и получателей (рис. 13.3). Очередь хранит отправленные сообщения, пока они не будут потреблены, отправитель и получатель не имеют временной зависимости друг от друга. Это означает, что отправитель может производить сообщения и отправлять их в очередь, когда ему удобно, и получатель может использовать их в то время, когда удобно ему. После создания получателя он получит все сообщения, которые были отправлены в очередь, даже те, которые были отправлены до его создания.

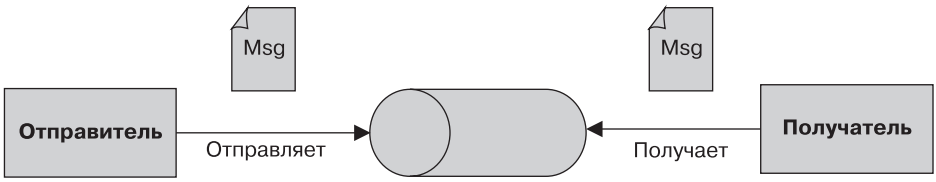


Рис. 13.3. Модель P2P

Каждое сообщение посылается в отдельную очередь, и получатель извлекает сообщение из очереди. Очереди сохраняют все отправленные сообщения до момента их потребления или истечения срока их существования.

Модель P2P используется, если существует только один получатель для каждого сообщения. Обратите внимание, что очередь может иметь несколько потребителей, но как только получатель потребляет сообщение из очереди, оно оттуда извлекается и никакой другой потребитель не может получить его. На рис. 13.4 можно увидеть, как один отправитель производит три сообщения.

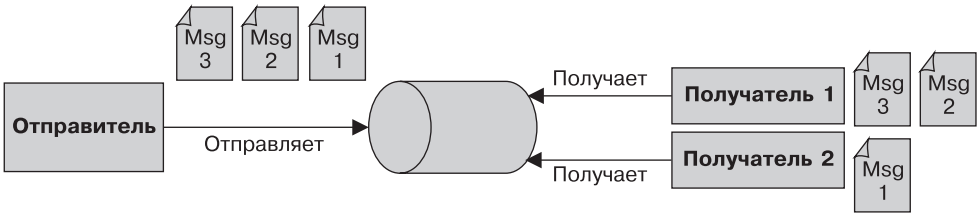


Рис. 13.4. Несколько получателей

Обратите внимание, что модель P2P не гарантирует доставку сообщений в определенном порядке (то есть порядок не определен). Поставщик может выбирать их по мере прибытия, или случайным образом, или каким-то другим способом.

Публикация-подписка

В модели публикации-подписки одно сообщение отправляется нескольким потенциальным потребителям одним производителем. Модель построена вокруг концепции тем, отправителей и подписчиков (рис. 13.5). Потребители называются *подписчиками*, так как им в первую очередь следует подписаться на тему. Поставщик управляет механизмом подписки/отписки, это происходит динамически.

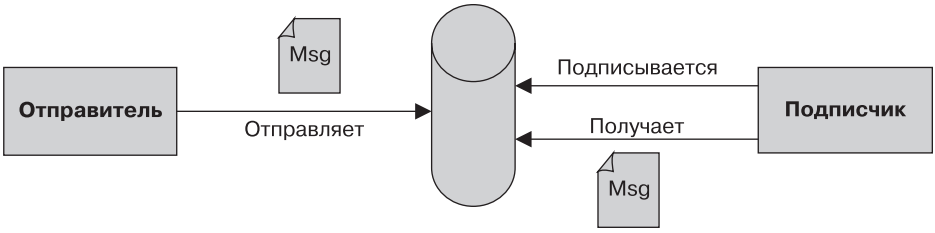


Рис. 13.5. Модель публикации/подписки

Тема сохраняет сообщения, пока они не распределятся между всеми абонентами. В отличие от модели P2P, временные зависимости между отправителями и подписчиками существуют — подписчики не получают сообщения, отправленные до того, как они подписались на тему, и если подписчик неактивен в течение определенного периода времени, он не получит прошлые сообщения, когда снова станет активным. Обратите внимание, что этого можно избежать, поскольку JMS API поддерживает концепцию стойких подписчиков, как вы увидите позже.

Несколько подписчиков могут потребить одно сообщение. Модель публикации/подписки может быть использована для широковещательных приложений, в которых одно сообщение доставляется нескольким потребителям. На рис. 13.6 отправляется три сообщения, которые получит каждый подписчик (в неопределенном порядке).

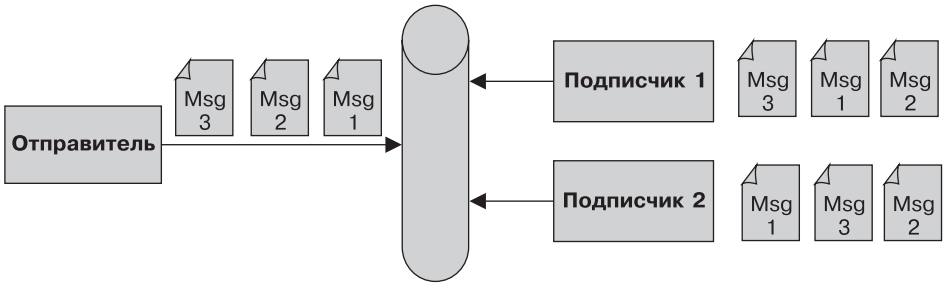


Рис. 13.6. Несколько подписчиков

Администрируемые объекты

Администрируемые объекты — это объекты, которые конфигурируются административно, а не программно. Поставщик сообщений настраивает эти объекты и делает их доступными в пространстве имен JNDI. Как и источники данных JDBC, администрируемые объекты создаются только один раз. Существует два типа администрируемых объектов:

- *фабрики соединений* — используются клиентами для создания подключения к пункту назначения;
- *места назначения* — точки распространения сообщений, которые получают, хранят и распространяют сообщения; места назначения могут быть очередями (P2P) или темами (pub-sub).

Клиенты получают доступ к этим объектам через портативные интерфейсы, выполнив их поиск в пространстве имен JNDI или посредством внедрения. В GlassFish есть несколько способов создания таких объектов, как вы увидите далее: с помощью консоли администрирования, командной строки с параметром `asadmin` или интерфейса REST. Начиная с версии JMS 2.0, вы даже можете использовать аннотации `@JMSConnectionFactoryDefinition` и `@JMSDestinationDefinition`, чтобы определить эти объекты программно.

Компоненты, управляемые сообщениями

Компоненты, управляемые сообщениями (Message-Driven Beans, MDB) являются асинхронными потребителями сообщений, которые выполняются внутри контейнера EJB. Как вы уже знаете из глав 7–9, контейнер EJB выполняет несколько задач (транзакции, безопасность, параллельный доступ, подтверждение сообщений и т. д.), а MDB фокусируются на потреблении сообщений. MDB не сохраняют свое состояние, что означает, что контейнер EJB может иметь множество их объектов,

которые выполняются параллельно, для обработки сообщений, приходящих от различных производителей. Даже если они выглядят как компоненты, не сохраняющие состояние, клиентские приложения не могут получить доступ непосредственно к MDB; единственный способ общения с MDB — отправить сообщение по месту назначения, которое слушают MDB.

В общем MDB слушают место назначения (очередь или тему) и при поступлении сообщения потребляют и обрабатывают его. Они также могут безопасно делегировать выполнение бизнес-логики другим компонентам, не сохраняющим состояние, с помощью транзакций. Они не сохраняют свое состояние, поэтому MDB не поддерживают свое состояние между отдельными вызовами при получении сообщений. MDB отвечают на сообщения, полученные от контейнера, в то время как сессионные компоненты, не сохраняющие состояние, отвечают на запросы клиентов через соответствующий интерфейс (локальный, удаленный или без интерфейса).

Обзор спецификаций обмена сообщениями

Обмен сообщениями в Java представлен в основном JMS, который может использоваться в приложениях, работающих в стандартном (Java SE) или производственном (Java EE) окружении. MDB предоставляют простой способ для сессионных EJB, не сохраняющих свое состояние, являться потребителями сообщений и быть связанными со спецификацией EJB.

Краткая история обмена сообщениями

Вплоть до конца 1980-х годов компании не имели простого способа связи для разных приложений. Разработчикам приходилось писать отдельные адаптеры для систем, чтобы преобразовать данные одного формата в другой, который смогла бы распознать система на месте назначения (и наоборот). Из-за несоответствия возможностей серверов и требований к ним были созданы буферы, позволившие ослабить связывание при обработке так, что общее время обмена сообщениями не увеличилось. Отсутствие однородных транспортных протоколов породило низкоуровневые адаптеры протоколов. К концу 1980-х годов стало появляться промежуточное программное обеспечение, которое решило этот вопрос интеграции. Первые MOM были созданы как отдельное программное обеспечение, которое может располагаться между приложениями и управлять «связыванием» систем. Они были в состоянии управлять разными платформами, языками программирования, сетевыми протоколами и аппаратными средствами.

Спецификация JMS была впервые опубликована в августе 1998 года. Она была создана основными поставщиками промежуточного программного обеспечения, чтобы организовать возможность обмена сообщениями в Java. В JSR 914 произошли незначительные изменения (JMS 1.0.1, 1.0.2 и 1.0.2b), что позволило наконец достичь версии 1.1 в апреле 2002 года. JMS 1.1 был интегрирован в J2EE 1.2 и с тех пор стал частью Java EE. Тем не менее JMS и MDB не являются частью спецификации Web Profile. Это означает, что они доступны только на серверах приложений, реализующих полную платформу Java EE 7.

Что нового в JMS 2.0

JMS 1.1 не изменялись в течение более чем десяти лет. Все API Java EE 5, за исключением JMS, медленно модернизировались, чтобы соответствовать изменениям языка (аннотации, обобщенные типы...). Настало время и для JMS следовать по тому же пути, использовать аннотации и упрощать API. Действительно, в JMS API были внесены некоторые изменения, чтобы сделать его проще в использовании:

- `Connection`, `Session` и другие объекты с методом `close()` теперь реализуют интерфейс `java.jang.AutoCloseable`, чтобы их можно было использовать в утверждении `try-with-resources` в Java SE 7;
- добавлен новый «упрощенный API», который предлагает более простую альтернативу стандартному и устаревшему API;
- внедрен новый метод `getBody`, что позволяет приложению извлечь тело непосредственно из объекта класса `Message` без необходимости преобразовать его к соответствующему подтипу;
- создан набор новых непроверенных исключений, которые наследуются от типа `JMSRuntimeException`;
- добавлены новые методы, позволяющие приложению отправлять сообщения асинхронно.

Что нового в EJB 3.2

В EJB 2.0 были введены MDB, которые были улучшены в EJB 3.0, следуя общей парадигме Java EE 5 «простота использования». Они не изменились внутренне, поскольку продолжали быть потребителями сообщений, но введение аннотаций и конфигурации с помощью исключений упростило их создание. Новая спецификация EJB 3.2 (JSR 345) привнесла некоторые изменения в MDB, добавив больше портативной конфигурации (подробнее об этом — позже).

Как вы узнали из главы 7, асинхронные вызовы в настоящее время возможны в рамках сессионных компонентов, не сохраняющих состояния (с помощью аннотации `@Asynchronous`). В предыдущих версиях Java EE нельзя было осуществить асинхронный вызов между EJB. Таким образом, единственным возможным способом было применять JMS и MDB — это дорогое решение, поскольку используется много ресурсов (место назначения JMS, соединения, фабрики и т. д.), только чтобы вызвать метод асинхронно. Сегодня асинхронные вызовы возможны между сессионными компонентами без необходимости использования MDB, позволяя им сосредоточиться на интеграции систем на основе обмена сообщениями.

Примеры реализации

Open Message Queue (OpenMQ) — это эталонная реализация JMS. Она имела открытый исходный код с 2006 года и может быть использована в автономных приложениях JMS или встроена в сервер приложений. OpenMQ является поставщиком сообщений по умолчанию для GlassFish. Она также добавляет много нестандартных функций, таких как универсальная служба обмена сообщениями (UMS), шаблонные места хранения тем, проверка сообщений XML, кластеризация и многое другое.

Java Messaging Service API

JMS является стандартным API Java, который позволяет приложениям создавать, отправлять, получать и читать сообщения асинхронно. Он определяет общий набор интерфейсов и классов, которые позволяют программам общаться с другими поставщиками сообщений. JMS аналогичен JDBC: последний соединяется с несколькими базами данных (Derby, MySQL, Oracle, DB2 и др.), а JMS — с несколькими поставщиками (OpenMQ, MQSeries, SonicMQ и т. д.).

JMS API развивался с момента своего создания. По историческим причинам JMS предлагает три альтернативных набора интерфейсов для производства и потребления сообщений. Эти очень разные интерфейсы развивались в версиях JMS 1.0, 1.1 и 2.0 и являются устаревшим, классическим и упрощенным API.

JMS 1.0 провел четкое различие между моделями «от точки к точке» и «публикация-подписка». Были определены два доменно-ориентированных API, один для модели «от точки к точке» (очереди) и другой — для модели «публикация-подписка» (темы). Вот почему вы можете найти, например, `QueueConnectionFactory` и `TopicConnectionFactory` API вместо стандартного `ConnectionFactory`. Обратите также внимание на разную лексику: потребитель вызывает получателя в модели P2P и подписчика в модели pub-sub.

JMS 1.1 API (называемый классическим API) предоставил единый набор интерфейсов, который можно использовать как для P2P, так и для модели pub-sub. В табл. 13.1 приведены общие имена интерфейсов (например, `Session`) и устаревшие имена для каждой модели (`QueueSession`, `TopicSession`).

Таблица 13.1. Интерфейсы в зависимости от версии JMS

Классический API	Упрощенный API	Устаревший API (P2P)	Устаревший API (pub-sub)
ConnectionFactory	ConnectionFactory	QueueConnection-Factory	TopicConnection-Factory
Connection	JMSContext	QueueConnection	TopicConnection
Session	JMSContext	QueueSession	TopicSession
Destination	Destination	Queue	Topic
Message	Message	Message	Message
MessageConsumer	JMSConsumer	QueueReceiver	TopicSubscriber
MessageProducer	JMSProducer	QueueSender	TopicPublisher
JMSException	JMSRuntime-Exception	JMSException	JMSException

Однако JMS 1.1 все еще был многословным и низкоуровневым API по сравнению с JPA или EJB. JMS 2.0 предоставил упрощенный API, который предлагает все функции классического API, но требует меньшего количества интерфейсов, а также проще в использовании. Выше (см. табл. 13.1) подчеркнуты различия между этими API (все они расположены в пакете `javax.jms`).

Я не буду обсуждать устаревшие API, но следует представить классический API, во-первых, потому, что вы увидите миллионы строк кода с использованием

классического API JMS 1.1, во-вторых, потому, что технически упрощенный API опирается на классический.

Классический API

Классический API JMS предоставляет классы и интерфейсы для приложений, требующих наличия системы обмена сообщениями (рис. 13.7). Он позволяет выполнять асинхронный обмен данными между клиентами, обеспечивая соединение с поставщиком и сеанс, в котором сообщения могут быть созданы, отправлены или получены. Эти сообщения могут содержать текст или другие объекты различного вида.

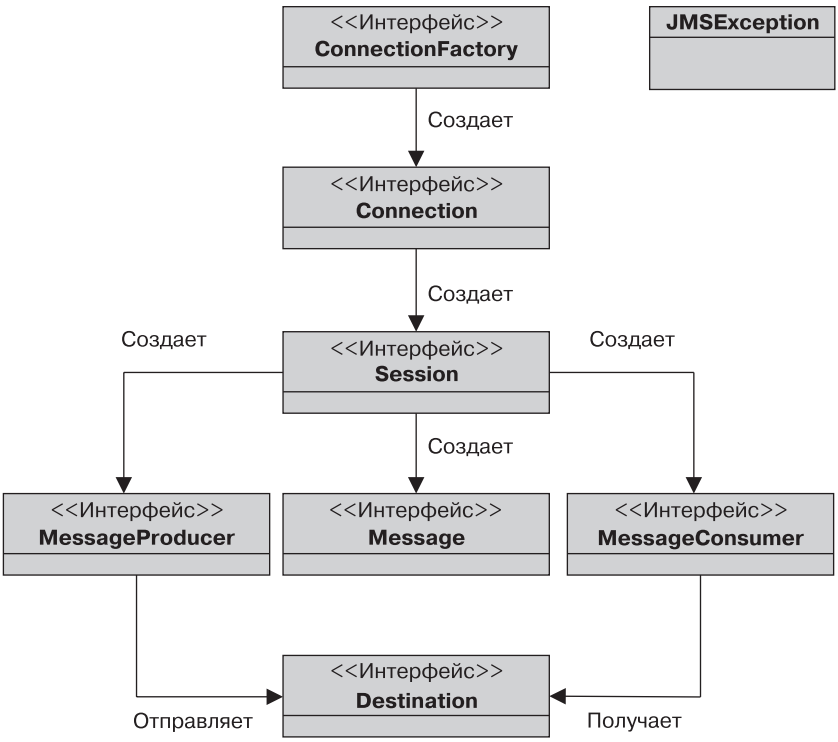


Рис. 13.7. Классический API JMS

ConnectionFactory

Фабрики соединений являются администрируемыми объектами, которые позволяют приложению соединяться с поставщиком путем программного создания объекта `Connection`. `javax.jms.ConnectionFactory` — это интерфейс, инкапсулирующий параметры конфигурации, которые были определены администратором.

Для использования администрируемых объектов, таких как `ConnectionFactory`, клиент должен выполнить поиск JNDI (или использовать внедрение). Например, следующий фрагмент кода получает объект JNDI `InitialContext` и использует его для поиска объекта типа `ConnectionFactory` под именем JNDI:

```
Context ctx = new InitialContext();
ConnectionFactory ConnectionFactory =
                 (ConnectionFactory) ctx.lookup("jms/javaee7/
ConnectionFactory");
```

Методы, доступные в этом интерфейсе (листинг 13.1), — `createConnection`, которые возвращают объект `Connection`, и `createContext`, появившиеся в JMS 2.0, возвращающие объект `JMSContext`. Вы можете создать объект типа `Connection` или `JMSContext` с помощью идентификатора пользователя по умолчанию или указания имени пользователя и пароля.

Листинг 13.1. Интерфейс ConnectionFactory

```
public interface ConnectionFactory {
  Connection createConnection() throws JMSException;
  Connection createConnection(String userName, String password) throws
JMSException;
  JMSContext createContext();
  JMSContext createContext(String userName, String password);
  JMSContext createContext(String userName, String password, int sessionMode);
  JMSContext createContext(int sessionMode);
}
```

Место назначения

Место назначения — это администрируемый объект, содержащий информацию о конфигурации конкретного поставщика, такую как адрес места назначения. Эта конфигурация скрыта от клиента JMS с помощью стандартного интерфейса `javax.jms.Destination`. Как и в случае с фабрикой соединений, для возврата таких объектов требуется выполнить поиск JNDI:

```
Context ctx = new InitialContext();
Destination queue = (Destination) ctx.lookup("jms/javaee7/Queue");
```

Соединение

Объект типа `Javax.jms.Connection`, который вы создаете, используя метод `createConnection()` фабрики соединений, инкапсулирует соединение с провайдером JMS. Подключения являются потокобезопасными и разработаны разделяемыми, поскольку открытие нового соединения требует значительного количества ресурсов. Тем не менее сеанс (`javax.jms.Session`) обеспечивает однопоточный контекст для отправки и получения сообщений, используя подключение для создания одного или нескольких сеансов. Если у вас есть фабрика соединений, вы можете использовать ее для создания соединения следующим образом:

```
Connection connection = connectionFactory.createConnection();
```

Перед тем как адресат сможет получить сообщения, он должен вызвать метод `start()`. Если вам нужно временно остановить получение сообщений, закрывать соединение не требуется — вы можете вызвать метод `stop()`:

```
connection.start();
connection.stop();
```

Когда приложение завершается, необходимо закрыть все созданные соединения. Закрытие соединения закрывает все его сеансы, а также производителей и потребителей:

```
connection.close();
```

Сеанс

Вы создаете сеанс из соединения с использованием метода `createSession()`. Сеанс предоставляет транзакционный контекст, в котором набор сообщений, которые будут отправлены или получены, сгруппирован в атомарную единицу работы. Это означает, что, если вы отправите несколько сообщений в течение одного сеанса, JMS гарантирует, что все они будут отправлены либо не будет отправлено ни одного. Такое поведение устанавливается при создании сеанса:

```
Session session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
```

Первый параметр метода определяет, будет ли сеанс поддерживать транзакции. В данном примере параметр имеет значение `true`, а это означает, что запрос на отправку сообщений не будет выполнен либо до вызова метода `commit()`, либо до закрытия сеанса. Если бы значение параметра было `false`, сеанс не был бы транзакционным — сообщения будут отправляться по вызову метода `send()`. Второй параметр означает, что сеанс автоматически подтверждает сообщения, если они были получены успешно. Сеанс имеет один поток и используется для создания сообщений, производителей и потребителей.

Сообщения

Для общения клиенты обмениваются сообщениями; производитель шлет сообщение к месту назначения, а потребитель получает его. Сообщения представляют собой объекты, которые инкапсулируют информацию и состоят из трех частей (рис. 13.8):

- *заголовок* — содержит стандартную информацию для идентификации и маршрутизации сообщений;
- *свойства* — пары «имя/значение», которые приложение может установить или считать; свойства также позволяют месту назначения фильтровать сообщения на основе их значений;
- *тело* — фактически содержит сообщение и может иметь один из нескольких форматов (текст, байты, объект и т. д.).

Заголовок

Заголовок имеет предопределенные пары «имя/значение», общие для всех сообщений, которые клиенты и поставщики используют для идентификации и маршрутизации сообщений. Они могут рассматриваться в качестве метаданных сообщения, поскольку дают информацию о нем. Каждое поле связано с геттером и сеттером, определенным в интерфейсе `javax.jms.Message`. Некоторые поля заголовка предусматривают задание значений клиентом, но многие из них устанавливаются автоматически с помощью методов `send()` и `publish()`. В табл. 13.2 описаны все поля заголовков сообщений JMS.

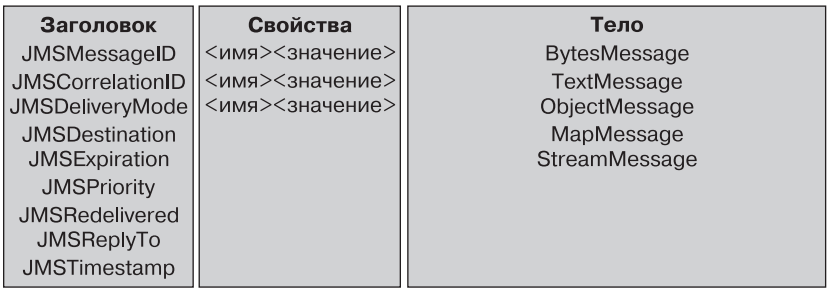


Рис. 13.8. Структура сообщения JMS

Таблица 13.2. Поля, содержащиеся в заголовке

Поле	Описание	Устанавливается
JMSDestination	Указывает место назначения, по которому было отправлено сообщение	Методами send() или publish()
JMSDeliveryMode	JMS поддерживает два режима доставки сообщений. PERSISTENT указывает поставщику убедиться, что сообщение не потерялось при передаче из-за ошибки. NON_ PERSISTENT — это режим с наименьшими накладными расходами, поскольку требует хранить сообщение в постоянном хранилище	Методами send() или publish()
JMSMessageID	Предоставляет значение, которое уникально идентифицирует каждое сообщение, отправленное поставщиком	Методами send() или publish()
JMSTimestamp	Содержит время, в которое сообщение поступило поставщику	Методами send() или publish()
JMSCorrelationID	Клиент может использовать его, чтобы связать одно сообщение с другим, например связать ответное сообщение с сообщением-запросом	Клиентом
JMSReplyTo	Содержит место назначения, по которому должно быть отправлено сообщение-ответ	Клиентом
JMSRedelivered	Это двоичное значение устанавливается поставщиком чтобы указать, что сообщение должно быть доставлено повторно	Поставщиком
JMSType	Это поле идентифицирует тип сообщения	Клиентом
JMSExpiration	Когда сообщение отправляется, время его существования рассчитывается и устанавливается на основе указанного времени существования в методе send()	Методами send() или publish()
JMSPriority	JMS имеет десятиуровневую шкалу приоритета, где 0 — минимальное значение, а 9 — максимальное	Методами send() или publish()

Свойства

Помимо полей заголовка, интерфейс `javax.jms.Message` поддерживает значения свойств, которые похожи на заголовки, но явно создаются конкретным приложением, вместо того чтобы быть стандартным для всех сообщений. Это обеспечивает механизм добавления дополнительных полей заголовка сообщения, которые клиент получит либо не получит с помощью селекторов. Значения свойств могут иметь тип `boolean`, `byte`, `short`, `int`, `long`, `float`, `double` и `String`. Код для установки и получения значения свойств выглядит следующим образом:

```
message.setFloatProperty("orderAmount", 1245.5f);
message.getFloatProperty("orderAmount");
```

Тело сообщения

Тело сообщения не является обязательным и включает данные, которые следует отправить или получить. В зависимости от используемого интерфейса оно может содержать различные форматы данных, как показано в табл. 13.3.

Таблица 13.3. Типы сообщений

Интерфейс	Описание
StreamMessage	Сообщение, тело которого содержит поток примитивных значений Java. Заполняется и считывается последовательно
MapMessage	Сообщение, тело которого содержит набор пар «имя/значение», чьи имена имеют тип String, а значения — примитивные значения Java
TextMessage	Сообщение, тело которого содержит строку (например, XML)
ObjectMessage	Сообщение, которое содержит сериализуемый объект или коллекцию сериализуемых объектов
BytesMessage	Сообщение, которое содержит поток байтов

Можно создать собственный формат сообщения, если вы создадите класс, наследующийся от интерфейса `javax.jms.Message`. Следует отметить, что при получении сообщения его тело доступно только для чтения. В зависимости от типа сообщения существуют различные методы, позволяющие получить доступ к его содержимому. Текстовое сообщение будет иметь методы `getText()` и `setText()`, сообщение-объект — методы `getObject()` и `setObject()` и т. д.:

```
textMessage.setText("This is a text message");
textMessage.getText();
bytesMessage.readByte();
objectMessage.getObject();
```

С появлением версии JMS 2.0 новый метод `<T>T getBody(Class<T> c)` возвращает тело сообщения как объект указанного типа.

Отправка и получение сообщений с помощью классического API

Рассмотрим простой пример, чтобы получить представление о том, как нужно использовать классический JMS API для отправки и получения сообщений. JMS

работает с производителями, потребителями и местами назначения. Производитель посылает сообщение по месту назначения, где потребитель ждет прибытия сообщения. Места назначения могут быть двух видов: очередь (для связи «от точки к точке») и тема (для связи «публикация-подписка»). В листинге 13.2 производитель посылает текстовое сообщение в очередь, которую слушает потребитель.

Листинг 13.2. Класс-производитель создает сообщение и помещает его в очередь с помощью классического API

```
public class Producer {
  public static void main(String[] args) {

      try {
       // Получает контекст JNDI
       Context jndiContext = new InitialContext();

       // Выполняет поиск администрируемых объектов
       ConnectionFactory connectionFactory = (ConnectionFactory) ➥
                         jndiContext.lookup("jms/javaee7/ConnectionFactory" );
       Destination queue = (Destination) jndiContext.lookup("jms/javaee7/Queue");

       //Создает необходимые артефакты для соединения с очередью
       Connection connection = connectionFactory.createConnection();
       Session session = connection.createSession(false, Session.AUTO_
ACKNOWLEDGE);
       MessageProducer producer = session.createProducer(queue);
       // Отправляет текстовое сообщение в очередь
      TextMessage message = session.createTextMessage("Сообщение отправлено " + new
Date());
       producer.send(message);

       connection.close();

     }  catch (NamingException | JMSException e) {
       e.printStackTrace();
     }
   }
}
```

Данный код представляет класс `Producer`, который имеет только метод `main()`. Первое, что происходит в этом методе, — создается контекст JNDI и затем используется для получения объектов типа `ConnectionFactory` и `Destination`.

Фабрики соединений и места назначения (очереди и темы) являются администрируемыми объектами и должны быть созданы и объявлены в поставщике сообщений (в нашем случае в OpenMQ GlassFish). Они оба имеют имя JNDI (например, очередь называется `jms/javaee7/Queue`); их следует искать в дереве JNDI.

Когда получены два администрируемых объекта, класс `Producer` использует класс `ConnectionFactory`, чтобы создать объект типа `Connection`, из которого будет получен объект типа `Session`. С помощью этого сеанса в очереди по месту назначения создаются объект типа `MessageProducer` и сообщение (`session.createProducer(queue)`).

Производитель посылает это сообщение (текстового типа). Обратите внимание, что класс `main` ловит исключение JNDI `NamingException`, а также `JMSException`.

К счастью, код получения сообщения выглядит примерно так же, как и код для его отправки. Действительно, первые строки класса `Consumer` в листинге 13.3 точно такие же: создается контекст JNDI, выполняется поиск фабрики соединений и места назначения, затем осуществляется подключение. Различия заключаются в том, что вместо `MessageProducer` используется `MessageConsumer` и приемник входит в бесконечный цикл, чтобы слушать очередь (вы позже увидите, что зацикливания можно избежать, если использовать более стандартный слушатель сообщений). Когда приходит сообщение, оно потребляется и отображается его содержимое.

Листинг 13.3. Класс-потребитель считывает сообщение из очереди с помощью классического API

```
public class Consumer {
  public static void main(String[] args) {
    try {
      // Получение контекста JNDI
      Context jndiContext = new InitialContext();

      // Выполняется поиск администрируемых объектов
      ConnectionFactory connectionFactory = (ConnectionFactory) ➥
                        jndiContext.lookup("jms/javaee7/ConnectionFactory");
      Destination queue = (Destination) jndiContext.lookup("jms/javaee7/Queue");

      // Создаются необходимые для подключения к очереди артефакты
      Connection connection = connectionFactory.createConnection();
      Session session = connection.createSession(false, Session.AUTO_
ACKNOWLEDGE);
      MessageConsumer consumer = session.createConsumer(queue);
      connection.start();

      // Выполняется цикл получения сообщений
      while (true) {
        TextMessage message = (TextMessage) consumer.receive();
        System.out.println("Сообщение получено: " + message.getText());
      }
    }  catch (NamingException | JMSException e) {
      e.printStackTrace();
    }
  }
}
```

Упрощенный API

Как видите, код, приведенный выше (см. листинги 13.2 и 13.3), довольно подробный и низкоуровневый. Вам нужно несколько артефактов, чтобы производить или потреблять сообщения (`ConnectionFactory`, `Connection`, `Session`...). Вдобавок вы должны обрабатывать исключение `JMSException`, которое является проверенным исключе-

нием (JMSException имеет несколько подклассов). Этот API был создан в JMS 1.1 в 2002 году и не изменялся до версии JMS 2.0.

JMS 2.0 вводит новый, упрощенный API, который состоит в основном из трех новых интерфейсов (JMSContext, JMSProducer и JMSConsumer). Они основаны на ConnectionFactory и других классических API, но не пользуются шаблонным кодом. Благодаря новому исключению JMSRuntimeException, которое является непроверенным, код отправления и получения сообщений стало гораздо проще писать и читать (взгляните на код следующих примеров).

На рис. 13.9 показана упрощенная диаграмма классов этого нового API. Обратите внимание, что устаревший, классический и упрощенный API находятся в пакете javax.jms.

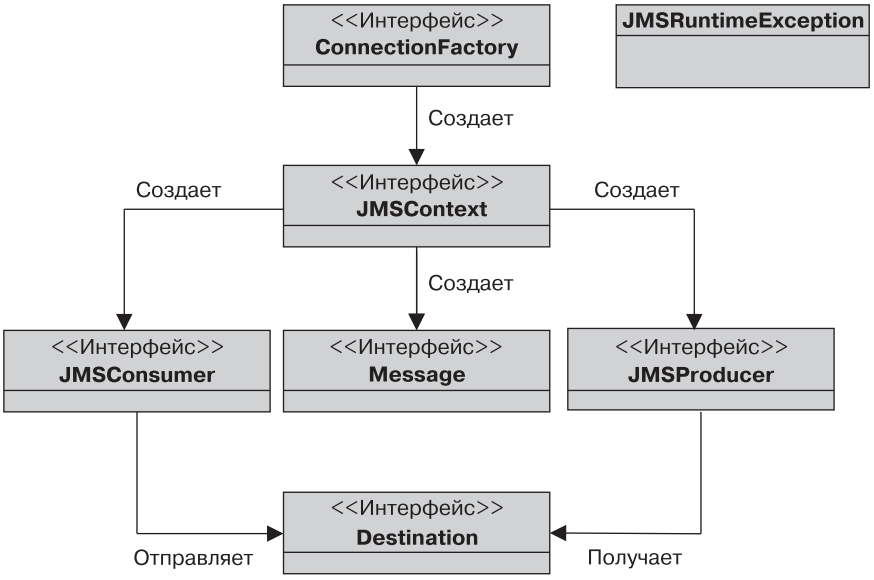


Рис. 13.9. Упрощенный JMS API

Упрощенный API предоставляет ту же функциональность для обмена сообщениями, что и классический, но требует использования меньшего количества интерфейсов и в целом проще в использовании. Его основные интерфейсы:

- JMSContext — активное подключение к поставщику JMS и однопоточный контекст для отправки и получения сообщений;
- JMSProducer — объект, созданный JMSContext, который используется для отправки сообщений в очередь или тему;
- JMSConsumer — объект, созданный JMSContext, который применяется для приема сообщений, отправленных в очередь или тему.

JMSContext

JMSContext — это основной интерфейс упрощенного JMS API, введенный в JMS 2.0. Он сочетает в себе функциональность двух отдельных объектов классического API

JMS 1.1: Connection (физическое соединение с поставщиком JMS) и Session (однопоточный контекст для отправки и получения сообщений).

JMSContext может быть создан приложением путем вызова одного из нескольких методов CreateContext фабрики соединений ConnectionFactory (см. листинг 13.1), а затем закрыт (то есть управляется приложением). Кроме того, если приложение работает в контейнере (EJB или Web), JMSContext может быть внедрен с помощью аннотации @Inject (управляется контейнером).

Когда приложению необходимо отправить сообщение, оно использует метод createProducer для создания JMSProducer, который предоставляет методы настройки и отправления сообщения. Сообщения могут быть отправлены либо синхронно, либо асинхронно. Для приема сообщений приложение может использовать один из нескольких методов createConsumer для создания объекта типа JMSConsumer. В табл. 13.4 приведено подмножество API JMSContext.

Таблица 13.4. Подмножество API JMSContext

Свойство	Описание
void start()	Начинает (или повторяет) отправку приходящих сообщений
void stop()	Временно приостанавливает доставку приходящих сообщений
void close()	Закрывает JMSContext
void commit()	Фиксирует все сообщения, отправляемые как транзакции, и высвобождает все блокировки
void rollback()	Откатывает все сообщения, отправляемые как транзакции, и высвобождает все блокировки
ByteMessage createByteMessage()	Создает объект типа ByteMessage
MapMessage create MapMessage()	Создает объект типа MapMessage
Message createMessage()	Создает объект типа Message
ObjectMessage create ObjectMessage()	Создает объект типа ObjectMessage
StreamMessage createStreamMessage()	Создает объект типа StreamMessage
TextMessage createTextMessage()	Создает объект типа TextMessage
TopicMessage createTopic(String topicName)	Создает объект типа Topic
Queue createQueue(String queueName)	Создает объект типа Queue
JMSConsumer create-Consumer(Destination destination)	Создает объект типа JMSConsumer для указанного места назначения
JMSConsumer createConsumer(Destination destination, String message-Selector)	Создает объект типа JMSConsumer для указанного места назначения с использованием селектора сообщений

Свойство	Описание
JMSProducer createProducer()	Создает объект типа JMSProducer, который может быть использован для конфигурации и отправки сообщений
JMSContext createContext(int sessionMode)	Создает объект типа JMSContext с указанным режимом сеанса

JMSProducer

`JMSProducer` используется для отправки сообщений от имени `JMSContext`. Он предоставляет различные методы для отправки сообщения по указанному месту назначения. Экземпляр `JMSProducer` создается путем вызова метода `JMSContext createProducer`. Он также предоставляет методы, позволяющие настроить параметры отправки, свойства и заголовки сообщения (см. рис. 13.8) перед его отправкой. В табл. 13.5 приведено подмножество `JMSProducer` API.

Таблица 13.5. Подмножество JMSProducer API

Свойство	Описание
get/set[Type]Property	Устанавливает и возвращает свойство сообщения, где [Type] — это тип свойства, который может быть Boolean, Byte, Double, Float, Int, Long, Object, Short, String
JMSProducer clearProperties()	Очищает все установленные свойства
Set<String> getPropertyNames()	Возвращает неизменяемое представление всех имен свойств, для которых были заданы значения
boolean propertyExists(String name)	Указывает, имеет ли свойство с заданным именем значение
get/set[Message Header]	Устанавливает и возвращает заголовок сообщения, где [Message Header] может иметь тип DeliveryDelay, DeliveryMode, JMSCorrelationID, JMSReplyTo, JMSType, Priority, TimeToLive
JMSProducer send(Destination destination, Message message)	Отправляет сообщение по указанному месту назначения, используя заданные параметры отправки, свойства и заголовки сообщения
JMSProducer send(Destination destination, String body)	Отправляет сообщение типа TextMessage с определенным телом по определенному месту назначения

JMSConsumer

`JMSConsumer` используется для получения сообщений из очереди или темы. Он создается с помощью одного из методов `JMSContext createConsumer`, которому в качестве параметра передается очередь или тема. Как вы увидите позже, `JMSConsumer` может быть создан с селектора сообщения, чтобы он мог ограничивать доставляемые сообщения.

Клиент может получать сообщения синхронно или асинхронно по мере их поступления. Для асинхронной доставки клиент может зарегистрировать с помощью `JMSConsumer` объект `MessageListener`. По мере поступления сообщений поставщик

доставляет их, вызывая метод OnMessage класса MessageListener. В табл. 13.6 приведено подмножество JMSConsumer API.

