Дебу Панда
Реза Рахман
Райан Купрак
Майкл Ремижан

EJB 3
В ДЕЙСТВИИ

ДМК ПРЕСС
издательство

MANNING

Дебу Панда, Реза Рахман, Райан Купрак, Майкл Ремижан

EJB 3
в действии

EJB 3 in Action
Second Edition

DEBU PANDA
REZA RAHMAN
RYAN CUPRAK
MICHAEL REMIJAN

MANNING
SHELTER ISLAND

EJB 3 в действии

ДЕБУ ПАНДА
РЕЗА РАХМАН
РАЙАН КУПРАК
МАЙКЛ РЕМИЖАН

ДМК ПРЕСС
издательство
Москва, 2015

УДК 004.455.2
ББК 32.973.41
П16

П16 Дебу Панда, Реза Рахман, Райан Купрак, Майкл Ремижан
EJB 3 в действии. / Пер. с англ. Киселев А. Н. – М.: ДМК Пресс, 2015. – 618 с.: ил.

ISBN 978-5-97060-135-8

Фреймворк EJB 3 предоставляет стандартныий способ оформления прикладной логики в виде управляемых модулей, которые выполняются на стороне сервера, упрощая тем самым создание, сопровождение и расширение приложений Java EE. Версия EJB 3.2 включает большее число расширений и более тесно интегрируется с другими технологиями Java, такими как CDI, делая разработку еще проще. Книга знакомит читателя с EJB на многочисленных примерах кода, сценариях из реальной жизни и иллюстрациях. Помимо основ в ней описываются некоторые особенности внутренней реализации, наиболее эффективные приемы использования, шаблоны проектирования, даются советы по оптимизации производительности и различные способы доступа, включая веб-службы, службы REST и веб-сокеты.

Издание предназначено программистам, уже знающим язык Java. Опыт работы с EJB или Java EE не требуется.

УДК 004.455.2
ББК 32.973.41

ISBN 978-1-93518-299-3 (англ.)
ISBN 978-5-97060-135-8 (рус.)

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Первоначально технология EJB появилась под влиянием идей, заложенных в технологии распределенных вычислений, таких как CORBA, и предназначалась для поддержки масштабирования серверных приложений. Во время бума доткомов вокруг EJB и J2EE была поднята большая рекламная шумиха.

Первоначальной целью EJB было предоставить простую альтернативу технологии CORBA на основе стандартной инфраструктуры разработки и компонентов многократного пользования. К моменту выхода EJB 2 стало очевидно, что эта инфраструктура превратилась в новый стандарт разработки серверных приложений. Она предоставляет промышленным разработчикам все необходимое – поддержку удаленных взаимодействий, механизмы управления транзакциями, средства обеспечения безопасности, обработки и хранения информации, и веб-службы – но все это были тяжеловесные механизмы, требующие от разработчиков основное внимание уделять особенностям взаимодействий с самой инфраструктурой, чем реализации бизнес-логики приложений. Из-за включения в EJB все новых особенностей, основатели этой инфраструктуры оказались не в состоянии справиться со все возрастающей сложностью.

С появлением недовольств, вызванных ограничениями в EJB 2, стали появляться новые инструменты с открытым исходным кодом. Эти инструменты являются ярким признаком растущего недовольства сложностью Java EE. Однако, несмотря на благие намерения, эти инструменты еще больше осложнили разработку промышленных приложений, так как они отклонялись от стандартов, лежащих в основе сервера приложений, где эти инструменты должны использоваться. В этот период была запущен процесс Java Community Process (JCP) и сформирована экспертная группа для выполнения работ по упрощению разработки на основе Java EE. Это была единственная причина начала разработки Java EE 5 и EJB 3.

Для технологии с такой широкой областью применения, изменения в EJB 3 стали просто оглушительными. EJB 3 благополучно объединила в себе инновационные приемы, существенно упрощающие разработку компонентов. В число этих приемов входят: использование аннотаций, метапрограммирование, внедрение зависимостей, AspectJ-подобные интерцепторы и интеллектуальное использование значений по умолчанию. Произошел отказ от тяжеловесной модели программирования на основе наследования в пользу более легковесной модели на основе простых объектов Java (Plain Old Java Object, POJO), а подробные описания настроек на языке XML ушли с пути разработчика.

Еще более существенными оказались изменения в модели хранения данных. В EJB 3 произошел отказ от несовершенной модели Entity Beans, использовавшейся в EJB 2, в пользу легковесного программного интерфейса Java Persistence API (JPA). В отличие от Entity Beans, интерфейс JPA не основывается на контейнерах. Он больше похож на инструменты объектно-реляционного отображения (Object Relational Mapping, ORM), созданные сообществом в ответ на сложность Entity Beans. JPA может использоваться и внутри, и за пределами сервера Java Enterprise, и в настоящее время де-факто считается стандартом доступа к хранимым данным для Java. Его язык запросов Java Persistence Query Language (JPQL) стандартизует объектно-реляционные запросы, а также поддерживает запросы на языке SQL.

Изменения в EJB 3 были благосклонно восприняты сообществом Java. Упрощение спецификации привело к ее широкому распространению среди новых проектов. Все больше компаний повторно рассматривают возможность применения прежде «неудачной» технологии EJB и склоняются к положительному решению. С выходом версии EJB 3.2 число положительных решений выросло еще больше. Спецификация EJB 3.2 сделала поддержку EJB 2 необязательной, что обусловило скорый закат старой технологии и дальнейший рост числа инноваций в EJB 3. Спецификация EJB 3.2 также внесла существенные расширения в компоненты, управляемые событиям (Message-Driven Bean, MDB), значительно упростив реализацию обмена сообщениями. Версия EJB 3.2 улучшила поддержку сеансовых компонентов с сохранением состояния (stateful session bean) и их локальных интерфейсов, а также внесла важнейшие улучшения в службы таймеров. Все это и многое другое ждет вас в EJB 3.2.

Так как EJB опирается на POJO, любой разработчик на Java легко сможет превратиться в EJB-разработчика. Простые аннотации привносят в прикладной код надежные транзакции, механизмы поддержки безопасности и возможность выступать в качестве веб-служб, с целью упростить взаимодействия программных продуктов в пределах компании. Мы старались сделать нашу книгу отличной от других по EJB, наполнив ее практическими примерами, описанием эффективных приемов, а также рекомендациями по улучшению производительности. Мы особенно отметим новинки, появившиеся в спецификации EJB версии 3.2, дающие дополнительные инструменты для использования в процессе разработки. Мы надеемся, что это второе издание книги поможет вам быстро понять, как эффективнее использовать EJB 3 в вашем следующем промышленном приложении.

Благодарности

Работа над книгой требует усилий многих людей и порой бывает сложно перечислить всех, кто помогал в ее создании. Прежде всего мы хотели бы поблагодарить всех сотрудников издательства Manning за их одобрение и поддержку, и особенно владельца издательства Марджана Бейса (Marjan Bace), совладельца Майкла Стефенса (Michael Stephens) и нашего редактора Нермину Миллер (Nermina Miller). Нам также хотелось бы выразить благодарность другим сотрудникам издательства Manning, участвовавших в работе над проектом на разных этапах: редактору-

рецензенту Оливии Бус (Olivia Booth); редактору проекта Джоди Аллен (Jodie Allen); менеджеру по развитию Маурину Спенсеру (Maureen Spencer); техническому корректору Дипаку Вохра (Deepak Vohra), выполнившему окончательную правку книги непосредственно перед выпуском в печать; Линде Ректенволд (Linda Recktenwald) и Мелоди Долаб (Melody Dolab), выполнившими литературное редактирование и корректуру; и верстальщику Денису Далиннику (Dennis Dalinnik), превратившему наши документы Word в настоящую книгу! Спасибо также всем, кто незримо участвовал в подготовке книги к публикации.