Таблица 13.6. Подмножество JMSConsumer API

Свойство	Описание
void close()	Закрывает JMSConsumer
Message receive()	Получает следующее созданное сообщение
Message receive(long timeout)	Получает следующее сообщение, которое поступит в заданном временном интервале
<T> T receiveBody(Class<T> c)	Получает следующее сообщение и возвращает его тело как объект указанного типа
Message receiveNoWait()	Получает следующее сообщение, если оно доступно прямо сейчас
void setMessageListener(MessageListener listener)	Устанавливает значение MessageListener
MessageListener getMessageListener()	Получает значение MessageListener
String getMessageSelector()	Получает значение селектора сообщений

Написание производителей сообщений

Новый упрощенный JMS API позволяет писать производителей и потребителей менее подробно, чем в случае классического API. Однако он все еще нуждается в обоих администрируемых объектах: ConnectionFactory и Destination. В зависимости от того, работаете вы за пределами или внутри контейнера (EJB , Web или ACC), вам придется использовать либо поиск JNDI, либо внедрение. Как вы видели ранее, API JMSContext является центральным API для производства и потребления сообщений. Если приложение работает за пределами контейнера, вам необходимо управлять жизненным циклом JMSContext (путем создания и закрытия его программно). Если вы работаете внутри контейнера, можете просто внедрить его и поручить контейнеру управлять его жизненным циклом.

Производство сообщения вне контейнера

Производитель сообщений (JMSProducer) является объектом, созданным JMSContext, и используется для отправки сообщений по месту назначения. Следующие шаги объясняют, как создать производителя, который посылает сообщение в очередь (листинг 13.4) вне какого-либо контейнера (в чистой среде Java SE).

1. Получить фабрики соединений и очереди с помощью поиска JNDI.
2. Создать объект JMSContext с помощью фабрики соединений (обратите внимание на утверждение try-with-resources, которое будет автоматически закрывать объект JMSContext).
3. Создать javax.jms.JMSProducer с помощью объекта JSMContext.
4. Отправить текстовое сообщение в очередь с помощью метода JMSProducer.send().

Листинг 13.4. Производитель класса выдал сообщение в очередь

```
public class Producer {
  public static void main(String[] args) {

    try {
      // Получение контекста JNDI
      Context jndiContext = new InitialContext();
         // Выполняется поиск администрируемых объектов
      ConnectionFactory connectionFactory = (ConnectionFactory) ➥
                        jndiContext.lookup("jms/javaee7/ConnectionFactory");
      Destination queue = (Destination) jndiContext.lookup("jms/javaee7/Queue");

      // Отправление текстового сообщения в очередь
      try (JMSContext context = connectionFactory.createContext()) {
        context.createProducer().send(queue, "Сообщение отправлено " + new
Date());
      }

    } catch (NamingException e ) {
      e.printStackTrace();
    }
  }
}
```

Если вы сравните данный код с тем, который использует классический API (см. листинг 13.2), вы заметите, что код стал менее подробным. Обработка исключений также стала аккуратнее, поскольку в новом API используется `JMSRuntimeException`, который является непроверенным исключением.

Производство сообщения внутри контейнера

Фабрики соединений и места назначения являются администрируемыми объектами, которые находятся в поставщике сообщений и должны быть объявлены в пространстве имен JNDI, поэтому вы используете JNDI API для их поиска. Когда код клиента выполняется внутри контейнера, вместо поиска может быть использовано внедрение зависимостей. Java EE 7 имеет несколько контейнеров: EJB, сервлет, и клиентский контейнер приложения (ACC). Если код работает в одном из этих контейнеров, аннотация `@Resource` может применяться для внедрения ссылки на этот ресурс в контейнере. С Java EE 7 использовать ресурсы намного проще, так как сложность ниже, чем у JNDI, а также не требуется настраивать ссылки на ресурсы в дескрипторах развертывания. Вы просто полагаетесь на возможности контейнера по внедрению.

В табл. 13.7 перечислены атрибуты, которые принадлежат к аннотации `@Resource`.

Таблица 13.7. API аннотации @javax.annotation.Resource

Элемент	Описание
name	Имя ресурса JNDI (зависит от реализации и не переносимо)
type	Тип ресурса в Java (например, javax.sql.DataSource или javax.jms.Topic)

Продолжение ⇩

Таблица 13.7 *(продолжение)*

Элемент	Описание
authentication-Type	Тип аутентификации, используемый в ресурсе (как для контейнера, так и для приложения)
shareable	Определяет, является ли ресурс разделяемым
mappedName	Характерное для продукта имя, которое ресурс должен преобразовывать
lookup	Имя ресурса в JNDI, с которым будет связан определяемый ресурс. Оно может связывать любой совместимый ресурс с использованием имен JNDI
description	Описание ресурса

Для использования аннотации `@Resource`, возьмем производителя из приведенного выше примера (см. листинг 13.4), изменим его компонент на тот, который не сохраняет свое состояние, и используем внедрение вместо поиска JNDI. В том примере (см. листинг 13.4) для обеих фабрик соединений и очереди выполняется поиск JNDI. В листинге 13.5 имя JNDI находится в аннотации `@Resource`. Когда `ProducerEJB` работает в контейнере, ссылки на `ConnectionFactory` и `Queue` внедряются при инициализации.

Листинг 13.5. ProducerEJB, запущенный внутри контейнера и использующий аннотацию @Resource

```
@Stateless
public class ProducerEJB {
  @Resource(lookup = "jms/javaee7/ConnectionFactory")
  private ConnectionFactory connectionFactory;
  @Resource(lookup = "jms/javaee7/Queue")
  private Queue queue;

  public void sendMessage() {

    try (JMSContext  context = connectionFactory.createContext()) {
      context.createProducer().send(queue, "Сообщение отправлено " + new Date());
    }
  }
}
```

Данный код проще, чем приведенный ранее (см. листинг 13.4), поскольку он не имеет дела с поиском JNDI или классом `NamingException`. Контейнер выполняет внедрение администрируемых объектов, как только инициализируется EJB.

Производство сообщений внутри контейнера с помощью CDI

Когда производитель выполняется в контейнере (EJB или контейнере сервлетов) с доступным CDI, он может внедрить `JMSContext`. Контейнер будет управлять его жизненным циклом (нет необходимости создавать или закрыть `JMSContext`). Это можно сделать благодаря аннотациям `@Inject` и `@JMSConnectionFactory`.

Аннотацию javax.jms.JMSConnectionFactory можно применить, чтобы указать имя ConnectionFactory для поиска JNDI, которое используется для создания JMSContext (листинг 13.6). Если аннотация JMSConnectionFactory опущена, по умолчанию задействуется стандартная фабрика соединений JMS.

Листинг 13.6. Управляемый компонент производит сообщение с помощью аннотации @Inject

```
public class Producer {
  @Inject
  @JMSConnectionFactory("jms/javaee7/ConnectionFactory")
  private  JMSContext context;
  @Resource(lookup = "jms/javaee7/Queue")
  private Queue queue;

  public void sendMessage() {
    context.createProducer().send(queue, "Сообщение отправлено " + new Date());
  }
}
```

Данный код довольно минималистичен. Контейнер выполняет всю работу по внедрению необходимых компонентов и управлению их жизненным циклом. Как разработчику вам нужна только одна строка кода, чтобы отправить сообщение.

Аннотация javax.jms.JMSPasswordCredential также может быть использована, чтобы указать имя пользователя и пароль, когда создан JMSContext:

```
@Inject
@JMSConnectionFactory("jms/connectionFactory")
@JMSPasswordCredential(userName="admin",password="mypassword")
private JMSContext context;
```

Написание потребителей сообщений

Клиент использует JMSConsumer, чтобы получать сообщения из места назначения. JMSConsumer создается путем передачи объекта типа Queue или Topic в метод createConsumer() класса JMSContext. Обмен сообщениями является асинхронным по своей сути, поэтому не существует временной зависимости между производителями и потребителями. Тем не менее сам клиент может получать сообщения двумя способами:

- *синхронно* — приемник явно получает сообщение от адресанта, вызвав метод receive();
- *асинхронно* — приемник решает зарегистрироваться на получение события, которое срабатывает всякий раз, когда появляется сообщение; он должен реализовать интерфейс MessageListener, и, когда приходит сообщение, поставщик доставляет его, вызвав метод onMessage().

На рис. 13.10 показаны эти два типа потребителя.

Синхронная доставка

Синхронный потребитель должен запустить JMSContext, работать в цикле, пока не придет новое сообщение, и запросить полученное сообщение с помощью одного из

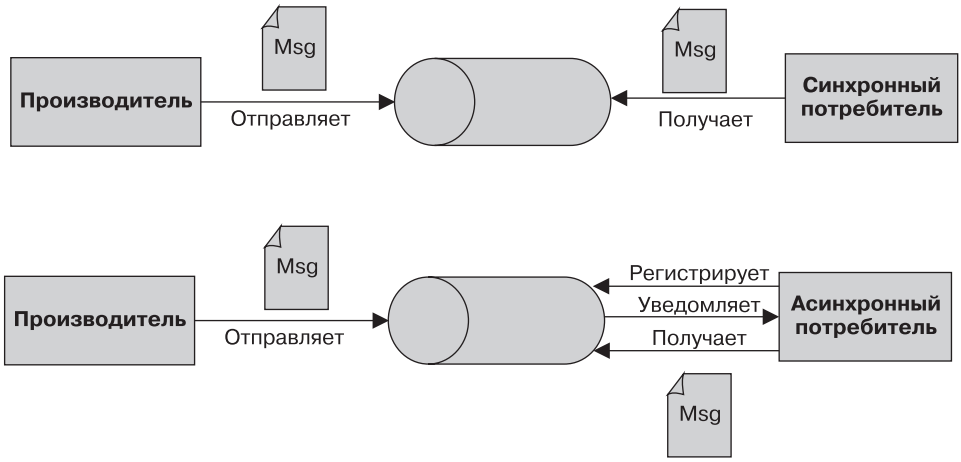


Рис. 13.10. Синхронные и асинхронные потребители

методов `receive()` (см. табл. 13.6). Есть несколько вариантов метода `receive()`, которые позволяют клиенту получить сообщение или подождать следующего. Описанные далее шаги объясняют, как создать синхронного потребителя, который потребляет сообщение из очереди (листинг 13.7).

1. Получить фабрику соединений и тему, используя поиск JNDI (или внедрение).
2. Создать объект `JMSContext` с помощью фабрики соединений.
3. Создать `javax.jms.JMSConsumer` посредством объекта `JSMContext`.
4. Запустить цикл и вызвать метод `receive()` (или в данном случае `receiveBody`) объекта-потребителя. Методы `receive()` будут заблокированы, если очередь пуста, и станут ждать прибытия сообщения. Здесь бесконечный цикл ждет появления других сообщений.

Листинг 13.7. Класс-потребитель потребляет сообщения в синхронном режиме

```
public class Consumer {
  public static void main(String[] args) {
    try {
      // Получение контекста JNDI
      Context jndiContext = new InitialContext();

      // Поиск администрируемых объектов
      ConnectionFactory connectionFactory = (ConnectionFactory) ➥
                        jndiContext.lookup("jms/javaee7/ConnectionFactory");
      Destination queue = (Destination) jndiContext.lookup("jms/javaee7/Queue");

      // Цикл получения сообщений
      try (JMSContext context = connectionFactory.createContext()) {
        while (true) {
          String message = context.createConsumer(queue).receiveBody
(String.class);
        }
```

```
        }

      }  catch (NamingException e) {
        e.printStackTrace();
      }
    }
  }
```

Опять же, сравнив этот код с приведенным ранее (см. листинг 13.3), вы увидите, насколько новый API проще в использовании и более выразителен.

ПРИМЕЧАНИЕ

Как и в случае производителей, которые могут использовать внедрение с помощью аннотаций @Resource, @Inject или @JMSConnectionFactory при выполнении внутри контейнера (см. листинги 13.5 и 13.6), потребители могут пользоваться теми же функциональными возможностями. Здесь я просто показываю, как потребитель может получить сообщение в чистой среде Java SE, но вы можете сами придумать, как упростить код, чтобы он работал внутри контейнера и использовал внедрение.

Асинхронная доставка

Асинхронное потребление основано на обработке событий. Клиент может зарегистрировать объект (включая себя), который реализует интерфейс `MessageListener`. *Слушатель сообщений* — это объект, который выступает в качестве асинхронного обработчика события для сообщений. По мере поступления сообщений поставщик доставляет их слушателю, вызвав метод `onMessage()`, который принимает один аргумент типа `Message`. С помощью этой модели событий потребителю не нужно запускать бесконечный цикл для получения сообщения. Данную модель событий использует MDB (подробнее об этом — позже).

Следующие шаги описывают процесс создания асинхронного слушателя сообщений (листинг 13.8).

1. Реализация классом интерфейса `javax.jms.MessageListener`, который определяет один метод — `onMessage()`.
2. Получение фабрики соединений и темы с использованием поиска JNDI (или внедрения).
3. Создание `javax.jms.JMSConsumer` с помощью объекта `JSMContext`.
4. Вызов метода `setMessageListener()` с передачей ему экземпляра интерфейса `MessageListener` (в листинге 13.8 сам класс `Listener` реализует интерфейс `MessageListener`).
5. Реализация метода `onMessage()` и обработка полученных сообщений. Каждый раз, когда приходит сообщение, поставщик будет вызывать этот метод, передавая сообщение.

Листинг 13.8. Потребитель является слушателем сообщений

```
public class Listener implements MessageListener {
  public static void main(String[] args) {

    try {
```

```
      // Получение контекста JNDI
       Context jndiContext = new InitialContext();

       // Поиск администрируемых объектов
       ConnectionFactory connectionFactory = (ConnectionFactory) ➥
                         jndiContext.lookup("jms/javaee7/ConnectionFactory");
       Destination queue = (Destination) jndiContext.lookup("jms/javaee7/
Queue");

       try (JMSContext context = connectionFactory.createContext()) {
         context.createConsumer(queue).setMessageListener(new Listener());
       }

     }  catch (NamingException e) {
       e.printStackTrace();
     }
   }

   public void onMessage(Message message) {
     System.out.println("Асинхронное сообщение получено: " + ➥
                        message.getBody(String.class));
   }
}
```

Механизмы надежности

Вы уже знаете, как подключаться к провайдеру, создавать различные типы сообщений, отправлять их в очереди или темы и получать их. Однако что, если вы сильно зависите от JMS и вам необходимо обеспечить надежность или другие дополнительные функции? JMS определяет несколько уровней надежности, которые обеспечивают доставку ваших сообщений, даже если поставщик выйдет из строя, или будет слишком загружен, или если место назначения уже заполнено сообщениями, срок жизни которых должен был истечь? Механизмы достижения надежной доставки сообщений следующие:

- *фильтрация сообщений* — используя селекторы, вы можете фильтровать сообщения, которые хотите получать;
- *настройка времени жизни сообщений* — установите время окончания срока жизни сообщений так, что они не будут доставлены, если устарели;
- *определение стойкости сообщения* — укажите, что сообщения не пропадают в случае сбоя поставщика;
- *управление подтверждением* — задайте различные уровни подтверждения сообщений;
- *создание стойких подписчиков* — убедитесь, что сообщения доставлены недоступному абоненту в модели pub-sub;
- *определение приоритетов* — установите приоритет для доставки сообщений.

Фильтрация сообщений

Некоторые приложения для обмена сообщениями нуждаются в фильтрации получаемых сообщений. Когда сообщение рассылается множеству клиентов, полезно задать параметры так, что оно будет получено только определенными адресатами. Это снижает затраты времени и пропускной способности, которые поставщик в противном случае потратит зря, доставляя клиентам ненужные сообщения.

Вы видели, что сообщения состоят из трех частей: заголовка, свойств и тела (см. рис. 13.8). Заголовок содержит фиксированное количество полей (метаданные сообщения), а свойства представляют собой набор пользовательских пар «имя/значение», значения которых может устанавливать приложение. Выборка может выполняться по этим двум областям. Производители устанавливают одно или несколько значений свойств или полей заголовка, а потребитель указывает критерии отбора сообщений с использованием выражений-селекторов. Доставляются только сообщения, которые соответствуют селектору. Селекторы сообщений делегируют работу по фильтрации сообщений поставщику JMS, а не приложению.

Селектор сообщений является строкой, которая содержит выражение. Синтаксис выражения основан на подмножестве условного синтаксиса выражений SQL92 и выглядит следующим образом:

```
context.createConsumer(queue, "JMSPriority < 6").receive();
context.createConsumer(queue, "JMSPriority < 6 AND orderAmount < 200").
receive();
context.createConsumer(queue, "orderAmount BETWEEN 1000 AND 2000" ).receive();
```

В предыдущем фрагменте кода потребитель создается с помощью метода `JMSContext.createConsumer()`, получив в качестве параметра строку-селектор. Она может использовать поля заголовка (`JMSPriority < 6`) или пользовательские свойства (`orderAmount < 200`). Производитель устанавливает значения этих свойств сообщения следующим образом:

```
context.createTextMessage().setIntProperty("orderAmount", 1530);
context.createTextMessage().setJMSPriority(5);
```

Выражение селектора может использовать логические операторы (`NOT`, `AND`, `OR`), операторы сравнения (=, >, >=, <, <=, <>), арифметические операторы (+, -, *, /), выражения (`[NOT] BETWEEN`, `[NOT] IN`, `[NOT] LIKE`, `IS [NOT] NULL`) и т. д.

Настройка параметров времени существования сообщений

При большой нагрузке можно указать время существования сообщений, чтобы убедиться, что поставщик удалит их из места назначения по мере устаревания. Это делается либо с помощью API `JMSProducer`, либо путем установки поля заголовка `JMSExpiration`. `JMSProducer` включает в себя метод `setTimeToLive()`, который принимает количество миллисекунд:

```
context.createProducer().setTimeToLive(1000).send(queue, message);
```

Задание стойкости сообщения

JMS поддерживает два режима доставки сообщений: стойкую и нестойкую. *Стойкая* доставка гарантирует, что сообщение доставляется потребителю только один раз, в случае *нестойкой* сообщение может быть и не доставлено. Стойкая доставка (используемая по умолчанию) надежнее, но это стоит производительности, так как она предотвращает потери сообщения, если произошел сбой поставщика. Режим доставки можно указать с помощью метода `setDeliveryMode()` интерфейса `JMSProducer`:

```
context.createProducer().setDeliveryMode(DeliveryMode.NON_PERSISTENT).
send(queue, message);
```

Управление подтверждением

До сих пор сценарии, которые мы рассматривали, предполагали, что сообщения отправляются и принимаются без подтверждения. Однако иногда вам может понадобиться, чтобы приемник подтверждал полученные сообщения (рис. 13.11). Фазу подтверждения может инициировать либо поставщик JMS, либо клиент, в зависимости от режима подтверждения.

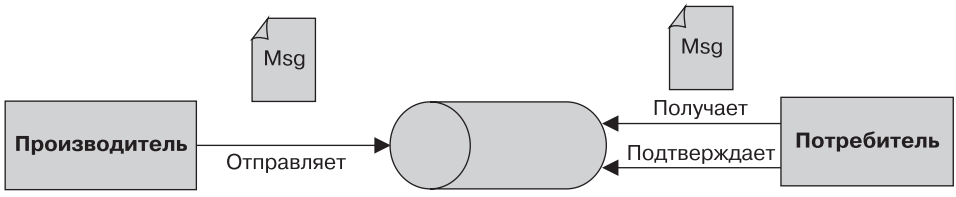


Рис. 13.11. Потребитель подтверждает сообщение

В транзакционных сессиях подтверждение происходит автоматически при успешном завершении транзакции. Если транзакция откатывается, все потребленные сообщения доставляются повторно. В нетранзакционных же сеансах должен быть указан режим подтверждения.

- `AUTO_ACKNOWLEDGE` — сеанс автоматически подтверждает получение сообщения.
- `CLIENT_ACKNOWLEDGE` — клиент подтверждает сообщение явным вызовом метода `Message.acknowledge()`.
- `DUPS_OK_ACKNOWLEDGE` — этот параметр указывает сеансу неспешно подтвердить доставку сообщения. Это может привести к многократной доставке некоторых сообщений, если происходит сбой поставщика JMS, поэтому его следует использовать потребителям, которые могут допускать дублирование сообщений. Если сообщение было доставлено повторно, поставщик устанавливает значение поля заголовка `JMSRedelivered` в `true`.

Следующий код использует аннотацию `@JMSSessionMode` для установки режима подтверждения `JMSContext` у производителя. Потребитель явно подтверждает сообщение, вызвав метод `acknowledge()`:

```
// Производитель
@Inject
```

```
@JMSConnectionFactory("jms/connectionFactory")
@JMSSessionMode(JMSContext.AUTO_ACKNOWLEDGE)
private JMSContext context;
...
context.createProducer().send(queue, message);

  // Потребитель
message.acknowledge();
```

Создание стойких потребителей

Недостатком использования модели pub-sub является то, что потребитель сообщения должен быть запущен, когда сообщение отправляется в тему, — в противном случае он его не получит. С помощью стойких потребителей JMS API предоставляет способ сохранить сообщения в теме, пока все подписчики не получат их. Используя такой способ подписки, потребитель может находиться в автономном режиме в течение некоторого времени, но, когда он подключится, он получит сообщения, которые прибыли за время его отсутствия. Чтобы достичь этого, клиент создает стойкого потребителя с использованием `JMSContext`:

```
context.createDurableConsumer(topic, "javaee7DurableSubscription").receive();
```

В этот момент клиентская программа запускает подключение и принимает сообщения. Название `javaee7DurableSubscription` используется в качестве идентификатора стойкой подписки. Каждый стойкий потребитель должен иметь уникальный идентификатор, в результате чего происходит объявление уникальной фабрики соединений для каждого потенциального стойкого потребителя.

Определение приоритетов

Вы можете использовать уровни приоритета, чтобы указать поставщику JMS доставлять срочные сообщения в первую очередь. JMS определяет десять значений приоритета, начиная с `0` (самый низкий) и заканчивая `9` (самый высокий). Вы можете задать значение приоритета с помощью метода `setPriority()` `JMSProducer`:

```
context.createProducer().setPriority(2).send(queue, message);
```

Большинство этих методов возвращают `JMSProducer`. Это позволяет объединять вызовы методов, что дает возможность использовать «текучий» стиль программирования. Например:

```
context.createProducer().setPriority(2)
                        .setTimeToLive(1000)
                        .setDeliveryMode(DeliveryMode.NON_PERSISTENT)
                        .send(queue, message);
```

Написание компонентов, управляемых сообщениями

До сих пор в этой главе демонстрировалось, как асинхронный обмен сообщениями обеспечивает слабую связанность и повышенную гибкость между системами

с использованием JMS API. Компоненты, управляемые сообщениями (MDB), обеспечивают стандарт этой модели асинхронных сообщений для промышленных приложений, работающий в контейнере EJB.

MDB — это асинхронный потребитель, который вызывается контейнером в результате прихода сообщений. Для поставщика сообщений MDB являются простым потребителем сообщений, скрытым за местом назначения, которое они слушают.

MDB — часть спецификации EJB, и их модель близка к модели сессионных компонентов, поскольку они не имеют состояния и работают внутри контейнера EJB. Контейнер слушает место назначения и делегирует вызов MDB по мере прибытия сообщений. Как и любой другой EJB, MDB могут получить доступ к ресурсам, управляемым контейнером (другие EJB, соединения JDBC, ресурсы JMS, диспетчер объектов и т. д.).

Зачем использовать MDB, когда можно задействовать отдельные клиенты JMS, как мы уже видели ранее? Это следует делать, так как контейнер управляет многопоточностью, безопасностью и операциями, что существенно упрощает код вашего потребителя JMS. Он также управляет входящими сообщениями для нескольких экземпляров MDB (доступных в хранилище), которые сами по себе не имеют специального кода для работы с многопоточностью. Как только новое сообщение достигнет места назначения, экземпляр MDB извлекается из хранилища для обработки сообщения. Простой потребитель MDB показан в листинге 13.9.

Листинг 13.9. Простой потребитель MDB

```
@MessageDriven(mappedName = "jms/javaee7/Topic")
public class BillingMDB implements  MessageListener {

  public void onMessage(Message message) {
    System.out.println("Сообщение получено: " + message.getBody(String.
class));
  }
}
```

Данный код (обработка исключений опущена для простоты) показывает, что использование MDB освобождает программиста от всех механических аспектов обработки типов сообщений, которые объяснялись ранее. MDB реализуют интерфейс `MessageListener` и метод `onMessage()`, при этом не нужен никакой другой код для подключения к провайдеру или начала потребления сообщений. MDB также полагаются на механизм конфигурации с помощью исключений, и необходимо только несколько аннотаций, чтобы он работал (см. аннотацию `@MessageDriven`).

Структура MDB

MDB отличаются от сессионных компонентов, поскольку не реализуют локальные или удаленные бизнес-интерфейсы. Вместо этого они реализуют интерфейс `javax.jms.MessageListener`. Клиенты не могут вызывать непосредственно методы MDB, однако, как и в случае сессионных компонентов, MDB имеют богатую модель программирования, которая включает в себя жизненный цикл, аннотации функций

обратного вызова, перехватчики, внедрение и транзакции. Использование этой модели предоставляет приложению больше функциональности.

Важно знать, что MDB не являются частью модели EJB Lite, что означает, что они не могут быть развернуты в простом веб-профиле сервера приложений, а нуждаются в полном стеке технологий Java EE.

Требования к разработке класса MDB:

- класс должен иметь аннотацию `@javax.ejb.MessageDriven` или ее эквивалент в дескрипторе развертывания XML;
- класс должен реализовывать, прямо или косвенно, интерфейс `MessageListener`;
- класс должен быть определен как общедоступный и не должен быть окончательным или абстрактным;
- класс должен иметь общедоступный конструктор без аргументов, который контейнер будет использовать для создания экземпляров MDB;
- класс не должен реализовывать метод `finalize()`.

Классу MDB разрешено выполнять иные методы, вызывать другие ресурсы и т. д. MDB развертывается в контейнере и может дополнительно поставляться с файлом `ejb-jar.xml`. Следуя модели «простота использования» Java EE 7, MDB может быть всего лишь аннотированным POJO, устраняя необходимость выполнять большую часть конфигурации. Однако если вам все еще нужно настроить конфигурацию JMS, можете использовать элементы с аннотациями `@MessageDriven` и `@ActivationConfigProperty` (XML или эквивалент).

@MessageDriven

MDB являются одним из самых простых видов EJB с точки зрения разработки, так как они поддерживают наименьшее количество аннотаций. Аннотация `@MessageDriven` (или эквивалентный XML) обязательна, потому что является частью метаданных, которые требуются, чтобы контейнер распознал, что класс Java на самом деле MDB.

API аннотации `@MessageDriven`, показанный в листинге 13.10, очень прост, и все элементы являются необязательными.

Листинг 13.10. API аннотации @MessageDriven

```
@Target(TYPE) @Retention(RUNTIME)
public @interface MessageDriven {
  String name() default "";
  Class messageListenerInterface default Object.class;
  ActivationConfigProperty[] activationConfig() default {};
  String mappedName();
  String description();
}
```

Элемент `name` указывает имя MDB (по умолчанию это название класса). `messageListenerInterface` определяет, какой слушатель сообщений реализует MDB (если MDB реализует несколько интерфейсов, то он сообщает контейнеру EJB, какой из них является интерфейсом `MessageListener`). Элемент `mappedName` — это имя

JNDI места назначения, которое должны слушать MDB. `Description` — строка, которая используется, чтобы дать описание MDB после развертывания. Элемент `activationConfig` задействуется для задания свойств конфигурации и принимает массив аннотаций `@ActivationConfigProperty`.

@ActivationConfigProperty

JMS позволяет настроить определенные свойства, такие как селекторы сообщений, режим подтверждения, стойкие подписчики и т. д. В MDB эти свойства можно установить с помощью аннотации `@ActivationConfigProperty`. Эта необязательная аннотация может быть предоставлена как один из параметров аннотации `@MessageDriven`, и, по сравнению с эквивалентом JMS, аннотация `@ActivationConfigProperty` очень проста — она состоит из пары «имя/значение» (листинг 13.11).

Листинг 13.11. API аннотации ActivationConfigProperty

```
@Target({}) @Retention(RUNTIME)
public @interface ActivationConfigProperty {
  String propertyName();
  String propertyValue();
}
```

Свойство `activationConfig` позволяет вам предоставить стандартную и нестандартную (зависящую от поставщика) конфигурацию. Код в листинге 13.12 устанавливает флаг режима подтверждения и селектор сообщения.

Листинг 13.12. Установка значений свойств MDB

```
@MessageDriven(mappedName = "jms/javaee7/Topic", activationConfig = {
  @ActivationConfigProperty(propertyName = "acknowledgeMode",
                            propertyValue = "Auto-acknowledge"),
  @ActivationConfigProperty(propertyName = "messageSelector",
                            propertyValue = "orderAmount < 3000")
})
public class BillingMDB implements MessageListener {
  public void onMessage(Message message) {
    System.out.println("Сообщение получено: " +
                       message.getBody(String.class));
  }
}
```

Каждое свойство активизации — это пара «имя/значение», которую понимает лежащий в основе поставщик сообщений и использует для настройки MDB. В табл. 13.8 перечислены некоторые стандартные свойства.

Таблица 13.8. Свойства активизации OpenMQ

Свойство	Описание
acknowledgeMode	Режим подтверждения (по умолчанию AUTO_ACKNOWLEDGE)
messageSelector	Селектор сообщений, используемый MDB
destinationType	Тип места назначения, TOPIC или QUEUE
destinationLookup	Имя административно определенной очереди или темы

Свойство	Описание
connectionFactory-Lookup	Имя административно определенной очереди или темы фабрики соединений
destination	Имя места назначения
subscriptionDurability	Стойкость подписки (по умолчанию NON_DURABLE)
subscriptionName	Имя подписки потребителя
shareSubscriptions	Используется, если управляемый сообщениями компонент развернут и может быть разделенным
clientId	Идентификатор клиента, который будет использован при соединении с поставщиком JMS

Внедрение зависимостей

Как и все EJB, описанные ранее (см. главу 7), MDB могут использовать внедрение зависимостей, чтобы получить ссылки на ресурсы, такие как источники данных JDBC, EJB или другие объекты. *Внедрение* — это средство, с помощью которого контейнер вставляет зависимости автоматически после создания объекта. Эти ресурсы должны быть доступны в контейнере или среде контекста, поэтому следующий код можно написать в MDB:

```
@PersistenceContext
private EntityManager em;
@Inject
private InvoiceBean invoice;
@Resource(lookup = "jms/javaee7/ConnectionFactory")
private ConnectionFactory connectionFactory;
```

Контекст MDB также может быть внедрен с помощью аннотации `@Resource`:

```
@Resource private MessageDrivenContext context;
```

Контекст MDB

Интерфейс `MessageDrivenContext` обеспечивает доступ к среде выполнения контекста, которую контейнер предоставляет объекту MDB. Контейнер передает этому объекту интерфейс `MessageDrivenContext`, с которым объект остается связанным на протяжении всего срока службы MDB. Это дает возможность MDB явно откатывать транзакции, получать принципал пользователя и т. д. Интерфейс `MessageDrivenContext` расширяет интерфейс `javax.ejb.EJBContext` без добавления каких-либо дополнительных методов.

Если MDB внедряют ссылку на их контекст, они будут иметь возможность ссылаться на методы, перечисленные в табл. 13.9.

Таблица 13.9. Методы интерфейса MessageDrivenContext

Метод	Описание
getCallerPrincipal	Возвращает java.security.Principal, связанный с вызовом
getRollbackOnly	Проверяет, помечена ли для отката текущая транзакция

Продолжение ⇩

Таблица 13.9 *(продолжение)*

Метод	Описание
getTimerService	Возвращает интерфейс javax.ejb.TimerService
getUserTransaction	Возвращает интерфейс javax.transaction.UserTransaction, который будет использован для разграничения транзакций. Только MDB, транзакции которых управляются компонентами (BMT), могут применять этот метод
isCallerInRole	Проверяет, имеет ли вызывающая сторона заданную роль безопасности
Lookup	Позволяет MDB выполнить поиск по записям среды контекста именования JNDI
setRollbackOnly	Позволяет объекту пометить текущую транзакцию для отката. Только MDB с BMT могут использовать этот метод

Жизненный цикл и аннотации функций обратного вызова

Цикл жизни MDB (рис. 13.12) идентичен жизненному циклу сессионного компонента: либо MDB существует и готов потреблять сообщения, либо не существует. Перед выходом контейнер сначала создает экземпляр MDB и, если это возможно, внедряет необходимые ресурсы, как указано в аннотациях метаданных (`@Resource`, `@Inject`, `@EJB` и т. д.) или дескрипторах развертывания. Затем контейнер вызывает метод обратного вызова компонента `@PostConstruct`, если таковой имеется. После этого MDB находится в состоянии готовности и ожидает, чтобы потребить входящие сообщения. Метод обратного вызова `@PreDestroy` вызывается, когда MDB удаляется из хранилища или уничтожается.

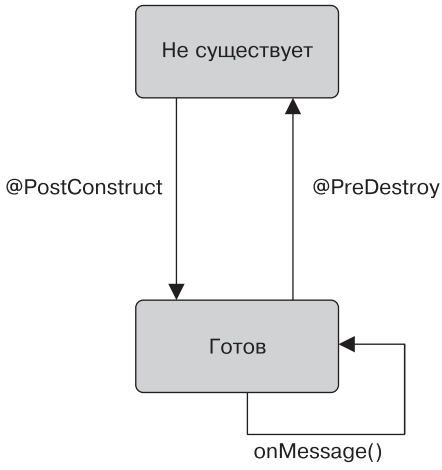


Рис. 13.12. Жизненный цикл MDB

Такое поведение идентично поведению сессионных компонентов, не сохраняющих состояние (для получения более подробной информации о методах обратного вызова см. главу 8), и, как и для других компонентов EJB, вы можете добавить перехватчики с помощью аннотации `@javax.ejb.AroundInvoke`.

MDB как потребитель

Как было показано в разделе «Написание потребителей сообщений» ранее в этой главе, потребители могут получить сообщение синхронно, запустив цикл и ожидая поступления сообщения, или асинхронно путем реализации интерфейса `MessageListener`. По своей природе MDB рассчитан на то, чтобы быть асинхронным потребителем сообщений. MDB реализует интерфейс слушателя сообщений, который по приходе сообщения вызывается контейнером.

Может ли MDB быть синхронным потребителем? Да, но это не рекомендуется. Синхронные потребители сообщений блокируют и связывают ресурсы сервера (EJB застрянут в цикле, не выполняя никакой работы, и контейнер не сможет освободить их). MDB, как и сессионные компоненты, не сохраняющие состояния, живут в хранилище определенного размера. Когда контейнеру нужен объект, он выбирает один из хранилища и использует его. Если каждый экземпляр уйдет в бесконечный цикл, хранилище в итоге опустеет и все доступные экземпляры будут заняты в цикле. Контейнер EJB также может начать производить больше объектов MDB, что увеличит хранилище и съест больше памяти. По этой причине сессионные компоненты и MDB не должны использоваться в качестве синхронных потребителей сообщений. В табл. 13.10 показаны разные режимы для MDB и сессионных компонентов.

Таблица 13.10. Сравнение MDB с сеансовыми компонентами

Компонент	Производитель	Синхронный потребитель	Асинхронный потребитель
Сессионный компонент	Да	Не рекомендуется	Невозможно
MDB	Да	Не рекомендуется	Да

MDB как производитель сообщений

MDB способны стать производителями сообщений, что часто происходит, когда они участвуют в рабочем процессе, так как они получают сообщение от одного адресата, обрабатывают его и отправляют в другое место назначения. Чтобы использовать эту возможность, следует применить JMS API.

Место назначения и фабрику соединений можно внедрить с помощью аннотаций `@Resource` и `@JMSConnectionFactory` или через поиск JNDI. Затем могут быть вызваны методы объекта `javax.jms.JMSContext` для создания и отправки сообщения. Код `BillingMDB` (листинг 13.13) слушает тему (`jms/javaee7/Topic`), получает сообщения (метод `onMessage()`) и посылает новое сообщение в очередь (`jms/javaee7/Queue`).

Листинг 13.13. MDB, потребляющий и производящий сообщения

```
@MessageDriven(mappedName = "jms/javaee7/Topic", activationConfig = {
        @ActivationConfigProperty(propertyName  = "acknowledgeMode",
                                  propertyValue = "Auto-acknowledge"),
        @ActivationConfigProperty(propertyName  = "messageSelector",
                                  propertyValue = "orderAmount BETWEEN 3 AND 7")
} )
public class BillingMDB implements MessageListener {

  @Inject
  @JMSConnectionFactory("jms/javaee7/ConnectionFactory")
  @JMSSessionMode(JMSContext.AUTO_ACKNOWLEDGE)
  private JMSContext context;
  @Resource(lookup = "jms/javaee7/Queue")
  private Destination printingQueue;

  public void onMessage(Message message) {
    System.out.println("Сообщение получено: " +
                       message.getBody(String.class));
    sendPrintingMessage();
  }

  private void sendPrintingMessage() throws JMSException {
    context.createProducer().send(printingQueue, "Сообщение ,было получено
и снова отправлено");
  }
}
```

Этот MDB использует большинство понятий, изученных к данному моменту. Во-первых, он применяет аннотацию `@MessageDriven` для определения имени JNDI темы, которую слушает (`mappedName = "jms/javaee7/Topic"`). В этой же аннотации он определяет набор свойств, таких как режим подтверждения и селектор сообщений с использованием массива аннотаций `@ActivationConfigProperty`, и реализует интерфейс `MessageListener` и его метод `onMessage()`.

Этот MDB также должен производить сообщения. Таким образом, в него внедряются два необходимых администрируемых объекта: фабрика соединений (с использованием `JMSContext`) и место назначения (очередь с именем `jms/javaee7/Queue`). Наконец, бизнес-метод, который посылает сообщения (метод `sendPrintingMessage()`), выглядит так, как вы видели ранее: создается объект типа `JMSProducer` и используется для создания и отправки текстового сообщения. Для лучшей читаемости обработка исключений была опущена во всем классе.

Транзакции

MDB — это разновидность EJB (для получения дополнительной информации см. главы 7 и 8). MDB могут использовать BMT или управляемые контейнером транзакции (CMT), они могут явно откатывать транзакции с помощью метода `MessageDrivenContext.setRollbackOnly()` и т. д. Однако у MDB есть некоторые особенности, на которых следует остановиться.

Говоря о транзакциях, мы всегда думаем о реляционных базах данных. Однако другие ресурсы, такие как системы обмена сообщениями, также могут быть транзакционными. Если две или более операции должны завершиться успехом или неудачей вместе, они образуют транзакцию (см. главу 9). Если рассматривать обмен сообщениями, то в случае отправки двух или более они должны быть отправлены успешно (фиксация) или неудачно (откат) вместе. Как это работает на практике? Ответ заключается в том, что сообщения не отправляются потребителям, пока транзакция не завершена. Контейнер начнет транзакцию перед вызовом метода `onMessage()` и закончит транзакцию по его возвращении (кроме случаев, когда транзакция помечена для отката с помощью метода `setRollbackOnly()`).

Хотя MDB являются транзакционными, они не могут выполняться в контексте транзакции клиента, так как клиента не имеют. Никто явно не вызывает методы MDB, они просто слушают место назначения и получают сообщения.

Контекст не передается от клиента к MDB, и аналогично транзакционный контекст клиента не может быть передан в методе `onMessage()`. В табл. 13.11 сравниваются сессионные компоненты, CMT и MDB.

Таблица 13.11. Транзакции MDB в сравнении с транзакциями сеансовых компонентов

Атрибут транзакции	Сессионный компонент	MDB
NOT_SUPPORTED	Да	Да
REQUIRED	Да	Да
MANDATORY	Да	Нет
REQUIRES_NEW	Да	Нет
SUPPORTS	Да	Нет
NEVER	Да	Нет

В CMT MDB могут использовать аннотацию `@javax.ejb.TransactionAttribute` на бизнес-методах со следующими двумя атрибутами:

- `REQUIRED` (по умолчанию) — если MDB вызывает другие компоненты, контейнер вместе с вызовом передает контекст транзакции; контейнер пытается зафиксировать транзакцию, когда завершается метод слушателя сообщений;
- `NOT_SUPPORTED` — если MDB вызывает другие компоненты, контейнер не передает контекст транзакции вместе с вызовом.

Обработка исключений

Во фрагментах кода, приведенных в этой главе, обработка исключений была опущена, поскольку в JMS API она может быть слишком объемной. Классический API определяет двенадцать различных исключений, которые наследуются от типа `javax.jms.JMSException`. В упрощенном API определены десять исключений времени выполнения, которые наследуются от `javax.jms.JMSRuntimeException`.

Важно отметить, что `JMSException` является проверенным исключением (см. главу 9 (обсуждение исключений приложения в разделе «Обработка исключений»)),

а JMSRuntimeException — непроверенным. Спецификация EJB описывает два типа исключений:

- *исключения приложения* — проверенные исключения, которые наследуются от класса Exception и не вызывают отката контейнера;
- *системные исключения* — непроверенные исключения, которые наследуются от класса RuntimeException и вызывают откат контейнера.

Генерация исключения JMSRuntimeException вызовет откат контейнера, а генерация JMSException — нет. Если выполнить откат необходимо, должен быть явно вызван метод setRollBackOnly() или сгенерировано системное исключение (например, EJBException):

```
public void onMessage(Message message) {
  try {
    System.out.println("Сообщение получено: " + message.getBody(String.
class));
  } catch (JMSException e) {
    context.setRollBackOnly();
  }
}
```

Все вместе

В этой главе мы рассмотрели основные концепции обмена сообщениями (модели P2P и pub-sub) и администрируемые объекты (фабрики соединений и места назначения), узнали, как подключиться к поставщику, производить и потреблять сообщения, использовать некоторые механизмы надежности, а также применять управляемые контейнером компоненты (MDB) для прослушивания мест назначения. Теперь посмотрим, как эти понятия работают вместе, на примере, который мы скомпилируем и упакуем с помощью Maven и развернем на GlassFish.

Будем использовать отдельный класс (OrderProducer), который посылает сообщения в тему (под названием jms/javaee7/Topic). Эти сообщения являются объектами, представляющими заказ клиента на покупку книг и компакт-дисков. Заказ (OrderDTO) имеет несколько атрибутов, в том числе и общую сумму заказа. Потребители, которые слушают тему, имеют типы OrderConsumer и ExpensiveOrderMDB (рис. 13.13). OrderConsumer получает любые заказы, но MDB потребляет только те, которые имеют общую стоимость более $1000 (это делается с помощью селектора).

Maven необходимо структурировать код, основанный на окончательных артефактах упаковки, поэтому ExpensiveOrderMDB и OrderDTO будут упакованы в один файл .jar (chapter13-MDB-1.0.jar), а затем развернуты на GlassFish. OrderProducer, OrderConsumer и опять же OrderDTO будут работать в среде Java SE.

Написание класса OrderDTO

Объект, который будет отправлен в сообщении JMS, является POJO, и для него необходимо реализовать интерфейс Serializable. Класс OrderDTO, показанный в листинге 13.14, дает некоторую информацию о заказе, в том числе об общей сумме.

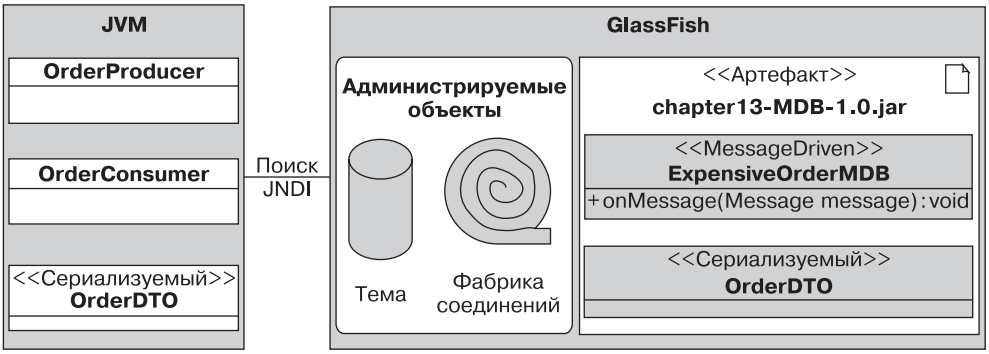


Рис. 13.13. Собираем все воедино

Он является объектом, который будет установлен в JMS `ObjectMessage` и отправлен из `OrderProducer` в тему, а затем потреблен `OrderConsumer` и `ExpensiveOrderMDB`.

Листинг 13.14. OrderDTO передается в JMS ObjectMessage

```
public class OrderDTO implements Serializable {
  private Long orderId;
  private Date creationDate;
  private String customerName;
  private Float totalAmount;
  // Конструкторы, методы работы со свойствами
}
```

Написание класса OrderProducer

`OrderProducer`, показанный в листинге 13.15, представляет собой автономный клиент, который использует упрощенный JMS API для отправки `ObjectMessage` в тему `jms/javaee7/Topic`. Он выполняет поиск необходимых фабрик соединений и мест назначения, и в методе `main()` создает экземпляр класса `OrderDTO`. Обратите внимание, что параметр `totalAmount` заказа — это аргумент, передаваемый классу (`args[0]`). `JSMProducer` устанавливает свойство сообщения `orderAmount`, чтобы позволить выполнять выборку сообщений далее, и заказ затем отправляется в тему.

Листинг 13.15. OrderProducer посылает заказ в OrderDTO

```
public class OrderProducer {
  public static void main(String[] args) throws NamingException {
    // Создает объект orderDto с параметром totalAmount
    Float totalAmount = Float.valueOf(args[0]);
    OrderDTO order = new OrderDTO(12341, new Date(), "Бетти Морей", totalAmount);

    // Получает контекст JNDI
    Context jndiContext = new InitialContext();

    // Поиск администрируемых объектов
    ConnectionFactory connectionFactory = (ConnectionFactory) ➥
                    jndiContext.lookup("jms/javaee7/ConnectionFactory");
```

```
    Destination topic = (Destination) jndiContext.lookup("jms/javaee7/Topic");

    try (JMSContext jmsContext = connectionFactory.createContext()) {
     // Посылает сообщение в тему
     jmsContext.createProducer().setProperty("orderAmount", totalAmount).
send(topic, order);
    }
  }
}
```

Написание OrderConsumer

OrderConsumer, показанный в листинге 13.16, также является отдельным клиентом JMS, который слушает тему jms/javaee7/Topic и использует JMSConsumer API, чтобы получить все OrderDTO (без селекторов).

Листинг 13.16. OrderConsumer потребляет все сообщения OrderDTO

```
public class OrderConsumer {
  public static void main(String[] args) throws NamingException {

    // Получение контекста JNDI
    Context jndiContext = new InitialContext();

    // Выполняется поиск администрируемых объектов
    ConnectionFactory connectionFactory = (ConnectionFactory) ➥
                    jndiContext.lookup("jms/javaee7/ConnectionFactory");
    Destination topic = (Destination) jndiContext.lookup("jms/javaee7/Topic");

    // Цикл получения сообщений
    Try (JMSContext jmsContext = connectionFactory.createContext()) {
      while (true) {
        OrderDTO order = jmsContext.createConsumer(topic).receiveBody(OrderDTO.
class);
        System.out.println("Заказ получен: " + order);
      }
    }
  }
}
```

Написание класса ExpensiveOrderMDB

Класс ExpensiveOrderMDB (листинг 13.17) является MDB с аннотацией @MessageDriven, который слушает место назначения jms/javaee7/Topic. Этот MDB интересуют только заказы, сумма которых больше $1000, что достигается с помощью селектора сообщений (orderAmount > 1000). По прибытии сообщения метод onMessage() потребляет его, приводит к типу OderDTO (getBody(OrderDTO.class)) и получает тело сообщения. В данном примере только отображается сообщение (system.out.println), но может быть и другая обработка (например, путем ее делегирования сессионному компоненту, не сохраняющему состояние).

Листинг 13.17. ExpensiveOrderMDB, потребляющий только заказы на сумму более $1000

```
@MessageDriven(mappedName = "jms/javaee7/Topic", activationConfig = {
        @ActivationConfigProperty(propertyName  = "acknowledgeMode",
                                  propertyValue = "Auto-acknowledge"),
        @ActivationConfigProperty(propertyName  = "messageSelector",
                                  propertyValue = "orderAmount > 1000")
} )
public class ExpensiveOrderMDB implements MessageListener {

  public void onMessage(Message message) {
    try {
      OrderDTO order = message.getBody(OrderDTO.class);
      System.out.println("Большой заказ получен: " + order.toString());
    }  catch (JMSException e) {
      e.printStackTrace();
    }
  }
}
```

Компиляция и упаковка с помощью Maven

`ExpensiveOrderMDB` и `OrderDTO` должны быть упакованы в один файл с расширением `.jar`, чтобы в дальнейшем быть развернутыми на GlassFish. MDB использует аннотации из пакета EJB (`@MessageDriven`) и JMS API (`ConnectionFactory`, `Destination` и т. д.), поэтому файл `pom.xml`, показанный в листинге 13.18, использует зависимость `glassfish-embedded-all` (которая содержит все API Java EE 7). Эта зависимость имеет область действия `provided`, поскольку GlassFish, как контейнер EJB и поставщик JMS, предоставляет эти API во время выполнения. Maven должен знать, что вы используете Java SE 7, в результате конфигурирования надстройки `maven-compiler-plugin`.

Листинг 13.18. Файл pom.xml, предназначенный для создания и упаковки MDB

```
<?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/xsd/maven-4.0.0.xsd" >
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <artifactId>chapter13</artifactId>
    <groupId>org.agoncal.book.javaee7</groupId>
    <version>1.0</version>
  </parent>

  <groupId>org.agoncal.book.javaee7.chapter13</groupId>
  <artifactId>chapter13-mdb</artifactId>
  <version>1.0</version>

  <dependencies>
    <dependency>
```

```
        <groupId>org.glassfish.main.extras</groupId>
        <artifactId>glassfish-embedded-all<artifactId>
        <version>4.0</version>
        <scope>provided</scope>
      </dependency>
    </dependencies>

    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>2.5.1</version>
          <configuration>
            <source>1.7</source>
            <target>1.7</target>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </project>
```

Для компиляции и упаковки классов откройте интерпретатор командной строки в каталоге, содержащем файл `pom.xml`, и введите следующую команду Maven:

```
$mvn package
```

Перейдите в директорию target, где вы должны увидеть файл `chapter13-MDB-1.0.jar`. Открыв его, вы увидите, что он содержит файл класса для классов `ExpensiveOrderMDB` и `OrderDTO`.

Создание администрируемых объектов

Администрируемые объекты, необходимые для отправки и получения сообщений, должны быть созданы в поставщике JMS. Каждый из них имеет имя JNDI, что позволяет клиентам получить ссылку на объект через поиск JNDI:

- фабрика соединений называется `jms/javaee7/ConnectionFactory`;
- тема называется `jms/javaee7/Topic`.

Эти объекты создаются административно, поэтому GlassFish должен быть запущен и работать во время создания. Убедившись, что командная строка `asadmin` находится в вашем каталоге, выполните следующую команду в консоли:

```
$ asadmin create-jms-resource --restype javax.jms.ConnectionFactory ➥
                                    jms/javaee7/ConnectionFactory
$ asadmin create-jms-resource --restype javax.jms.Topic jms/javaee7/Topic
```

Веб-консоль GlassFish может использоваться для создания фабрики соединений и очереди. Однако, по моему опыту, самый простой и быстрый способ управлять GlassFish — использовать сценарий `asadmin`. Задействуйте другую команду, чтобы перечислить все JMS-ресурсы и обеспечить успешное создание администрируемых объектов:

```
$ asadmin list-jms-resources
jms/javaee7/Topic
jms/javaee7/ConnectionFactory
```

Начиная с версии JMS 2.0, существует программный способ объявить администрируемые объекты. Идея состоит в том, чтобы аннотировать любой управляемый компонент (управляемый компонент, EJB, MDB...) аннотациями `@JMSConnectionFactoryDefinition` и `@JMSDestinationDefinition`, развернуть компонент, и контейнер гарантированно создаст фабрику соединений и место назначения. Этот механизм подобен изученному вами ранее (см. главу 8, листинг 8.15), где описывалась аннотация `@DataSourceDefinition`. В листинге 13.19 показан класс `ExpensiveOrderMDB` с двумя аннотациями определения.

Листинг 13.19. ExpensiveOrderMDB, определяющий администрируемые объекты программно

```
@JMSConnectionFactoryDefinition(name = "jms/javaee7/ConnectionFactory",
                                className = "javax.jms.ConnectionFactory")
@JMSDestinationDefinition(name = "jms/javaee7/Topic",
                          className = "javax.jms.Topic")
public class ExpensiveOrderMDB implements MessageListener {...}
```

Развертывание MDB на GlassFish

Как только MDB упакован в файл с расширением `.jar`, он должен быть развернут в GlassFish. Это можно сделать несколькими способами, в том числе через консоль веб-администрирования. Тем не менее командная строка `asadmin` довольно просто может решить данную задачу: откройте командную строку, перейдите в директорию, в которой находится файл `chapter13-MDB-1.0.jar`, убедитесь, что GlassFish все еще работает, и введите такую команду:

```
$ asadmin deploy chapter13-MDB-1.0.jar
```

Если развертывание прошло успешно, следующая команда должна вернуть имя развернутого файла `.jar` и его тип (в этом примере EJB):

```
$ asadmin list-components
chapter13-MDB-1.0 <ejb>
```

Выполнение примера

MDB развернут на GlassFish и слушает место назначения `jms/javaee7/Topic`, ожидая появления сообщений. Пришло время запустить клиенты `OrderConsumer` и `OrderProducer`. Эти классы являются автономными приложениями, имеющими метод `main`, который должен быть выполнен вне GlassFish, в чистой среде Java SE. Введите приведенную далее команду для запуска `OrderConsumer`:

```
$ java -cp target/classes OrderConsumer
```

Потребитель работает в бесконечном цикле и ждет входящих заказов. Теперь введите следующую команду, чтобы отправить сообщение с заказом на сумму $2000:

```
$ java -cp target/classes OrderProducer 2000
```

Сумма больше $1000 (граница суммы определена в селекторе сообщения), поэтому и `OrderExpensiveMD`, и `OrderConsumer` должны получить сообщение. Проверьте журналы GlassFish, чтобы убедиться в этом. Если передать параметр меньше чем $1000, MDB не получит сообщение, а только `OrderConsumer`:

```
$ java -cp target/classes OrderProducer 500
```

Резюме

В этой главе показано, что интеграция с помощью обмена сообщениями является слабосвязанной, асинхронной формой общения между компонентами. MOM можно рассматривать в качестве буфера между системами, которые должны производить и потреблять сообщения в своем темпе. Это отличается от архитектуры RPC (например, RMI), в которых клиенты должны знать методы доступных служб.

В первом разделе этой главы мы рассмотрели JMS API (классический и упрощенный) и его словарь. Асинхронная модель является очень мощным API, который можно использовать в среде Java SE или Java EE. Он базируется на моделях P2P и pub-sub, фабриках соединений, местах назначения, соединениях, сеансах, сообщениях (заголовок, свойства, тело) различных типов (текст, объект, словарь, поток, байты), селекторах и других механизмах надежности, таких как подтверждение или стойкость.

В Java EE имеется специальный компонент для потребления сообщений — MDB. Во втором разделе этой главы демонстрируется, как MDB можно использовать в качестве асинхронного потребителя и как они полагаются на контейнер, чтобы решить некоторые задачи (управление жизненным циклом, перехватчиками, транзакциями, безопасностью, параллельным доступом, сообщениями о подтверждении и т. д.).

В этой главе также показано, как собрать эти элементы вместе с помощью Maven, GlassFish и OpenMQ, и приведен пример автономного передатчика и приемника MDB.

В следующих главах будут продемонстрированы другие технологии, используемые для взаимодействия с внешними системами, — SOAP и веб-службы RESTful.

Глава 14
Веб-службы SOAP

Термин «веб-служба» подразумевает «нечто» доступное в Сети («веб»), что может сослужить вам «службу». Первый пример, который приходит на ум, — это HTML-страница: она доступна в Интернете и после прочтения дает вам информацию, которую вы ищете. Другой вид веб-служб — это сервлеты. Они связаны с URL, следовательно, доступны в Интернете и выполняют любую обработку. Но термин «веб-службы» быстро распространился и был приравнен к сервис-ориентированной архитектуре (SOA). Сегодняшние веб-службы являются частью нашей повседневной жизни. Приложения веб-служб могут быть реализованы с помощью различных технологий, таких как SOAP, описанная в этой главе, или REST (см. следующую главу).

Веб-службы SOAP, как говорят, слабо связаны, потому что клиент веб-службы, известный как потребитель, не должен знать детали его реализации (например, язык, использованный при разработке, сигнатуры методов или платформа, на которой он запущен). Потребитель имеет возможность вызвать веб-службу SOAP с помощью простого интерфейса, описывающего доступные бизнес-методы (параметры и возвращаемые значения). Реализация может быть выполнена на любом языке (Visual Basic, C#, C, C++, Java и т. д.). Потребитель и поставщик веб-службы будут по-прежнему иметь возможность обмениваться данными слабосвязанным способом: с помощью XML-документов. Потребитель отправляет запрос веб-службе SOAP в виде документа XML и, возможно, получает ответ также в формате XML.

Веб-службы SOAP заботятся и о распределении. Распределенное программное обеспечение существовало в течение недолгого времени, но, в отличие от существующих распределенных систем, веб-службы SOAP приспособлены для работы в Сети. По умолчанию используется HTTP — известный и надежный протокол, не сохраняющий состояние.

Веб-службы SOAP есть везде. Они могут быть вызваны с простого компьютера или использовать интеграцию business-to-business (B2B), так что операции, которые раньше требовали ручного вмешательства, выполняются автоматически. Веб-службы SOAP объединяют приложения, используемые различными организациями через Интернет или в рамках одной компании (что известно как Enterprise Application Integration, или EAI). Во всех случаях они предоставляют стандартный способ соединения разных частей программного обеспечения.

В этой главе сначала будут изложены некоторые важные сведения, чтобы вы могли понять, что такое веб-службы SOAP, такие как WSDL или SOAP. Далее будет показано, как создать веб-службу SOAP, а затем — как ее использовать.

Основные сведения о веб-службах SOAP

Проще говоря, веб-службы SOAP — это своего рода бизнес-логика, предоставляемая через службу (поставщика услуг) клиенту (потребителю услуг). Однако, в отличие от объектов или компонентов EJB, веб-службы SOAP предоставляют слабосвязанный интерфейс с помощью XML. Стандарты веб-служб SOAP указывают, что интерфейс, которому посылается сообщение, должен определять формат сообщений запроса и ответа, а также механизмы для публикации и обнаружения интерфейсов веб-служб (реестр служб).

На рис. 14.1 вы можете увидеть высокоуровневую картину взаимодействия веб-служб SOAP. Они могут зарегистрировать свой интерфейс в реестре (UDDI), чтобы потребитель смог обнаружить его. Как только потребитель узнает интерфейс службы и формат сообщения, он может отправить запрос к поставщику услуг и получить ответ.

Рис. 14.1. Потребитель обнаруживает службу с помощью реестра

Веб-службы SOAP зависят от нескольких технологий и протоколов для передачи и преобразования данных от потребителя к поставщику стандартным образом. Это такие технологии и протоколы, как:

- расширяемый язык разметки (XML) — основной фундамент, на котором строятся и определяются веб-службы SOAP (SOAP, WSDL и UDDI);
- Web Services Description Language (WSDL, язык описания веб-служб) — определяет протокол, интерфейс, типы сообщений и взаимодействия между потребителем и поставщиком;
- Simple Object Access Protocol (SOAP, простой протокол доступа к объекту) — протокол кодирования сообщения, основанный на технологиях XML, определяющих «конверт» для общения веб-служб;
- транспортный протокол — позволяет обмениваться сообщениями. Хотя протокол передачи гипертекста (HTTP) является наиболее популярным транспортным протоколом, другие — такие как SMTP или JMS — также могут использоваться;
- Universal Description Discovery and Integration (UDDI) — необязательный реестр служб и механизмов обнаружения, похожий на телефонный справочник. Он может быть использован для хранения и категоризации SOAP-интерфейсов веб-служб (WSDL).

С помощью этих стандартных технологий веб-службы SOAP обеспечивают практически неограниченный потенциал. Клиенты могут вызывать службу, которая может использоваться любой программой и размещать любой тип данных и структуру для обмена сообщениями с помощью XML.

XML

Я уже описал XML в главе 12, и теперь вы знаете, как управлять XML-документами, анализировать и связывать их. Поскольку XML является идеальной технологией интеграции, которая решает проблему независимости данных и совместимости, он считается ДНК веб-служб SOAP. Помимо того, что XML используется в качестве формата сообщений, с его помощью веб-службы определяются (WSDL) или обмениваются (SOAP). Связанные с этими XML-документами схемы (XSD) применяются для проверки обмена данными между потребителем и поставщиком. Исторически веб-службы SOAP развились из основной идеи RPC (Remote Procedure Call, удаленный вызов процедур) с использованием XML.

WSDL

WSDL — это язык определения интерфейса (IDL), который определяет взаимодействие между потребителями и веб-службами SOAP (рис. 14.2). Он является главным для веб-служб SOAP, поскольку описывает тип сообщения, порт, протокол коммуникации, поддерживаемые операции, расположение и то, что потребитель должен получить взамен. Данный язык определяет контракт, которому будет соответствовать служба. Вы можете рассматривать WSDL как интерфейс Java, который написан на XML.

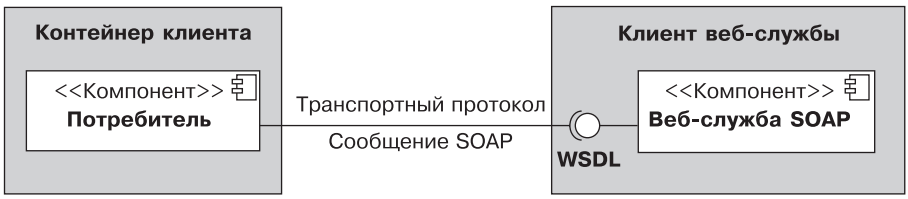


Рис. 14.2. Интерфейс WSDL между потребителем и веб-службой

Для обеспечения совместимости стандартный интерфейс веб-служб необходим потребителю и производителю, чтобы они могли поделиться сообщением и понять его. В этом заключается роль WSDL. В листинге 14.1 показан пример, который представляет собой интерфейс веб-службы SOAP, проверяющей кредитную карту (эта служба принимает в качестве входных данных сведения о кредитной карте и проверяет их). Документ WSDL состоит из чистого XML, и если вы прочтете его внимательно, то увидите всю информацию, необходимую для того, чтобы потребитель смог найти веб-службу (`soap:address location`), вызвать метод (`operation name="validate"`) и использовать соответствующий транспортный протокол (`soap:binding transport`).

Листинг 14.1. Файл WSDL, представляющий службу проверки кредитной карточки

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions ➥
      xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata" ➥
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" ➥
      xmlns:tns="http://chapter14.javaee7.book.agoncal.org/" ➥
      xmlns:xsd="http://www.w3.org/2001/XMLSchema" ➥
      xmlns="http://schemas.xmlsoap.org/wsdl/" ➥
      targetNamespace="http://chapter14.javaee7.book.agoncal.org/" ➥
      name="CardValidatorService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://chapter14.javaee7.book.agoncal.org/" ➥
        schemaLocation="http://localhost:8080/chapter14/
CardValidatorService?xsd=1"/>
    </xsd:schema>
  </types>
  <message name="validate">
    <part name="parameters" element="tns:validate"/>
  </message>
  <message name="validateResponse">
    <part name="parameters" element="tns:validateResponse"/>
  </message>
  <portType name="CardValidator">
    <operation name="validate">
      <input message="tns:validate"/>
      <output message="tns:validateResponse"/>
    </operation>
  </portType>
  <binding name="CardValidatorPortBinding" type="tns:CardValidator">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/
http"style="document"/>
    <operation name="validate">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="CardValidatorService">
    <port name="CardValidatorPort" binding="tns:CardValidatorPortBinding">
      <soap:address location="http://localhost:8080/chapter14/
CardValidatorService"/>
    </port>
  </service>
</definitions>
```

WSDL использует XML для описания того, что служба делает, как вызвать ее операции и где ее найти. Он следует фиксированной структуре, включающей не-

сколько частей (types, message, portType, binding, service). В табл. 14.1 перечислено подмножество элементов и атрибутов WSDL. WSDL — богатый язык, который не ограничивается тем, что описано в табл. 14.1 и определяется W3C. Если вы хотите узнать больше о WSDL, его структуре и типах данных, перейдите на соответствующий сайт W3C.

Таблица 14.1. Элементы и атрибуты WSDL

Элемент	Описание
definitions	Корневой элемент WSDL, определяющий глобальные описания пространств имен, которые видны на протяжении всего документа
types	Определяет типы данных, которые будут использованы в сообщениях. В этом примере приведено описание схемы XML (CardValidatorService?xsd=1) с параметрами, переданными запросу и ответу веб-служб
message	Определяет формат данных, которые передаются между потребителем веб-службы и самой веб-службой. В примере у вас есть запрос (метод validate) и ответ (validateResponse)
portType	Определяет операции веб-служб (метод validate). Каждая операция ссылается на входное и выходное сообщение
binding	Описывает конкретный протокол (в данном случае SOAP) и форматы данных для операций и сообщений, определенных для конкретного типа порта
service	Содержит коллекцию элементов <port>, где каждый порт связан с конечной точкой (сетевым адресом или URL)
port	Указывает адрес для связывания, таким образом определяя конечную точку коммуникации

Элемент <xsd:import namespace> относится к XML-схеме, которая должна быть доступна в сети потребителям WSDL. В листинге 14.2 показана схема определения типов данных, используемая в веб-службе: структура объекта CreditCard (с номером, сроком годности и т. д.), отправленная в запросе (validate), и двоичное значение (действительна карточка или нет), полученное в ответ (validateResponse).

Листинг 14.2. Схема импортируемых файлов WSDL

```
<xs:schema xmlns:tns="http://chapter14.javaee7.book.agoncal.org/" ➥
           xmlns:xs="http://www.w3.org/2001/XMLSchema" ➥
           targetNamespace="http://chapter14.javaee7.book.agoncal.org/
"version="1.0">
  <xs:element name="validate" type="tns:validate"/>
  <xs:element name="validateResponse" type="tns:validateResponse"/>
  <xs:complexType name="validate">
    <xs:sequence>
      <xs:element name="arg0" type="tns:creditCard" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  < xs:complexType name="creditCard">
    <xs:sequence/>
    <xs:attribute name="number" type="xs:string" use="required"/>
    <xs:attribute name="expiry_date" type="xs:string" use="required"/>
```

```
      <xs:attribute name="control_number" type="xs:int" use="required"/>
      <xs:attribute name="type" type="xs:string" use="required"/>
    </xs:complexType>
    <xs:complexType name="validateResponse">
      <xs:sequence>
        <xs:element name="return" type="xs:boolean"/>
      </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Если вы помните JAXB, описанный в главе 12, то поймете, что XSD в листинге 14.2 имеет всю необходимую для связывания с классами Java информацию. С помощью WSDL и схемы любой потребитель может генерировать требуемые для вызова веб-службы артефакты (как вы увидите позже, JAX-WS поставляется с утилитами, которые автоматически генерируют эти артефакты).

SOAP

WSDL описывает абстрактный интерфейс веб-службы, в то время как SOAP предоставляет конкретную реализацию, определяя XML-сообщения, которыми обмениваются потребитель и поставщик (рис. 14.3). SOAP — это стандартный протокол для приложений веб-служб. Он обеспечивает механизм связи для подключения веб-служб, обмена форматированными XML-данными через сетевой протокол, обычно HTTP. Как и WSDL, SOAP в значительной мере опирается на XML, поскольку сообщение SOAP — это XML-документ с несколькими элементами (конверт, заголовок, тело и т. д.). Вместо того чтобы использовать HTTP, для выполнения запроса веб-страниц с помощью браузера SOAP отправляет XML-сообщения, помещая их в запрос HTTP, и получает ответ HTTP.

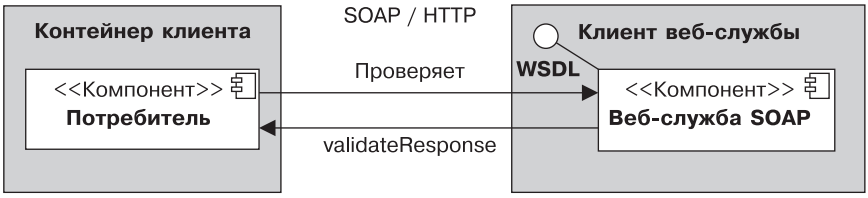


Рис. 14.3. Потребитель, вызывающий веб-службу SOAP

SOAP предназначен для обеспечения независимого, абстрактного протокола связи с возможностью подключения распределенных служб. Связанные службы могут быть построены с использованием любой комбинации аппаратного и программного обеспечения, которые поддерживает данный транспортный протокол.

Вернемся к примеру проверки кредитной карты, показанному на рис. 14.3. Потребитель вызывает веб-службу SOAP для проверки кредитной карты, передавая все необходимые параметры (номер кредитной карты, срок действия, тип и контрольный номер), получает логическое значение и информирует потребителя о том, действительна ли карта. В листингах 14.3 и 14.4 соответственно показана структура этих двух сообщений SOAP.

Листинг 14.3. SOAP-конверт, отправленный в качестве запроса

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/ ➥
               xmlns:cc="http://chapter14.javaee7.book.agoncal.org/">
  <soap:Header/>
  <soap:Body>
    <cc:validate>
      <arg0 number="1234567 89011" expiry_date="10/12" control_number="544"
type="Visa"/>
    </cc:validate>
  </soap:Body>
</soap:Envelope>
```

Листинг 14.4. SOAP-конверт, полученный в качестве ответа

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" ➥
               xmlns:cc="http://chapter14.javaee7.book.agoncal.org/">
  <soap:Body>
    <cc:validateResponse>
      <return>true</return>
    </cc:validateResponse>
  </soap:Body>
</soap:Envelope>
```

Потребитель передает всю информацию о кредитной карте в SOAP-конверт (см. листинг 14.3), методу `validate` веб-службы проверки кредитной карты. Служба возвращает другой конверт SOAP (см. листинг 14.4) с результатом проверки (истина или ложь).

В табл. 14.2 перечислено подмножество элементов и атрибутов SOAP. Как и WSDL, SOAP определяется стандартом W3C.

Таблица 14.2. Элементы и атрибуты SOAP

Элемент	Описание
Envelope	Определяет сообщение и пространство имен, использованное в документе. Этот обязательный элемент является корневым
Header	Содержит любые необязательные атрибуты сообщения или характерную для приложения инфраструктуру, такую как информация о безопасности или о сетевой маршрутизации
Body	Содержит сообщение, которым обмениваются приложения
Fault	Предоставляет информацию об ошибках, которые произошли при обработке сообщений. Этот элемент необязателен

UDDI

Потребители и поставщики, взаимодействующие друг с другом через Интернет, должны быть в состоянии найти информацию, которая позволит их соединить. Любой потребитель знает о точном местоположении службы, которую хочет вызвать или найти. UDDI предоставляет стандартный подход к поиску информации о размещении веб-службы и о том, как ее вызвать. Поставщик служб публикует WSDL в реестре UDDI, доступном в Интернете. Тогда служба может быть обнаружена

и загружена потенциальными потребителями. Это не обязательно, поскольку можно вызвать веб-службу без UDDI, если вы уже знаете ее расположение.

UDDI — это основанный на XML реестр веб-служб, похожий на телефонный справочник, где предприятия могут зарегистрировать свои службы. Такая регистрация включает в себя вид деятельности, географическое положение, сайт, номер телефона и т. д. Другие предприятия могут выполнять поиск по реестру и узнавать информацию о конкретных веб-службах. Эта информация предоставляет дополнительные метаданные о службе, описывающие ее поведение и фактическое местонахождение документа WSDL.

ПРИМЕЧАНИЕ

UDDI не получила столь широкого распространения, несмотря на надежды его дизайнеров (IBM, Microsoft и SAP). В январе 2006 года три компании объявили, что они закрывают свои общедоступные реестры UDDI. В конце 2007 года группа, определяющая UDDI в OASIS, объявила о закрытии технического комитета.

Транспортный протокол

Чтобы потребитель смог общаться с веб-службой, понадобится способ для отправки сообщений. Сообщения SOAP могут транспортироваться через сеть по протоколу, который могут поддерживать обе стороны. Учитывая, что веб-службы требуются в основном в Интернете, они обычно используют HTTP, но могут быть задействованы и другие сетевые протоколы, такие как HTTPS (HTTP Secure), TCP/IP, SMTP (Simple Mail Transport Protocol, простой протокол пересылки почты), FTP (File Transfer Protocol, протокол обмена файлами) и т. д.

Обзор спецификаций веб-служб SOAP

Как показано в главе 4, устойчивость в основном обеспечивается благодаря одной спецификации — JPA. Для веб-служб ситуация более сложная, поскольку вам приходится иметь дело с большим количеством спецификаций, поступающих из различных стандартных органов. Кроме того, поскольку другие языки программирования также используют веб-службы, не все эти спецификации непосредственно связаны с Java Community Process (JCP).

Краткая история спецификаций веб-служб SOAP

Веб-службы SOAP являются стандартным способом общения по сети для предприятий. У этих веб-служб были предшественники: Common Object Request Broker Architecture (CORBA), которая первоначально использовалась в UNIX-системах, и Distributed Component Object Model (DCOM), его конкурент от компании Microsoft. На уровне пониже работают технологии Remote Procedure Call (RPC) и более близкая к нашему миру Java Remote Method Invocation (RMI).

До появления Интернета основным производителям было трудно договориться, какой транспортный протокол следует использовать. Когда протокол HTTP был выбран стандартом, он постепенно стал универсальным средством бизнес-коммуникации. Примерно в то же время XML также стал стандартом официально,

когда W3C объявил, что XML 1.0 пригоден для развертывания в приложениях. К 1998 году обе составляющие, HTTP и XML, готовы были работать вместе.

SOAP 1.0, работа над которым началась в Microsoft в 1998 году, был выпущен в конце 1999 года. Он моделировал типизированные ссылки и массивы в схеме XML. К 2000 году компания IBM начала работу над SOAP 1.1, а WSDL представили на рассмотрение W3C в 2001 году. UDDI была написана в 2000 году усилиями организации по развитию стандартов структурированной информации (OASIS), чтобы позволить предприятиям предоставлять и обнаруживать веб-службы. SOAP, WSDL и UDDI стали стандартами де-факто, используемыми для создания веб-служб. Их поддерживали основные IT-компании.

Java представила возможности для работы с веб-службами в Java API для основанного на XML RPC 1.0 (JAX-RPC 1.0) в июне 2002 года, а JAX-RPC 1.1 был добавлен в J2EE 1.4 в 2003 году. Эта спецификация оказалась довольно избыточной и сложной в использовании. С появлением Java EE 5 и аннотаций в Java новая спецификация Java API для основанных на XML веб-службах версии 2.0 (JAX-WS 2.0) была представлена в качестве предпочтительной модели веб-служб SOAP. Сегодня Java EE 7 поставляется с JAX-WS 2.2a.

Спецификации, связанные с веб-службами SOAP

Чтобы освоить все стандарты веб-служб, вам придется потратить некоторое время на чтение всех спецификаций, перечисленных в табл. 14.3, начиная с W3C, JCP и OASIS.

Таблица 14.3. Спецификации и главные компании, связанные с веб-службами SOAP

Спецификация	Версия	Компания	JSR	URL
JAX-WS	2.2a	JCP	224	http://jcp.org/en/jsr/detail?id=224
Веб-службы	1.3	JCP	109	http://jcp.org/en/jsr/detail?id=109
Метаданные веб-служб	2.1	JCP	181	http://jcp.org/en/jsr/detail?id=181
JAXB	2.2	JCP	222	http://jcp.org/en/jsr/detail?id=222
SAAJ	1.3	JCP	67	http://jcp.org/en/jsr/detail?id=67
JAX-RPC	1.1	JCP	101	http://jcp.org/en/jsr/detail?id=101
JAXR	1.1	JCP	93	http://jcp.org/en/jsr/detail?id=93
SOAP	1.2	W3C	—	http://www.w3.org/TR/soap/
XML	1.1	W3C	—	http://www.w3.org/TR/xml
WSDL	1.1	W3C	—	http://www.w3.org/TR/wsdl
UDDI	1.0	OASIS	—	http://uddi.org/pubs/uddi_v3.htm

W3C — это консорциум, который разрабатывает и поддерживает веб-технологии, такие как HTML, XHTML, RDF, CSS и т. д., и, что более интересно для веб-служб, XML, XML-схемы, SOAP и WSDL.