У нас было много рецензентов, потративших свое драгоценное время на чтение рукописей на разных этапах их подготовки, и их отзывы помогли значительно повысить качество книги. Мы очень признательны вам: Артур Новак (Artur Nowak), Азиз Рахман (Aziz Rahman), Боб Касацца (Bob Casazza), Кристоф Мартини (Christophe Martini), Дэвид Стронг (David Strong), Джит Марвах (Jeet Marwah), Джон Гриффин (John Griffin), Джонас Банди (Jonas Bandi), Джозеф Лехнер (Josef Lehner), Юрген де Коммер (Jürgen De Commer), Каран Малхи (Karan Malhi), Халид Муктар (Khalid Muktar), Корай Гюслю (Koray Güclü), Луис Пенья (Luis Peña), Матиас Агезли (Matthias Agethle), Палак Матхур (Palak Mathur), Павел Розенблюм (Pavel Rozenblioum), Рик Вагнер (Rick Wagner), Сумит Пал (Sumit Pal), Виктор Агилар (Victor Aguilar), Веллингтон Пинхейро (Wellington Pinheiro) и Зороджай Макая (Zorodzayi Mukuya).

Наконец, спасибо редакторам из Manning Early Access Program (MEAP), вычитывавшим наши главы по мере их готовности и присылавшим комментарии и исправления на форум книги. Ваша помощь сделала эту книгу намного лучше.

Дебу Панда (Debu Panda)

Я хочу поблагодарить мою супругу Ренуку (Renuka), за ее одобрение и поддержку, за то, что терпела, когда я поздно ложился и рано вставал, и когда все выходные проводил в работе над первым изданием книги. Я также хочу сказать спасибо моим детям, Ништхе (Nistha) и Нишиту (Nisheet), которым приходилось делить их папу с компьютером.

Большое спасибо моим соавторам, Резу Рахману (Reza Rahman), Райану Купраку (Ryan Cuprak) и Майклу Ремижану (Michael Remijan), которые упорно трудились над вторым изданием этой книги.

Реза Рахман (Reza Rahman)

Даже путь в тысячу ли начинается с первого шага.
– Лао-Цзы

Не думаю, что приступая к работе над первым изданием этой книги, кто-то из авторов представлял себе, насколько успешной станет книга и как это отразится на нас самих. Сегодня, оглядываясь назад, я могу сказать, что книга добилась оглушительного успеха, а ее создание стало первым шагом на долгом пути, длиной последние несколько лет. Должен признаться, что продолжаю получать удовольствие от каждой минуты на этом пути. После завершения работы над первым изданием

я стал все больше и больше участвовать в жизни сообщества Java. Я вошел в состав различных экспертных групп по Java EE, включая экспертную группу EJB, и получил уникальную возможность написать свободно распространяемый контейнер EJB практически с нуля, а теперь еще оказался в центре команды SunOracle, занимающейся популяризацией и продвижением Java EE.

В результате всего этого у меня практически не осталось свободного времени, которого было в достатке, когда я работал над первым изданием. Это основная причина, почему нам пришлось пропустить издание книги, посвященное Java EE 6 и EJB 3.1. И, как мне кажется, что к лучшему, потому что Java EE 7 оказалась еще более мощной и привлекательной платформой, что мы и попытались продемонстрировать в этом издании книги. Я очень благодарен Майклу и Райану, что приняли решение и сыграли важную роль в создании достойного второго издания. Я также благодарен многим членам сообщества Java EE, с которыми я имел честь работать. Наконец, я благодарен моей жене Николь (Nicole) и дочери Зехре (Zehra), что безоговорочно позволили мне поддаться моей страсти. Итак, мой путь продолжается.

Райан Купрак (Ryan Cuprak)

Работа над этой книгой стала бы для меня невозможной без поддержки семьи и друзей. Я особенно хотел бы поблагодарить Эльзу (Elsa), любовь всей моей жизни, которая поддерживала и подбадривала меня в течение длинного и трудного периода, когда я проводил ночи напролет, сгорбившись за компьютером. Наконец, я хочу сказать спасибо Резу, что предложил мне поработать над этим проектом и подвигнул меня обратить внимание на Java EE много лет тому назад.

Майкл Ремижан (Michael Remijan)

Моя красавица жена Келли (Kelly) и моя дочка София (Sophia) – первые, кому я хочу сказать спасибо. Они разделили со мной это приключение и без их поддержки ранними утрами, поздними вечерами и на выходных я не смог бы спокойно заниматься исследованиями и работой над этим проектом. Келли – мой лучший друг, она подбадривала меня во всех моих начинаниях, и это добавляет мне сил и уверенности. Я люблю ее очень нежно. И я счастлив, что у меня такая замечательная семья.

Мои соавторы – Дебу, Райан и Реза – следующие в моем списке: большое спасибо вам. Эта книга является плодом коллективных усилий. EJB – обширная технология, обладающая огромным множеством возможностей; отсюда и такой размер книги. Написать такую книгу в одиночку практически невозможно, поэтому наше сотрудничество сыграло решающую роль в ее создании. Для меня было большой честью работать с такими талантливыми коллегами.

Наконец, спасибо всем сотрудникам издательства Manning, потративших бесчисленные часы на обзоры этой книги и поддерживавшими процесс ее подготовки, особенно хотелось бы отметить Маурина Спенсера (Maureen Spencer) и Джоди Аллен (Jodie Allen). Спасибо также Кристине Рудлофф (Christina Rudloff), пригласившей меня в проект – без этого я не стал бы одним из соавторов книги.

О книге

EJB 3 позволяет взглянуть на разработку серверных Java-приложений с неожиданной стороны. Поэтому мы старались наполнить эту книгу о EJB неожиданными для вас фактами.

Большинство книг о Java-программировании на стороне сервера чрезвычайно тяжелы в чтении – они наполнены теорией, слегка сдобренной нравоучениями, и предназначаются для опытных разработчиков. Хотя мы легко укладываемся в стереотип гиков и не обладаем талантами комедиантов или конферансье, мы все же попытались добавить живых красок в нашу книгу, чтобы сделать ее простой и практичной, насколько это возможно. Мы избрали дружественный и неформальный тон общения с читателем и постарались уделить основное внимание обсуждению примеров решения задач, с которыми многие из нас сталкиваются в своей повседневной работе. В большинстве случаев мы сначала рассказываем о задаче, которую требуется решить; показываем код, решающий ее с помощью EJB 3; а затем исследуем особенности этой технологии, использованные в коде.

Мы будем рассказывать о теоретических предпосылках, только когда это действительно необходимо. Мы старались избегать теоретических рассуждений ради самих рассуждений, чтобы сделать наше с вами общение как можно более живым. Цель этой книги – помочь вам быстро и эффективно освоить EJB 3, а не служить исчерпывающим справочником. Мы не будем рассматривать редко используемые возможности. Зато мы глубоко будем проникать в наиболее полезные особенности EJB 3 и родственные технологии. Мы будем обсуждать разные варианты, чтобы вы могли делать осознанный выбор, и расскажем о наиболее распространенных ловушках, а также покажем испытанные приемы.