OASIS хранит несколько связанных с веб-службами стандартов, таких как UDDI, WS-Addressing, WS-Security, WS-Reliability и многие другие (известные как WS-*).

Вернемся к Java. JCP имеет набор спецификаций, которые являются частью Java EE 7 и Java SE 7. Они включают в себя JAX-WS 2.2a (JSR 224), Web Services 1.3 (JSR 109), Web Services Metadata 2.3 (JSR 181) и JAXB 2.2 (JSR 222). Вместе эти спецификации обычно неофициально называют Java Web Services (JWS — веб-службы Java). SAAJ (SOAP with Attachments API for Java — API SOAP с вложениями для Java), определенный в JSR 67, является частью Java SE и позволяет разработчикам создавать и использовать сообщения SOAP с вложениями.

Другие соответствующие спецификации были частью предыдущей версии Java EE и удалены или развиваются сейчас отдельно от Java EE. JAX-WS стал преемником JAX-RPC (JSR 101), который был слишком громоздким и сложным. JAX-RPC удален из Java EE 6, что означает, что его нет и в Java EE 7. Спецификация Java API для XML-реестров (Java API for XML Registers, JAXR) определяет стандартный набор API для Java, которые позволяют клиентам получать доступ к UDDI. Поскольку UDDI не имел ожидаемого успеха, этот JSR (93) был удален и не включен в Java EE 7.

Глядя на этот огромный список спецификаций, вы можете подумать, что писать веб-службы SOAP в Java трудно, особенно когда речь заходит об API. Тем не менее вся прелесть в том, что вам не нужно беспокоиться о базовых технологиях (XML, WSDL, SOAP, HTTP и т. д.) — несколько стандартов JWS будут делать эту работу за вас, что вы увидите в следующих подразделах.

JAX-WS 2.2a

JAX-WS 2.2a определяет набор API (основные пакеты перечислены в табл. 14.4) и аннотаций, которые позволяют создавать и использовать веб-службы в Java. Он предоставляет потребителям и службам возможности по отправке и получению запросов веб-служб через SOAP, маскируя сложность протокола. Таким образом, ни потребитель, ни служба не должны генерировать или анализировать SOAP-сообщения, поскольку низкоуровневой обработкой занимается JAX-WS. Спецификации JAX-WS зависят от других спецификаций, таких как архитектура Java для связывания XML (JAXB), которые вы видели в главе 12.

Таблица 14.4. Основные пакеты JAX-WS

Пакет	Описание
javax.xml.ws	В этом пакете содержатся основные API JAX-WS
javax.xml.ws.http	Определяет API, характерные для связывания XML/HTTP
javax.xml.ws.soap	Определяет API, характерные для связывания SOAP 1.1 / HTTP или SOAP 1.2 / HTTP
javax.xml.ws.handler	В этом пакете определяются API для обработчиков сообщений

Web Services 1.3

JSR 109 определяет модель программирования и поведение во время выполнения веб-служб в контейнере Java EE. Он также определяет упаковку для обеспечения переносимости веб-служб на разные реализации серверов.

WS-Metadata 2.3

Web Services Metadata (WS-Metadata, спецификации JSR 181) предоставляет аннотации, помогающие определять и развертывать веб-службы (основные пакеты перечислены в табл. 14.5). Основная цель JSR 181 — упростить развитие веб-служб. Он обеспечивает отображение объектов между WSDL и интерфейсами Java и наоборот, что достигается благодаря аннотациям. Эти аннотации могут быть использованы в простых классах Java или EJB.

Таблица 14.5. Основные пакеты WS-Metadadta

Пакет	Описание
javax.jws	Содержит аннотации для перехода от Java к WSDL
javax.jws.soap	API, предназначенные для преобразования веб-служб к протоколу сообщений SOAP

Что нового в спецификации веб-служб SOAP

К сожалению, ничего нового не появилось в JAX-WS и WS-Metadata в Java EE 7. Спецификации веб-служб SOAP не были обновлены в этой версии Java EE. Например, в Java EE 7 веб-службы SOAP не используются управляемыми компонентами, поэтому вы не можете задействовать перехватчики, связывание перехватчиков, внедрение контекста и т. д. Bean Validation также не был интегрирован, то есть вы не можете выполнить проверку на уровне метода (см. главу 3).

Примеры реализаций

Metro — это пример реализации спецификаций веб-служб Java с открытым исходным кодом. Он состоит из JAX-WS и JAXB, а также поддерживает устаревшие API JAX-RPC. Он позволяет создавать и развертывать безопасные, надежные, транзакционные, совместимые с SOAP веб-службы, а также потребители веб-служб SOAP. Metro создается сообществом GlassFish, но может быть использован и вне GlassFish, в среде Java SE или других веб-контейнерах (например, Tomcat или Jetty).

Apache CXF (ранее известный как XFire) и Apache Axis2 также реализуют стек JWS. Кроме того, не являясь примерами реализации, оба фреймворка также активно используются в веб-службах SOAP.

Создание веб-служб SOAP

До сих пор мы рассмотрели много низкоуровневых понятий, таких как протокол HTTP, документы WSDL, сообщения SOAP или XML. Но как создать веб-службу SOAP с помощью всех этих спецификаций? Вы можете либо начать с WSDL, либо сразу перейти к кодированию на Java.

Поскольку документ WSDL — это контракт между потребителем и службой, он может быть использован для создания кода Java для потребителя и для службы. Этот подход называется *нисходящим*, также известным как *«сначала контракт»*. Он начинается с создания контракта (WSDL) путем определения операций, сообщений

и т. д. Когда потребитель и поставщик договорились о контракте, можно реализовывать классы Java на основе этого договора. Metro предоставляет некоторые инструменты (`wsimport`), позволяющие создать классы на базе WSDL.

При другом подходе — *восходящем* — реализация класса уже существует и нужно лишь создать WSDL. Опять же Metro предоставляет утилиты (`wsgen`) для генерации WSDL из существующих классов. В обоих случаях код, возможно, должен быть скорректирован в соответствии с WSDL и наоборот. В этой ситуации приходит на помощь JAX-WS. Благодаря простой модели развития и нескольким аннотациям преобразования Java к WSDL могут быть скорректированы. Но будьте осторожны: восходящий подход может привести к созданию очень неэффективных приложений, поскольку методы и классы Java не имеют никакого отношения к идеальной детализации сообщений, перемещающихся по сети. Если имеют место высокая задержка и/или низкая пропускная способность, следует посылать меньшее количество более крупных сообщений, и это может быть более эффективно выполнено при использовании первого подхода.

Несмотря на все эти спецификации, концепции, стандарты и организации, написание и использование веб-служб с применением нисходящего подхода очень легко реализуется. Веб-службы SOAP следуют парадигме Java EE 7 «простота разработки» и не требуют от вас писать WSDL или SOAP. Веб-службы — это просто аннотированный POJO, который должен быть развернут в контейнере веб-службы. В листинге 14.5 показан код веб-службы, которая проверяет кредитную карту.

Листинг 14.5. Веб-служба CardValidator

```
@WebService
public class CardValidator {
  public boolean validate(CreditCard creditCard) {
    Character lastDigit = creditCard.getNumber().charAt( ➥
                          creditCard.getNumber().length() - 1);

    if (Integer.parseInt(lastDigit.toString()) % 2 != 0) {
      return true;
    } else {
      return false;
    }
  }
}
```

Как и сущности или EJB, веб-службы SOAP используют модель аннотированного POJO, который следует политике конфигурации с помощью исключений. Это означает, что веб-служба может быть простым классом Java, который имеет аннотацию `@javax.jws.WebService`, если все значения по умолчанию вам подходят. Тогда среда выполнения JAX-WS сгенерирует код, необходимый для преобразования вызова метода Java в XML, который отправляется через HTTP. Эта веб-служба SOAP `CardValidator` имеет один метод, предназначенный для проверки кредитной карты. Он принимает в качестве параметра информацию о кредитной карте и возвращает двоичное значение в зависимости от того, действительна карта или нет. В этом примере он предполагает, что кредитные карты с четным номером действительны, а карты с нечетным — нет.

Объект CreditCard (листинг 14.6) — это объект, которым обмениваются потребитель и веб-служба SOAP. При описании архитектуры веб-службы данные для обмена должны представлять собой XML-документ, поэтому требуется метод для преобразования объекта Java в документ XML. Здесь в действие вступает JAXB, который имеет простые аннотации и мощный API. Как показано в главе 12, объект типа CreditCard должен иметь аннотацию @javax.xml.bind.annotation.XmlRootElement, а также несколько других аннотаций для преобразования (например, @XmlAttribute), если вам нужно настроить преобразование. Тогда JAXB сможет преобразовать XML в Java и наоборот.

Листинг 14.6. Класс CreditCard с аннотациями JAXB

```
@XmlRootElement
public class CreditCard {
  @XmlAttribute(required = true)
  private String number;
  @XmlAttribute(name = "expiry_date", required = true)
  private String expiryDate;
  @XmlAttribute(name = "control_number", required = true)
  private Integer controlNumber;
  @XmlAttribute(required = true)
  private String type;
  // Конструкторы, методы работы со свойствами
}
```

Благодаря аннотациям JAXB можно избежать разработки низкоуровневых средств анализа XML, и это происходит неявно. Веб-служба манипулирует объектом Java, и то же верно для потребителя.

Структура веб-службы SOAP

Как и большинство компонентов Java EE 7, веб-службы SOAP полагаются на парадигму конфигурации с помощью исключений, которая указывает, что настройка компонента является исключением. Необходима только одна аннотация, чтобы превратить POJO в веб-службу SOAP @WebService. К написанию веб-службы предъявляются следующие требования:

- класс должен иметь аннотацию @javax.jws.WebService или XML-эквивалент в дескрипторе развертывания (webservices.xml);
- класс (он же реализация компонента службы) может реализовать нуль или более интерфейсов (интерфейс конечной точки службы), которые должны иметь аннотацию @WebService;
- класс должен быть определен как общедоступный и не должен иметь спецификаторы final или abstract;
- класс должен иметь общедоступный конструктор по умолчанию;
- класс не должен определять метод finalize();
- для того чтобы преобразовать веб-службу SOAP в компонент-конечную точку, класс должен иметь аннотацию @javax.ejb.Stateless или @javax.ejb.Singleton (см. главу 7);

- служба должна быть объектом, не сохраняющим состояние, и не должна сохранять характерное для клиента состояние во время вызовов методов.

Спецификация WS-Metadata (JSR 181) утверждает, что, пока объект отвечает этим требованиям, POJO может быть использован для реализации веб-службы, развернутой в контейнере сервлета. Его обычно называют *сервлетом конечной точки*. Компоненты, не сохраняющие состояния, или синглтоны также могут быть использованы для реализации веб-службы, которая будет развернута в контейнере EJB (конечная точка EJB).

Конечные точки веб-служб SOAP

JAX-WS позволяет использовать в качестве веб-служб обычные классы Java и EJB. Мы называем это интерфейсами конечной точки службы (SEI). Сравнивая код POJO (см. листинг 14.5) и веб-службы EJB (листинг 14.7), сложно увидеть какие-либо различия, за исключением того, что веб-служба EJB имеет дополнительные аннотации `@Stateless` (или `@Singleton`). Однако упаковка может отличаться, что вы увидите дальше.

Листинг 14.7. Веб-служба CardValidator в качестве конечной точки EJB

```
@WebService
@Stateless
public class CardValidator {

  public boolean validate(CreditCard creditCard) {
    Character lastDigit = creditCard.getNumber().charAt( ➥
                          creditCard.getNumber().length() - 1);

    if (Integer.parseInt(lastDigit.toString()) % 2 != 0) {
      return true;
    } else {
      return false;
    }
  }
}
```

Обе конечные точки имеют почти идентичное поведение, но если использовать конечные точки EJB, можно получить несколько преимуществ. Поскольку веб-служба — это тоже EJB, транзакции и безопасность будут управляться контейнером автоматически, а также могут быть задействованы перехватчики, что невозможно в том случае, если в качестве конечной точки используется сервлет. Бизнес-код может быть представлен в виде веб-службы и в виде EJB в одно и то же время. Это означает, что бизнес-логика может быть доступна через SOAP, а также через RMI, если добавить удаленный интерфейс.

Преобразование WSDL

На уровне служб системы определяются в терминах XML-сообщений, операций WSDL и сообщений SOAP. Между тем на уровне Java приложения определяются в терминах объектов, интерфейсов и методов. Необходимо выполнить преобразова-

ние объектов Java в операции WSDL. Среда выполнения JAXB использует аннотации, чтобы определить, как осуществить маршалинг/демаршалинг класса в XML и из него. Кроме того, JWS применяет аннотации для преобразования классов Java в WSDL и для определения того, как произвести маршалинг вызова метода в запрос SOAP и демаршалинг ответа SOAP в экземпляр возвращаемого методом типа.

В спецификациях JAX-WS (JSR 224) и WS-Metadata (JSR 181) определены два различных типа аннотаций:

- *аннотации отображения WSDL* — относятся к пакету `javax.jws` и позволяют вам изменить преобразование WSDL/Java. Аннотации `@WebMethod`, `@WebResult`, `@WebParam` и `@OneWay` используются веб-службой для настройки сигнатуры доступных методов;
- *аннотации связывания SOAP* — относятся к пакету `javax.jws.soap` и позволяют настроить связывание SOAP (`@SoapBinding` и `@SOAPMessageHandler`).

Как и все другие спецификации Java EE 7, аннотации веб-служб могут быть переопределены в необязательном дескрипторе развертывания XML (`webservices.xml`). Внимательнее посмотрим на аннотации преобразования WSDL.

@WebService

Аннотация `@javax.jws.WebService` помечает класс Java или интерфейс как веб-службу. Если аннотация использована непосредственно для класса (как во всех примерах выше), то процессор аннотаций контейнера сгенерирует интерфейс, так что следующие фрагменты кода эквивалентны. Вот пример аннотации для класса:

```
@WebService
public class CardValidator {...}
```

А следующий фрагмент кода показывает аннотацию на интерфейсе, который реализует класс. Как вы можете видеть, реализация должна определить полное имя интерфейса в атрибуте `endpointInterface`:

```
@WebService
public interface Validator {...}
@WebService (endpointInterface = "org.agoncal.book.javaee7.chapter14.
Validator")
public class CardValidator implements Validator {...}
```

Аннотация `@WebService` имеет набор атрибутов (листинг 14.8), которые позволяют настроить имя веб-службы в файле WSDL (элементы `<wsdl:portType>` или `<wsdl:service>`) и в его пространстве имен, а также изменить расположение самой WSDL (атрибут `wsdlLocation`).

Листинг 14.8. @WebService API

```
@Retention(RUNTIME) @Target(TYPE)
public @interface WebService {
  String name() default "";
  String targetNamespace() default "";
  String serviceName() default "";
  String portName() default "";
```

```
  String wsdlLocation() default "";
  String endpointInterface() default "";
}
```

Поэтому, когда правила преобразования WSDL по умолчанию не подходят для вашей веб-службы SOAP, просто используйте необходимые атрибуты. Приведенный ниже код изменяет порт и имя службы:

```
@WebService(portName = "CreditCardValidator", ➥
            serviceName = "ValidatorService")
public class CardValidator {...}
```

Если вы сравните этот код со стандартным WSDL, описанным в листинге 14.1, то увидите следующие изменения:

```
<service name="ValidatorService">
  <port name="CreditCardValidator" binding="tns:CreditCardValidatorBinding">
  <soap:address location="http://localhost:8080/chapter14/ValidatorService"/>
  </port>
</service>
```

Когда вы используете аннотацию `@WebService`, будут предоставлены все общедоступные методы веб-служб, за исключением случаев, когда применяется аннотация `@WebMethod`.

@WebMethod

По умолчанию все общедоступные методы веб-служб SOAP попадают в WSDL и используют все правила отображения по умолчанию. Чтобы настроить некоторые элементы этого преобразования, вы можете добавить аннотацию `@javax.jws.WebMethod` для методов. API этой аннотации достаточно прост: он позволяет переименовать метод или исключить его из WSDL. В листинге 14.9 показано, как в веб-службе `CardValidator` переименовываются два первых метода и исключается последний.

Листинг 14.9. Два метода переименованы, последний исключен

```
@WebService
public class CardValidator {
  @WebMethod(operationName = "ValidateCreditCard")
  public boolean validate(CreditCard creditCard) {
    // Бизнес-логика
  }

  @ WebMethod(operationName = "ValidateCreditCardNumber")
  public void validate(String creditCardNumber) {
    // Бизнес-логика
  }

  @WebMethod(exclude = true)
  public void validate(Long creditCardNumber) {
    // Бизнес-логика
  }
}
```

Как вы можете видеть во фрагменте WSDL, описанном в листинге 14.10, определены только два метода: `ValidateCreditCard` и `ValidateCreditCardNumber`. Последний был исключен из WSDL.

Листинг 14.10. Фрагмент WSDL с переименованными методами

```
<message name="ValidateCreditCard">
  <part name="parameters" element="tns:ValidateCreditCard"/>
</message>
<message name="ValidateCreditCardResponse">
  <part name="parameters" element="tns:ValidateCreditCardResponse"/>
</message>
<message name="ValidateCreditCardNumber">
   <part name="parameters" element="tns:ValidateCreditCardNumber"/>
</message>
<message name="ValidateCreditCardNumberResponse">
  <part name="parameters" element="tns:ValidateCreditCardNumberResponse"/>
</message>
<portType name="CardValidator">
  <operation name="ValidateCreditCard">
    <input message="tns:ValidateCreditCard"/>
    <output message="tns:ValidateCreditCardResponse"/>
  </operation>
  <operation name="ValidateCreditCardNumber">
    <input message="tns:ValidateCreditCardNumber"/>
    <output message="tns:ValidateCreditCardNumberResponse"/>
  </operation>
</portType>
```

@WebResult

Аннотация `@javax.jws.WebResult` управляет сгенерированным именем сообщения, возвратившего значение в WSDL. В листинге 14.11 возвращаемый результат метода `validate()` переименован в `IsValid`.

Листинг 14.11. Возвращаемый результат метода переименован

```
@WebService
public class CardValidator {
  @WebResult(name = "IsValid")
  public boolean validate(CreditCard creditCard) {
    // Бизнес-логика
  }
}
```

По умолчанию имя возвращаемого значения в WSDL — `return`. Но с помощью аннотации `@WebResult` вы можете быть конкретнее и иметь более выразительный контракт:

```
<!-- По умолчанию -->
<xs:element name="return" type="xs:boolean"/>
<!-- После переименования на IsValid -->
<xs:element name="IsValid" type="xs:boolean"/>
```

Эта аннотация также включает и другие элементы, предназначенные для настройки WSDL, такие как пространство имен XML для возвращенного значения, и выглядит как аннотация `@WebParam`, показанная в листинге 14.12.

Листинг 14.12. `@WebParam` API

```
@Retention(RUNTIME) @Target(PARAMETER)
public @interface WebParam {
  String name() default "";
  public enum Mode {IN, OUT, INOUT};
  String targetNamespace() default "";
  boolean header() default false;
  String partName() default "";
};
```

@WebParam

Аннотация `@javax.jws.WebParam`, показанная в листинге 14.12, похожа на аннотацию `@WebResult`, поскольку она настраивает параметры для методов веб-службы. Ее API позволяет изменить имя параметра в WSDL (листинг 14.13), пространство имен и тип. Допустимые типы: `IN`, `OUT` или `INOUT` (оба), которые определяют, является параметр входящим или возвращаемым (по умолчанию `IN`).

Листинг 14.13. Параметр метода переименован

```
@WebService
public class CardValidator {
  public boolean validate(@WebParam(name= "Credit-Card", mode = IN) ➥
                          CreditCard creditCard) {
    // Бизнес-логика
  }
}
```

Опять же если вы сравните это со стандартным XSD, определенным в листинге 14.2, то заметите, что по умолчанию имя этого параметра — `arg0`. Аннотация `@WebParam` переопределяет это значение по умолчанию именем `Credit-Card`:

```
<!-- По умолчанию -->
<xs:element name="arg0" type="tns:creditCard" minOccurs="0"/>
<!-- После переименования на Credit-Card -->
<xs:element name="Credit-Card" type="tns:creditCard" minOccurs="0"/>
```

@OneWay

Аннотация `@OneWay` может быть использована для методов, которые не имеют возвращаемого значения, например возвращающих значение типа `void`. Эта аннотация не содержит элементов и может рассматриваться как интерфейс разметки, который информирует контейнер, что вызов может быть оптимизирован, поскольку нет возвращения (можно, например, использовать асинхронный вызов).

@SoapBinding

Связывание описывает, как веб-служба связана с протоколом обмена сообщениями, в частности с протоколом SOAP. Существует два стиля программирования для

связывания SOAP, определенного в WSDL 1.1: RPC и документ (он же обмен сообщениями). От этого выбора зависит то, как будет структурировано тело содержимого SOAP.

- Документ — сообщение SOAP включает в себя документ. Оно посылается в виде одного документа в элементе `<soap:Body>` без дополнительных правил форматирования и содержит все, о чем договорились отправитель и получатель. Выбирается стандартный стиль документа.
- RPC — сообщение SOAP содержит параметры и возвращаемые значения. `<soap:Body>` включает элемент с именем метода или удаленной процедуры. Этот элемент, в свою очередь, хранит элемент для каждого параметра данной процедуры.

При связывании SOAP (независимо от того, какой стиль задан) следует выбрать один из двух различных форматов сериализации/десериализации.

- Буквальный — данные сериализуются в соответствии с XML-схемой.
- Закодированный — кодирование SOAP определяет, как должны быть сериализованы объекты, структуры, массивы и деревья объектов.

Это позволяет вам выбрать один из четырех вариантов пар «стиль/использование»:

- документ/буквальная сериализация (по умолчанию);
- документ/закодированная сериализация (не совместима с WS-*);
- RPC/буквальная сериализация;
- RPC/закодированная сериализация.

По умолчанию сгенерированная WSDL, которую вы видели ранее, использует первую модель (документ/буквальная сериализация). Применение аннотации `@SoapBinding` для класса, как показано в листинге 14.14, может изменить это.

Листинг 14.14. Веб-служба, использующая модель «RPC/буквальная сериализация»

```
@WebService
@SOAPBinding(style = RPC, use = LITERAL)
public class CardValidator {
  public boolean validate(CreditCard creditCard) {
    // Бизнес-логика
  }
}
```

В листинге 14.14 переопределяется стандартное связывание, вместо документа используется стиль RPC. Это оказывает влияние на WSDL и XML-схему, которая генерируется для поставщика веб-служб SOAP и потребителя. В листинге 14.15 продемонстрированы эти различия.

Листинг 14.15. Различия в WSDL между стилями «документ» и RPC

```
<!-- Стиль документа -->
<message name="validate">
  <part name="parameters" element="tns:validate"/>
</message>
<message name="validateResponse">
```

```
    <part name="parameters" element="tns:validateResponse"/>
  </message>
  ...
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
  style="document"/>

  <!-- Стиль RPC-->
  <message name="validate">
    <part name="arg0" type="tns:creditCard"/>
  </message>
  <message name="validateResponse">
    <part name="return" type="xsd:boolean"/>
  </message>
  ...
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/>
```

Собираем все преобразования воедино

Чтобы иметь более полное понимание того, какое влияние оказывают эти аннотации на веб-службы SOAP, соберем их вместе и посмотрим на различные артефакты. Я буду использовать простую веб-службу `CardValidator`, определенную в листинге 14.5, и добавлю большую часть рассмотренных аннотаций для преобразования (листинг 14.16).

Листинг 14.16. Веб-служба CardValidator, к которой применены аннотации преобразования

```
@WebService(portName = "CreditCardValidator", ➥
            serviceName = "ValidatorService")
@SOAPBinding(style = RPC, use = LITERAL)
public class CardValidator {
  @WebResult(name = "IsValid")
  @WebMethod(operationName = "ValidateCreditCard")
  public boolean validate(@WebParam(name = "Credit-Card") ➥
                          CreditCard creditCard) {
    // Бизнес-логика
  }

  @WebResult(name = "IsValid")
  @WebMethod(operationName = "ValidateCreditCardNumber")
  public void validate(@WebParam(name = "Credit-Card-Number") ➥
                       String creditCardNumber) {
    // Бизнес-логика
  }

  @WebMethod(exclude = true)
  public void validate(Long creditCardNumber) {
    // Бизнес-логика
  }
}
```

В листинге 14.16 определена веб-служба, которая следует модели «RPC/буквальное кодирование» и предоставляет только два метода (обратите внимание, что метод `validate(Long CreditCardNumber)` не виден из-за аннотации `@WebMethod(exclude=true)`). Каждый параметр метода и возвращаемые значения переименованы, чтобы WSDL получилась более выразительной. В листинге 14.17 показан результирующий WSDL-документ, который вы можете сравнить с исходным, приведенным в листинге 14.1 (различия выделены полужирным шрифтом).

Листинг 14.17. WSDL после настройки

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions ➥
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" ➥
      xmlns:tns="http://chapter14.javaee7.book.agoncal.org/" ➥
      xmlns:xsd="http://www.w3.org/2001/XMLSchema" ➥
      xmlns="http://schemas.xmlsoap.org/wsdl/" ➥
      targetNamespace="http://chapter14.javaee7.book.agoncal.org/" ➥
      name="ValidatorService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://chapter14.javaee7.book.agoncal.org/" ➥
        schemaLocation="http://localhost:8080/chapter14/
ValidatorService?xsd=1"/>
    </xsd:schema>
  </types>
  <message name="ValidateCreditCard">
    <part name="Credit-Card" type="tns:creditCard"/>
  </message>
  <message name="ValidateCreditCardResponse">
    <part name="IsValid" type="xsd:boolean"/>
  </message>
  <message name="ValidateCreditCardNumber">
    <part name="Credit-Card-Number" type="xsd:string"/>
  </message>
  <message name="ValidateCreditCardNumberResponse"/>
  <portType name="CardValidator">
    <operation name="ValidateCreditCard">
      <input message="tns:ValidateCreditCard"/>
      <output message="tns:ValidateCreditCardResponse"/>
    </operation>
    <operation name="ValidateCreditCardNumber">
      <input message="tns:ValidateCreditCardNumber"/>
      <output message="tns:ValidateCreditCardNumberResponse"/>
    </operation>
  </portType>
  <binding name="CreditCardValidatorBinding" type="tns:CardValidator">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" ➥
      style="rpc"/>
    <operation name="ValidateCreditCard">
      <soap:operation soapAction=""/>
      <input>
```

```
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
    <operation name="ValidateCreditCardNumber">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="ValidatorService">
    <port name="CreditCardValidator" ➥
       binding="tns:CreditCardValidatorBinding">
      <soap:address location="http://localhost:8080/chapter14/
ValidatorService"/>
    </port>
  </service>
</definitions>
```

Аннотация `@WebService` переименовывает элементы WSDL `<portType name>` и `<port name>`. Аннотация `@WebMethod` переименовывает элемент `<operation name>`, а `@WebResult` и `@WebParam` — элемент `<part name>`, который является частью элемента `<message>`.

XML-схема изменяется, как и запрос и ответ, определенные в элементе `<xs:complexType>`. Можно увидеть (листинг 14.18), что она значительно отличается от XSD, заданной в листинге 14.2. Это произошло потому, что веб-служба SOAP использует стиль RPC (не определены документы запроса/ответа, вместо этого применяются типы `creditCard` и `boolean`).

Листинг 14.18. XML-схема после настройки

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://chapter14.javaee7.book.agoncal.org/"
version="1.0">
  <xs:complexType name="creditCard">
    <xs:sequence/>
    <xs:attribute name="number" type="xs:string" use="required"/>
    <xs:attribute name="expiry_date" type="xs:string" use="required"/>
    <xs:attribute name="control_number" type="xs:int" use="required"/>
    <xs:attribute name="type" type="xs:string" use="required"/>
  </xs:complexType>
</xs:schema>
```

WSDL (см. листинг 14.17) и XSD (см. листинг 14.18) используются для определения контракта между потребителем и поставщиком службы. Но во время выполнения они не применяются и потребитель и поставщик обмениваются только

конвертами SOAP. В листинге 14.19 показан SOAP-запрос, который отправляется веб-службе. Он определяет вызываемый метод (`ValidateCreditCard`) и параметры, передаваемые этому методу (`Credit-Card`).

Листинг 14.19. Конверт SOAP для запроса ValidateCreditCard после настройки

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" ➥
               xmlns:cc="http://chapter14.javaee7.book.agoncal.org/">
  <soap:Header/>
  <soap:Body>
    <cc:ValidateCreditCard>
      <Credit-Card number="1234567 89011" expiry_date="10/12" ➥
                   control_number="544" type="Visa"/>
    </cc:ValidateCreditCard>
  </soap:Body>
</soap:Envelope>
```

В листинге 14.20 показан ответ SOAP, который отправляется обратно потребителю. Он указывает, что кредитная карта, переданная в запросе, действительна (`<IsValid>true</IsValid>`).

Листинг 14.20. Конверт SOAP для ответа ValidateCreditCard после настройки

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               xmlns:cc="http://chapter14.javaee7.book.agoncal.org/">
  <soap:Body>
    <cc:ValidateCreditCardResponse>
      <IsValid>true</IsValid>
    </cc:ValidateCreditCardResponse>
  </soap:Body>
</soap:Envelope>
```

Обработка исключений

До этого момента все работало хорошо: обмен данными между потребителем и поставщиком был корректным, веб-служба не пропадала, а сеть работала надежно. Но так бывает не всегда. В Java, когда что-то идет не так, генерируется исключение и некоторые классы внутри JVM должны разобраться с ним. Для веб-служб SOAP этот механизм не работает, поскольку потребитель и служба могут быть написаны на разных языках и находиться в разных сетях. Таким образом, возникла идея использовать SOAP Fault в сообщении SOAP. Среда выполнения JAX-WS автоматически преобразует исключения Java в сообщения SOAP Fault, которые возвращаются клиенту. Эта функция позволяет сэкономить много времени и энергии, которые вы затратите на написание кода, преобразующего ваши исключения в сообщения SOAP Fault.

Если вы посмотрите на метод `validate` веб-службы `CardValidator`, определенный в листинге 14.5, то заметите, что, когда параметр `CreditCard` пуст, проверка прекращается и генерируется исключение `NullPointerException`. В этом случае среда выполнения JAX-WS ловит исключение `NullPointerException` на сервере, создает сообщение SOAP Fault (листинг 14.21) и отправляет его обратно потребителю.

Листинг 14.21. SOAP Fault отправляется в ответе SOAP

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>java.lang.NullPointerException</faultstring>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

Как вы можете видеть в листинге 14.21, среда выполнения JAX-WS автоматически устанавливает значение параметра `faultstring`, соответствующее имени исключения в Java. Спецификация также предоставляет механизм различения типов неисправностей с использованием элемента `faultCode`. В этом случае он имеет значение `soap:Server`, указывая, что за неисправности отвечает сервер (другой вариант — `soap:Client`).

Еще один способ возвращения SOAP Fault — генерация исключения приложения (листинг 14.22). Здесь веб-служба генерирует проверенное исключение (но тот же самый механизм относится и к непроверенным исключениям), если кредитная карта имеет нечетный номер. Это исключение приложения будет автоматически преобразовано в соответствующее сообщение `soap:Fault`, обернуто в тело SOAP и возвращено клиенту.

Листинг 14.22. Проверка генерирует исключение

```
@WebService
public class CardValidator throws CardValidatorException {
public boolean validate(CreditCard creditCard) {
   Character lastDigit = creditCard.getNumber().charAt( ➥
                         creditCard.getNumber().length() - 1);

    if (Integer.parseInt(lastDigit.toString()) % 2 == 0) {
      return true;
    }  else {
      throw new CardValidatorException("Неверный номер кредитной карты");
    }
  }
}
```

Исключение приложения может наследовать от типов `Exception`, `RuntimeException` или исключений веб-служб SOAP, таких как `javax.xml.ws.WebServiceException` или один из его подклассов (например, `javax.xml.ws.soap.SOAPFaultException`). Эти исключения также могут поддерживать аннотацию `@WebFault`, чтобы создать более подробный конверт SOAP (листинг 14.23).

Листинг 14.23. Исключение с аннотацией WebFault

```
@WebFault(name = "CardValidationFault")
public class CardValidatorException extends Exception {
  public CardValidatorRTException() {
    super();
  }
```

```
  public CardValidatorRTException(String message) {
    super(message);
  }
}
```

Когда веб-служба SOAP генерирует исключение, определенное в листинге 14.23, среда выполнения JAX-WS генерирует сообщение об ошибке SOAP, определенное в листинге 14.24.

Листинг 14.24. SOAP Fault в конверте SOAP

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>org.agoncal.book.javaee7.chapter14.CardValidatorException</
faultstring>
      <detail>
        <ns2:CardValidationFault ➥
         xmlns:ns2="http://chapter14.javaee7.book.agoncal.org/">
          <message>Неверный номер кредитной карты</message>
        </ns2:CardValidationFault>
      </detail>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

Но если вы хотите создать более точное сообщение об ошибке SOAP с другим значением `faultCode` и т. д., то можете использовать API `javax.xml.soap.SOAPFactory` для создания объекта типа `javax.xml.soap.SOAPFault` (это показано в листинге 14.25).

Листинг 14.25. Проверка использует SOAPFactory для создания SOAPFault

```
@WebService
public class CardValidator {
  public boolean validate(CreditCard creditCard) {
    Character lastDigit = creditCard.getNumber().charAt( ➥
                          creditCard.getNumber().length() - 1);

    if (Integer.parseInt(lastDigit.toString()) % 2 == 0) {
      return true;
    }  else {
      SOAPFactory soapFactory = SOAPFactory.newInstance();
      SOAPFault fault = soapFactory.createFault("Неверный номер ➥
            кредитной карты", new QName("ValidationFault"));
      throw new CardValidatorException(fault);
    }
  }
}
```

В листинге 14.25 создается объект типа `SOAPFault` с текстом `Номер кредитной карты недействителен` и кодом ошибки (`ValidationFault`), формирующим конверт SOAP (листинг 14.26).

Листинг 14.26. Объект SOAPFault, создающий конверт SOAP

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <soap:Fault>
      <faultcode>ValidationFault</faultcode>
      <faultstring>Неверный номер кредитной карты</faultstring>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

Жизненный цикл и функции обратного вызова

Как вы можете видеть на рис. 14.4, веб-службы SOAP также имеют жизненный цикл, который напоминает управляемые компоненты. Такой жизненный цикл присущ компонентам, не хранящим состояния: либо они не существуют, либо готовы обрабатывать запрос. Этим жизненным циклом управляет контейнер.

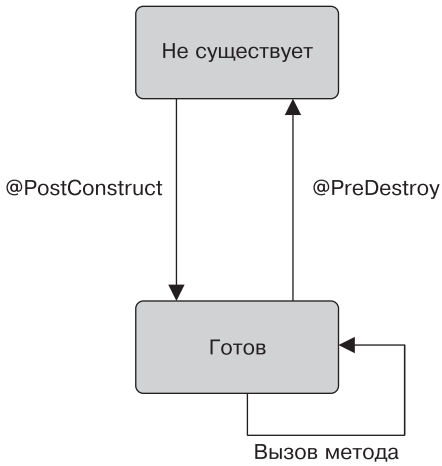


Рис. 14.4. Жизненный цикл веб-службы SOAP

Конечные точки-сервлеты и EJB поддерживают внедрение зависимостей (потому что работают в контейнере) и методы жизненного цикла, такие как `@PostConstruct` и `@PreDestroy`. Контейнер вызывает метод обратного вызова `@PostConstruct`, если таковой существует, когда создает экземпляр веб-службы, и вызывает метод обратного вызова `@PreDestroy` при его разрушении. Одно из различий между типами конечных точек заключается в том, что могут использовать перехватчики (описанные в главе 2).

WebServiceContext

Веб-службы SOAP имеют контекст среды и могут получить доступ к нему при добавлении ссылки на `javax.xml.ws.WebServiceContext` с аннотацией `@Resource`. Внутри этого контекста веб-служба во время выполнения может получить информацию,

такую как класс конечной реализации, контекст сообщения и информацию о безопасности относительно обслуживаемого запроса.

ПРИМЕЧАНИЕ

Спецификации JAX-WS и WS-Metadata не были обновлены в Java EE 7, поэтому, даже если ваше приложение использует CDI, вы не можете задавать аннотацию @Inject, чтобы внедрить WebServiceContext: вам все равно придется обратиться к аннотации @Resource.

Код в листинге 14.27 использует `WebServiceContext`, чтобы узнать, имеет ли вызывающая сторона роль `Admin` для проверки кредитной карты. В табл. 14.6 перечислены методы, определенные в интерфейсе `javax.xml.ws.WebServiceContext`.

Листинг 14.27. Веб-служба SOAP, использующая WebServiceContext

```
@WebService
public class CardValidator {
  @Resource
  private WebServiceContext context;

  public boolean validate(CreditCard creditCard) {
    if (!context.isUserInRole("Admin"))
      throw new SecurityException("Только администратор может проверить карту");
    // Бизнес-логика
    }
  }
```

Таблица 14.6. Методы интерфейса WebServiceContext

Метод	Описание
getMessageContext	Возвращает MessageContext для запроса, обслуживаемого в момент вызова метода. Может быть использован для доступа к частям сообщения SOAP: к заголовку, телу и т. д.
getUserPrincipal	Возвращает Principal, идентифицирующий отправителя запроса, который обслуживается в данный момент
isUserInRole	Возвращает двоичное значение, указывающее, является ли аутентифицированный пользователь представителем определенной логической роли
getEndpointReference	Возвращает EndpointReference, связанный с заданной конечной точкой

Дескриптор развертывания

Как и большинство технологий Java EE 7, веб-службы SOAP позволяют определить метаданные с использованием аннотаций (то, что мы делали во всех примерах), а также с помощью XML. Расположенный в каталоге WEB-INF файл `webservices.xml` замещает или дополняет аннотации. Как и большинство дескрипторов развертывания в Java EE 7, файл `webservices.xml` обязателен, поскольку аннотации могут быть использованы для указания большей части данных, заданных в этом дескрипторе развертывания. Например, в листинге 14.28 показано, как можно переопределить порт WSDL (`OverriddenPort`) для веб-службы `CardValidator`.