Коль скоро вы выбрали эту книгу, весьма маловероятно, что вы новичок в Java. Мы полагаем, что у вас уже есть опыт программирования на этом языке, и, возможно, вам доводилось заниматься разработкой веб-приложений с применением таких технологий, как JSF, Struts, JSP или сервлеты (servlets). Мы также предполагаем, что вы знакомы с технологиями баз данных, такими как JDBC, и имеете хотя бы минимальное представление о языке SQL. Для работы с этой книгой вам не понадобится опыт использования EJB 2.x; EJB 3 полностью отличается от нее. Мы не рассчитываем, что вы знакомы с какими-либо технологиями Java EE, на которых основывается технология EJB, такими как Java Naming and Directory Interface (JNDI), Java Remote Method Invocation (RMI) и Java Messaging Service (JMS). Более того, мы будем исходить из предположения, что вы не знакомы с понятиями промежуточного слоя, такими как удаленные взаимодействия (remoting), организация пулов (pooling), конкурентное (или параллельное) программирование, безопасность и распределенные транзакции. Эта книга идеально подходит для разработчиков на Java, имеющих двух-трех летний опыт работы, интересующихся технологией EJB 3.

Вы можете найти в этой книге одно важное отличие от других подобных ей книг. EJB – это серверная связующая (или промежуточная) технология. Это означает, что она существует не в вакууме и должна интегрироваться с другими технологи-

ями для выполнения своей миссии. На протяжении всей книги мы будем рассказывать о том, как EJB 3 интегрируется с такими технологиями, как JNDI, JMS, JSF, JSP, Servlets, AJAX и даже с клиентами Java SE на основе Swing.

Эта книга рассказывает о EJB 3, как о стандарте, а не как о конкретной технологии сервера приложений. По этой причине мы не будем пытаться обсуждать какую-то определенную реализацию – все примеры программного кода в этой книге построены так, что смогут выполняться в любом контейнере EJB 3 и взаимодействовать с любым провайдером доступа к хранимым данным. На веб-сайте поддержки книги, по адресу: www.manning.com/EJB3inActionSecondEdition, вы найдете инструкции, как опробовать примеры в GlassFish и Oracle Application Server 10g. Размещение инструкций по использованию конкретных серверов приложений на веб-сайте вместо книги позволит нам поддерживать их в актуальном состоянии и дополнять новыми сведениями о последних реализациях.

Структура книги

Эта книга делится на четыре части.

В первой части дается общий обзор EJB. В главе 1 вы познакомитесь с EJB 3 и типами EJB, и составите некоторое представление о EJB 3. Здесь также рассказывается об изменениях в версии EJB 3.2. В главе 2 вы впервые попробуете EJB на вкус, реализовав свое первое решение с применением технологии EJB.

Вторая часть охватывает особенности работы с компонентами EJB при реализации прикладной логики. Глава 3 описывает тонкости сеансовых компонентов (session beans) и знакомит с наиболее удачными приемами. Глава 4 представляет собой краткое введение в организацию обмена сообщениями, JMS и подробно рассказывает о MDB. Глава 5 охватывает дополнительные темы, такие как контекст EJB, JNDI, ресурсы и внедрение EJB, интерцепторы AOP и контейнер клиентского приложения. Глава 6 обсуждает транзакции и безопасность. Глава 7 знакомит с таймерами и новыми возможностями реализации отложенных вычислений. Глава 8 демонстрирует возможность экспорта прикладной логики EJB в виде веб-служб SOAP и RESTful.

В третьей части подробно рассказывается о взаимоотношениях EJB 3 с JPA и CDI. В главе 9 дается введение в предметное моделирование и особенности отображения сущностей JPA в предметную область. Глава 10 охватывает управление сущностями JPA через CRUD-операции. Глава 11 рассказывает о языке JPQL и более подробно рассказывает о приемах извлечения данных. Глава 12 служит введением в технологию CDI и рассказывает, как она дополняет EJB.

В четвертой части содержатся рекомендации по внедрению EJB 3 в работу на предприятии. Глава 13 обсуждает подготовку компонентов EJB к развертыванию на сервере. Глава 14 знакомит с веб-сокетами, их взаимосвязях с компонентами EJB, а также с возможностью асинхронного выполнения прикладной логики с применением утилит EJB. Глава 15 охватывает вопросы модульного и интеграционного тестирования без необходимости развертывания приложений на действующем сервере.

Книга имеет три приложения. Приложение A – это справочник по дескриптору развертывания *ejb-jar.xml*. Приложение B содержит пошаговые инструкции по загрузке и установке пакета Java EE 7 SDK, включающего Java SE 7, GlassFish 4 и NetBeans. Приложение C содержит информацию о процессе сертификации по технологии EJB в Oracle и сертификационных экзаменах по EJB.

Загружаемый исходный код

Приложение B содержит пошаговые инструкции по загрузке и установке пакета Java EE 7 SDK. Исходный код примеров для этой книги доступен для загрузки по адресу: http://code.google.com/p/action-bazaar/. Здесь вы сможете либо получить копию репозитория Git со всеми примерами, либо загрузить подготовленный ZIP-файл со всеми примерами внутри. Программный код разрабатывался в основном в среде NetBeans, но все примеры собирались с помощью Maven, поэтому они должны сохранить работоспособность и в других средах разработки.

ZIP-файл с исходным кодом также доступен для загрузке на сайте издательства, по адресу: www.manning.com/EJB3inActionSecondEdition.

Соглашения по оформлению исходного кода

Так как обсуждение разных тем в этой книге в значительной степени опирается на примеры, исходному коду было уделено самое пристальное внимание. Большие фрагменты кода представлены в виде отдельных листингов. Весь программный код набран `моноширинным шрифтом`, как этот фрагмент, для большей наглядности. Все, что находится внутри фрагментов программного кода, например: элементы XML; имена методов, типов данных, пакетов, переменных и прочее, набраны шрифтом Courier. Некоторые элементы программного кода набраны **`жирным моноширинным шрифтом`** – это сделано, чтобы выделить наиболее важные фрагменты. Также в коде могут содержаться комментарии, описывающие некоторые важные понятия. В некоторых случаях мы сократили код примеров, чтобы сделать его более простым и компактным. В любом случае, полный код примеров можно найти в загружаемых zip-файлах. Дополнительно мы рекомендуем настраивать среду разработки перед прочтением каждой главы.

Автор в сети

Приобретая книгу «EJB 3 в действии, второе издание», вы получаете бесплатный доступ на частный веб-форум издательства Manning Publications, где вы сможете оставлять свои отзывы о книге, задавать вопросы технического характера и получать помощь от авторов и других пользователей. Чтобы получить доступ к форуму и зарегистрироваться на нем, откройте в браузере страницу www.manning.com/EJB3inActionSecondEdition. На этой странице «Author Online» (Автор в сети) описывается, как попасть на форум после регистрации, какие виды помощи доступны и правила поведения на форуме.

Издательство Manning обязуется предоставить своим читателям место встречи, где может состояться содержательный диалог между отдельными читателями и между читателями и автором. Но со стороны автора отсутствуют какие-либо обязательства уделять форуму какое-то определенное внимание – его присутствие на форуме остается добровольным (и неоплачиваемым). Мы предлагаем задавать автору стимулирующие вопросы, чтобы его интерес не угасал!