Листинг 14.28. Дескриптор развертывания webservices.xml

```
<webservices 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/javaee_web_services_1_3.xsd" ➥
             version="1.3">

  <webservice-description>
    <webservice-description-name>CardValidatorWS</webservice-description-name>

    <port-component>
      <port-component-name>CardValidator</port-component-name>
      <wsdl-port>OverriddenPort</wsdl-port>
      <service-endpoint-interface>
        org.agoncal.book.javaee7.chapter14.Validator
      </service-endpoint-interface>
      <service-impl-bean>
        <servlet-link>CardValidatorServlet</servlet-link>
      </service-impl-bean>
    </port-component>
  </webservice-description>
</webservices>
```

Упаковка

Веб-службы SOAP могут быть упакованы в файл с расширением `.war` или `.jar` (для EJB). Службы, упакованные в файл `.war`, могут использовать конечные точки-сервлеты или EJB Lite. Веб-службы, упакованные в файл `.jar` EJB, могут использовать только сессионные компоненты, не сохраняющие состояния, либо синглтоны.

Разработчик несет ответственность за упаковку таких элементов, как:

- реализация службы компонента и ее зависимые классы;
- интерфейсы конечной точки службы (не обязательно);
- файл WSDL либо путем включения, либо с помощью ссылки (не требуется, если используются аннотации, поскольку WSDL может быть автоматически сгенерирована средой выполнения JAX-WS);
- сгенерированные артефакты для запроса и ответа SOAP (не обязательно, поскольку они автоматически генерируются средой выполнения JAX-WS);
- опциональный дескриптор развертывания.

Публикация веб-служб SOAP

Как только веб-служба SOAP упакована, ее публикация — всего лишь вопрос развертывания архива в таком контейнере Java EE, как GlassFish или JBoss. Это потому, что JAX-WS является частью Java EE 7 и среда выполнения связана с сервером приложений. Но вы также можете опубликовать веб-службу SOAP в таком веб-контейнере, как Tomcat или Jetty, если встроите реализацию JAX-WS, например Metro, CXF или Axis2.

Но помните, что JAX-WS также поставляется в Java SE и иногда вам не потребуется подключать сервлеты или контейнер EJB (например, при тестировании веб-службы). В этом случае вы можете использовать API `javax.xml.ws.Endpoint` для программной публикации веб-службы SOAP. Метод `Endpoint.publish` (листинг 14.29) использует по умолчанию легкий сервер HTTP, который включен в Oracle JVM (определен в пакете `com.sun.net.httpserver`).

Листинг 14.29. Веб-служба SOAP, публикующая себя и принимающая входящие запросы

```
@WebService
public class CardValidator {
  public boolean validate(CreditCard creditCard) {
    // Бизнес-логика
  }

  public static void main(String[] args) {
    Endpoint.publish("http://localhost:8080/cardValidator", ➥
                     new CardValidator());
  }
}
```

В листинге 14.29 метод `publish` используется для публикации веб-службы SOAP `CardValidator`, после чего он начинает принимать входящие запросы по адресу http://localhost:8080/cardValidator. После этого можно вызывать методы вашей веб-службы. Существует также метод `stop`, который может быть использован для прекращения приема входящих запросов и отключения конечной точки (как вы увидите позже при разработке интеграционного теста).

Вызов веб-служб SOAP

До сих пор вы видели, как писать, упаковывать и публиковать веб-службы SOAP. Теперь рассмотрим, как их вызывать. С помощью WSDL и некоторых инструментов для создания заглушек клиентов Java (или прокси) вы можете легко вызвать веб-службы, не заботясь о соединении их с HTTP или SOAP. Вызов веб-службы похож на вызов распределенных объектов с помощью RMI. Как и RMI, JAX-WS позволяет программисту использовать локальный вызов метода для вызова распределенной службы. Разница заключается в том, что на удаленном узле веб-служба может быть написана на другом языке программирования (обратите внимание, что вы можете также вызвать код, написанный не на Java, с применением RMI-IIOP). WSDL является стандартным контрактом между потребителем и службой. Metro предоставляет инструмент преобразования WSDL в Java (`wsimport`), который генерирует Java-интерфейсы и классы из WSDL. Такие прокси дают вам Java-представление конечной точки веб-службы (сервлет или EJB). Этот прокси затем перенаправляет локальный вызов Java удаленной веб-службе, используя HTTP.

Когда с помощью такого прокси вызывается метод (рис. 14.5), он преобразует параметры метода в сообщения SOAP (запрос) и отправляет их конечной точке веб-службы. Чтобы получить результат, ответ SOAP преобразуется обратно в экземпляр

возвращаемого типа. Вам не нужно понимать внутреннюю работу прокси и даже смотреть на его код. Перед компиляцией вашего клиента-потребителя нужно сгенерировать SEI, чтобы получить прокси-класс, который можно вызывать в коде.

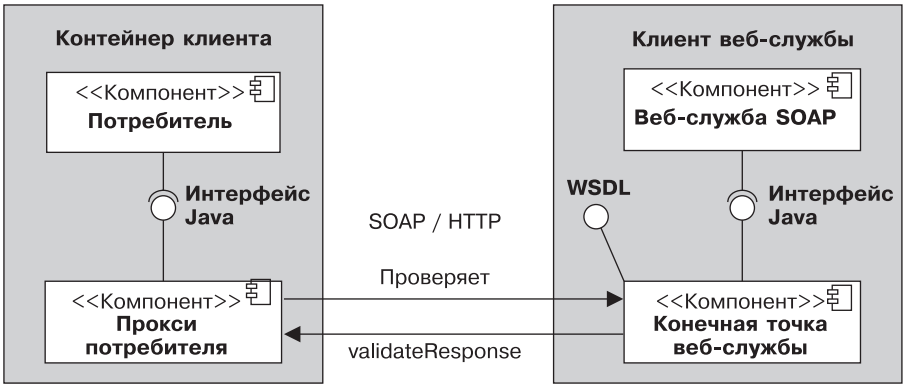


Рис. 14.5. Потребитель, вызывающий веб-службу с помощью прокси

При разработке потребителей SOAP с использованием нисходящего подхода клиент должен получить WSDL, создать необходимые артефакты, вызвать веб-службу `CardValidator` для проверки кредитной карты (SOAP-сообщение `validate`) и получить ответ (SOAP-сообщение `validateResponse`).

ПРИМЕЧАНИЕ

Инструменты wsimport и wsgen поставляются в JDK 1.7, а также GlassFish или Metro. wsimport в качестве входного аргумента принимает WSDL и формирует такие артефакты JAX-WS, как SEI; wsgen читает класс конечной точки веб-службы и генерирует WSDL. Вы можете получить доступ к этим инструментам непосредственно после установки Java SE 7 либо с помощью интерфейса командной строки (CLI) GlassFish, задачи Ant или плагина Maven.

Структура потребителя SOAP

Поскольку JAX-WS доступен в Java SE, использовать веб-службы SOAP может любой код Java, начиная с класса `Main`, запущенного в JVM, и заканчивая любым из компонентов Java EE, работающим в контейнере (сетевом, EJB или контейнере клиентских приложений). Если он работает в контейнере, потребитель может получить экземпляр прокси либо путем внедрения, либо создав его программно. Чтобы внедрить веб-службу, вам нужно использовать аннотацию `@javax.xml.ws.WebServiceRef` или производитель CDI.

Программный вызов

Если ваш потребитель выполняется вне контейнера, необходимо программно вызывать вашу веб-службу SOAP. Как вы можете видеть в листинге 14.30, веб-служба `CardValidator` не используется непосредственно. Потребитель добавляет экземпляр

типа CardValidatorService (который был сформирован из WSDL благодаря wsimport) с помощью ключевого слова new. Затем он должен получить прокси-класс CardValidator (getCardValidatorPort()) для вызова бизнес-методов локально. Локальные вызовы выполняются благодаря методу прокси validate(), который, в свою очередь, вызовет удаленную веб-службу, создаст запрос SOAP, выполнит маршалинг сообщений кредитной карты и т. д. Прокси находит целевую службу, потому что URL конечной точки по умолчанию встроен в файл WSDL и впоследствии интегрирован в реализацию прокси.

Листинг 14.30. Класс Java SE, программно вызывающий веб-службу SOAP

```
public class WebServiceConsumer {
  public static void main(String[] args) {

    CreditCard creditCard = new CreditCard();
    creditCard.setNumber("12341234");
    creditCard.setExpiryDate("10/12");
    creditCard.setType("VISA");
    creditCard.setControlNumber(1234);

    CardValidator cardValidator = ➥
    new CardValidatorService().getCardValidatorPort();
    cardValidator.validate(creditCard);
  }
}
```

Хотя этот код очень прост, многое происходит за кулисами. Несколько артефактов были сгенерированы из файла WSDL, чтобы выполнить данную работу. Они содержат всю информацию, необходимую для подключения к URL, где располагается веб-служба, выполняют маршалинг объекта CreditCard в XML, вызывают веб-службу через запрос SOAP и получают результат из ответа SOAP.

Вызов с помощью внедрения

С другой стороны, если ваш потребитель работает в контейнере, для получения ссылки на прокси веб-службы SOAP можно использовать внедрение. В листинге 14.31 показан простой Java-класс main, который работает в контейнере клиентского приложения (ACC) и применяет аннотацию @WebServiceRef. Он следует шаблону аннотаций @Resource и @EJB, показанному в предыдущих главах, но с точки зрения веб-служб. Когда эта аннотация применяется для атрибута (или для геттера), контейнер внедряет экземпляр прокси клиента веб-службы при инициализации приложения.

Листинг 14.31. Класс Java SE, запущенный в ACC и использующий внедрение

```
public class WebServiceConsumer {
   @WebServiceRef
   private static CardValidatorService cardValidatorService;

   public static void main(String[] args) {
     CreditCard creditCard = new CreditCard();
```

```
        creditCard.setNumber("12341234");
        creditCard.setExpiryDate("10/12");
        creditCard.setType("VISA");
        creditCard.setControlNumber(1234);

        CardValidator cardValidator = ➥
        cardValidatorService.getCardValidatorPort();
        cardValidator.validate(creditCard);
      }
}
```

Обратите внимание, что код в листинге 14.31 будет похожим, если вашим потребителем окажется EJB, сервлет или компонент-подложка JSF. Чтобы выполнить внедрение, вам нужно запустить свой код в контейнере, в противном случае аннотация `@WebServiceRef` не может быть использована.

Вызов с помощью CDI

Спецификация JAX-WS не была обновлена в Java EE 7 и, следовательно, не охватывает все возможности CDI. Например, вы не можете напрямую внедрить ссылку на прокси веб-службы с помощью аннотации `@Inject`. Однако, как видно в главе 2, вы можете воспользоваться производителем CDI, чтобы добавить такую ссылку (листинг 14.32).

Листинг 14.32. Вспомогательный класс, который выводит ссылку на веб-службу

```
public class WebServiceProducer {
  @Produces
  @WebServiceRef
  private CardValidatorService cardValidatorService;
}
```

Благодаря вспомогательному классу `WebServiceProducer` в листинге 14.32 любые EJB или сервлеты теперь могут внедрять ссылку `CardValidatorService` с помощью аннотации `@Inject` и вызова бизнес-метода (листинг 14.33). С помощью такого механизма вы даже можете использовать альтернативы CDI, чтобы перенаправить вызов вашего метода альтернативной веб-службе SOAP, как описано в главе 2.

Листинг 14.33. EJB, использующий производитель CDI для внедрения ссылки на веб-службу

```
@Stateless
public class EJBConsumerWithCDI {
  @Inject
  private CardValidatorService cardValidatorService;
  public boolean validate(CreditCard creditCard) {
    CardValidator cardValidator = cardValidatorService.getCardValidatorPort();
    return cardValidator.validate(creditCard);
  }
}
```

Все вместе

Теперь соберем все эти понятия вместе и опишем потребителя, веб-службу SOAP, затем протестируем ее, развернем в GlassFish и вызовем. Мы будем использовать аннотации JAXB и JAX-WS, а также создадим интерфейс конечной точки службы с целью для Maven `wsimport`. Чтобы написать веб-службу, требуется выполнить несколько шагов. Я продемонстрирую их, повторно рассмотрев веб-службу `CardValidator`.

Веб-служба SOAP `CardValidator` проверяет, что кредитная карта действительна. Она включает один метод, который принимает в качестве параметра объект `CreditCard`, применяет определенный алгоритм и возвращает значение `true`, если карта действительна, или `false`, если это не так. Мы протестируем эту бизнес-логику, а также проведем интеграционное тестирование (с помощью встроенного HTTP-сервера). Как только веб-служба будет протестирована и развернута на GlassFish, `wsimport` задействуется для создания всех необходимых артефактов для потребителя, который может затем вызвать веб-службу для проверки кредитных карт.

Как показано на рис. 14.6, вы будете использовать два проекта Maven: один для упаковки веб-службы в файл `.war` (`chapter14-service-1.0.war`), а другой — для упаковки потребителя в файл `.jar` (`chapter14-consumer-1.0.jar`). Эти архивы будут содержать не только разработанный вами код, но и код, сгенерированный как для потребителя (`wsimport`), так и для службы (среда выполнения JAX-WS).

Написание класса CreditCard

Класс `CreditCard`, показанный в листинге 14.34, является POJO, используемым в качестве параметра метода веб-службы `validate()`. Веб-службы SOAP обмениваются XML-сообщениями, а не объектами Java. Класс `CreditCard` имеет несколько аннотаций JAXB (например, `@XmlRootElement`), что позволяет ему быть преобразованным в XML так, чтобы его можно было передать в запросе SOAP. Объект `CreditCard` содержит некоторые базовые обязательные атрибуты, такие как номер кредитной карты, дата истечения срока действия (в формате ММ/ГГ), тип кредитной карты (Visa, Master Card, American Express и т. д.), а также контрольное число.

Листинг 14.34. Класс CreditCard с аннотациями JAXB

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class CreditCard {
   @XmlAttribute(required = true)
   private String number;
   @XmlAttribute(name = "expiry_date", required = true)
   private String expiryDate;
   @XmlAttribute(name = "control_number", required = true)
   private Integer controlNumber;
   @XmlAttribute(required = true)
   private String type;
   // Конструкторы, методы работы со свойствами
}
```

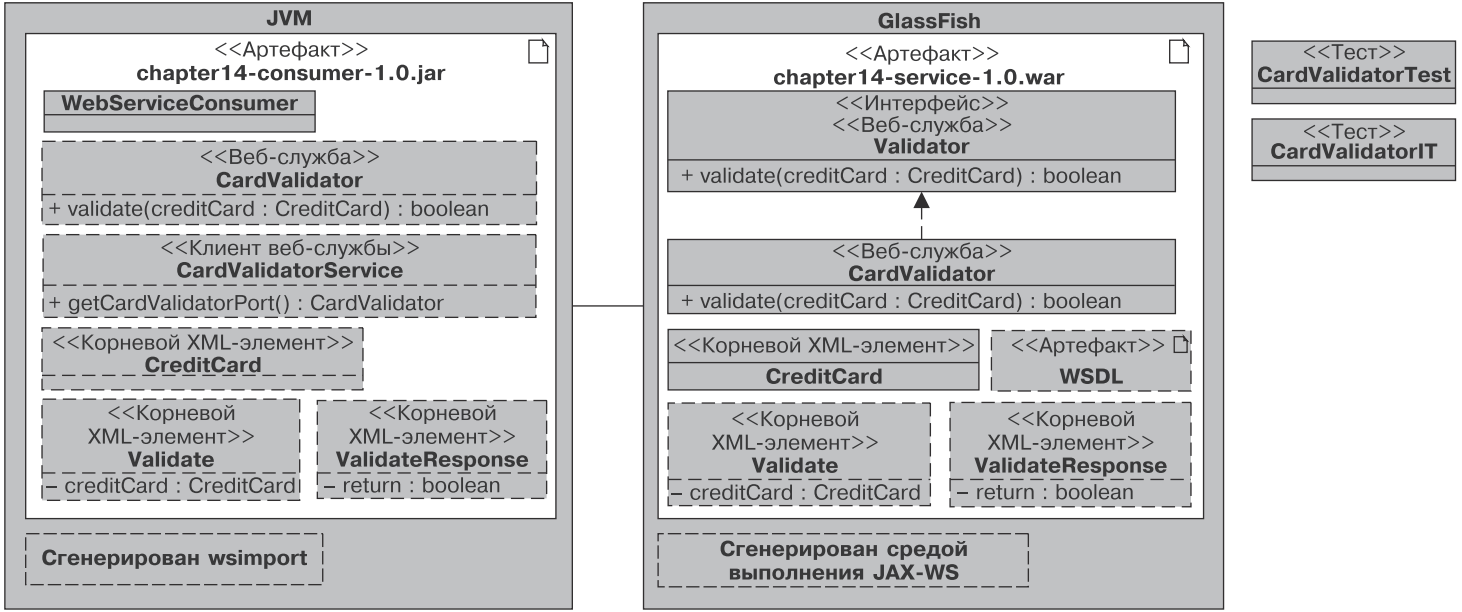


Рис. 14.6. Собираем все вместе

Написание веб-службы SOAP CardValidator

Веб-служба `CardValidator` (листинг 14.36) реализует интерфейс `Validator` (листинг 14.35), и оба имеют JAX-WS-аннотации `@WebService`. `CardValidator` является обычным POJO, поэтому считается конечной точкой-сервлетом, а не EJB (поскольку не имеет аннотаций `@Stateless` или `@Singleton`). Служба поддерживает метод `validate()`, который принимает в качестве параметра объект `CreditCard`. Алгоритм проверки действительности карты заключается в следующем: четные номера действительны, нечетные — нет. Метод возвращает значение типа `boolean`. Класс и интерфейс будут упакованы в файл с расширением `.war` (`chapter14-service-1.0.war`).

Листинг 14.35. Интерфейс веб-службы проверки

```
@WebService
public interface Validator {
  public boolean validate(CreditCard creditCard);
}
```

Листинг 14.36. Компонент веб-службы CardValidator

```
@WebService(endpointInterface = "org.agoncal.book.javaee7.chapter14.
Validator")
public class CardValidator implements Validator {
  public boolean validate(CreditCard creditCard) {

    Character lastDigit = creditCard.getNumber().charAt( ➥
                          creditCard.getNumber().length() - 1);
    if (Integer.parseInt(lastDigit.toString()) % 2 == 0) {
      return true;
    } else {
      return false;
    }
  }
}
```

Для простоты здесь не использовалось дополнительное преобразование от Java к WSDL, поэтому вы не заметили аннотаций `@WebMethod`, `@WebResult` или `@WebParam`, что позволило вам увидеть, как легко можно написать веб-службу, придерживаясь всех стандартных правил преобразования.

Написание модульного теста CardValidatorTest

Модульное тестирование веб-службы SOAP может быть довольно простым, если вы будете тестировать только бизнес-логику (то есть без таких возможностей контейнера, как внедрение, безопасность и т. д.). Модульное тестирование POJO `CardValidator` означает тестирование алгоритма проверки кредитных карт. Как показано в листинге 14.37, тест создает экземпляр POJO `CardValidator`, объект `CreditCard` с четным номером, проверяет его и указывает, что карта действительна. Тогда номер изменяется на нечетный, карта проверяется снова и указывается, что она недействительна.

Листинг 14.37. Модульный тест CardValidatorTest

```
public class CardValidatorTest {
  @Test
  public void shouldCheckCreditCardValidity() {
    CardValidator cardValidator = new CardValidator();

    CreditCard creditCard = new CreditCard("12341234", "10/10", 1234, "VISA");
    assertTrue("Кредитная карта действительна", cardValidator.
validate(creditCard));

    creditCard.setNumber("12341233");
    assertFalse("Кредитная карта недействительна", cardValidator.
validate(creditCard));
  }
}
```

Но что вы будете делать, если, например, вашей веб-службе SOAP нужно выполнить внедрение? Если просмотреть код в листинге 14.27, вы заметите, что контейнер выполняет внедрение `WebServiceContext`, поэтому компонент может проверить, имеет ли пользователь роль `Admin`. Без контейнера контекст `WebServiceContext` будет иметь значение `null` и тест будет считаться проваленным. Для решения этой проблемы вы можете либо создать видимость служб контейнера (например, внедрением `WebServiceContext`), либо протестировать свои веб-службы SOAP в режиме интеграции, то есть внутри контейнера.

Написание интеграционного теста CardValidatorIT

Интеграционное тестирование веб-службы SOAP означает ее развертывание на веб-сервер, доступ к ее WSDL с использованием всех возможностей контейнера и т. д. Ранее это было трудоемкой задачей, поскольку вы должны были запустить среду выполнения контейнера, упаковать ваше приложение, развернуть и протестировать его. Но с появлением Java SE 6 вы можете использовать встроенный веб-сервер благодаря API `javax.xml.ws.Endpoint`.

В листинге 14.38 показан класс теста `JUnit`, который публикует нашу веб-службу `CardValidator` SOAP в заданный URL (http://localhost:8080/cardValidator). После публикации (`endpoint.isPublished()`) он может получить доступ к сгенерированной WSDL через URL http://localhost:8080/cardValidator?wsdl и использовать его для создания службы (`Service.create(wsdlDocumentLocation, serviceQN)`). Метод `service.getPort` возвращает прокси клиента веб-службе SOAP, где можно затем вызывать метод (`cardValidator.validate(creditCard)`). Прежде чем закончить тестирование, необходимо свернуть веб-службу (`endpoint.stop()`).

Листинг 14.38. Интеграционный тест CardValidatorIT

```
public class CardValidatorIT {
  @Test
  public void shouldCheckCreditCardValidity() throws MalformedURLException {
    // Публикует веб-службу SOAP
    Endpoint endpoint = ➥
```

```
        Endpoint.publish("http://localhost:8080/cardValidator", ➥
                         new CardValidator());
        assertTrue(endpoint.isPublished());
        assertEquals("http://schemas.xmlsoap.org/wsdl/soap/http", ➥
                     endpoint.getBinding().getBindingID());

        // Необходимые свойства для доступа к веб-службе
        URL wsdlDocumentLocation = ➥
          new URL("http://localhost:8080/cardValidator?wsdl");
        String namespaceURI = "http://chapter14.javaee7.book.agoncal.org/";
        String servicePart = "CardValidatorService";
        String portName = "CardValidatorPort";
        QName serviceQN = new QName(namespaceURI, servicePart);
        QName portQN = new QName(namespaceURI, portName);

        // Создает объект службы
        Service service = Service.create(wsdlDocumentLocation, serviceQN);
        Validator cardValidator = service.getPort(portQN, Validator.class);

        // Вызывает веб-службу
        CreditCard creditCard = new CreditCard("12341234", "10/10", 1234, "VISA");
        assertTrue("Кредитная карта действительна", ➥
        cardValidator.validate(creditCard));

        creditCard.setNumber("12341233");
        assertFalse("Кредитная карта недействительна", ➥
        cardValidator.validate(creditCard));

        // Отключает видимость веб-службы SOAP
        endpoint.stop();
        assertFalse(endpoint.isPublished());
      }
    }
```

Теперь, когда веб-служба `CardValidator` SOAP разработана и протестирована, можно упаковать и развернуть ее с помощью GlassFish.

Компиляция, тестирование и упаковка с помощью Maven

Веб-служба `CardValidator` (интерфейс в листинге 14.35 и реализации в листинге 14.36) должна быть скомпилирована, проверена и упакована в архив `.war` (`<packaging>war</packaging>`). Файл `pom.xml` (листинг 14.39) объявляет зависимость `glassFish-embedded-all`, которая является удобным способом получить доступ ко всем спецификациям Java EE 7, в том числе к метаданным JAX-WS и веб-службам. На самом деле `glassFish-embedded-all` содержит полную реализацию GlassFish 4.0 в одном файле `.jar`, полезную для компиляции и встраивания. Установка версии 1.7 в `maven-compiler-plugin` явно указывает, что вы хотите использовать Java SE 7 (`<source>1.7</source>`).

Листинг 14.39. Файл pom.xml для компиляции, тестирования и упаковки веб-службы

```
<?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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <artifactId>chapter14</artifactId>
    <groupId>org.agoncal.book.javaee7</groupId>
    <version>1.0</version>
  </parent>

  <groupId>org.agoncal.book.javaee7.chapter14</groupId>
  <artifactId>chapter14-service</artifactId>
  <version>1.0</version>
  <packaging>war</packaging>

  <dependencies>
    <dependency>
      <groupId>org.glassfish.main.extras</groupId>
      <artifactId>glassfish-embedded-all</artifactId>
      <version>4.0</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
     <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.5.1</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-war-plugin</artifactId>
        <version>2.2</version>
```

```
        <configuration>
          <failOnMissingWebXml>false</failOnMissingWebXml>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>2.12.4</version>
        <executions>
          <execution>
            <id>integration-test</id>
            <goals>
              <goal>integration-test</goal>
              <goal>verify</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

При использовании Java EE 7 дескрипторы развертывания необязательны, поэтому вам не нужны файлы `web.xml` или `webservices.xml`. Однако, поскольку Maven по умолчанию обязывает вас добавить файл `web.xml` в файл с расширением `.war`, необходимо изменить атрибут `failOnMissingWebXml` `maven-war-plugin` на `false`, иначе Maven не сможет создать файл.

Для компиляции и упаковки веб-службы откройте в командной строке корневой каталог, содержащий файл `pom.xml`, и введите следующую команду Maven:

```
$  mvn package
```

Перейдите в папку target, где вы должны получить файл `chapter14-service-1.0.war`. Если вы откроете его, то увидите, что `Validator.class`, `CardValidator.class` и `CreditCard.class` находятся в каталоге `WEB-INF\classes`. Файл `.war` не содержит ничего более, даже файла WSDL (который будет генерироваться средой выполнения JAX-WS).

Вы можете выполнить модульный тест (см. листинг 14.37) и интеграционный тест (см. листинг 14.38) с помощью Maven Surefire и плагина Failsafe, введя следующую команду Maven:

```
$ mvn integration-test
```

Теперь, когда SOAP веб-служба `CardValidator` разработана и протестирована, мы можем развернуть ее на GlassFish.

Развертывание на GlassFish

Как только веб-служба упакована в архив `.war`, она должна быть развернута в GlassFish. Это может быть сделано в командной строке `asadmin`. Откройте консоль и перейдите в папку вашего проекта, в которой находится файл `chapter14-service-1.0.war`, убедитесь, что GlassFish работает, и введите следующую команду:

```
$ asadmin deploy chapter14-service-1.0.war
Application deployed with name chapter14-service-1.0.
Command deploy executed successfully.
```

Если развертывание прошло успешно, следующая команда должна вернуть имена развернутых компонентов и их типы:

```
$ asadmin list-components
chapter14-service-1.0 <webservices, web>
```

Интересно отметить, что GlassFish распознает веб-модуль (файл `.war` может содержать веб-страницы, сервлеты и т. д.) как веб-службу. Если вы зайдете в консоль администрирования GlassFish, показанную на рис. 14.7 (http://localhost:4848/), то увидите, что файл `chapter14-service-1.0` развернут в меню Applications (Приложения).

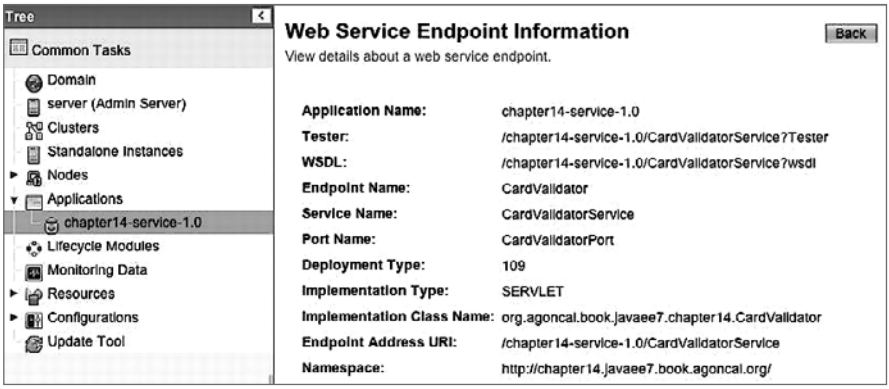


Рис. 14.7. Веб-службы, развернутые в консоли администрирования GlassFish

На этой странице, если вы выберете ссылку WSDL, откроется окно браузера, в котором будет находиться следующий URL и отобразится сгенерированный WSDL веб-службы SOAP `CardValidator`:

```
http://localhost:8080/chapter14-service-1.0/CardValidatorService?wsdl
```

Интересно отметить, что вы не создавали эту WSDL и не развертывали ее в файле `.war`. Стек Metro автоматически генерирует WSDL на основе аннотаций, содержащихся в веб-службе (а также запрос SOAP `Validate` и ответ SOAP `ValidateResponse`, показанные на рис. 14.6). Этот WSDL мы будем применять для генерации клиентского прокси, так что потребитель может получить удаленный доступ к веб-службе.

Написание класса WebServiceConsumer

Веб-служба уже развернута, GlassFish запущен и работает, и вы знаете, на каком URL сгенерирован WSDL. Благодаря этому WSDL потребитель будет иметь возможность генерировать необходимые артефакты для вызова веб-службы с помощью инструмента `wsimport`. Во-первых, напишем код потребителя, как показано в листинге 14.40.

Листинг 14.40. Класс WebServiceConsumer, вызывающий веб-службы с помощью внедрения

```
public class WebServiceConsumer {
  @WebServiceRef
  private static CardValidatorService cardValidatorService;
  public static void main(String[] args) {

    CreditCard creditCard = new CreditCard();
    creditCard.setNumber("12341234");
    creditCard.setExpiryDate("10/12");
    creditCard.setType("VISA");
    creditCard.setControlNumber(1234);

    CardValidator cardValidator = cardValidatorService.getCardValidatorPort();

    System.out.println(cardValidator.validate(creditCard));
  }
}
```

Класс `WebServiceConsumer` создает экземпляр объекта `CreditCard`, устанавливает значения некоторых его свойств, внедряет ссылку на веб-службу, вызывает метод `validate()` и отображает результат (`true` или `false` в зависимости от того, действительна ли кредитная карта). Интересно то, что потребитель не имеет ни одного из этих классов. `CardValidatorService`, `CardValidator` и `CreditCard` совершенно ему незнакомы. Код не будет скомпилирован, пока все эти классы не сгенерируются.

Создание артефактов потребителя и упаковка с помощью Maven

Перед компиляцией класса `WebServiceConsumer` вам нужно сгенерировать артефакты с помощью инструмента `wsimport`. Хорошая новость состоит в том, что Maven имеет плагин JAX-WS с целью `wsimport`, которая выполняется автоматически во время фазы жизненного цикла `generate-resources`. Maven использует богатый жизненный цикл для создания приложений. Фаза `generate-resources` применяется для генерации кода и выполняется перед компиляцией. Единственное, что нужно сделать, — это указать цели `wsimport`, где можно найти документ WSDL. У вас есть эта информация, потому что вы развернули веб-службу в GlassFish и отобразили содержимое WSDL. Он располагается здесь:

```
http://localhost:8080/chapter14-service-1.0/CardValidatorService?wsdl
```

Файл `pom.xml` в листинге 14.41 определяет необходимые зависимости, `glassfish-embedded-all` версии 4.0, а также версию JDK (1.7). Класс `WebServiceConsumer` упакован в файл `chapter14-consumer-1.0.jar`. Как и любой файл с расширением `.jar`, он включает файл `META-INF/MANIFEST.MF`, который может быть использован для определения метаданных. Вы можете добавить элемент `Main-Class` в манифест, указывая на класс `WebServiceConsumer`. Это позволит выполнить файл `.jar` (например, с помощью команды `java -jar`).

Листинг 14.41. Файл pom.xml генерирует и упаковывает артефакты потребителя

```
<?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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <artifactId>chapter14</artifactId>
    <groupId>org.agoncal.book.javaee7</groupId>
    <version>1.0</version>
  </parent>
  <groupId>org.agoncal.book.javaee7.chapter14</groupId>
  <artifactId>chapter14-consumer</artifactId>
  <packaging>jar</packaging>
  <version>1.0</version>
  <dependencies>
    <dependency>
      <groupId>org.glassfish.main.extras</groupId>
      <artifactId>glassfish-embedded-all</artifactId>
      <version>4.0</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.4</version>
        <configuration>
          <archive>
            <manifest>

<mainClass>org.agoncal.book.javaee7.chapter14.WebServiceConsumer</mainClass>
            </manifest>
          </archive>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.jvnet.jax-ws-commons</groupId>
        <artifactId>jaxws-maven-plugin</artifactId>
        <version>2.2</version>
        <executions>
          <execution>
            <goals>
              <goal>wsimport</goal>
            </goals>
            <configuration>
              <wsdlUrls>
                <wsdlUrl>
```

```
http://localhost:8080/chapter14-service-1.0/CardValidatorService?wsdl
                </wsdlUrl>
              </wsdlUrls>
              <keep>true</keep>
            </configuration>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.5.1</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Для лучшего понимания того, что происходит за кулисами, сначала сгенерируйте артефакты, введя такую команду Maven:

```
$  mvn generate-sources
```

Эта команда выполняет фазу жизненного цикла Maven `generate-sources` и, следовательно, определенную с ним цель `wsimport`. Цель подключается к URL веб-службы WSDL, загружает ее и генерирует все артефакты. Вот вывод команды Maven:

```
[INFO] --- jaxws-maven-plugin:2.2:wsimport (default) @ chapter14-consumer ---
[INFO] Processing: http://localhost:8080/chapter14-service-1.0/
CardValidatorService?wsdl ➥
[INFO] jaxws:wsimport args: [-keep, ➥
chapter14-consumer/target/generated-sources/wsimport, ➥
  -encoding, UTF-8, -Xnocompile, ➥
  http://localhost:8080/chapter14-service-1.0/CardValidatorService?wsdl]
parsing WSDL...
Generating code...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

Если вам интересно, вы можете перейти в каталог `target/generated-sources/wsimport` и проверить сгенерированные классы. Вы, конечно, найдете там классы `CardValidator`, `CardValidatorService` и `CreditCard`, а также один класс для запроса SOAP (`Validate`) и еще один для ответа (`ValidateResponse`). Классы полны аннотациями JAXB и JAX-WS, поскольку они выполняют маршалинг объекта типа `CreditCard` и подключение к удаленной веб-службе. Вам не придется беспокоиться о сгенерированном коде. Файл `.jar` может быть скомпилирован и упакован с помощью такой команды:

```
$ mvn package
```

Она создаст файл chapter14-consumer-1.0.jar, который будет содержать написанный вами класс WebServiceConsumer, а также все сгенерированные классы (см. рис. 14.6, чтобы иметь полное представление обо всех созданных классах). Этот файл самостоятелен и может быть выполнен для вызова веб-службы.

Запуск класса WebServiceConsumer

Помните, что класс WebServiceConsumer использует аннотацию @WebServiceRef, чтобы в него можно было внедрить ссылку на интерфейс конечной точки веб-службы. Это означает, что код должен быть выполнен в контейнере клиентского приложения (ACC). Помните также, что файл chapter14-consumer-1.0.jar является исполняемым, поскольку вы добавили элемент Main-Class в файл MANIFEST.MF. Единственное, что вам нужно сделать, — вызвать утилиту appclient, которая поставляется вместе с GlassFish, и передать ей файл .jar следующим образом:

```
$ appclient -client chapter14-consumer-1.0.jar
```

Она вызовет веб-службу через HTTP и выдаст ответ, говоря вам о том, действительна ли кредитная карта.

Резюме

Обмен данными между компаниями должен быть безопасным, надежным, совместимым и выполняться с помощью транзакций. Вот почему существовали такие технологии, как CORBA, DCOM, RPC или RMI. По мере широкого внедрения HTTP несколько компаний (W3C, OASIS) разработали веб-службы SOAP как слабосвязанный (XML) стандартный способ общения по сети для предприятий. Сегодня многие организации широко используют веб-службы SOAP для интеграции приложений, запущенных различными внешними организациями (Интернет) или внутренними отделами (интранет).

В этой главе были описаны некоторые связанные с веб-службами SOAP стандарты (WSDL, SOAP и т. д.). Кроме того, мы сосредоточились на таких спецификациях Java EE, как JAX-WS, JAXB, WS-Metadata и т. д. Эти спецификации жизненно важны, если вы хотите скрыть сложность сети и упростить разработку. Оставив эти спецификации позади, JAX-WS использует только небольшой набор аннотаций для регулировки преобразования Java в WSDL.

В конце главы мы рассмотрели пример того, как написать веб-службу, скомпилировать, протестировать (модульное и интеграционное тестирование) и упаковать ее с помощью Maven. Благодаря WSDL и некоторым инструментам можно создать артефакты потребителя для удаленного вызова веб-служб SOAP.

Amazon, eBay, Google, Yahoo! и многие другие компании предоставляют своим клиентам веб-службы SOAP. Но в последние годы они все перешли на веб-службы RESTful (описанные в следующей главе), в основном из соображений производительности.

Глава 15
Веб-службы в стиле REST

Веб-службы SOAP (SOAP, WSDL, WS-*), описанные в предыдущей главе, обеспечивают интероперабельность как в интеграции обмена сообщениями, так и в стиле межпроцессной коммуникации. Веб-службы SOAP, по-прежнему широко используемые в B2B-индустрии, так и не достигли того пика, о перспективе которого в свое время много говорили в Интернете. С развитием Web 2.0 появились новые веб-фреймворки, вместе с которыми стали более востребованы гибкая веб-разработка и пользовательские интерфейсы с обратной связью. В нашу жизнь прочно вошли мобильные устройства, на которых применяются самые разные оригинальные и сетевые приложения для агрегирования данных. HTML5 и JavaScript произвели настоящую революцию в практике интернет-серфинга. В таких условиях значительно повысилась популярность нового вида веб-служб. Это веб-службы с передачей состояния представления, известные как RESTful. Учитывая это, многие ключевые игроки современной Сети, в частности Amazon, eBay, Google и Yahoo!, отказались от прежних веб-служб SOAP в пользу RESTful-служб, ориентированных на работу с ресурсами.

Передача состояния представления (REST) — это стиль архитектуры, основанный на принципах работы Сети. Говоря о веб-службах, можно утверждать, что в таком стиле мы стараемся максимально учесть в работе служб характерные черты Сети. Чтобы разработать веб-службу с передачей состояния представления, необходимо хорошо разбираться в протоколе передачи гипертекста (HTTP) и уникальных идентификаторах ресурсов (URI), а также придерживаться определенных принципов проектирования. В сущности, это означает, что каждый URI является представлением определенного объекта. С данным объектом можно взаимодействовать с помощью запросов HTTP `GET` (для получения его содержимого), `DELETE`, `POST` (для создания содержимого) или `PUT` (для обновления содержимого).

RESTful-архитектуры быстро приобрели популярность, так как они основаны на очень надежном протоколе передачи данных: HTTP. Веб-службы с передачей состояния представления снижают жесткость связи между клиентом и сервером, что позволяет впоследствии развернуть целый REST-интерфейс, не нарушая при этом работу имеющихся клиентов. Веб-службы RESTful, как и протокол, на котором они основаны, не сохраняют состояния и могут задействовать HTTP-кэш и прокси-серверы, чтобы вам было проще справляться с высокими нагрузками, а система могла легко масштабироваться. Более того, выстраивать такие службы сравнительно просто, так как не требуется никакого специального инструментария (в отличие, например, от WSDL).

В начале этой главы мы рассмотрим несколько концепций, которые помогут вам понять, что такое REST. Затем мы объединим весь изученный материал, научимся писать веб-службы с передачей состояния представления и использовать их в работе.

Понятие о веб-службах RESTful

Веб-службы SOAP создавались в расчете на использование сразу нескольких протоколов передачи данных, в том числе HTTP. В результате они задействовали лишь небольшую часть возможностей этого протокола. Напротив, в веб-службах стиля RESTful протокол HTTP имеет центральное значение: они используют его богатые возможности практически в полном объеме. В архитектурах стиля REST каждая информационная единица считается ресурсом, и эти ресурсы адресуются с помощью универсальных идентификаторов (URI). Как правило, идентификаторы представляют собой веб-ссылки. Действия над ресурсами выполняются при использовании ограниченного набора четко определенных операций. REST как стиль клиент-серверной архитектуры разработан для обмена представлениями этих ресурсов с применением определенного интерфейса и протокола. Благодаря данным принципам приложения в стиле RESTful получаются простыми и легковесными, но при этом отличаются высокой производительностью.

Практика работы в браузере

Поскольку REST построен на базе Сети, для более полного понимания этого стиля приведу аналогию с работой в Интернете (через браузер). Например, вас интересует список технических книг по языку Java на русском языке. Как вы его найдете? Откроете в браузере сайт издательства «Питер» (www.piter.com). На этой странице вряд ли найдется именно та информация, которую вы искали, но с нее вам будет проще попасть к списку книг по Java, вышедшему в нашем издательстве. На домашней странице вы найдете строку для поиска по всем заголовкам издательства «Питер». Набираете в строке слово `Java` и оказываетесь на следующей странице: http://www.piter.com/search/index.php?q=Java&s.x=21&s.y=12&s=%CF%EE%E8%F1%EA&ext=&inSaleOnly=0&searchAs=0&orderBy=r&order=ASC&itemsPerPage=10. Здесь вашему вниманию предлагается список из десяти книг, но на самом деле их гораздо больше (видите внизу список страниц?).

Итак, вы сохраняете эту ссылку в вашем любимом менеджере закладок и начинаете просматривать список. Вас заинтересовала книга М. Вербурга и М. Эванса «Java. Новое поколение разработки». Нажимаете название-гиперссылку и попадаете на отдельную страницу с описанием книги (http://www.piter.com/book.phtml?9785496 00544). Здесь вы можете прочесть аннотацию и просмотреть отрывок из книги.

Примерно так строится наша повседневная работа с браузером. В REST при работе со службами применяются точно такие же принципы; книги, результаты поиска, оглавление, обложка и т. п. трактуются как ресурсы.

Ресурсы и URI

В RESTful-архитектурах ресурсам отводится центральная роль. В предыдущем разделе я указал, что ресурс — это любая сущность, на которую клиент может поставить ссылку или с которой может попытаться взаимодействовать. Словом, любая информационная единица, стоящая того, чтобы на нее сделать ссылку (книга, результат поиска, оглавление). Ресурс может храниться в файле, базе данных фактически по любому адресу. Следует максимально избегать представления аб-

страктных концепций в качестве ресурсов; в таких случаях лучше удовлетвориться простыми объектами. Итак, какие ресурсы мы можем встретить в приложении, предназначенном для продажи книг?

- Список книг по Java.
- Книга «Java. Новое поколение разработки».
- Оглавление этой книги.

В Сети ресурсы определяются по URI, который состоит из имени и структурированного адреса, указывающего, где находится данный ресурс. Существуют различные разновидности URI: WWW-адреса, универсальные идентификаторы документов, универсальные идентификаторы ресурсов и, наконец, комбинации единых указателей ресурсов (URL) и единообразных имен ресурсов (URN). Примеры ресурсов и URI перечислены в табл. 15.1.

Таблица 15.1. Примеры ресурсов и URI

Ресурс	URI
Каталог компьютерной литературы издательства «Питер»	http://books.piter.com/collection/kompyutery-i-internet
Обложка книги «Java. Новое поколение разработки»	http://static2.insales.ru/images/products/1/3744/25513632/49600544.jpg
Информация о сотрудничестве с издательством «Питер»	http://books.piter.com/page/sotrudnichestvo
Погода в Санкт-Петербурге	http://pogoda.ru/Sankt-Peterburg/
Интересные фотографии с Flickr по состоянию на 1 января 2014 года	http://www.flickr.com/explore/2014/01/01
Интересные фотографии с Flickr за последнюю неделю	http://www.flickr.com/explore/interesting/7days
Список приключенческих фильмов	http://www.movies.com/categories/adventure

URI должны максимально четко описывать содержащуюся по ним информацию, а также указывать на уникальный ресурс. Обратите внимание: разные URI, идентифицирующие самостоятельные ресурсы, могут вести к одному и тому же адресу. Стандартный формат URI таков:

```
http://host:port/path?queryString#fragment
```

`http` — это протокол, `host` — это DNS-имя или IP-адрес, а часть `port` необязательна. `Path` — это множество символов, образующих текстовый сегмент; разделительным знаком в данном случае является /. Далее идет строка запроса, также являющаяся необязательной (это список параметров, представленных в форме «имя/значение», где разделительным символом между такими парами является уже `&`). Последняя часть, отделенная символом #, — это фрагмент, указывающий на конкретное место в документе. Следующий URI предоставляет информацию о погоде в Лиссабоне 1 января 2014 года:

```
http://www.weather.com:8080/weather/2014/01/01?location=Lisbon,Portugal&time=
morning
```

Представления

Итак, мы выяснили, что такое ресурсы и где они находятся. Но что такое *представление* того ресурса, на который указывает URI? Существует ли несколько представлений для одного ресурса? Кстати, вам могут понадобиться разные представления одного и того же ресурса: в виде текста, в форматах JSON, XML, PDF, JPG и др. Работая с ресурсом, клиент всегда имеет дело с тем или иным его представлением; сам ресурс остается на сервере. Представление — это любая полезная информация о состоянии ресурса. Например, у списка книг по Java на сайте издательства Apress есть как минимум два представления:

- HTML-страница, отображаемая в браузере: http://www.apress.com/java;
- файл с книгами в формате «значения, разделенные запятыми» (CSV): http://www.apress.com/resource/csv/bookcategory?cat=32.

Как выбрать одно из возможных представлений заданного ресурса? Есть два решения. Сервис может давать отдельный URI на каждое представление, как показано выше. Но две эти ссылки действительно очень разные и почти не напоминают друг друга. Ниже приведен более красивый набор ссылок:

- http://www.apress.com/java;
- http://www.apress.com/java/csv;
- http://www.apress.com/java/xml.

Первый URI по умолчанию используется для представления ресурса, а в дополнительных вариантах URI приведен формат соответствующего представления `/csv` (следует понимать: `text/csv`), `/xml`, `/pdf` и т. д.

Другое решение — предложить один URI для всех вариантов представления (например, http://www.apress.com/java) и задействовать механизм, называемый *согласованием содержимого*. Мы подробно поговорим о нем ниже в этой главе. Например, URI может существовать в виде человекочитаемого и машинообрабатываемого представлений.

Адресуемость

Важный принцип, которого следует придерживаться при проектировании веб-служб в стиле RESTful, — это адресуемость. Ваша веб-служба должна делать приложение максимально адресуемым. Это означает, что каждый значимый информационный элемент в вашем приложении должен быть ресурсом и иметь URI, по которому можно легко попасть к этому ресурсу. URI — единственный информационный фрагмент, который необходимо опубликовать для обеспечения доступа к ресурсу. В таком случае ваш бизнес-партнер сможет не гадать, где находится интересующая информация, а просто и быстро открыть нужный ресурс.

Допустим, вы пытаетесь справиться с ошибкой в вашем приложении. Вы провели кое-какие тесты и в результате пришли к строке 42 в коде класса `CreditCardValidator.java` — именно здесь и возникает ошибка. Поскольку вы не отвечаете за данную область приложения, то хотите переадресовать эту проблему квалифицированному коллеге, который сможет с ней справиться. Как вы укажете ему эту зловредную строку? Можно сказать: «посмотри строку 42 в классе `CreditCardValidator`», но если ваш

исходный код адресуется через браузер репозитория, то вы можете сохранить URI этой строки и указать ее в отчете об ошибке. Здесь мы подходим к проблеме определения детализации ваших REST-ресурсов в приложении: адресуемость прослеживается на уровне строки, метода, класса, уровня и т. д.

Уникальные URI обеспечивают связываемость ваших ресурсов, а поскольку они предоставляются через единообразный интерфейс, все точно знают, как с ними взаимодействовать. Соответственно, пользователи вполне могут работать с вашим приложением таким образом, о котором вы даже не задумывались.

Связность

В теории графов граф называется *связным*, если между любой парой его вершин существует как минимум один путь. Граф называется *сильно связным*, если он содержит прямой путь от вершины *u* к *v* и от вершины *v* к *u* для любой пары вершин *u*, *v*. Согласно принципам REST ресурсы должны быть связаны максимально сильно.

Опять же это утверждение появилось в результате причин успеха Интернета. На всех веб-страницах есть ссылки для перехода на другие страницы, они расположены логичным и интуитивно понятным образом, и весь Интернет является очень хорошо связанным. Если между двумя ресурсами существует четкая взаимосвязь, то они должны быть соединены. REST постулирует, что веб-службы также должны пользоваться преимуществами гипермедийности, информируя клиента о том, какие ресурсы доступны и куда двигаться дальше. REST стимулирует легкую обнаруживаемость служб. Располагая единственным URI, пользовательский агент, который обращается к хорошо связанной веб-службе, может обнаружить все доступные действия, ресурсы, их различные представления и т. д.

Например, когда пользовательский агент (браузер) ищет представление CD (листинг 15.1), в этом представлении может содержаться не только имя исполнителя, но и ссылка на его биографию (в виде URI). Пользовательский агент определяет, переходить по этой ссылке или нет. Представление также может содержать и ссылки на другие представления ресурса или доступных действий.

Листинг 15.1. Представление CD, связанное с другими службами

```
<cd>
  <title>Элла и Луи</title>
  <year ref="http://music.com/year/1956">1956</year>
  <artist ref="http://music.com/artists/123">Элла Фицжеральд</artist>
  <artist ref="http://music.com/artists/456">Луи Армстронг</artist>
  <link rel="self" type="text/json" href="http://music.com/album/789"/>
  <link rel="self" type="text/xml" href="http://music.com/album/789"/>
  <link rel="http://music.com/album/comments" type="text/xml" ➥
        href="http://music.com/album/789/comments"/>
  </cd>
```

Еще одна важнейшая особенность гипермедийности — это состояние приложения, которое определяется в контексте гипермедиа. Проще говоря, сам тот факт, что веб-служба предоставляет набор ссылок, позволяет клиенту переводить приложение из одного состояния в другое. И для этого достаточно просто перейти по ссылке.

В предыдущем фрагменте XML-кода клиент может изменить состояние приложения, оставив комментарий об альбоме. Список комментариев к альбому — это ресурс, адресуемый по URI http://music.com/album/789/comments. Поскольку эта веб-служба использует единообразный интерфейс, клиенту достаточно знать формат URI, доступные типы контента и формат данных — и он также узнает, как следует оперировать этой информацией. Метод `GET` будет получать список имеющихся комментариев, метод `PUT` — обновлять комментарий и т. д. Исходя из единственного запроса, клиент может предпринять множество действий: гипермедиа управляет состоянием приложения.

Единообразный интерфейс

Одно из важнейших ограничений, оформляющих архитектуру RESTful, — использование единообразного интерфейса для управления всеми вашими ресурсами. Выберите подходящий вам интерфейс, но применяйте его одинаково во всех контекстах, от ресурса к ресурсу, от службы к службе. Никогда не отступайте от выбранного принципа, не изменяйте исходного значения. При использовании единообразного интерфейса «общая архитектура системы упрощается, а также повышается прозрачность взаимодействий» (Рой Томас Филдинг, «Архитектурные стили и проектирование архитектур программных систем, поддерживающих работу в сети» (*Architectural Styles and the Design of Network-based Software Architectures*)). Ваши службы становятся частью сообщества служб, использующих совершенно одинаковую семантику.

Основным веб-протоколом де-факто является HTTP. Это основанный на работе с документами стандартизированный протокол, действующий по принципу «запрос — ответ», обеспечивающий взаимодействие клиента и сервера. HTTP — это единообразный интерфейс веб-служб с передачей состояния представления. Веб-службы, построенные на базе SOAP, WSDL и других стандартов WS-*, также используют HTTP в качестве «транспортного уровня», но задействуют лишь немногие из его возможностей (ведь SOAP-службы могут работать и с другими транспортными протоколами, например с JMS). При использовании этих альтернатив приходится выяснять семантику службы, анализируя WSDL, а потом вызывать нужные методы. Веб-службы с передачей состояния представления имеют единообразный интерфейс (HTTP-методы и URI), поэтому если вы знаете, где находится ресурс (URI), то можете вызвать HTTP-метод (`GET`, `POST` и т. д.).

Единообразный интерфейс не только способствует узнаваемости, но и улучшает интероперабельность между приложениями. Протокол HTTP широко поддерживается, а связанных с ним клиентских библиотек так много, что вам вряд ли придется сталкиваться с проблемами при обмене информацией.

Отсутствие сохранения состояния

Последняя из важнейших черт REST — отсутствие сохранения состояния. Она означает, что каждый HTTP-запрос происходит в полной изоляции от других, так как серверу никогда не приходится отслеживать запросы, выполненные ранее. Эту мысль можно сформулировать и яснее: состояние ресурса и состояние приложения обычно различаются. Состояние ресурса должно находиться на сервере и совместно исполь-

зоваться всеми, а состояние приложения — оставаться на клиенте и принадлежать только ему. Возвращаясь к примеру из листинга 15.1, отмечу: состояние приложения заключается в том, что клиент выбрал представление альбома «Элла и Луи», но сервер не должен об этом знать. Состояние ресурса — это сама информация об альбоме, и сервер, конечно, должен обеспечивать ее поддержку. Клиент может изменить состояние приложения. Если корзина для заказов — это ресурс с ограниченным доступом (то есть доступный всего одному клиенту), то приложение должно отслеживать идентификатор корзины заказов в рамках клиентского сеанса.

Отсутствие сохранения состояния связано с многочисленными преимуществами. В частности, эта черта улучшает масштабируемость: не приходится обрабатывать информацию о сеансах, маршрутизировать последующие запросы на тот же сервер, что и предыдущие (балансирование нагрузки), обрабатывать отказы (например, связанные с перебоями в работе) и т. д. Если вам требуется сохранять состояние, то клиенту приходится выполнять дополнительную работу, связанную с этим.

HTTP

HTTP — это протокол для распределенных совместно работающих гипермедийных информационных систем. Именно он породил Всемирную паутину с ее URI-адресами, язык HTML, а также программы-браузеры. HTTP стал плодом скоординированной работы организаций W3C и IETF. Протокол был разработан на основе нескольких «рабочих предложений» (RFC), среди которых следует особо отметить RFC 216, где определяется версия этого протокола HTTP 1.1.

Запрос и ответ

Протокол HTTP основан на запросах и ответах, которыми обмениваются клиент и сервер. Клиент отправляет на сервер запрос и ожидает получить от него ответ (рис. 15.1). Сообщения, которыми они обмениваются, состоят из конверта и тела. Тело также называется полезной информацией или сущностью.

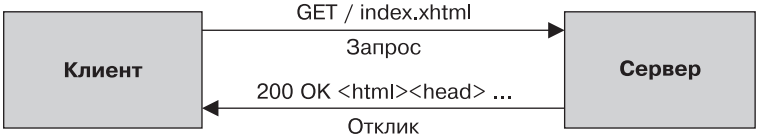


Рис. 15.1. Запрос и ответ в протоколе HTTP

Если вы переходите по страницам сайта Apress, то видите в браузере только веб-страницы, но не технические детали запросов и ответов HTTP. Чтобы понять, что же происходит «за кулисами», вы можете воспользоваться другим инструментом, например cURL. Вот информация о запросе, отправленном на сервер при переходе по ссылке http://www.apress.com/java?limit=all&mode=list:

```
$ curl -v -X GET http://www.apress.com/java?limit=all&mode=list
> GET /java?limit=all&mode=list HTTP/1.1
> User-Agent: curl/7.23.1 (x86_64-apple-darwin11.2.0) libcurl/7.23.1
zlib/1.2.5
```

```
> Host: www.apress.com
> Accept: */*
```

В этом запросе присутствует несколько информационных фрагментов, отсылаемых с клиента на сервер:

- HTTP-метод, в данном случае `GET`;
- путь, в данном случае `/java?limit=all&mode=list`;
- другие заголовки запроса (`User-Agent`).

Обратите внимание: в этом запросе отсутствует тело сообщения. На самом деле запросы `GET` в принципе не имеют такого тела. На вышеуказанный запрос сервер выдаст следующий ответ:

```
< HTTP/1.1200 OK
< Date : Sat, 17 Nov 201217:42:15 GMT
< Server: Apache/2.2.3 (Red Hat)
< X-Powered-By: PHP/5.2.17
< Vary: Accept-Encoding,User-Agent
< Transfer-Encoding: chunked
< Content-Type : text/html; charset=UTF-8
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
  <title>
  ...
```

В состав этого ответа входит следующая информация:

- код отклика — в данном случае `200-OK`;
- несколько заголовков отклика — в предыдущем коде содержатся заголовки отклика `Date`, `Server`, `Content-Type`. В данном случае тип содержимого — `text/html`, но это может быть и информация в других форматах, например в XML (`application/xml`) или в форматах изображений (`image/jpeg`);
- тело объекта или представление — содержимым возвращенной веб-страницы в данном примере является тело объекта (здесь я продемонстрировал фрагмент HTML-страницы).

ПРИМЕЧАНИЕ

cURL (http://curl.haxx.se/) — это инструмент командной строки для передачи файлов с помощью URL-синтаксиса по различным протоколам — в частности, HTTP, FTP, SFTP, SCP и др. С ее помощью можно отправлять HTTP-команды, изменять HTTP-заголовки и т. д. Этот инструмент очень удобен для имитации пользовательских действий в браузере.

Заголовки

Поля HTTP-заголовков — это компоненты заголовка сообщения, содержащиеся в запросах и ответах. Файлы заголовков состоят из разделенных двоеточиями пар «имя/значение», которые записываются обычным текстом, завершаются переходом на новую строку и последовательностью символов перехода на новую строку. Основной набор таких полей стандартизирован организацией IETF и должен быть

внедрен во всех реализациях, соответствующих протоколу HTTP. Но при необходимости в каждом приложении могут определяться и дополнительные имена полей. В табл. 15.2 приведено несколько распространенных заголовочных значений, которые могут встретиться вам в запросе или ответе.

Таблица 15.2. Распространенные значения заголовков

Имя заголовка	Описание
Accept	Допустимые типы содержимого (например, text/plain)
Accept-Charset	Допустимые наборы символов (например, utf-8)
Accept-Encoding	Допустимые варианты кодировки (например, gzip, deflate)
Accept-Language	Допустимый язык отклика (en-US)
Cookie	Файл HTTP-cookie, ранее отосланный сервером
Content-Length	Длина тела запроса в байтах
Content-Type	MIME-тип тела запроса (например, text/xml)
Date	Дата и время отправки сообщения
ETag	Идентификатор конкретной версии ресурса (например, 8af7ad3082f20958)
If-Match	Действие производится лишь в том случае, если клиент предоставил объект, совпадающий с таким же объектом на сервере
If-Modified-Since	Допускает возврат кода состояния 304 — Не изменялось, если контент не изменялся с указанной даты
User-Agent	Строка, соответствующая пользовательскому агенту (например, Mozilla/5.0)

HTTP-методы

Сеть состоит из четко идентифицированных ресурсов, связанных друг с другом и доступных благодаря простым HTTP-запросам. Основными стандартизированными типами HTTP-запросов являются `GET`, `POST`, `PUT`, `DELETE`. Они также именуются операциями, командами или методами. HTTP определяет и другие, реже используемые, методы: `HEAD`, `TRACE`, `OPTIONS`, `CONNECT`.

- `GET` — это обычная операция считывания, которая запрашивает представление ресурса. Операция `GET` должна реализовываться безопасным образом, то есть не изменять состояние ресурса. Кроме того, запрос `GET` должен быть идемпотентным. Это означает, что он должен оставлять ресурс в одном и том же состоянии, независимо от того, сколько раз он вызывался: единожды, дважды или больше. Безопасность и идемпотентность обеспечивают значительную стабильность. Когда клиент не получает ответа (например, из-за отказа сети), он может обновить свои запросы, и эти новые запросы будут ожидать того же ответа, который был бы получен при первой попытке, если бы все пошло нормально. При этом состояние ресурса на сервере не повреждается.
- Если вызвать метод `POST`, имея определенное представление (текст, XML и т. д.), то будет создан новый ресурс, идентифицируемый по URI, заявленному в запросе. Например, `POST` может вызываться при прикреплении сообщения к файлу

журнала, записи комментария в блог, занесении книги в список и т. д. Следовательно, метод POST не является ни безопасным (состояние ресурса изменяется), ни идемпотентным (при двукратной отправке этого запроса мы получим два новых подчиненных объекта). Если ресурс был создан на исходном сервере, то в ответ должен быть получен код состояния 201 – Создан. В большинстве браузеров генерируются лишь два вида запросов — GET и POST.

- Запрос PUT предназначен для обновления состояния ресурса, сохраненного по указанному URI. Если URI запроса ссылается на несуществующий ресурс, то ресурс будет создан именно по этому URI. Метод PUT может применяться, например, при обновлении цены на книгу или адреса клиента. Метод PUT небезопасен (так как состояние ресурса обновляется), но идемпотентен: можно многократно отослать один и тот же запрос PUT, а конечное состояние ресурса останется неизменным.
- Запрос DELETE удаляет ресурс. Ответом на DELETE может быть статусное сообщение, получаемое в теле более подробного сообщения, либо отсутствие статуса вообще. Запрос DELETE также является идемпотентным, но не является безопасным.
- Как было указано выше, существуют и другие методы HTTP, которые используются реже.
  - Запрос HEAD практически идентичен GET с той оговоркой, что сервер не возвращает в ответ тело сообщения. HEAD может быть полезен при проверке валидности ссылки или размера объекта без передачи этой ссылки или объекта.
  - Когда сервер получает от клиента запрос TRACE, он отражает полученный запрос на клиент. TRACE может быть полезен для проверки того, какую информацию добавляет к запросу (или изменяет) промежуточный сервер, прокси-сервер или брандмауэр.
  - Запрос OPTIONS представляет собой запрос информации о возможностях коммуникации, доступных в той цепочке запросов/ответов, на которую указывает данный URI. Этот метод позволяет клиенту определять варианты и/или требования, связанные с ресурсом, либо возможности сервера. При этом не предполагается никаких действий ресурса или операции по получению ресурса.
  - CONNECT используется с прокси-серверами, которые могут динамически переключаться на работу в режиме туннеля (в таком случае протокол HTTP служит оберткой для различных сетевых протоколов).

Согласование содержимого

Согласование содержимого[1] — механизм, описанный в разделе 12 стандарта HTTP. Согласование содержимого определяется как «механизм автоматического определения необходимого представления при наличии нескольких разнотипных вариантов представлений». Потребности, пожелания и способности клиента могут отличаться; наилучшее представление для японского пользователя мобильного устройства вполне может не совпадать с оптимальным вариантом для американского читателя новостной ленты.

[1] Существует также перевод «обсуждение содержимого». — *Примеч. пер.*

Согласование содержимого основано, в частности (но не только), на использовании заголовков HTTP-запросов `Accept`, `Accept-Charset`, `Accept-Encoding`, `Accept-Language` и `User-Agent`. Например, чтобы получить представление списка книг издательства «Питер» по теме Java, клиентское приложение (пользовательский агент) запросит страницу http://books.piter.com/search?q=Java, для заголовка `Accept` которой будет задано значение `text/csv`. Легко представить, что в другом случае на основе заголовка `Accept-Language` сервер выберет подходящий документ в формате CSV для отображения на заданном языке (например, японском или английском).

Типы содержимого

HTTP использует медиатипы, применяемые в Интернете (более известные как MIME-типы). Они записываются в полях заголовков `Content-Type` и `Accept` для обеспечения открытой и расширяемой типизации данных и согласования типов. Медиатипы Интернета подразделяются на пять самостоятельных основных категорий: `text`, `image`, `audio`, `video` и `application`. Эти типы далее классифицируются на несколько подтипов (`text/plain`, `text/xml`, `text/xhtml` и т. д.). Рассмотрим наиболее распространенные общеупотребительные типы содержимого:

- `text/plain` — используется по умолчанию, им записываются простые текстовые сообщения;
- `text/html` — очень часто применяется в наших браузерах. Данный тип информирует пользовательский агент, что содержимое — это веб-страница на языке HTML;
- `image/gif`, `image/jpeg`, `image/png` — этот обобщающий тип, соответствующий изображениям нескольких типов, требует устройства для отображения (графический дисплей, графический принтер) и для просмотра информации;
- `text/xml`, `application/xml` — формат, используемый для обмена XML-сообщениями;
- `application/json` — объектная нотация JavaScript (JSON). Это легковесный текстовый формат для обмена данными, не зависящий от конкретного языка программирования (см. главу 12).

Коды состояния

При получении каждого ответа с ним ассоциируется HTTP-код. Спецификация определяет около 60 таких кодов. Элемент `Status-Code` — это трехзначное целое число, описывающее контекст, в котором произошел ответ. Код состояния входит в состав обертки сообщения. Первая цифра указывает один из классов ответа:

- `1xx` — информационный: запрос получен, процесс продолжает работу;
- `2xx` — успех: действие было успешно получено, интерпретировано и принято;
- `3xx` — перенаправление: необходимо выполнить дополнительные действия для удовлетворения запроса;
- `4xx` — клиентская ошибка: запрос содержит ошибочный синтаксис или не может быть удовлетворен;
- `5xx` — серверная ошибка: серверу не удалось выполнить запрос, который на первый взгляд был совершенно правильным.

В табл. 15.3 перечислены коды состояния, которые уже могли вам встречаться.

Таблица 15.3. Коды состояния HTTP

Код	Описание
100 — Продолжить	Сервер получил заголовки запроса, далее клиент должен прислать тело запроса
101 — Переключение протоколов	Запрашивающая сторона просит сервер переключиться на другой протокол, и сервер соглашается это сделать
200 — Хорошо	Запрос завершен успешно. Тело объекта, если таковое имеется, содержит представление ресурса
201 — Создан	Запрос был выполнен, что привело к созданию нового ресурса
204 — Нет содержимого	Сервер успешно обработал запрос, но не возвратил никакого содержимого
206 — Частичное содержимое	Сервер доставил лишь часть содержимого; это было обусловлено диапазонами заголовков, заданными клиентом
301 — Перемещено навсегда	Запрошенному ресурсу был присвоен новый постоянный URI, и при любых последующих ссылках на данный ресурс следует использовать один из возвращенных URI
304 — Не изменялось	Указывает, что ресурс не изменялся с момента последнего запроса
307 — Временное перенаправление	Запрос должен быть повторен с другим URI; однако в будущих запросах следует по-прежнему использовать исходный URI
308 — Постоянное перенаправление	Этот, а также все последующие запросы должны направляться уже по новому URI
400 — Неверный запрос	Запрос не может быть выполнен из-за синтаксической ошибки
401 — Не авторизован	Похож на 403, но указывает, что для запроса требовалась аутентификация и она не была выполнена
403 — Запрещено	Запрос был допустимым, но сервер отказался на него отвечать
404 — Не найдено	Сервер не нашел каких-либо ресурсов, соответствующих URI, содержащемуся в запросе
405 — Метод не поддерживается	Запрос был сделан с использованием метода, который не поддерживается ресурсом
406 — Неприемлемо	Запрошенный ресурс может сгенерировать только такое содержимое, которое является неприемлемым в соответствии с заголовками Accept, указанными в запросе
500 — Внутренняя ошибка сервера	Сервер столкнулся с неожиданным условием, помешавшим ему выполнить запрос
501 — Не реализовано	Сервер либо не распознает метод запроса, либо не способен выполнить запрос
503 — Служба недоступна	В настоящее время сервер недоступен, поскольку он перегружен запросами или остановлен для техобслуживания; как правило, это временное состояние
505 — Версия HTTP не поддерживается	Сервер не поддерживает ту версию протокола HTTP, которая использовалась в запросе

Кэширование и условные запросы

В большинстве распределенных систем роль кэширования невозможно переоценить. Цель кэширования — оптимизация производительности, достигаемая методом исключения ненужных запросов или снижения объема данных в откликах. Протокол HTTP предоставляет специальные механизмы для реализации кэширования и обеспечения корректности кэшированных данных. Но если клиент решает не использовать механизм кэширования, то все данные ему всякий раз придется запрашивать заново, независимо от того, были ли они изменены с момента последнего запроса.

При отправке ответа на запрос `GET` отсылаемая информация может содержать заголовок `Last-Modified`. Он указывает время последнего изменения ресурса. Когда пользовательский агент в следующий раз запросит данный ресурс, он сможет передать эту дату в заголовке `If-Modified-Since`. Веб-сервер (или прокси-сервер) сравнит эту дату с датой последнего изменения ресурса. Если дата, указанная пользовательским агентом, окажется такой же или более поздней, то будет возвращен код состояния `304 – Не изменялось` без тела отклика. В противном случае запрошенная информация будет выполнена или перенаправлена.

Но манипуляции с датами могут быть сложны; в частности, они подразумевают, что все взаимодействующие агенты постоянно остаются синхронизированными. Эта проблема решается с помощью заголовка ответа ETag. ETag фактически является MD5- или SHA1-хешем всех байтов в представлении. Стоит измениться хотя бы одному байту в представлении, как изменится и ETag.

На рис. 15.2 показано, как использовать ETag. Чтобы получить ресурс с информацией о книге, вы используете действие `GET`, в котором указываете URI ресурса (`GET /book/12345`). Сервер возвратит ответ с XML-представлением книги, код состояния `200 – Хорошо`, а также сгенерированный ETag. Когда вы повторно запросите тот же ресурс, попутно сообщив это же значение ETag в заголовке `If-None-Match`, сервер не будет отправлять вам представление ресурса, если с момента последнего запроса этот ресурс не изменился. Вместо этого он вернет код состояния `304 – Не изменялось`, сообщая клиенту, что с момента последнего обращения ресурс не изменился.

Рис. 15.2. Применение кэширования и кода состояния 304 — Не изменялось

Запросы, использующие HTTP-заголовки `If-Modified-Since`, `If-Unmodified-Since`, `If-Match`, `If-None-Match` и `If-Range`, называются условными. Условные запросы

помогают экономить пространство полосы доступа и ресурсы процессора (как на сервере, так и на клиенте), так как исключают ненужные «круговые» операции передачи данных. Заголовки `If-*` чаще всего используются с запросами `GET` и `PUT`.

От Сети к веб-службам

Все мы пользуемся Сетью и знаем, как она работает. Отчего же веб-службы должны работать иначе? В конце концов, службы также часто обмениваются уникально идентифицируемыми ресурсами, которые связываются с другими ресурсами по принципу, напоминающему расстановку гиперссылок. Масштабируемость веб-архитектуры проверена временем; зачем же заново изобретать велосипед. Так почему бы не использовать при создании, обновлении или удалении ресурса о книге давно знакомые HTTP-операции? Например:

- используем `POST` для создания ресурса о книге (передаем при этом XML, JSON или любой другой формат) с URI http://www.apress.com/book/. В ответ получаем с сервера URI нового ресурса: http://www.apress.com/book/1 23456;
- применим `GET` для считывания ресурса (а также для считывания тех ссылок на другие ресурсы, которые мы, возможно, встретим в теле объекта) по адресу http://www.apress.com/book/1 23456;
- используем `PUT` для обновления данных о ресурсе по адресу http://www.apress.com/book/1 23456;
- применим `DELETE` для удаления ресурса по адресу http://www.apress.com/book/1 23456.

Пользуясь HTTP-операциями, мы можем совершать над ресурсом любые действия из разряда CRUD.

WADL

SOAP-службы работают на базе языка WADL, используемого для описания формата возможных запросов к конкретной веб-службе. При этом язык описания веб-приложений (WADL) применяется и для обеспечения возможных взаимодействий с заданной веб-службой в стиле REST. Этот язык упрощает клиентскую разработку, в ходе которой мы можем напрямую загружать ресурсы и взаимодействовать с ними. Язык WADL был представлен на рассмотрение W3C, но пока эта организация не планирует его стандартизировать, поскольку он не слишком широко поддерживается. В листинге 15.2 показан пример кода на этом языке.

Листинг 15.2. Код на языке WADL, определяющий операции, которые можно выполнить с ресурсом

```
<application xmlns="http://wadl.dev.java.net/2009/02">
  <doc xmlns:jersey="http://jersey.java.net/" jersey:generatedBy="Jersey:
2.0"/>

  <resources base="http://www.apress.com/">

    <resource path="{id}">
      <param name="id" style="template" type="xs:long"/>
```

```
        <method name="GET">
          <response>
            <representation element="book" mediaType="application/xml"/>
            <representation element="book" mediaType="application/json"/>
          </response>
        </method>
        <method name="DELETE"/>
      </resource>

      <resource path="book">
        <method name="GET">
          <response>
            <representation element="book" mediaType="application/xml"/>
            <representation element="book" mediaType="application/json"/>
          </response>
        </method>
      </resource>

    </resources>
  </application>
```

В листинге 15.2 описан корень ресурса (http://www.apress.com/), которому вы можете передать идентификатор (`{id}`) для получения (`GET`) или удаления (`DELETE`) книги. Другой ресурс позволяет получить (`GET`) все записи о книгах с сайта APress в формате JSON или XML.

Обзор спецификаций веб-служб с передачей состояния представления

В отличие от стеков SOAP и WS-*, опирающихся на стандарты W3C, REST не имеет стандарта и является просто стилем архитектуры, в котором есть свои принципы проектирования. Приложения в стиле REST значительно зависят от других стандартов:

- HTTP;
- URI, URL;
- XML, JSON, HTML, GIF, JPEG и т. д. (представления ресурсов).

Технологии со стороны Java были описаны в спецификации JAX-RS (API Java для веб-служб с передачей состояния представления), но REST напоминает шаблон проектирования: многоразовое решение для распространенной проблемы, которое может быть реализовано на нескольких языках.

Краткая история REST

Термин REST впервые появился в диссертации Роя Томаса Филдинга «Архитектурные стили и проектирование архитектур программных систем, поддерживающих работу в сети», в главе 5 (работа доступна в Интернете на сайте Калифорнийского университета: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm). Диссертация представляет собой ретроспективное описание архитектуры, на базе которой была

создана Всемирная паутина. В этой работе Филдинг рассматривает те части Сети, которые функционируют особенно хорошо, и вычленяет принципы проектирования, которые позволили бы создать другую распределенную гипермедийную систему. Предполагается, что она должна быть не менее эффективной, чем Интернет, но может быть и не связана с ним.

Итак, в основе проекта REST лежало стремление создать подобную сетевую архитектурную модель. Рой Филдинг также является одним из авторов спецификации HTTP, поэтому совершенно неудивительно, что HTTP так хорошо укладывается в архитектурные принципы, описанные в этой диссертации.

API Java для веб-служб с передачей состояния представления

Для написания веб-служб REST на языке Java вам понадобятся лишь клиент и сервер, поддерживающий взаимодействие по протоколу HTTP. Любой браузер и контейнер HTTP-сервлетов решит такую задачу — вам придется лишь озаботиться определенной конфигурацией XML и состряпать связующий код для синтаксического анализа запросов и ответов HTTP. Но когда такой код окажется написан, его будет практически невозможно читать и поддерживать. Вот тут-то нам и пригодится JAX-RS. Как видите, написав несколько аннотаций, вы получаете широчайшие возможности для вызова HTTP-ресурсов и их синтаксического анализа.

Первая версия спецификации JAX-RS (запрос на спецификацию JSR 311) была готова в октябре 2008 года. В ней был определен набор API, на основе которых строится архитектурный стиль REST. Но в данном варианте спецификации была описана лишь серверная часть REST. С появлением платформы Java EE 7 спецификация JAX-RS была обновлена до версии 2.0. Среди других ее нововведений можно назвать описание клиентского API.

Что нового в JAX-RS 2.0

Спецификация JAX-RS 2.0 (запрос на спецификацию JSR 339) — это основной релиз, сосредоточенный на интеграции с платформой Java EE 7 и ее новыми возможностями. К числу основных новых возможностей JAX-RS 2.0 относятся:

- клиентский API, отсутствовавший во всех технических версиях JAX-RS 1.x, поэтому в каждой реализации применялся свой собственный проприетарный API. JAX-RS 2.0 ликвидирует этот пробел благодаря низкоуровневому API для выстраивания запросов;
- в JAX-RS 2.0 появились фильтры и перехватчики, напоминающие обработчики SOAP или перехватчики управляемых компонентов. Поэтому теперь вы можете перехватывать запрос и ответ и обрабатывать их, как вам требуется;
- появился новый механизм асинхронной обработки, который позволяет внедрять интерфейсы для длинного опроса или реализовывать отправку данных по инициативе сервера;
- достигнута интеграция с технологией валидации компонентов, позволяющая ограничивать ваши веб-службы с передачей состояния представления.