Форум и архивы предыдущих дискуссий будут оставаться доступными, пока книга продолжает издаваться.

О названии

Сочетая в себе введение, краткий обзор и примеры использования, книги из серии «в действии» предназначены для изучения и запоминания. Согласно исследованиям в когнитивистике, люди лучше запоминают вещи, которые они узнают в процессе самостоятельного изучения.

В издательстве Manning нет ученых-когнитивистов, тем не менее, мы уверены, что для надежного усваивания необходимо пройти через стадии исследования, игры и, что интересно, пересказа всего, что было изучено. Люди усваивают новые знания до уровня овладения ими, только после их активного исследования. Люди учатся в действии. Особенность учебников из серии «в действии» в том, что они основываются на примерах. Это побуждает читателя пробовать, играть с новым кодом, и исследовать новые идеи.

Есть и другая, более прозаическая причина выбора такого названия книги: наши читатели – занятые люди. Они используют книги для выполнения работы или для решения проблем. Им нужны книги, позволяющие легко перепрыгивать с места на место и изучать только то, что они хотят, и только когда они этого хотят. Им нужны книги, помогающие в действии. Книги этой серии создаются для таких читателей.

Об авторах

Дебу Панда (Debu Panda) – опытный руководитель, инженер и лидер сообщества. Он написал более 50 статей о технологиях Enterprise Java, Cloud и SOA, и две книги о промежуточном ПО уровня предприятия (Enterprise middleware). Следуйте за Дебу в Твиттере *@debupanda*.

Реза Рахман (Reza Rahman) – прежде долгое время работал независимым консультантом, а ныне является официальным пропагандистом Java EE/GlassFish в Oracle. Реза часто выступает в группах пользователей Java и на конференциях по всему миру. Он часто пишет статьи для индустриальных журналов, таких как «JavaLobby/DZone» и «TheServerSide». Реза входил в состав экспертных групп, занимавшихся разработкой спецификаций Java EE, EJB и JMS. Реализовал контейнер EJB для Resin, открытого сервера приложений Java EE.

Райан Купрак (Ryan Cuprak) – аналитик в компании Dassault Systèmes (DS), автор руководства «NetBeans Certification Guide», выпущенного издательством McGraw-Hill, и президент группы пользователей «Connecticut Java Users Group» с 2003 года. Удостоен награды JavaOne 2011 Rockstar Presenter. В DS занимается

разработкой средств интеграции данных, обеспечивающих преобразование клиентских данных, а также созданием пользовательских интерфейсов. До поступления на работу в компанию DS он работал в начинающей компании, специализирующейся на распределенных вычислениях, в TurboWorx и в группе Eastman Kodak Molecular Imaging Systems, ныне являющейся частью компании Carestream Health. В TurboWorx служил инженером службы сбыта и занимался предпродажной поддержкой и разработкой программ на языке Java. Купрак получил степень бакалавра информатики и биологии в университете Лойола в Чикаго. Получил в компании Sun сертификат специалиста по NetBeans IDE.

Майкл Ремижан (Michael Remijan) – директор-распорядитель и технический руководитель в больнице BJC Hospital. Начал работать с Java EE в конце 1990-х. Занимался разработкой систем B2C и B2B для коммерции, производства, астрономии, сельского хозяйства, телекоммуникации, национальной обороны и здравоохранения. Получил степень бакалавра информатики и математики в университете штата Иллинойс в Урбана-Шампейн и степень магистра управления научно-техническим развитием в университете города Феникс. Обладает множеством сертификатов компании Sun Microsystems и является автором статей в журналах «Java Developer Journal» и «Javalobby/DZone». Его блог можно найти по адресу: mjremijan.blogspot.com.

Об илллюстрации на обложке

На обложке книги «EJB 3 в действии» изображена «Русская девушка с мехом». Этот рисунок взят из Французского туристического путеводителя «Encyclopedie des Voyages» (автор J. G. St. Saveur), выпущенного в 1796 году. Путешествия ради удовольствия были сравнительно новым явлением в то время и туристические справочники, такие как этот, были популярны, они позволяли знакомиться с жителями других регионов Франции и других стран, не вставая с кресла.

Многообразие рисунков в путеводителе «Encyclopedie des Voyages» отчетливо демонстрирует уникальные и индивидуальные особенности городов и районов мира, существовавших 200 лет назад. Это было время, когда по одежде можно было отличить двух людей, проживающих в разных регионах, расположенных на расстоянии всего нескольких десятков километров. Туристический справочник позволяет почувствовать изолированность и отдаленность того периода от любого другого исторического периода отличного от нашей гиперподвижной современности.

С тех пор мода изменилась, а региональные различия, такие существенные в те времена, исчезли. Сейчас зачастую сложно отличить жителей разных континентов. Возможно, пытаясь рассматривать это с оптимистической точки зрения, мы обменяли культурное и визуальное разнообразие на более разнообразную личную жизнь. Или более разнообразную и интересную интеллектуальную жизнь и техническую вооруженность.

Мы в издательстве Manning славим изобретательность, предприимчивость и радость компьютерного бизнеса обложками книг, изображающими богатство региональных различий двухвековой давности.

Часть I

ОБЗОР ЛАНДШАФТА EJB

Эта книга рассказывает о технологии Enterprise Java Beans (EJB) 3 и охватывает версии спецификации вплоть до EJB 3.2. Версия EJB 3.2 была выпущена, как продолжение развития спецификации EJB, целью которой является полнота удовлетворения всех потребностей корпоративного сектора и улучшение архитектуры EJB путем уменьшения ее сложности для разработчика.

Первая часть книги представляет EJB 3 как мощную и удобную платформу, по достоинству занимающую свое место стандарта разработки ответственных решений в корпоративном секторе. В этой части мы познакомимся с Java Persistence API (JPA 2.1) – технологией Java EE, целью которой является стандартизация Java ORM и тесная интеграция с EJB 3. В этой части мы также коротко познакомимся с технологией управления контекстами и внедрением зависимостей (Contexts and Dependency Injection) для Java (CDI 1.1), универсальной технологией следующего поколения внедрения зависимостей для Java EE.

В главе 1 мы представим вам элементы, составляющие EJB 3, коснемся уникальных особенностей EJB, как платформы разработки, и расскажем о новых возможностях, увеличивающих продуктивность. Мы даже продемонстрируем пример приложения «Hello World».

В главе 2 будут даны более практичные примеры кода и представлено приложение ActionBazaar, воображаемая корпоративная система, которая будет разрабатываться на протяжении всей книги. Мы постараемся дать вам возможность максимально быстро составить свое представление о EJB 3. Будьте готовы, вас ждет масса кода!

ГЛАВА 1.
Что такое EJB 3

Эта глава охватывает следующие темы:

- контейнер EJB и его роль в корпоративных приложениях;
- разные типы компонентов Enterprise Java Beans (EJB);
- тесно связанные технологии, такие как Java Persistence API (JPA);
- разные среды выполнения EJB;
- инновации, появившиеся в EJB 3;
- новые изменения в EJB 3.2.

Однажды, когда Бог осматривал свои создания, он заметил мальчика по имени Садху (Sadhu), юмор и острота ума которого понравились ему. В этот день Бог чувствовал себя особенно щедрым и предложил мальчику загадать три желания. Садху попросил три воплощения: одно в качестве божьей коровки, одно в качестве слона и одно – в качестве коровы. Удивленный этими желаниями, Бог попросил Садху объяснить их. Мальчик ответил: «Я хочу быть божьей коровкой, потому что все в мире восхищаются ее красотой и прощают ей ее беззаботность. Я хочу быть слоном, потому что он может съесть огромное количество пищи, и никто не будет высмеивать его. Я хочу быть коровой, потому что она любима всеми и полезна всем.». Бог был очарован этим ответом и дал мальчику возможность прожить все три воплощения. Затем он превратил Садху в утреннюю звезду за его служение человечеству в роли коровы.

Спецификация EJB тоже пережила три воплощения. Когда появилась первая версия, индустрия была ослеплена ее инновациями. Но, как и божья коровка, EJB 1 имела ограниченные функциональные возможности. Во втором воплощении EJB была также тяжеловесна и прожорлива, как всеми любимый толстокожий. Храбрецы, кому нужна была сила слона, должны были приручить жуткую сложность EJB 2. И, наконец, в третьем воплощении EJB стала более полезной для широкого круга разработчиков, так же как нежно любимая и почитаемая индусами корова, чье молоко кормит нас.

Благодаря упорному труду множества людей, версия EJB 3 стала более простой и легковесной, не потеряв так ценимую в ней мощь. Компоненты EJB теперь могут быть простыми объектами Java (Plain Old Java Objects, POJO) и близко похожими на код в примере программы «Hello World». В следующих главах мы опишем самую яркую звезду среди фреймворков индустриального уровня.

Мы стремились сохранить эту книгу максимально практичной, не в ущерб содержимому. Книга создавалась так, чтобы помочь вам быстро изучить EJB 3, не пренебрегая основами. Мы вместе с вами будем участвовать в погружении и рассказывать вам о наиболее восхитительных особенностях, а также предостерегать о скрывающихся опасностях.

В мире Java технология EJB расценивается как важнейшая и влиятельнейшая, особенно после появления радикально измененной версии 3. Мы не будем тратить слишком много времени на описание EJB 2. Возможно, вы уже знакомы с более ранними версиями EJB или вообще не имеете о них представления. Уделение слишком пристального внимания предыдущим версиям – пустая трата времени. EJB 3 и EJB 2 имеют слишком мало общего. Более того, в нынешней спецификации EJB 3.2 поддержка EJB 2 вообще объявлена необязательной. Но, если вы испытываете интерес к EJB 2, мы рекомендуем вам приобрести одну из множества замечательных книг, посвященных предыдущим версиям EJB.

В этой главе мы расскажем, что из себя представляет EJB 3, объясним, почему следует всерьез подумать о ее использовании, и обозначим наиболее важные новшества в самой последней версии, такие как аннотации, приоритет соглашений перед настройками, и внедрение зависимостей. Мы используем эту главу как трамплин для прыжка к главе 2. Итак. Давайте начнем с общего обзора EJB.

1.1. Обзор EJB

Первое, что следует выяснить при оценке новой технологии, – что она в действительности дает. Что же такого особенного в EJB? Помимо технологий уровня представления, таких как JavaServer Pages (JSP), JavaServer Faces (JSF) или Struts, разве нельзя создать веб-приложение на языке Java с использованием некоторого API, такого как Java Database Connectivity (JDBC) для доступа к базам данных? Конечно можно, если вы не ограничены жесткими временными рамками и ресурсами. До появления EJB именно так и работали. По опыту прошлых лет мы знаем, что в этом случае придется потратить массу времени на решение типовых задач системного уровня, вместо того, чтобы сосредоточиться на решении прикладных задач. Этот же опыт показывает, что типовые задачи имеют типовые решения. Это именно то, что с готовностью выкладывает на стол EJB. EJB – это коллекция «фиксированных» решений типовых проблем, возникающих при разработке серверных приложений, а также проверенная временем схема реализации серверных компонентов. Эти фиксированные решения, или службы, предоставляются контейнером EJB. Для доступа к этим службам необходимо создать специализированные компоненты, используя декларативные и программные EJB API, и развернуть их в контейнере.

1.1.1. EJB как модель компонентов

В этой книге под EJB подразумеваются серверные компоненты, которые можно использовать для построения прикладных компонентов. У некоторых разработчиков возникают ассоциации с разработкой сложного и тяжеловесного кода, использующего технологию CORBA или Microsoft COM+, когда они слышат термин *компонент* (component). В дивном мире EJB 3 компонент – это ничто иное, как POJO с некоторыми специальными возможностями. Что еще более важно, эти возможности остаются невидимыми, пока они не станут необходимы и не отвлекают от главной цели компонента. На протяжении всей книги вы будете находить подтверждение этому, начиная с главы 2.

Чтобы воспользоваться службами EJB, ваш компонент должен быть объявлен с типом, распознаваемым фреймворком EJB. EJB распознает два конкретных типа компонентов: сеансовые компоненты (session beans) и компоненты, управляемые сообщениями (message-driven beans). Сеансовые компоненты в свою очередь подразделяются на сеансовые компоненты без сохранения состояния (stateless session beans), сеансовые компоненты с сохранением состояния (stateful session beans) и компоненты-одиночки (синглтоны – singletons). Компоненты каждого типа имеют определенное предназначение, область видимости, состояние, жизненный цикл и особенности использования на уровне прикладной логики. Мы будем обсуждать эти типы компонентов на протяжении всей книги и особенно пристальное внимание уделим им во второй части. При обсуждении CRUD-операций (create, read, update, delete – создать, прочитать, изменить, удалить) на уровне хранения данных в третьей части мы будем говорить о сущностях JPA и их взаимоотношениях с компонентами EJB. Начиная с версии EJB 3.1, все компоненты EJB являются управляемыми. Управляемые компоненты по сути являются обычными Java-объектами в окружении Java EE. Механизм управления контекстами и внедрения зависимостей (Contexts and Dependency Injection, CDI) позволяет осуществлять внедрение зависимостей в любые управляемые компоненты, включая компоненты EJB. Подробнее о механизме CDI и управляемых компонентах будет рассказываться в третьей части 3.

1.1.2. Службы компонентов EJB

Как уже упоминалось выше, фиксированные службы являются наиболее ценной частью EJB. Некоторые службы автоматически подключаются зарегистрированным компонентам, так как специально предназначены для использования компонентами на уровне прикладной логики. В число этих служб входят: внедрение зависимостей, транзакции, поддержка многопоточной модели выполнения и организация пулов. В большинстве случаев для подключения служб разработчик явно должен объявить о своих намерениях с использованием аннотаций/XML или обращением к EJB API во время выполнения. К числу таких служб относятся: поддержка безопасности, планирование, асинхронная обработка, удаленные взаимодействия и веб-службы. Значительная часть этой книги посвящена описанию приемов эксплуатации служб EJB. Мы не можем позволить себе роскошь деталь-

но описать в этой главе каждую службу, но перечислим наиболее важные из них в табл. 1.1 и поясним их назначение.