В табл. 15.4 перечислены основные пакеты, определенные в спецификации JAX-RS.

Таблица 15.4. Основные пакеты JAX-RS

Пакет	Описание
javax.ws.rs	Высокоуровневые интерфейсы и аннотации, используемые для создания веб-служб с передачей состояния представления
javax.ws.rs.client	Классы и интерфейсы нового клиентского API JAX-RS
javax.ws.rs.container	Контейнер-специфичный API JAX-RS
javax.ws.rs.core	Низкоуровневые интерфейсы и аннотации, используемые для создания веб-ресурсов с передачей состояния представления
javax.ws.rs.ext	API, предоставляющие расширения для типов, поддерживаемых в JAX-RS API

Справочная реализация

Справочная реализация JAX-RS называется Jersey. Это свободный проект, который одновременно подчиняется двум лицензиям: CDDL и GPL. Jersey также предоставляет специальный API для расширения самой Jersey.

Существуют и другие реализации JAX-RS, в частности CXF (Apache), RESTEasy (JBoss) и Restlet (сравнительно старый проект, появившийся еще до того, как была завершена подготовка спецификации JAX-RS).

Написание веб-служб с передачей состояния представления

Здесь обсуждались некоторые настолько низкоуровневые концепции (взять хотя бы протокол HTTP), что у вас вполне могли накопиться самые разные предположения о том, как может выглядеть код при разработке веб-службы в стиле REST. Начнем с хорошего: вам не придется ни писать служебный код для переваривания HTTP-запросов, ни генерировать HTTP-ответы вручную. JAX-RS — это очень красивый API, позволяющий описать веб-службу с передачей состояния представления с помощью всего нескольких аннотаций. Веб-службы в стиле REST — это обычные POJO, имеющие минимум один метод, аннотированный `javax.ws.rs.Path`. В листинге 15.3 показан типичный ресурс такого рода.

Листинг 15.3. Простая веб-служба в стиле REST для работы с информацией о книгах

```
@Path("/book")
public class BookRestService {
  @GET
  @Produces("text/plain")
  public String getBookTitle() {
    return "H2G2";
  }
}
```

`BookRestService` — это класс Java, аннотированный `@Path`. Это означает, что путь к ресурсу будет лежать через путь URI `/book`. Метод `getBookTitle()` помечен как предназначенный для обработки HTTP-операций `GET` (на это указывает аннотация `@GET`). Данный метод генерирует текст (MIME-тип такого содержимого — `text/plain`; я также мог бы воспользоваться константой `MediaType.TEXT_PLAIN`). Для доступа к этому ресурсу вам понадобится HTTP-клиент, например браузер, в котором можно открыть URL http://www.myserver.com/book.

Спецификация JAX-RS является по своей природе HTTP-ориентированной и имеет несколько четко определенных классов и аннотаций для работы с HTTP и URI. Ресурс может включать несколько представлений, поэтому API обеспечивает поддержку разнообразных типов содержимого, а также использует JAXB для маршалинга и демаршалинга XML-представлений в объекты и обратно. JAX-RS также не зависит от контейнера. Поэтому развертывание ресурсов можно выполнять в GlassFish и во многих других сервлетных контейнерах.

Структура веб-службы с передачей состояния представления

Из листинга 15.3 понятно, что REST-служба не реализует никаких интерфейсов, а также не расширяет какого-либо класса. Единственная обязательная аннотация, нужная для превращения POJO в REST-службу, — это `@Path`. JAX-RS работает на основе «конфигурации по исключениям», то есть располагает набором аннотаций для конфигурирования стандартного поведения. Ниже перечислены требования для написания REST-службы:

- класс должен быть аннотирован с помощью `@javax.ws.rs.Path` (в JAX-RS 2.0 отсутствует XML-эквивалент метаданных, поскольку нет дескриптора развертывания);
- класс должен быть определен как общедоступный (публичный), при этом не должен быть ни финальным, ни абстрактным;
- классы корневых ресурсов (имеющие аннотацию `@Path`) должны иметь действующий по умолчанию общедоступный конструктор. Классы некорневых ресурсов не требуют такого конструктора;
- класс ни в коем случае не должен определять метод `finalize()`;
- для добавления возможностей EJB к REST-службе класс должен иметь аннотацию `@javax.ejb.Stateless` или `@javax.ejb.Singleton` (см. главу 7);
- служба должна быть объектом, не имеющим состояния, и не должна сохранять клиентоспецифичное состояние между вызовами методов.

Выполнение операций CRUD над веб-службами в стиле REST

В листинге 15.3 было показано, как написать очень простую REST-службу, возвращающую строку. Но в большинстве случаев вам нужно будет обращаться к базе данных, получая или сохраняя информацию по транзакционному принципу. Для

таких целей также можно написать REST-службу и добавлять возможности сеансовых компонентов, не сохраняющих состояния. Чтобы обеспечить такой функционал, потребуется аннотация `@Stateless`. Так мы получим доступ к уровню сохраняемости (сущностям JPA), как показано в листинге 15.4.

Листинг 15.4. REST-служба Book, создающая, удаляющая и получающая информацию о книгах из базы данных

```
@Path("book")
@Stateless
public class BookRestService {

  @Context
  private UriInfo uriInfo;
  @PersistenceContext(unitName = "chapter15PU")
  private EntityManager em;

  @GET
  @Produces(MediaType.APPLICATION_XML)
  public Books getBooks() {
    TypedQuery<Book> query = em.createNamedQuery(Book.FIND_ALL, Book.class);
    Books books = new Books(query.getResultList());
    return books;
  }

  @POST
  @Consumes(MediaType.APPLICATION_XML)
    public Response createBook(Book book) {
    em.persist(book);
    URI bookUri = ➥
    uriInfo.getAbsolutePathBuilder().path(book.getId().toString()).build();
    return Response.created(bookUri).build();
  }

  @DELETE
  @Path("{id}")
  public Response deleteBook(@PathParam("id") Long bookId) {
    em.remove(em.find(Book.class, bookId));
    return Response.noContent().build();
  }
}
```

В коде из листинга 15.4 представлена REST-служба, которая может использовать и производить XML-представление книги. Метод `getBooks()` получает из базы данных список книг и возвращает XML-представление этого списка (используя согласование содержимого). Данное представление доступно через метод `GET`. Метод `createBook()` принимает XML-представление книги и сохраняет его в базе данных. Он вызывается с помощью HTTP-операции `POST` и возвращает ответ (`Response`) с URI (`bookUri`) новой книги, а также статус `created`. Метод `deleteBook` принимает идентификатор книги в качестве параметра и удаляет его из базы данных.

Код из листинга 15.4 построен по очень простой модели JAX-RS и использует набор очень мощных аннотаций. Теперь подробнее рассмотрим концепции, представленные в этом коде.

Определение URI и URI связывания

Аннотация `@Path` представляет относительный URI и может сопровождать класс или метод. При использовании с классами она ссылается на корневой ресурс, указывая корень дерева ресурсов и предоставляя доступ к подресурсам. В листинге 15.5 показана REST-служба, расположеная по ссылке http://www.myserver.com/items. Все методы этой службы будут иметь `/items` в качестве корня.

Листинг 15.5. Корневой путь к ресурсу Item

```
@Path("/items")
public class ItemRestService {
  @GET
  public Items getItems() {
    // ...
  }
}
```

Потом вы можете добавлять к вашим методам подпути. Это может быть полезно для объединения в группы общих функциональностей нескольких ресурсов, как показано в листинге 15.6 (пока можно абстрагироваться от того, какую роль играют аннотации `@GET`, `@POST` и `@DELETE`, так как они будут описаны подробнее в разделе «Сопоставление HTTP-методов»).

Листинг 15.6. Несколько подпутей у ItemRestService

```
@Path("/items")
public class ItemRestService {
  @GET
  public Items getItems() {
    // URI : /items
  }

  @GET
  @Path("/cds")
  public CDs getCDs() {
    // URI : /items/cds
  }

  @GET
  @Path("/books")
  public Books getBooks() {
    // URI: /items/books
  }

  @POST
  @Path("/book")
  public Response createBook(Book book) {
```

```
    // URI: /items/book
  }
}
```

В листинге 15.6 показана веб-служба с передачей состояния представления, которая будет предоставлять вам методы для получения всех элементов (компакт-дисков и книг) из приложения CD-Bookstore. При запросе корневого ресурса `/items` выбирается всего один метод без подчиненной аннотации `@Path` (`getItems()`). Затем, когда `@Path` будет сопровождать и класс и метод, относительный путь к методу будет получен конкатенацией обоих элементов. Например, путь для получения всех CD будет таким: `/items/cds`. При запросе `/items/books` будет вызван метод `getBooks()`. Чтобы создать новую запись о книге, необходимо указать на `/items/book`.

Если бы информация `@Path("/items")` сопровождала лишь класс, но не касалась ни одного метода, то для доступа к любому из методов использовался бы один и тот же путь. Единственным отличительным признаком стала бы HTTP-операция (`GET`, `PUT` и т. д.). Об этом мы поговорим ниже.

Извлечение параметров

При работе с REST для доступа к ресурсу очень важно иметь аккуратные URI, полученные при конкатенации путей. Но одних лишь путей и подпутей недостаточно. При оперировании веб-службами в стиле REST им необходимо передавать параметры, а также извлекать параметры и обрабатывать их во время исполнения. В листинге 15.4 было показано, как получить параметр из пути с аннотацией `@javax.ws.rs.PathParam`. В JAX-RS предоставляется широкий набор аннотаций для извлечения различных параметров, которые могут присутствовать в запросе (`@PathParam`, `@QueryParam`, `@MatrixParam`, `@CookieParam`, `@HeaderParam` и `@FormParam`).

В листинге 15.7 продемонстрировано, как аннотация `@PathParam` используется для извлечения значения параметра шаблона URI. Параметр имеет имя и представляется в виде переменной, заключенной в фигурные скобки, либо переменной, за которой следует регулярное выражение. Метод `searchCustomers` принимает любой строковый параметр, тогда как в параметрах метода `getCustomerByLogin` допускаются только строчные и прописные буквы алфавита (`[a-zA-Z]*`), а в параметрах метода `getCustomerById` — только цифры (`\\d+`).

Листинг 15.7. Извлечение параметров пути и регулярных выражений

```
@Path("/customer")
@Produces(MediaType.APPLICATION_JSON)
public class CustomerRestService {

  @Path("search/{text}")
  public Customers searchCustomers(@PathParam("text") String textToSearch) {
    // URI : /customer/search/smith
  }
  @GET
  @Path("{login: [a-zA-Z]*}")
```

```
    public Customer getCustomerByLogin(@PathParam("login") String login) {
      // URI : /customer/foobarsmith
    }

    @GET
    @Path("{customerId : \\d+}")
    public Customer getCustomerById(@PathParam("customerId") Long id) {
      // URI : /customer/12345
    }
}
```

Аннотация `@QueryParam` извлекает значение параметра запроса, содержащегося в URI. Параметры запроса — это пары «ключ/значение», разделительным символом между которыми служит `&`. Например: `http://www.myserver.com/customer?zip=75012&city=Paris`. Аннотация `@MatrixParam` действует подобно `@QueryParam`, однако извлекает значение параметра матрицы URI (в качестве разделительного знака используется `;`). В листинге 15.8 показано, как извлечь из URI оба параметра — содержащий запрос и содержащий матрицу.

Листинг 15.8. Извлечение параметров со значением запроса и матрицы

```
@Path("/customer")
@Produces(MediaType.APPLICATION_JSON)
public class CustomerRestService {
  @GET
  public Customers getCustomersByZipCode(@QueryParam("zip") Long zip, ➥
                                         @QueryParam("city") String city) {
    // URI : /customer?zip=75012&city=Paris
  }

  @GET
  @Path("search")
  public Customers getCustomersByName(@MatrixParam("firstname") String ➥
                      firstname, @MatrixParam("surname") String surname) {
    // URI : /customer/search;firstname=Antonio;surname=Goncalves
  }
}
```

Две другие аннотации относятся к внутренней части HTTP, такая информация не отражается непосредственно в URI. Речь идет о cookie и HTTP-заголовках. Аннотация `@CookieParam` извлекает значение cookie, а аннотация `@HeaderParam` — значение поля заголовка. В листинге 15.9 мы извлекаем сеансовый ID из cookie и информацию о пользовательском агенте из HTTP-заголовка.

Листинг 15.9. Извлечение значения из cookie и HTTP-заголовка

```
@Path("/customer")
@Produces(MediaType.TEXT_PLAIN)
public class CustomerRestService {

  @GET
  public String extractSessionID(@CookieParam("sessionID") String sessionID) {
```

```
    // ...
  }
  @GET
  public String extractUserAgent(@HeaderParam("User-Agent") String userAgent) {
    // ...
  }
}
```

Аннотация `@FormParam` указывает, что значение параметра должно быть извлечено из формы, содержащейся в теле объекта в запросе. Поддержка аннотации `@FormParam` у полей и свойств не требуется.

Имея все эти аннотации, можно добавить и аннотацию `@DefaultValue` для определения значения, которое будет действовать по умолчанию для ожидаемого параметра. Такое значение задействуется в соответствующем параметре, если подобное значение отсутствует в запросе. В листинге 15.10 устанавливаются значения, которые будут действовать по умолчанию для параметров запроса и матрицы. Например, если при работе с методом `getCustomersByAge` в запросе не окажется параметра `age`, то по умолчанию будет применено значение `50`.

Листинг 15.10. Определение значений, действующих по умолчанию

```
@Path("/customer")
public class CustomerRestService {
  @GET
  public Customers getCustomersByAge(@DefaultValue("50") @QueryParam("age") ➥
                                     int age) {
    // ...
  }

  @GET
  public Customers getCustomersByCity(@DefaultValue("Paris") ➥
                                      @MatrixParam("city") String city) {
    // ...
  }
}
```

Использование и создание типов содержимого

При работе с REST один и тот же ресурс может обладать несколькими представлениями. Так, книга может иметь вид веб-страницы, PDF-документа или изображения, на котором показана лишь обложка. В JAX-RS указывается несколько типов Java, способных представлять ресурсы, например `String`, `InputStream` и компоненты `JAXB`. Аннотации `@javax.ws.rs.Consumes` и `@javax.ws.rs.Produces` могут применяться с таким ресурсом, который способен иметь несколько представлений. Здесь определяются медиатипы представления, которыми могут обмениваться клиент и сервер. В JAX-RS есть класс `javax.ws.rs.core.MediaType`, действующий в качестве абстракции для MIME-типа. Он содержит несколько методов и определяет константы, перечисленные в табл. 15.5.

Таблица 15.5. MIME-типы, определяемые в классе MediaType

Имя константы	MIME-тип
APPLICATION_ATOM_XML	"application/atom+xml"
APPLICATION_FORM_URLENCODED	"application/x-www-form-urlencoded"
APPLICATION_JSON	"application/json"
APPLICATION_OCTET_STREAM	"application/octet-stream"
APPLICATION_SVG_XML	"application/svg+xml"
APPLICATION_XHTML_XML	"application/xhtml+xml"
APPLICATION_XML	"application/xml"
MULTIPART_FORM_DATA	"multipart/form-data"
TEXT_HTML	"text/html"
TEXT_PLAIN	"text/plain"
TEXT_XML	"text/xml"
WILDCARD	"*/*"

Если в методе используются аннотации `@Consumes` и `@Produces`, то они переопределяют любые аннотации, которыми может сопровождаться класс ресурса для аргумента метода или возвращаемого типа. При отсутствии любой из двух этих аннотаций предполагается поддержка любого медиатипа (*/*). По умолчанию `CustomerRestService` порождает обычные текстовые представления, которые переопределяются в некоторых методах (листинг 15.11). Обратите внимание: `getAsJsonAndXML` порождает массив представлений (XML или JSON).

Листинг 15.11. Ресурс Customer с несколькими представлениями

```
@Path("/customer")
@Produces(MediaType.TEXT_PLAIN)
public class CustomerRestService {
  @GET
  public Response getAsPlainText() {
    // ...
  }

  @GET
  @Produces(MediaType.TEXT_HTML)
  public Response getAsHtml() {
    // ...
  }

  @GET
  @Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
  public Response getAsJsonAndXML() {
    // ...
  }
  @PUT
  @Consumes(MediaType.TEXT_PLAIN)
```

```
  public void putName(String customer) {
    // ...
  }
}
```

Если веб-служба REST в состоянии произвести более одного медиатипа, то целевой метод будет соответствовать наиболее приемлемому медиатипу согласно информации, указанной клиентом в заголовке `Accept` HTTP-запроса. Например, если заголовок `Accept` таков:

```
Accept: text/plain
```

а URI при этом — `/customer`, то будет вызван метод `getAsPlainText()`. Но клиент мог бы использовать и такой HTTP-заголовок:

```
Accept: text/plain; q=0.8, text/html
```

Он объявляет, что клиент может принимать медиатипы `text/plain` и `text/html`, но предпочитает второй. Для указания выбора используется коэффициент качества (он же — вес предпочтения), равный 0,8 («Я предпочитаю `text/html`, но пришлите мне `text/plain`, если это наилучший доступный вариант после 80%-ного снижения качества»). При включении такого заголовка и указании на URI `/customer` будет вызван метод `getAsHtml()`.

Возвращаемые типы

До сих пор мы в основном говорили о том, как вызывать метод (с помощью параметров, медиатипа, HTTP-операций...), а возвращаемый тип нас не интересовал. Что же может возвращать веб-служба в стиле REST? Как и любой класс Java, метод может возвратить любой стандартный тип Java, компонент JAXB и вообще любой объект, обладающий текстовым представлением, которое можно передать по протоколу HTTP. В данном случае среда времени исполнения определяет MIME-тип возвращаемого объекта и вызывает соответствующий поставщик объектов (Entity Provider, см. ниже) для получения нужного представления. Среда времени исполнения также определяет подходящий код состояния, который HTTP должен вернуть потребителю (например, `204 – Нет содержимого`, если возвращаемый тип метода ресурса равен `void` или `null`; либо `200 – Хорошо`, если возвращено ненулевое значение). Но иногда требуется более тщательный контроль над возвращаемой информацией: разумеется, вас интересует тело ответа, называемое в терминологии HTTP объектом, но также могут быть важны и код ответа, и/или заголовки и cookie ответа. В таких случаях вы возвращаете объект `Reponse`. Целесообразно возвращать `javax.ws.rs.core.Response`, поскольку так гарантируется тип возвращаемого содержимого. В листинге 15.12 показаны различные возвращаемые типы.

Листинг 15.12. Служба Customer, возвращающая типы данных, компонент JAXB и ответ

```
@Path("/customer")
public class CustomerRestService {
  @GET
  public String getAsPlainText() {
    return new Customer("Джон", "Смит", "jsmith@gmail.com", ➥
```

```
                        "1234565").toString();
    }

    @GET
    @Path("maxbonus")
    public Long getMaximumBonusAllowed() {
      return 1234L;
    }

    @GET
    @Produces(MediaType.APPLICATION_XML)
    public Customer getAsXML() {
      return new Customer("Джон", "Смит", "jsmith@gmail.com", "1234565");
    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getAsJson() {
      return Response.ok(new Customer("Джон", "Смит", "jsmith@gmail.com", ➥
                         "1234565"), MediaType.APPLICATION_JSON).build();
    }
}
```

Метод `getAsPlainText` возвращает строковое представление потребителя, а метод `getMaximumBonusAllowed` — числовую константу. Будут применяться настройки, заданные по умолчанию, поэтому код состояния для обоих методов при возврате будет равен `200 – Хорошо`, если не произойдет исключения. Метод `getAsXML` возвращает объект `Customer` JAXB POJO. Это означает, что среда времени исполнения выполнит маршалинг объекта в XML-представление.

Метод `getAsJson` возвращает не HTML-объект, а объект `javax.ws.rs.core.Response`. `Response` оборачивает HTML-объект, возвращаемый потребителю, и инстанцируется с помощью класса `ResponseBuilder`, используемого в качестве фабрики. В данном примере мы по-прежнему хотим вернуть JAXB-объект (`Customer`) с кодом состояния `200 – Хорошо` (метод `ok()`), но также собираемся указать JSON в качестве MIME-типа. При вызове метода `ResponseBuilder.build()` создается конечный экземпляр `Response`.

Рекомендуется возвращать пользовательский `Response` для всех запросов, а не для самого HTML-объекта (при необходимости вы можете затем задать нужный код состояния). В табл. 15.6 показано подмножество Response API.

Таблица 15.6. Response API

Метод	Описание
accepted()	Создает новый объект ResponseBuilder с состоянием 202 — Принято
created()	Создает новый объект ResponseBuilder для созданного ресурса (с его URI)
noContent()	Создает новый объект ResponseBuilder для пустого ответа
notModified()	Создает новый объект ResponseBuilder с состоянием 304 — Не изменялось

Метод	Описание
ok()	Создает новый объект ResponseBuilder с состоянием 200 — Хорошо
serverError()	Создает новый объект ResponseBuilder с состоянием 500 — Серверная ошибка
status()	Создает новый объект ResponseBuilder с предоставленным состоянием
temporaryRedirect()	Создает новый объект ResponseBuilder с временным перенаправлением
getCookies()	Получает cookie из сообщения ответа
getHeaders()	Получает заголовки из сообщения ответа
getLinks()	Получает ссылки, прикрепленные к сообщению в заголовке
getStatus()	Получает код состояния, ассоциированный с ответом
readEntity()	Получает объект сообщения как экземпляр указанного типа Java. При этом используется интерфейс MessageBodyReader, поддерживающий отображение сообщения на запрошенный тип

Response и ResponseBuilder следуют шаблону проектирования «текучий интерфейс». Это означает, что вы вполне можете написать ответ, воспользовавшись конкатенацией методов. В таком случае код также становится более удобочитаемым. Вот примеры кода, который мы можем написать с помощью этого API:

```
Response.ok().build();
Response.ok().cookie(new NewCookie("SessionID", "5G79GDIFY09")).build();
Response.ok("Plain Text").expires(new Date()).build();
Response.ok(new Customer ("Джон", "Смит"), MediaType.APPLICATION_JSON).
build();
Response.noContent().build();
Response.accepted(new Customer("Джон", "Смит", "jsmith@gmail.com",
"1234565")).build();
Response.notModified().header("Браузер", "Mozilla").build();
```

Сопоставление HTTP-методов

Итак, мы рассмотрели, как протокол HTTP работает со своими запросами, ответами и операциями (GET, POST, PUT и т. д.) JAX-RS определяет эти распространенные HTTP-методы с помощью аннотаций @GET, @POST, @PUT, @DELETE, @HEAD и @OPTIONS. В качестве методов ресурсов могут предоставляться только общедоступные методы. В листинге 15.13 показана пользовательская веб-служба в стиле REST, предоставляющая методы из разряда CRUD: @GET для получения ресурсов, @POST для создания нового ресурса, @PUT для обновления имеющегося ресурса и @DELETE для удаления ресурса.

Листинг 15.13. Ресурс Customer, предоставляющий CRUD-операции и возвращающий ответы

```
@Path("/customer")
@Produces(MediaType.APPLICATION_XML)
@Consumes(MediaType.APPLICATION_XML)
```

```
public class CustomerRestService {
  @GET
  public Response getCustomers() {
    // ..
    return Response.ok(customers).build();
  }

  @GET
  @Path("{customerId}")
  public Response getCustomer(@PathParam("customerId") String customerId) {
    // ..
    return Response.ok(customer).build();
  }

  @POST
  public Response createCustomer(Customer customer) {
    // ..
    return Response.created(createdCustomerURI).build();
  }

  @PUT
  public Response updateCustomer(Customer customer) {
    // ..
    return Response.ok(customer).build();
  }
  @DELETE
  @Path("{customerId}")
  public Response deleteCustomer(@PathParam("customerId") String customerId) {
    // ..
    return Response.noContent().build();
  }
}
```

В спецификации HTTP определяется, какие коды состояния HTTP должны выдаваться при успешном ответе. Вы можете быть уверены, что JAX-RS будет возвращать те же самые стандартные коды ответа.

- Методы GET получают (в виде объекта) любую информацию, на которую указывает запрошенный URI. GET должен возвращать код 200 – Хорошо.
- Метод PUT ссылается на уже существующий ресурс, который необходимо обновить. Если обновляется существующий ресурс, то должен быть возвращен один из следующих кодов состояния: 200 – Хорошо или 204 – Нет содержимого. Такие ответы означают успешное завершение запроса.
- Метод POST используется для создания нового ресурса, идентифицируемого URI запроса. В ответ должен возвращаться код 201 – Создано с URI нового ресурса или код 204 – Нет содержимого, если метод не создал ресурса, который можно было бы идентифицировать по URI.
- Метод DELETE требует, чтобы сервер удалил ресурс, на который указывает содержащийся в запросе URI. При успешном ответе должен возвращаться код

200 – Хорошо (если в ответе содержится объект), 202 – Принято (если действие пока не запущено) или 204 – Нет содержимого, если действие было запущено, но в ответе отсутствует объект.

Построение URI. Гиперссылки — центральная составляющая REST-приложений. Чтобы развиваться в процессе движения приложения между состояниями, веб-службы в стиле REST должны гибко управлять переходами и так же гибко строить URI. В JAX-RS предоставляется объект javax.ws.rs.core.UriBuilder, призванный заменить java.net.URI и упростить безопасное построение URI. Класс UriBuilder обладает набором методов, с помощью которых удобно строить новые URI с нуля или на основе уже существующих URI. В листинге 15.14 показаны примеры использования UriBuilder для создания любого URI с параметрами, указывающими путь, запрос или матрицу.

Листинг 15.14. Использование UriBuilder

```
public class URIBuilderTest {
  @Test
  public void shouldBuildURIs() {
    URI uri = ➥
    UriBuilder.fromUri("http://www.myserver.com").path("book").path("1234").
build();
    assertEquals("http://www.myserver.com/book/1234", uri.toString());

    uri = UriBuilder.fromUri("http://www.myserver.com").path("book ") ➥
          .queryParam("author", "Goncalves").build();
    assertEquals("http://www.myserver.com/book?author=Goncalves", ➥
                 uri.toString());

    uri = UriBuilder.fromUri("http://www.myserver.com").path("book") ➥
         .matrixParam("author", "Goncalves").build();
    assertEquals("http://www.myserver.com/book;author=Goncalves", uri.toString());
    uri = UriBuilder.fromUri("http://www.myserver.com").path("{path} ") ➥
         .queryParam("author", "{value}").build("book", "Goncalves");
    assertEquals("http://www.myserver.com/book?author=Goncalves", ➥
                 uri.toString());

    uri = UriBuilder.fromResource(BookRestService.class).path("1234").build();
    assertEquals("/book/1234", uri.toString());

    uri = UriBuilder.fromUri("http://www.myserver.com").fragment("book").build ();
    assertEquals("http://www.myserver.com/#book", uri.toString());
  }
}
```

Контекстная информация

Когда происходит обработка ресурса, поставщику ресурса для правильного выполнения запроса требуется контекстная информация. Аннотация @javax.ws.rs.core.Context предназначена для внедрения в атрибут или параметр метода следующих классов: HttpHeaders, UriInfo, Request, SecurityContext и Providers. Например,

в листинге 15.15 показан код, внедряющий UriInfo таким образом, что этот класс может строить URI, а также добавляющий HttpHeaders для возвращения определенной заголовочной информации.

Листинг 15.15. Ресурс Customer, получающий HttpHeaders и UriInfo

```
@Path("/customer")
public class CustomerRestService {
  @Context
  UriInfo uriInfo;

  @Inject
  private CustomerEJB customerEJB;

  @GET
  @Path("media")
  public String getDefaultMediaType(@Context HttpHeaders headers) {
    List<MediaType> mediaTypes = headers.getAcceptableMediaTypes();
    return mediaTypes.get(0).toString();
  }

  @GET
  @Path("language")
  public String getDefaultLanguage(@Context HttpHeaders headers) {
    List<String> mediaTypes = ➥
      headers.getRequestHeader(HttpHeaders.ACCEPT_LANGUAGE);
    return mediaTypes.get(0);
  }

  @POST
  @Consumes(MediaType.APPLICATION_XML)
  public Response createCustomer(Customer cust) {
    Customer customer = customerEJB.persist(cust);

    URI bookUri = ➥
      uriInfo.getAbsolutePathBuilder().path(customer.getId()).build();
    return Response.created(bookUri).build();
  }
}
```

Как было показано выше в примерах с HTTP, информация передается между клиентом и сервером не только в теле объекта, но и в заголовках (Date, Server, Content-Type и т. д.). HTTP-заголовки участвуют в работе единообразного интерфейса, а веб-службы REST используют их «в оригинальных значениях». Вам как разработчику ресурсов может понадобиться доступ к HTTP-заголовкам. Именно для этого служит интерфейс javax.ws.rs.core.HttpHeaders. Экземпляр HttpHeaders можно внедрить в атрибут или параметр метода с помощью аннотации @Context, так как класс HttpHeaders — это словарь с вспомогательными методами для доступа к значениям заголовков без учета регистра. В листинге 15.15 служба возвращает стандартные значения Accept-Language и MediaType.

ПРИМЕЧАНИЕ

В JAX-RS 2.0 аннотация @Context используется при внедрении контекстной информации. К сожалению, аннотация @Inject здесь не сработает, так как в данной версии выравнивание CDI не может быть полностью заархивировано. Остается надеяться, что в будущих версиях JAX-RS мы сможем широко использовать CDI и работать всего с одной аннотацией: @Inject.

Поставщик объектов

Когда мы получаем объекты в запросах или отсылаем в ответах, реализации JAX-RS требуется способ для преобразования представлений в тип Java и обратно. Эту работу выполняют поставщики объектов, обеспечивающие отображение между представлениями и ассоциированными с ними типами Java. В качестве примера можно привести JAXB, который отображает объект на XML-представление и наоборот. Если стандартных поставщиков XML и JSON недостаточно, то вы можете разработать собственные специальные поставщики объектов. Можете даже определить для этого свой формат. В таком случае потребуется предоставить среде времени исполнения JAX-RS способ для считывания/записи вашего специального формата в объект или из него, реализовав для этого собственный поставщик объектов. Существует две разновидности поставщиков объектов: `MessageBodyReader` и `MessageBodyWriter`.

Допустим, вы хотите перевести ваш компонент `Customer` в `custom/format`, чтобы он принял вид `1234/John/Smith`. Как видите, в качестве разделителя используется символ /, причем первый маркер — это пользовательский идентификатор, второй — имя, последний — фамилия. В первую очередь вам потребуется класс (записывающий), который будет принимать компонент `Customer` и отображать его в теле ответа. В листинге 15.16 показан класс `CustomCustomerWriter`, который должен реализовывать интерфейс `javax.ws.rs.ext.MessageBodyWriter` и сопровождаться аннотацией `@Provider`. Аннотация `@Produces` указывает наш специальный медиатип (`"custom/format"`). Как видите, метод `writeTo` преобразует компонент `Customer` в поток данных, следующий за специальным форматом.

Листинг 15.16. Поставщик, порождающий специальное представление Customer

```
@Provider
@Produces("custom/format")
public class CustomCustomerWriter implements MessageBodyWriter<Customer> {
  @Override
  public boolean isWriteable(Class<?> type, Type genericType, ➥
      Annotation[] annotations, MediaType mediaType) {
    return Customer.class.isAssignableFrom(type);
  }
  @Override
  public void writeTo(Customer customer, Class<?> type, Type gType, ➥
      Annotation[] annotations, MediaType mediaType, ➥
      MultivaluedMap<String, Object> httpHeaders, ➥
  OutputStream outputStream) throws IOException, WebApplicationException {

    outputStream.write(customer.getId().getBytes());
    outputStream.write('/');
```

```
    outputStream.write(customer.getFirstName().getBytes());
    outputStream.write('/');
    outputStream.write(customer.getLastName().getBytes());
  }

  @Override
  public long getSize(Customer customer, Class<?> type, Type genericType,
      Annotation[] annotations, MediaType mediaType) {
    return customer.getId().length() + 1 + customer.getFirstName().length()
        + 1 + customer.getLastName().length();
  }
}
```

С другой стороны, для отображения тела запроса на тип Java класс должен реализовывать интерфейс `javax.ws.rs.ext.MessageBodyReader` и сопровождаться аннотацией `@Provider`. По умолчанию предполагается, что такая реализация должна применять все медиатипы (`*/*`). Аннотация `@Consumes` в листинге 15.17 используется для указания нашего специального медиатипа. Метод `readFrom` принимает поток данных ввода, размечает его с помощью разделительного символа `/` и преобразует в объект `Customer`. Реализации `MessageBodyReader` и `MessageBodyWriter` могут генерировать исключение `WebApplicationException`, если они не способны создать какое-либо представление.

Листинг 15.17. Поставщик, использующий специальный поток данных и создающий объект Customer

```
@Provider
@Consumes("custom/format")
public class CustomCustomerReader implements MessageBodyReader<Customer> {
  @Override
  public boolean isReadable(Class<?> type, Type genericType,
      Annotation[] annotations, MediaType mediaType) {
    return Customer.class.isAssignableFrom(type);
  }

  @Override
  public Customer readFrom(Class<Customer> type, Type gType,
      Annotation[] annotations, MediaType mediaType,
      MultivaluedMap<String, String> httpHeaders, InputStream inputStream)
      throws IOException, WebApplicationException {
    String str = convertStreamToString(inputStream);
    StringTokenizer s = new StringTokenizer(str, "/");

    Customer customer = new Customer();
    customer.setId(s.nextToken());
    customer.setFirstName(s.nextToken());
    customer.setLastName(s.nextToken());

    return customer;
  }
}
```

Имея CustomCustomerWriter и CustomCustomerReader, мы можем обмениваться данными, представленными в нашем специальном формате. Обмен идет в обоих направлениях между службой и потребителем. Веб-службы в стиле REST должны просто объявлять верный медиатип, а среда времени исполнения JAX-RS сделает остальное:

```
@GET
@Produces("custom/format")
public Customer getCustomCustomer () {
  return new Customer("1234", "Джон", "Смит");
}
```

Наш пользовательский формат представляет собой особый случай, но при работе с обычными медиатипами реализация JAX-RS применяет несколько стандартных поставщиков объектов (табл. 15.7). Поэтому в наиболее распространенных случаях вам не придется реализовывать собственные механизмы для считывания и записи.

Таблица 15.7. Поставщики, применяемые в реализации JAX-RS по умолчанию

Тип	Описание
byte[]	Все медиатипы (*/*)
java.lang.String	Все медиатипы (*/*)
java.io.InputStream	Все медиатипы (*/*)
java.io.Reader	Все медиатипы (*/*)
java.io.File	Все медиатипы (*/*)
javax.activation.DataSource	Все медиатипы (*/*)
javax.xml.transform.Source	XML-типы (text/xml, application/xml)
javax.xml.bind.JAXBElement	Медиатипы для XML, относящиеся к классу JAXB (text/xml, application/xml)
MultivaluedMap<String,String>	Содержимое формы (application/x-www-form-urlencoded)
javax.ws.rs.core StreamingOutput	Все медиатипы (*/*), только с MessageBodyWriter

Обработка исключений

До сих пор мы обсуждали идеальный код, который исправно выполняется всегда и везде и не требует обработки исключений. К сожалению, жизнь не так хороша и рано или поздно вам придется столкнуться со сбоями, происходящими в ресурсах либо из-за получения невалидных данных, либо из-за ненадежности участков сети.

Метод ресурса может генерировать любое проверяемое или непроверяемое исключение. Непроверяемые исключения могут генерироваться повторно, а также проникать в базовый контейнер. Напротив, проверяемые исключения не могут генерироваться непосредственно и должны обертываться в исключения, специфичные для того или иного контейнера (ServletException, WebServiceException или WebApplicationException). Но вы также можете выдавать javax.ws.rs.WebApplicationException или его подклассы (BadRequestException, ForbiddenException, NotAcceptableException, NotAllowedException,

NotAuthorizedException, NotFoundException, NotSupportedException). Исключение будет отлавливаться реализацией JAX-RS и преобразовываться в HTTP-ответ. Стандартная ошибка имеет код 500 и соответствует пустому сообщению. Но класс javax.ws.rs.WebApplicationException предлагает различные конструкторы, позволяющие выбрать конкретный код состояния (этот код определяется в перечислении javax.ws.rs.core.Response.Status) или объект. В листинге 15.18 метод getCustomer генерирует непроверяемое исключение (IllegalArgumentException), если идентификатор пользователя меньше 1000, либо код 404 – Не найдено, если запись, соответствующая определенному пользователю, не будет найдена в базе данных. Вместо этого можно было бы также выдать NotFoundException.

Листинг 15.18. Метод, генерирующий исключения

```
@Path("/customer")
public class CustomerRestService {
  @Inject
  private CustomerEJB customerEJB;

  @Path("{customerId}")
  public Customer getCustomer(@PathParam("customerId") Long customerId) {
    if (customerId < 1000)
      throw new IllegalArgumentException("Id must be greater than 1000!");

    Customer customer = customerEJB.find(customerId);
    if (customer == null)
      throw new WebApplicationException(Response.Status.NOT_FOUND);
    return customer;
  }
}
```

Чтобы держать код сухим[1], можно предоставить специальные поставщики отображения исключений. Такой поставщик отображает общее исключение на Response. Если будет сгенерировано такое исключение, то реализация JAX-RS отловит его и вызовет соответствующий поставщик отображения исключений. Поставщик отображения исключений является реализацией ExceptionMapper<E extends java.lang.Throwable>, которая сопровождается аннотацией @Provider. В листинге 15.19 исключение javax.persistence.EntityNotFoundException отображается в коде состояния 404 – Не найдено.

Листинг 15.19. Исключение JPA, отображаемое в коде состояния 404 — Не найдено

```
@Provider
public class EntityNotFoundMapper implements ExceptionMapper<EntityNotFound-
Exception> {
  public Response toResponse(javax.persistence.EntityNotFoundException ex) {
    return Response.status(404).entity(ex.getMessage()).type(MediaType.TEXT_
PLAIN).build();
  }
}
```

[1] Английская аббревиатура DRY, которую можно перевести как «сухой», означает Don't Repeat Yourself — «Не повторяйся». — *Примеч. пер.*

Жизненный цикл и обратные вызовы

При поступлении запроса происходит разрешение той REST-службы, которой он был адресован, и создается новый экземпляр соответствующего класса. Соответственно, жизненный цикл такой службы является «позапросным»; служба может игнорировать проблемы, связанные с параллелизмом, и безопасно пользоваться переменными экземпляров.

При развертывании в контейнере Java EE (сервлетном или EJB) классы ресурсов и поставщики JAX-RS также могут использовать описанные в JSR 250 технологию управления жизненным циклом и аннотации безопасности: `@PostConstruct`, `@PreDestroy`, `@RunAs`, `@RolesAllowed`, `@PermitAll`, `@DenyAll` и `@DeclareRoles`. Жизненный цикл ресурса может использовать аннотации `@PostConstruct` и `@PreDestroy` для добавления бизнес-логики уже после создания ресурса или перед его удалением. На рис. 15.3 показан жизненный цикл, свойственный большинству компонентов платформы Java EE.

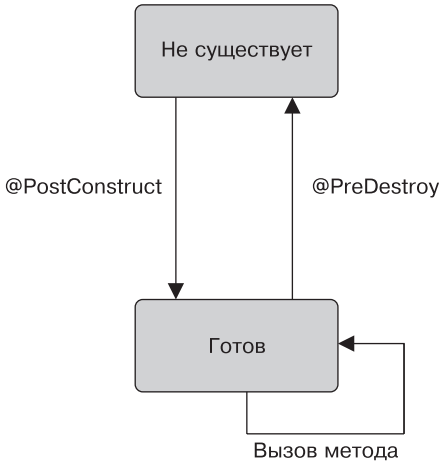


Рис. 15.3. Жизненный цикл ресурса

Упаковка

Веб-службы в стиле REST могут упаковываться в файлы WAR-архивов или EJB JAR-архивов в зависимости от того, готовите вы службу для веб-профиля или полномасштабную службу для Java EE 7. Не забывайте, что веб-служба в стиле REST также может сопровождаться аннотациями `@Stateless` или `@Singleton` — в таком случае с ней можно будет использовать службы сеансовых компонентов. У веб-служб SOAP было несколько сгенерированных артефактов для упаковки, но с REST складывается иная ситуация. Разработчик отвечает лишь за упаковку классов, аннотированных `@Path` (и других вспомогательных классов). Обратите внимание: в JAX-RS 2.0 отсутствует дескриптор развертывания.

Вызов веб-служб в стиле REST

Итак, вы уже научились писать веб-службы с передачей состояния представления, которые могут обрабатывать данные, выполнять в базе данных операции из разряда CRUD, доставлять информацию с сервера и т. д. Как получить к ним доступ? Как следует из рис. 15.1, для этого нужно просто сделать HTTP-запрос к URI. Воспользуйтесь любым инструментом, обеспечивающим такую возможность, и свободно вызывайте любые описанные выше службы в стиле REST.

Первый мыслимый инструмент такого рода — любой браузер. Откройте браузер, введите URI в адресную строку — и получите визуальное представление. К сожалению, браузеры работают только с HTTP-методами `GET` и `POST`. Если вы хотите добиться от браузера большего, к нему вполне можно подключить несколько модулей. Такие подключаемые модули поддерживают работу с методами `PUT` и `DELETE` (в качестве примера можно привести Postman из Chrome). Еще один очень функциональный инструмент, которым можно воспользоваться, — cUrl. Этот инструмент командной строки обеспечивает доступ к деталям HTTP-протокола и, в частности, позволяет вызывать веб-службы REST.

До выхода JAX-RS 2.0 в Java отсутствовал стандартный способ простого вызова веб-служб в стиле REST. Приходилось либо полагаться на низкоуровневый API `java.net.HttpURLConnection`, либо пользоваться проприетарным API того или иного фреймворка (например, Jersey, Resteasy или Restlet). JAX-RS ликвидирует этот пробел с помощью своего текучего, легкого в использовании API для выстраивания запросов.

Клиентский API

В JAX-RS 2.0 появился новый клиентский API, с помощью которого можно с легкостью делать HTTP-запросы к удаленным REST-службам (несмотря на все низкоуровневые функции протокола HTTP). Речь идет о текучем API для построения запросов (использующем шаблон проектирования «Построитель»), в котором задействуется небольшое количество классов и интерфейсов (обзор пакета `javax.ws.rs.client` представлен в табл. 15.8, а информация об API `Response` приводилась в табл. 15.6). Очень часто вам придется сталкиваться с тремя основными классами: `Client`, `WebTarget` и `Response`. Интерфейс `Client` (получаемый с помощью `ClientBuilder`) — это построитель интерфейсов `WebTarget`. Интерфейс `WebTarget` представляет самостоятельный URI, с которого вы можете инициировать запросы для получения `Response`. На основании этого ответа можно проверять HTTP-состояние, длину или cookie. Но гораздо важнее, что вы можете получать содержимое ответа. Это делается с помощью класса `Entity`.

Таблица 15.8. Основные классы и интерфейсы из пакета javax.ws.rs.client

Класс/интерфейс	Описание
Client	Это основная входная точка для текучего API, используемого для построения и выполнения клиентских ответов
ClientBuilder	Входная точка для клиентского API, используемая для начальной загрузки экземпляров Client

Класс/интерфейс	Описание
Configurable	Клиентская конфигурационная форма Client, WebTarget и Invocation
Entity	Инкапсулирует объект сообщения, включая ассоциированную информацию о вариантах
Invocation	Запрос, готовый к исполнению
Invocation.Builder	Построитель вызовов клиентского интерфейса
WebTarget	Цель ресурса, идентифицируемая URI ресурса

Пошагово рассмотрим этот список, чтобы разобраться, как работать с новым API. Затем объединим все приобретенные знания и протестируем веб-службу в стиле REST.

Начальная загрузка клиента

Основной входной точкой в данный API является интерфейс `Client`. Он конфигурирует HTTP-соединения и управляет ими. Кроме того, это фабрика для `WebTargets`, поэтому данный интерфейс обладает набором методов для создания ссылок на ресурсы и вызова ресурсов. Экземпляры `Client` создаются с помощью одного из статических методов класса `ClientBuilder`:

```
Client client = ClientBuilder.newClient();
```

В примере показан способ создания экземпляра `Client` с помощью стандартной реализации. Мы также можем запросить конкретную специальную реализацию `Client`, воспользовавшись нужной конфигурацией. Например, следующий код регистрирует поставщика `CustomCustomerReader` (см. листинг 15.17) и устанавливает некоторые свойства:

```
Client client = ClientBuilder.newClient();
client.configuration().register(CustomCustomerReader.class).
setProperty("MyProperty", 1234);
```

Цели и вызовы

Имея `Client`, вы можете выбрать в качестве цели URI определенной веб-службы в стиле REST и вызывать в ней конкретные HTTP-методы. Именно для таких операций нужны интерфейсы `WebTarget` и `Invocation`. Методы `Client.target()` — это фабрики для веб-целей, представляющие конкретный URI. Вы выстраиваете и выполняете запросы из экземпляра `WebTarget`. Можно создать `WebTarget` со строковым представлением URI:

```
WebTarget target = client.target("http://www.myserver.com/book");
```

Можно также получить `WebTarget` от `java.net.URI`, `javax.ws.rs.core.UriBuilder` или `javax.ws.rs.core.Link`:

```
URI uri = new URI("http://www.myserver.com/book");
WebTarget target = client.target(uri);
```

Теперь у вас есть URI, на который нужно нацелиться. Переходим к построению HTTP-запроса. `WebTarget` обеспечивает такие операции с помощью `Invocation.Builder`. Чтобы инициировать простой HTTP-запрос `GET` к URI, достаточно написать:

```
Invocation invocation = target.request().buildGet()
```

Invocation.Builder позволяет строить запросы GET, а также POST, PUT и DELETE. Кроме того, можно строить запрос для различных MIME-типов и даже добавлять параметры с информацией о пути, запросе и матрице. Для методов PUT и POST необходимо передавать параметр Entity, представляющий полезную нагрузку для отправки веб-службе в стиле REST:

```
target.request().buildDelete();
target.queryParam("author", "Eloise").request().buildGet();
target.path(bookId).request().buildGet();
target.request(MediaType.APPLICATION_XML).buildGet();
target.request(MediaType.APPLICATION_XML).acceptLanguage("pt").buildGet();
target.request().buildPost(Entity.entity(new Book()));
```

Этот код просто строит Invocation. Затем понадобится вызвать метод invoke() для активизации вашей удаленной веб-службы в стиле REST и получить обратно объект Response, который определяет соглашение (контракт) с возвращаемым экземпляром. Именно Response вы и применяете:

```
Response response = invocation.invoke();
```

Итак, если собрать все вместе, то следующие строки кода понадобятся для вызова метода GET в удаленной REST-службе, расположенной по ссылке http://www.myserver.com/book. При этом возвращается значение text/plain:

```
Client client = ClientBuilder.newClient();
WebTarget target = client.target("http://www.myserver.com/book");
Invocation invocation = target.request(MediaType.TEXT_PLAIN).buildGet();
Response response = invocation.invoke();
```

Благодаря API построителя и некоторым полезным сокращениям аналогичное поведение можно выразить всего в одной строке кода:

```
Response response = ➥
    ClientBuilder.newClient().target("http://www.myserver.com/book") ➥
    .request(MediaType.TEXT_PLAIN).get();
```

Далее рассмотрим, как манипулировать Response и использовать объекты.

Использование ответов

Класс Response позволяет потребителю в определенной степени управлять HTTP-ответами, возвращаемыми от веб-службы в стиле REST. С помощью этого API можно проверять код состояния HTTP, заголовки, cookie и, конечно же, тело сообщения (то есть объект, или сущность). В коде, приведенном ниже, используются встроенные методы для обращения к определенной низкоуровневой информации о HTTP — в частности, они позволяют узнать код состояния, длину тела сообщения, дату отправки или любой HTTP-заголовок:

```
assertTrue(response.getStatusInfo() == Response.Status.OK);
assertTrue(response.getLength() == 4);
assertTrue(response.getDate() != null);
assertTrue(response.getHeaderString("Content-type").equals("text/plain"));
```

Но в большинстве случаев нам требуется получить от Response тот объект, который был отправлен веб-службой в стиле REST. Метод readEntity считывает входной поток данных сообщения, как экземпляр указанного класса Java. Для этого используется интерфейс MessageBodyReader, поддерживающий отображение потока данных сообщения на требуемый тип. Таким образом, если вы укажете, что вам нужна строка (String), то среда времени исполнения JAX-RS применит действующий по умолчанию считыватель строк:

```
String body = response.readEntity(String.class);
```

Когда метод readEntity() вызывается с POJO, среде времени исполнения JAX-RS требуется интерфейс MessageBodyReader, определяющий тип содержимого ответа путем сопоставления. Например, если Book представляет собой компонент JAXB, а ваше содержимое относится к типу XML, то JAX-RS делегирует среде исполнения JAXB демаршалинг потока данных XML в POJO Book:

```
Book book = response.readEntity(Book.class);
```

Объединяя все вышеизложенное, следующий код вызывает удаленную веб-службу в стиле REST и получает возвращенную строковую сущность:

```
Response response = ➥
    ClientBuilder.newClient().target("http://www.myserver.com/book") ➥
                             .request().get();
String body = response.readEntity(String.class);
```

Но в методе GET предусмотрен сокращенный вариант записи, позволяющий указать желаемый тип и получить объект всего в одной строке кода (без использования объекта Response в качестве посредника):

```
String body =
    ClientBuilder.newClient().target("http://www.myserver.com/book") ➥
                             .request().get(String.class);
```

Структура потребителя REST

В отличие от JAX-WS, встроенного в Java SE и обеспечивающего вызов готовых веб-служб SOAP при наличии одного лишь JDK, при работе с JAX-RS необходимо указывать в пути к классу клиентский API. Но это единственное ограничение. В отличие от применения SOAP здесь вам не придется генерировать никаких артефактов. Поэтому потребители веб-служб в стиле REST могут происходить откуда угодно — как из любого класса Java, работающего на виртуальной машине JVM (основной класс, интеграционный тест, пакетная обработка), так и из любого компонента Java EE, работающего в контейнере (сервлет, EJB, управляемый компонент).

Разумеется, одна из сильных сторон веб-служб в стиле REST — их интероперабельность. Так, если вы разместите в Интернете набор ресурсов, то сможете обращаться к ним и с мобильных устройств (смартфоны, планшеты), и с помощью других веб-технологий (скажем, JavaScript).

Все вместе

Объединим все концепции, разобранные выше в этой главе, напишем веб-службу в стиле REST для работы с информацией о книгах, упакуем ее и развернем в GlassFish. Затем протестируем, воспользовавшись для этого cURL и интеграционным тестом (применим новый клиентский API). Наша цель — получить JAXB-компонент `Books`, представляющий собой список JPA-сущностей `Book`, отображаемых на базу данных. Служба `BookRestService` обеспечивает выполнение CRUD-операций над информацией о книге. Несмотря на то что это веб-служба с передачей состояния представления, она также является сеансовым компонентом и допускает разграничение транзакций (с применением менеджера сущностей). Выполнив развертывание, вы сможете создавать, получать или удалять записи о книгах, пользуясь HTTP-методами с помощью cURL и клиентского API JAX-RS. Благодаря JAXB и расширению Jersey вы сможете получить по два варианта представления этих книг — в XML и в JSON.

В данном примере структура каталогов строится по принципу, принятому в Maven. Все классы упаковываются в архив WAR (`chapter15-service-1.0.war`). Классы, показанные на рис. 15.4, должны находиться в следующих каталогах и файлах:

- `src/main/java` — каталог для `Books`, сущности `Book` и `BookRestService`, а также для классов, используемых для конфигурирования среды времени исполнения (подробнее о `ApplicationConfig` мы поговорим ниже);
- `src/main/resources` — включает файл `persistence.xml`, используемый ресурсом, отображающим объект `Book` на базу данных `Derby`;
- `src/test/java` — каталог для интеграционного теста `BookRestServiceIT`;
- `pom.xml` — объектная модель проекта Maven (POM), описывающая проект и его зависимости.

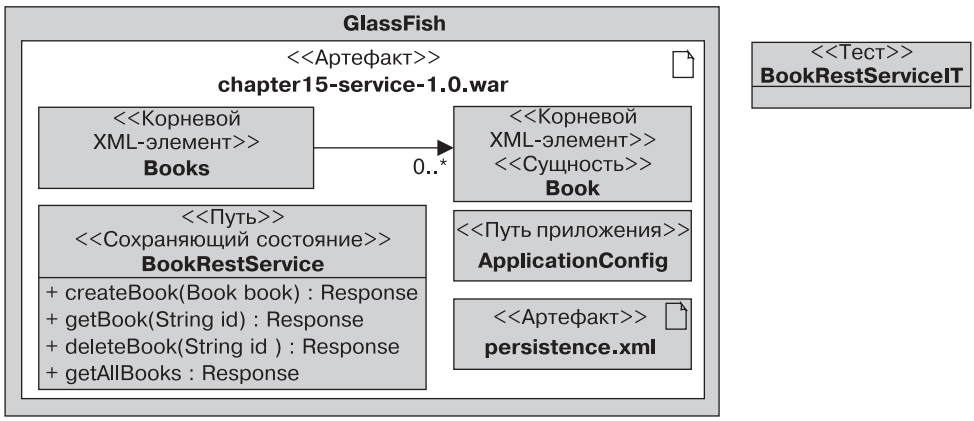


Рис. 15.4. Все вместе

Написание сущности Book

Теперь вы уже понимаете код сущности `Book`, но остается отметить еще один важный момент: данная сущность также сопровождается аннотацией JAXB `@XmlRootElement`

(листинг 15.20). Благодаря этому сущность может иметь XML-представление книги.

Листинг 15.20. Сущность Book с аннотацией JAXB

```
@Entity
@XmlRootElement
@NamedQuery(name = Book.FIND_ALL, query = "SELECT b FROM Book b")
public class Book {

  public static final String FIND_ALL = "Book.findAll";

  @Id
  @GeneratedValue
  private String id;
  @Column(nullable = false)
  private String title;
  private Float price;
  @Column(length = 2000)
  private String description;
  private String isbn;
  private Integer nbOfPage;
  private Boolean illustrations;

  // Конструкторы, геттеры, сеттеры
}
```

Эта сущность также может быть упакована с файлом `persistence.xml` (здесь он опущен ради упрощения примера).

Написание JAXB-компонента Books

Один из методов, которым обладает веб-служба в стиле REST, получает все книги из базы данных. Этот метод мог бы вернуть `List<Book>`, но в таком случае не получился бы JAXB-маршалинг. Чтобы иметь XML-представление списка книг, нам понадобится POJO, аннотированный JAXB. Как показано в листинге 15.21, класс `Books` наследует от `ArrayList<Book>` и имеет аннотацию `@XmlRootElement`.

Листинг 15.21. JAXB-компонент Books, содержащий список Book

```
@XmlRootElement
@XmlSeeAlso(Book.class)
public class Books extends ArrayList<Book> {
  public Books() {
    super();
  }

  public Books(Collection<? extends Book> c) {
    super(c);
  }

  @XmlElement(name = "book")
```

```
    public List<Book> getBooks() {
      return this;
    }

    public void setBooks(List<Book> books) {
      this.addAll(books);
    }
  }
```

Написание службы BookRestService

BookRestService — это веб-служба в стиле REST, реализованная как не сохраняющий состояние сеансовый компонент. Она использует менеджер объектов, позволяющий создавать, получать и удалять книги. Разделим эту службу на несколько частей и объясним каждую из них.

Заголовок

Заголовок BookRestService (листинг 15.22) очень важен, так как использует несколько аннотаций метаданных. В JAX-RS пользователи обращаются к службам, вызывая URI. Аннотация @Path("/book") указывает корневой путь ресурса (URL, по которому можно обратиться к этому ресурсу). В данном случае идентификатор ресурса будет примерно таким: http://localhost:8080/book.

Листинг 15.22. Заголовок BookRestService

```
@Path("/book")
@Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
@Stateless
public class BookRestService {
  @PersistenceContext(unitName = "chapter15PU")
  private EntityManager em;
  @Context
  private UriInfo uriInfo;
  // ...
```

Аннотации @Produces и @Consumes определяют задаваемый по умолчанию тип содержимого, который будет производиться или использоваться этим ресурсом: XML или JSON. Наконец, мы находим здесь аннотацию @Stateless, подробно рассмотренную в главе 7. Она информирует контейнер о том, что веб-служба в стиле REST должна интерпретироваться как EJB и обеспечивать разграничение транзакций при обращениях к базе данных. В эту службу внедрена ссылка на менеджер объектов, а также UriInfo.

Создание новой книги

Придерживаясь семантики REST, мы используем HTTP-операцию POST для создания нового ресурса на XML или JSON (в зависимости от того, что указано в заголовке с аннотацией @Consumes). По умолчанию любой метод использует XML или JSON, это же верно для метода createBook(). Как показано в листинге 15.23, метод принимает Book в качестве параметра; не забывайте, что сущность Book также является объектом

JAXB и после выполнения демаршалинга XML в объект Book менеджер объектов может долговременно сохранить его. Если параметр book является нулевым, генерируется исключение BadRequestException (код состояния 400 – Неверный запрос).

Листинг 15.23. Метод createBook службы BookRestService

```
// ...
@POST
public Response createBook(Book book) {
  if (book == null)
    throw new BadRequestException();

  em.persist(book);
  URI bookUri = uriInfo.getAbsolutePathBuilder().path(book.getId()).build();
  return Response.created(bookUri).build();
}
// ...
```

Этот метод возвращает Response, представляющий собой URI свежей записи о книге. Мы могли бы вернуть код состояния 200 – Хорошо (Response.ok()), указывающий, что создание записи о книге прошло успешно. Но в соответствии с принципами REST метод должен возвращать код 201 (или 204), указывая, что запрос был выполнен и привел к созданию нового ресурса (Response.created()). На новоиспеченный ресурс можно сослаться по URI, возвращенному в ответе (bookUri).

Чтобы создать ресурс с помощью кода, приведенного в листинге 15.23, нам потребуется отослать информацию в XML или JSON. Текст JSON менее пространен. Инструмент командной строки cURL использует метод POST и передает информацию в формате JSON. Эта информация должна соответствовать правилам отображения JSON/XML, действующим в Jersey (также не забывайте, что XML, в свою очередь, отображается с объекта Book по правилам JAXB):

```
$ curl -X POST --data-binary "{\"description\":\"Научно-фантастическая
книга\",
\"illustrations\":false,\"isbn\":\"1-84023-742-2\",\"nbOfPage\":354,
\"price\":12.5,\"title\":\"Автостопом по Галактике\"}"
-H "Content-Type: application/json"
http://localhost:8080/chapter15-service-1.0/rs/book -v
```

Развернутый режим вывода cURL (с аргументом -v) отображает HTTP-запрос и ответ (см. следующий вывод). В ответе вы видите URI созданной книги, причем идентификатор этого ресурса равен 601:

```
> POST /chapter15-service-1.0/rs/book HTTP/1.1
> User-Agent: curl/7.23.1 (x86_64-apple-darwin11.2.0) libcurl/7.23.1
> Host: localhost:8080
> Accept: */*
> Content-Type: application/json
> Content-Length: 165
>
< HTTP/1.1201 Created
< Server: GlassFish Server Open Source Edition 4.0
```

```
< Location: http://localhost:8080/chapter15-service-1.0/rs/book/601
< Date: Thu, 29 Nov 201221:49:44 GMT
< Content-Length: 0
```

Получение книги по ID

Чтобы получить книгу по ее идентификатору, нужно использовать в запросе URL /book/{id книги}. id применяется в качестве параметра для нахождения книги в базе данных. Если в листинге 15.24 книга не найдена, то генерируется исключение NotFoundException (404). В зависимости от MIME-типа метод getBook() вернет представление записи книги в XML или JSON.

Листинг 15.24. Метод getBook службы BookRestService

```
// ...
@GET
@Path("{id}")
public Response getBook(@PathParam("id") String id) {
  Book book = em.find(Book.class, id);

  if (book == null)
    throw new NotFoundException();

  return Response.ok(book).build();
}
// ...
```

Аннотация @Path указывает подпуть внутри уже заданного пути на уровне класса. При использовании синтаксиса {id} URL-элемент связывается с параметром метода. Воспользуемся cURL для доступа к книге с идентификатором 601 и получения JSON-представления:

```
$ curl -X GET -H "Accept: application/json" ➥
    http://localhost:8080/chapter15-service-1.0/rs/book/601
{"description":"Научно-фантастическая книга","id":"1", ➥
    "illustrations":false,"isbn":"1-84023-742-2","nbOfPage":354, ➥
    "price":12.5,"title":"H2G2"}
```

Изменив свойство заголовка Accept, мы сможем в этом же коде вернуть XML-представление книги с идентификатором 601:

```
$ curl -X GET -H "Accept: application/xml" ➥
    http://localhost:8080/chapter15-service-1.0/rs/book/601
<?xml version="1.0" encoding="UTF-8" ➥
    standalone="yes"?><book><description>Научно-фантастическая ➥
    книга</description><id>601</id><illustrations>false</illustrations> ➥
    <isbn>1-84023-742-2</isbn><nbOfPage>354</nbOfPage><price>12.5</price> ➥
    <title>H2G2</title></book>
```

Получение всех книг

Для получения из базы данных всех книг код из листинга 15.25 использует запрос TypedQuery, результат которого устанавливается в класс Books. Не забывайте, что

этот класс расширяет ArrayList<Book> и дает XML-представление списка благодаря аннотациям JAXB. В результате получаем ответ с кодом состояния 200 – Хорошо, причем в теле сообщения содержится представление книг.

Листинг 15.25. Метод getBooks службы BookRestService

```
// ...
@GET
public Response getBooks() {
  TypedQuery<Book> query = em.createNamedQuery(Book.FIND_ALL, Book.class);
  Books books = new Books(query.getResultList());
  return Response.ok(books).build();
}
// ...
```

Опять же, если вы пожелаете быстро протестировать код с помощью cUrl, откройте командную строку и введите в нее следующие команды, чтобы получить представление списка книг либо в XML, либо в JSON:

```
$ curl -X GET -H "Accept: application/json" ➥
http://localhost:8080/chapter15-service-1.0/rs/book
$ curl -X GET -H "Accept: application/xml" ➥
http://localhost:8080/chapter15-service-1.0/rs/book
```

Удаление книги

Метод deleteBook() в листинге 15.26 соответствует формату метода getBook(), так как использует подпуть и ID в качестве параметра. Единственное отличие от метода getBook() заключается в том, что во втором случае применяется HTTP-операция DELETE (а не GET). Если книга не найдена в базе данных, то генерируется исключение NotFoundException, в противном случае книга удаляется и возвращается код состояния 204 – Нет содержимого.

Листинг 15.26. Метод deleteBook службы BookRestService

```
// ...
@DELETE
@Path("{id}")
public Response deleteBook(@PathParam("id") String id) {
  Book book = em.find(Book.class, id);

  if (book == null)
    throw new NotFoundException();
  em.remove(book);
  return Response.noContent().build();
}
// ...
```

Если бы мы воспользовались развернутым режимом вывода cURL (с аргументом -v), то увидели бы отправку запроса DELETE. В ответе же присутствует код состояния 204 – Нет содержимого. Он указывает, что ресурс больше не существует:

```
$ curl -X DELETE http://localhost:8080/chapter15-service-1.0/rs/book/601 -v >
DELETE /chapter15-service-1.0/rs/book/601 HTTP/1.1
```

```
> User-Agent: curl/7.23.1 (x86_64-apple-darwin11.2.0) libcurl/7.23.1
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 204 No Content
< Server: GlassFish Server Open Source Edition 4.0
```

Конфигурирование JAX-RS

Перед развертыванием службы BookRestService и сущности Book нам потребуется зарегистрировать в Jersey паттерн url; это делается для перехватывания HTTP-вызовов, идущих к службам. Таким образом, запросы, отсылаемые к пути /rs, будут перехватываться Jersey. Вы можете либо задать этот паттерн url при конфигурировании сервлета Jersey в файле web.xml, либо использовать аннотацию @ApplicationPath (листинг 15.27). Класс ApplicationConfig должен расширять javax.ws.rs.core.Application и определять все веб-службы в стиле REST (здесь — c.add(BookRestService.class)), а также выполнять все прочие необходимые расширения (c.add(MOXyJsonProvider.class)).

Листинг 15.27. Класс ApplicationConfig, объявляющий URL-паттерн для /rs

```
@ApplicationPath("rs")
public class ApplicationConfig extends Application {
  private final Set<Class<?>> classes;

  public ApplicationConfig() {
    HashSet<Class<?>> c = new HashSet<>();
    c.add(BookRestService.class);
    c.add(MOXyJsonProvider.class);
    classes = Collections.unmodifiableSet(c);
  }

  @Override
  public Set<Class<?>> getClasses() {
    return classes;
  }
}
```

Этот класс должен находиться где-то в вашем проекте (конкретный пакет не имеет значения), и благодаря аннотации @ApplicationPath он будет отображать запросы на URL-шаблон /rs/*. Это означает, что всякий раз, когда очередной URL будет начинаться с /rs/, Jersey будет его обрабатывать. Действительно, во всех тех примерах, которые я использовал в cURL, все URL ресурсов начинались с /rs:

```
$ curl -X GET -H "Accept: application/json" ➥
    http://localhost:8080/chapter15-service-1.0/rs/book
```

ПРИМЕЧАНИЕ

В главе 12 были рассмотрены некоторые спецификации, связанные с XML и JSON. Спецификации делятся на две группы: касающиеся обработки (JAXP и JSON-P) и связывания (JAXB). Благодаря связыванию JAX-RS будет преобразовывать список компонентов в XML с помощью встроенных интерфейсов MessageBodyReader и MessageBodyWriter,

основанных на JAXB. К сожалению, пока не существует технологии связывания для работы с JSON (ожидается, что она появится в ближайшие годы и будет называться JSON-B). Поэтому на данный момент невозможно автоматически выполнить маршалинг компонента в JSON. Вот почему в классе ApplicationConfig нам требуется добавить Jersey-специфичный поставщик (MOXyJsonProvider). Так мы получаем JSON-интерфейсы MessageBodyReader и MessageBodyWriter. Но такой прием не обладает портируемостью на все реализации JAX-RS.

Компиляция и упаковка с помощью Maven

Теперь все классы должны быть скомпилированы и упакованы в WAR-файл (`<packaging>war</packaging>`). Файл `pom.xml`, показанный в листинге 15.28, объявляет все зависимости, необходимые для компиляции кода (эти зависимости находятся в `glassfish-embedded-all`). Как вы помните, JAXB входит в состав Java SE, и эту зависимость добавлять не требуется. Обратите также внимание, что в файле `pom.xml` объявляется плагин Failsafe, специально разработанный для выполнения интеграционных тестов (его мы рассмотрим ниже для запуска класса `BookRestServiceIT`, предназначенного именно для интеграционного тестирования).

Листинг 15.28. Файл pom.xml для компиляции, тестирования и упаковки веб-службы с передачей состояния представления

```
<?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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <artifactId>chapter15</artifactId>
    <groupId>org.agoncal.book.javaee7</groupId>
    <version>1.0</version>
  </parent>

  <groupId>org.agoncal.book.javaee7.chapter15</groupId>
  <artifactId>chapter15-service</artifactId>
  <version>1.0</version>
  <packaging>war</packaging>

  <dependencies>
    <dependency>
      <groupId>org.glassfish.main.extras</groupId>
      <artifactId>glassfish-embedded-all</artifactId>
      <version>4.0</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
```

```
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.5.1</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-war-plugin</artifactId>
        <version>2.2</version>
        <configuration>
          <failOnMissingWebXml>false</failOnMissingWebXml>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>2.12.4</version>
        <executions>
          <execution>
            <id>integration-test</id>
            <goals>
              <goal>integration-test</goal>
              <goal>verify</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Для компиляции и упаковки классов откройте командную строку в корневом каталоге, содержащем файл `pom.xml`, а затем введите следующую команду Maven:

```
$ mvn package
```

Перейдите в целевой каталог, найдите файл `chapter15-service-1.0.war` и откройте его. Обратите внимание: `Book.class`, `Books.class`, `ApplicationConfig.class` и `BookRestService.class` находятся в каталоге `WEB-INF\classes`. В названном WAR-архиве также упакован файл `persistence.xml`.

Развертывание в GlassFish

Когда код будет упакован, убедитесь, что у вас работают GlassFish и Derby, после чего разверните WAR-файл, воспользовавшись инструментом командной строки `asadmin`. Откройте консоль, перейдите в каталог target, где расположен файл `chapter15-service-1.0.war`, и введите:

```
$ asadmin deploy chapter15-service-1.0.war
```

Если развертывание пройдет успешно, следующая команда должна вернуть имя развернутого компонента и его тип:

```
$ asadmin list-components
chapter15-service-1.0 <ejb, web>
```

Теперь, когда приложение развернуто, можете воспользоваться инструментом cURL из командной строки, чтобы создавать книжные ресурсы с помощью запросов `POST`, а также получать все ресурсы (или только один конкретный) с помощью запросов `GET`. Запросы `DELETE` удаляют книжные ресурсы.

WADL

В начале этой главы я вкратце упомянул о WADL (языке описания веб-приложений) и сказал, что он пока не стандартизирован и не очень активно используется в архитектуре REST. Но если вы хотели бы взглянуть, как на этом языке можно описать нашу REST-службу `BookRestService`, обратите внимание на такой URL: http://localhost/8080/chapter15-service-1.0/rs/application.wadl. В листинге 15.29 показана выдержка из этого WADL-описания.

Листинг 15.29. Выдержка из описания на языке WADL, сгенерированного GlassFish для веб-службы BookRestService в стиле REST

```
<application xmlns="http://wadl.dev.java.net/2009/02">
  <resources base="http://localhost:8080/chapter15-service-1.0/rs/">
    <resource path="/book">
      <method id="POST" name="POST">
        <request>
          <representation element="book" mediaType="application/xml"/>
          <representation element="book" mediaType="application/json"/>
        </request>
      </method>
    <resource path="{id}">
      <param name="id" style="template" type="xs:string"/>
        <method id="GET{id}" name="GET">
          <response>
            <representation mediaType="application/xml"/>
            <representation mediaType="application/json"/>
          </response>
        </method>
        <method id="DELETE{id}" name="DELETE"/>
      </resource>
    </resource>
  ...
```

```
    </resources>
  </application>
```

Код WADL в листинге 15.29 описывает корневой путь (`http://localhost:8080/chapter15-service-1.0/rs/`) и все подпути, доступные в REST-службе (`/book` и `{id}`). Кроме того, здесь описаны HTTP-методы, которые вы можете вызывать (`POST`, `GET`, `DELETE`...).

Написание интеграционного теста BookRestServiceIT

Теперь, когда код упакован и развернут в GlassFish, мы можем написать интеграционный тест, в ходе которого будем выполнять HTTP-запросы к веб-службе REST. При этом воспользуемся новым API JAX-RS 2.0 `Client`. Интеграционные тесты отличаются от модульных, так как в них ваш код проверяется неизолированно. Этим тестам требуются все контейнерные службы. Как правило, если GlassFish и Derby в данный момент не работают, код из листинга 15.30 тоже не сработает.

Первые два метода тестируют отказы. Тест `shouldNotCreateANullBook` гарантирует, что вы не сможете создать книгу с нулевым объектом `Book`. Итак, он посылает нулевую сущность и ожидает получить код состояния `400 – Плохой запрос`. В тестовом случае `shouldNotFindTheBookID` передается неизвестный ID книги, после чего предполагается получить код состояния `404 – Не найдено`.

Тестовый случай `shouldCreateAndDeleteABook` немного сложнее первого, так как в нем вызывается несколько операций. Во-первых, этот тест отправляет XML-представление объекта `Book` и позволяет убедиться, что возвращается код состояния `201 – Создан`. Переменная `bookURI` соответствует URI только что созданной записи о книге. В тестовом случае этот URI применяется для запроса новой книги. Затем код считывает тело сообщения, приводит его к классу `Book` (`book = response.readEntity(Book.class)`) и выполняет утверждения, гарантирующие корректность значений. Затем вызывается метод `DELETE`, удаляющий запись о книге из базы данных и удостоверяющий, что в ответ приходит код состояния `204 – Нет содержимого`. Последний запрос `GET`, выполняемый с ресурсом, гарантирует удаление этого ресурса, проверяя наличие кода состояния `404 – Не найдено`.

Листинг 15.30. Класс ApplicationConfig, объявляющий URL-шаблон /rs

```
public class BookRestServiceIT {

  private static URI uri = UriBuilder.
  fromUri("http://localhost/chapter15-service-1.0/rs/book").port(8080).build ();
  private static Client client = ClientBuilder.newClient();

  @Test
  public void shouldNotCreateANullBook() throws JAXBException {

    // Отправка нулевой книги методом POST
    Response response = client.target(uri).request().post(Entity.entity(null, ➥
    MediaType.APPLICATION_XML));
```

```
    assertEquals(Response.Status.BAD_REQUEST, response.getStatusInfo());
  }

  @Test
  public void shouldNotFindTheBookID() throws JAXBException {

    // Получение книги с неизвестным ID методом GET
    Response response = client.target(uri).path("unknownID").request(). get ();
    assertEquals(Response.Status.NOT_FOUND, response.getStatusInfo());
  }

  @Test
  public void shouldCreateAndDeleteABook() throws JAXBException {

    Book book = new Book("H2G2", 12.5F, "Science book", "1-84023-742-2", 354,
false);
    // Посылаем книгу методом POST
    Response response = client.target(uri).request().post(Entity.entity(book, ➥
    MediaType.APPLICATION_XML));
    assertEquals(Response.Status.CREATED, response.getStatusInfo());
    URI bookURI = response.getLocation();

    // Имея местоположнение, получаем книгу методом GET
    response = client.target(bookURI).request().get();
    book = response.readEntity(Book.class);
    assertEquals(Response.Status.OK, response.getStatusInfo());
    assertEquals("H2G2", book.getTitle());

    // Получаем id книги и удаляем ее методом DELETE
    String bookId = bookURI.toString().split("/")[6];
    response = client.target(uri).path(bookId).request().delete();
    assertEquals(Response.Status.NO_CONTENT, response.getStatusInfo());

    // Методом GET получаем книгу Book и проверяем, была ли она удалена
    response = client.target(bookURI).request().get();
    assertEquals(Response.Status.NOT_FOUND, response.getStatusInfo());
  }
}
```

Убедитесь, что GlassFish и Derby работают нормально и приложение развернуто, а затем выполните этот тест с применением плагина Maven Failsafe. Для этого введите в командной строке следующую команду Maven:

```
$ mvn failsafe:integration-test
```

Резюме

В предыдущей главе были рассмотрены веб-службы SOAP. Теперь вы уже должны понимать разницу между веб-службами, работающими на базе JAX-RS и JAX-WS. При работе REST широко задействуется HTTP, поэтому данная глава началась

с общего знакомства с концепциями ресурсов, представления, адресуемости, связности, а также единообразных интерфейсов. Вы узнали, как с помощью простых HTTP-операций (GET, POST, PUT и др.) получать доступ к любому ресурсу, развернутому в Интернете.

Затем в этой главе был подробно рассмотрен протокол HTTP, работа которого строится на базе обмена запросами и ответами. По HTTP передаются сообщения, состоящие из заголовков, cookie и полезной нагрузки. С помощью HTTP-заголовков и согласования содержимого веб-службы в стиле REST могут выбирать из одного и того же URI подходящий тип содержимого. Можно использовать кэширование, помогающее оптимизировать сетевой трафик благодаря условным запросам (вспомните материал об использовании дат и меток ETag). Поскольку REST-службы построены на HTTP, такая оптимизация может применяться и с ними. Благодаря новому клиентскому API JAX-RS 2.0 мы смогли программно вызвать несколько веб-служб в стиле REST.

JAX-RS — это API языка Java, входящий в состав платформы Java EE 7 и упрощающий разработку веб-служб с передачей состояния представления. Работая с набором аннотаций, можно определить путь и подпути вашего ресурса, извлекать различные параметры или выполнять отображение на HTTP-методы (@GET, @POST и т. д.). При разработке веб-служб в стиле REST необходимо учитывать природу ресурсов, способы их связывания друг с другом, а также способы управления их состоянием с помощью HTTP. Теперь вы сами сумеете продемонстрировать в Сети несколько служб и посмотреть, на каких разнообразных устройствах можно будет их использовать.