Таблица 1.1. Службы EJB

Служба	Назначение
Регистрация, поиск и внедрение зависимостей	Помогает находить и связывать компоненты. Идеально подходит для реализации простых настроек. Позволяет изменять связи между компонентами для нужд тестирования.
Управление жизненным циклом	Дает возможность выполнять определенные действия в момент перехода компонента от одного этапа жизненного цикла к другому, например после создания и перед уничтожением.
Поддержка многопоточной модели выполнения	EJB наделяет все компоненты поддержкой многопоточной модели выполнения и обеспечивает высокую производительность, скрывая детали этой поддержки от вас. Это означает, что многопоточные серверные компоненты можно писать, как если бы они были однопоточными настольными приложениями. Сложность компонента не играет никакой роли: EJB в любом случае обеспечит поддержку многопоточной модели.
Транзакции	EJB автоматически наделяет все ваши компоненты поддержкой механизма транзакций. Это означает, что вам не придется писать какой-либо код управления транзакциями при работе с базами данных или серверами сообщений через JDBC, JPA или Java Message Service (JMS).
Организация пулов	EJB создает пул экземпляров компонента, совместно используемых клиентами. В любой момент времени каждый экземпляр может использоваться только одним клиентом. По завершении обслуживания клиента, экземпляр немедленно возвращается в пул для повторного использования, вместо передачи сборщику мусора для утилизации. Поддерживается возможность ограничивать размеры пулов, чтобы в случае их переполнения все последующие запросы автоматически ставились в очередь ожидания обработки. Благодаря этому ваша система останется отзывчивой, даже при взрывном росте числа запросов. Аналогично организации пулов экземпляров компонентов, EJB автоматически организует пулы потоков выполнения в рамках контейнера для снижения накладных расходов.
Управление состоянием	Контейнер обеспечивает прозрачное управление состоянием компонентов с поддержкой состояния, исключая необходимость писать подробный и чреватый ошибками код управления состоянием. Благодаря этому вы можете организовать управление состоянием в переменных экземпляра, как если бы писали обычное настольное приложение. Все хлопоты по обслуживанию сеансов/состояний возьмет на себя EJB.
Управление памятью	EJB предпринимает определенные действия по оптимизации использования памяти, сохраняя редко используемые компоненты с поддержкой состояния на диске для освобождения памяти. Этот прием называется пассивацией (passivation). Когда память вновь становится доступной и возникает потребность в пассивированных компонентах, EJB загружает их обратно в память. Это процесс называется активацией (activation).

Таблица 1.1. (окончание)

Служба	Назначение
Обмен сообщениями	EJB 3 позволяет создавать компоненты для обработки сообщений без необходимости задумываться о технических тонкостях JMS API.
Безопасность	EJB дает простую возможность обезопасить компоненты посредством обычной конфигурации.
Планирование	В EJB имеется возможность планировать автоматический вызов любых методов EJB с помощью простых циклических таймеров или cron-подобных выражений.
Асинхронная обработка	Вы можете настроить любой метод EJB для вызова в асинхронном режиме.
Интерцепторы	В EJB 3 добавлена поддержка интерцепторов (interceptors), упрощающих AOP (Aspect-Oriented Programming – аспектно-ориентированное программирование). Благодаря этому легко можно отделять сквозные задачи, такие как журналирование и аудит, на уровне конфигурации.
Веб-службы	EJB 3 поддерживает прозрачную возможность преобразования компонентов в веб-службы типа Simple Object Access Protocol (SOAP) или Representational State Transfer (REST) с минимальными изменениями в коде или вообще без таковых.
Удаленные взаимодействия	С помощью EJB 3 можно создавать компоненты с поддержкой удаленного доступа без необходимости писать хоть какой-то код. Кроме того, EJB 3 позволяет клиентскому коду обращаться к удаленным компонентам, как если бы они были локальными, с помощью механизма внедрения зависимостей (Dependency Injection, DI).
Тестирование	Любые компоненты EJB легко поддаются модульному и интеграционному тестированию с применением встроенных контейнеров в таких фреймворках, как JUnit.

1.1.3. Многоуровневые архитектуры и EJB

Корпоративные приложения проектируются для решения уникальных задач, и потому к ним предъявляется множество похожих требований. Большинство корпоративных приложений имеют некоторый пользовательский интерфейс, реализующий прикладные процессы, модель предметной области и хранение информации в базе данных. Так как эти требования являются общими, появляется возможность следовать обобщенной архитектуре, или принципам проектирования, известным как *шаблоны проектирования*.

При разработке серверных приложений часто используются *многоуровневые архитектуры*. В таких архитектурах компоненты группируются в уровни (или слои). Каждый уровень приложения служит четко определенным целям, по аналогии с участками сборочного конвейера. На каждом участке конвейера выполняются определенные операции и их результат передается дальше по конвейеру. В многоуровневых архитектурах каждый уровень делегирует выполнение операций следующему уровню, расположенному под ним.

EJB учитывает это обстоятельство и не способствует созданию универсальных компонентов на все случаи жизни. Вместо этого EJB следует модели специализированных компонентов, призванных решать задачи, свойственные определенному уровню в многоуровневой архитектуре. Существует две основные разновидности многоуровневых архитектур: традиционная четырехуровневая архитектура и проблемно-ориентированная архитектура (Domain-Driven Design, DDD). Давайте рассмотрим эти архитектуры в отдельности и выясним, как EJB соответствует им.

Традиционная четырехуровневая архитектура

На рис. 1.1 представлена традиционная четырехуровневая архитектура. Она достаточно известна и пользуется большой популярностью. В этой архитектуре *уровень представления* отвечает за отображение графического интерфейса пользователя (Graphical User Interface, GUI) и обработку ввода пользователя. Уровень представления передает все запросы *уровню прикладной логики*. Этот уровень является сердцем приложения и содержит основную логику обработки. Иными словами, прикладная логика – это уровень, где находятся компоненты, выполняющие различные прикладные операции, такие как учет, поиск и упорядочение данных, а также обслуживание учетных записей пользователей. Уровень прикладной логики извлекает данные и сохраняет их в базе данных, используя для этого уровень хранения. *Уровень хранения* обеспечивает высокоуровневые, объектно-ориентированные абстракции поверх уровня базы данных. *Уровень базы данных* обычно включает систему управления реляционными базами данных (Relational Database Management System, RDBMS), такую как Oracle, DB2 или SQL Server.

Рис. 1.1. Большинство корпоративных приложений состоят как минимум из четырех уровней: уровня представления – фактического пользовательского интерфейса, который может быть реализован с применением браузера или отдельного настольного приложения; прикладного уровня, определяющего бизнес-правила; уровня хранения, обеспечивающего взаимодействия с базой данных; и уровня базы данных, включающего реляционную базу данных, такую как Oracle, где находятся хранимые объекты

EJB не является технологией реализации уровня представления или хранения. Ее цель – обеспечить надежную поддержку компонентов прикладного уровня. На рис. 1.2 показано, как EJB поддерживает эти уровни посредством своих служб.

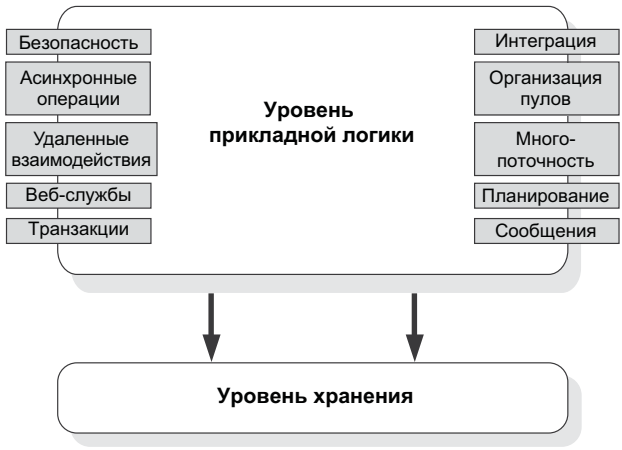

Рис. 1.2. Службы компонентов, предлагаемые EJB 3 на прикладном уровне. Обратите внимание, что все службы являются независимыми друг от друга, поэтому вы вольны (в большинстве случаев) выбирать службы, необходимые вашему приложению

В типичной системе на основе Java EE, на уровне представления используются JSF и CDI, EJB используется на прикладном уровне, а на уровне хранения данных используются JPA и CDI.

Традиционная четырехуровневая архитектура не лишена недостатков. Ее часто критикуют за то, что она подрывает объектно-ориентированные идеалы ОО моделирования предметной области, когда объекты инкапсулируют одновременно и данные и поведение. Из-за того, что традиционная архитектура концентрируется на моделировании прикладной обработки, а не предметной области, уровень прикладной логики часто больше напоминает приложение баз данных, написанное в процедурном стиле, нежели объектно-ориентированное приложение. Так как компоненты уровня хранения являются простыми хранилищами данных, они больше напоминают записи в базе данных, чем сущности объектно-ориентированного мира. Как вы увидите в следующем разделе, проблемно-ориентированная архитектура (DDD) предлагает альтернативное решение, в попытке преодолеть обозначенные недостатки.

Проблемно-ориентированная архитектура

На рис. 1.3 показано, как выглядит проблемно-ориентированная архитектура. Термин *проблемно-ориентированная архитектура* (domain-driven design) может быть и является относительно новым, но само понятие существует достаточно давно (см. книгу «Domain-Driven Design: Tackling Complexity in the Heart of Software», Эрика Эванса (Eric Evans) [Addison-Wesley Professional, 2003][1]). Архитектура DDD подчеркивает, что объекты предметной области не должны служить простой заменой записей базы данных, и должны содержать прикладную логику. Эти объекты могут быть реализованы как сущности в JPA. В архитектуре DDD, объект `Catalog` в приложении электронной торговли, помимо хранения всех данных

[1] Эрик Эванс, «Предметно-ориентированное проектирование (DDD): Структуризация сложных программных систем», Вильямс, 2010, ISBN: 978-5-8459-1597-9. – *Прим. перев.*

из таблицы каталога в базе данных, мог бы уметь анализировать и не возвращать записи из каталога, соответствующие отсутствующим на складе товарам. Будучи простыми объектами Java (POJO), сущности JPA поддерживают объектно-ориентированные возможности, такие как наследование и полиморфизм. Не составляет большого труда реализовать объектную модель хранилища на основе JPA и добавить в ее сущности прикладную логику. В настоящее время проблемно-ориентированная архитектура все еще использует *уровень служб*, или *уровень приложения* (см. книгу «Patterns of Enterprise Application Architecture» Мартина Фаулера (Martin Fowler) [Addison-Wesley Professional, 2002][2]). Уровень приложения подобен прикладному уровню в традиционной четырехуровневой архитектуре, но значительно тоньше. EJB прекрасно справляется с поддержкой компонентной модели с уровнем служб. Неважно, какую архитектуру вы выберете, традиционную четырехуровневую или проблемно-ориентированную, вы с успехом можете использовать сущности JPA для моделирования прикладных объектов, включая моделирование состояния и поведения. Подробнее о моделировании прикладных объектов с применением сущностей JPA мы поговорим в главе 7.

Несмотря на богатство предоставляемых служб, EJB 3 не является единственно доступным средством. Существуют и другие технологии, которые можно комбинировать и получить в результате более или менее похожий на EJB комплекс служб и инфраструктуры. Например, для создания приложений можно использовать Spring с другими открытыми технологиями, такими как Hibernate и AspectJ. Но, если дело обстоит так, есть ли смысл выбирать EJB 3? Отличный вопрос!

Рис. 1.3. Проблемно-ориентированная архитектура обычно состоит из четырех или более уровней. Уровень представления отвечает за пользовательский интерфейс и взаимодействия с уровнем служб/приложения. Уровень служб/приложения обычно очень тонкий и легковесный, и всего лишь обеспечивает взаимодействия между уровнем представления и предметным уровнем. Предметный уровень – сложный комплекс компонентов, представляющих модель прикладных данных, состоящих из сущностей, объектов значений, агрегатов, фабрик и репозиториев. Уровень инфраструктуры соответствует базе данных или иной технологии хранения данных

[2] Мартин Фаулер, «Шаблоны корпоративных приложений», Вильямс, 2009, ISBN: 978-5-8459-1611-2. – *Прим. перев.*

1.1.4. Почему стоит выбрать EJB 3?

В начале этой главы мы отмечали, что EJB является инновационной технологией, которая высоко подняла планку стандартов разработки серверных компонентов. Так же как и сам язык Java, EJB изменила окружающий мир, чтобы остаться в нем и послужить основой для множества других инноваций. Еще несколько лет назад серьезную конкуренцию EJB могла составить только платформа Microsoft .NET Framework. В этом разделе мы укажем на некоторые особенности EJB 3, которые по нашему мнению выводят эту последнюю версию в начало не такого уж и длинного списка.

Простота в использовании

Благодаря нацеленности на простоту использования, EJB 3 является, пожалуй, самой простой платформой для разработки серверных компонентов. Наиболее яркими аспектами с этой точки зрения являются: программирование POJO, преобладание аннотаций перед конфигурацией в формате XML, преимущественное использование значений по умолчанию и отказ от сложных парадигм. Несмотря на большое число служб в EJB, все они просты и понятны. По большей части технология EJB 3 выглядит весьма практично и не требует глубоко понимания всех ее теоретических хитросплетений. Фактически большинство служб EJB спроектировано так, чтобы вы могли не задумываться о них, сосредоточиться на решении прикладных задач и уходить домой в конце дня с осознанием, что достигнуто все желаемое.

Полный интегрированный стек решений

EJB 3 предлагает полный стек серверных решений, включая транзакции, безопасность, обмен сообщениями, планирование, удаленные взаимодействия, веб-службы, асинхронная обработка, тестирование, внедрение зависимостей и интерцепторы. Это означает, что вам не придется тратить время на поиск сторонних инструментов для интеграции в свое приложение. Все необходимые службы уже имеются в наличии, и вам не придется делать что-то особенное, чтобы активировать их. Что ведет почти к полному отказу от систем конфигурации.

Кроме того, EJB 3 обеспечивает бесшовную интеграцию с другими технологиями Java EE, такими как CDI, JPA, JDBC, JavaMail, Java Transaction API (JTA), JMS, Java Authentication and Authorization Service (JAAS), Java Naming and Directory Interface (JNDI), Remote Method Invocation (RMI) и другими. EJB также гарантирует простоту интеграции с технологиями уровня представления, такими как JSP, Servlets и JSF. При необходимости с помощью CDI можно интегрировать с EJB и сторонние инструменты.

Открытый стандарт Java EE

Спецификация EJB является важной частью стандарта Java EE. Это важно понимать, если вы собираетесь использовать EJB. EJB 3 имеет открытый прикладной интерфейс с общедоступным описанием и комплект тестов на совместимость, которые могут использоваться любыми организациями для создания своих реа-

лизаций контейнеров. Стандарт EJB 3 разрабатывался в рамках процесса Java Community Process (JCP), в который были вовлечены группы лиц, участвующих также в развитии стандарта Java. Открытость стандарта ведет к расширению его поддержки среди производителей, а это означает, что вы не будете зависеть от проприетарных решений.

Широкая поддержка производителей

EJB поддерживается широким кругом независимых организаций. В их числе наиболее технологически развитые, наиболее уважаемые и финансово сильные компании, такие как Oracle и IBM, а также активные и энергичные открытые проекты, такие как JBoss и Apache. Широкая поддержка производителей дает три важных преимущества. Во-первых, вы не зависите от взлетов и падений каких-то компаний или групп людей. Во-вторых, у многих людей есть конкретный долгосрочный интерес в максимальном сохранении конкурентоспособности технологии. Вы смело можете рассчитывать на возможность использования лучших технологий как внутри мира Java, так и за его пределами, в обозримом будущем. В-третьих, производители исторически конкурировали друг с другом, реализуя дополнительные нестандартные особенности. Все эти факторы помогают продвижению EJB по пути эволюции.

Кластеризация, балансировка нагрузки и отказоустойчивость

В число особенностей, добавленных многими производителями серверов приложений, входят поддержка кластеризации, балансировка нагрузки и отказоустойчивость. Серверы приложений с поддержкой EJB обладают доказанной поддержкой аспектов высокопроизводительных вычислений, таких как фермы серверов. Еще более важно, что вы можете использовать эту поддержку, не изменяя свой код, не привлекая сторонние инструменты интеграции и с относительно простыми конфигурациями (если не считать работы по настройке аппаратных кластеров). Это означает, что вы можете положиться на аппаратные кластеры для масштабирования приложений, основанных на EJB 3.

Производительность и масштабируемость

Корпоративные приложения имеют много общего с домами. И те, и другие строятся на достаточно долгие периоды времени, порой даже намного более долгие, чем можно было бы ожидать. Поддержка высокопроизводительных вычислений, отказоустойчивость, масштабируемость – важнейшие особенности, заложенные в платформу EJB. Вы не просто будете писать серверные приложения быстрее, вы также можете рассчитывать на дальнейший рост своей системы. Вы сможете увеличить число обслуживаемых пользователей без изменения кода; все хлопоты возьмет на себя контейнер EJB через такие особенности, как поддержка многопоточной модели выполнения, распределенные транзакции, организация пулов ресурсов, пассивация, асинхронная обработка, обмен сообщениями и удаленные

взаимодействия. От вас может потребоваться выполнить лишь минимальную оптимизацию, но чаще достаточно будет просто перенести приложение на ферму серверов и лишь немного изменить настройки.

Мы полагаем, что к настоящему времени вы достаточно воодушевились открывающимися перспективами EJB и вам не терпится узнать больше. Поэтому давайте двинемся дальше и посмотрим, как можно использовать EJB для создания уровня прикладной логики приложения, начав со знакомства с особенностями создания компонентов.

1.2. Основы типов EJB

На языке EJB компонент называется *bean* (зерно). Если ваш менеджер не нашел забавной игру слов «coffee bean» (кофейное зерно) в отношении Java, остается только посетовать в адрес маркетологов из Sun. Но, как бы то ни было, вам наверняка часто приходилось слышать от людей в строгих костюмах слова «Enterprise» (корпоративный) и «bean» (зерно) вместе, как нечто вполне нормальное.

Как уже отмечалось, EJB делит зерна (то бишь, компоненты) на два типа, исходя из их предназначения:

- сеансовые компоненты;
- компоненты, управляемые сообщениями.

Каждый тип компонентов служит определенной цели и может использовать определенное подмножество служб EJB. Истинная цель деления компонентов на типы состоит в том, чтобы избежать их перегрузки слишком большим количеством служб. Это как исключить неприятные последствия от действий чересчур любопытного бухгалтера в роговых очках, которому интересно, что произойдет, если накоротко замкнуть клеммы аккумулятора в его автомобиле. Классификация компонентов также помогает организовать приложение; например, деление компонентов на типы помогает создавать приложения с многоуровневой архитектурой. Давайте познакомимся с этими типами немного поближе, начав с сеансовых компонентов.

1.2.1. Сеансовые компоненты

Сеансовый компонент вызывается клиентом для выполнения вполне определенных операций, таких как проверка кредитной истории клиента. Слово «сеансовый» в названии предполагает, что экземпляр компонента доступен только пока выполняется некоторая единица работы и безвозвратно уничтожается в случае аварии или остановки сервера. Сеансовый компонент может реализовать любую прикладную логику. Всего существует три разновидности сеансовых компонентов: *с сохранением состояния*, *без сохранения состояния* и *«одиночки»* (singleton).

Сеансовый компонент с сохранением состояния автоматически сохраняет свое состояние между обращениями к нему от одного и того же клиента, при этом от вас не требуется писать какой-либо дополнительный код. Типичным примером компонента с сохранением состояния может служить корзина с покупками в ин-

тернет-магазине, таком как Amazon. Сеансовые компоненты с сохранением состояния завершают свое существование либо по таймауту, либо по явному запросу клиента. Сеансовые компоненты без сохранения состояния, напротив, не хранят никакой информации о своем состоянии и представляют прикладные службы, которые выполняют все необходимые действия в рамках единственного запроса. Сеансовые компоненты без сохранения состояния можно использовать для реализации таких операций, как перевод средств на кредитную карту или проверка кредитной истории клиента. Сеансовые компоненты-одиночки хранят информацию о своем состоянии, они совместно используются всеми клиентами и продолжают свое существование на протяжении всего времени работы приложения. Компонент-одиночку можно использовать, например, для реализации скидки, так как правила предоставления скидки обычно фиксированы и распространяются на всех клиентов. Обратите внимание, что поддержка компонентов-одиночек впервые была добавлена в EJB 3.1.

Сеансовые компоненты могут вызываться локально или удаленно, посредством Java RMI. Компоненты-одиночки и компоненты без сохранения состояния могут также экспортироваться в виде веб-служб SOAP и REST.

1.2.2. Компоненты, управляемые сообщениями

Подобно сеансовым компонентам, компоненты, управляемые сообщениями, (Message-Driven Bean, MDB) также реализуют некоторую прикладную логику, но имеют одно важное отличие: клиенты никогда не вызывают методы MDB непосредственно. Вместо этого компоненты MDB вызываются для обработки сообщений, отправленных на сервер сообщений, что открывает возможность организовать асинхронный обмен сообщениями между частями системы. Типичными примерами подобных серверов сообщений могут служить HornetQ, ActiveMQ, IBM WebSphere MQ, SonicMQ, Oracle Advanced Queueing и TIBCO. Компоненты MDB обычно применяются для повышения надежности интеграции систем и асинхронной обработки данных. В качестве примера сообщения можно привести запрос на пополнение товарных запасов от автоматизированной системы розничной торговли к системе управления поставками. Не надо пока пытаться глубоко вникнуть в суть организации обмена сообщениями – мы займемся этим вопросом позднее.

1.3. Связанные спецификации

EJB имеет две тесно связанные спецификации, которые мы покроем в этой книге. Первая из них: JPA, которая является стандартом хранения данных для Java EE и CDI, обеспечивает возможность внедрения зависимостей и предоставляет службы управления контекстом для всех компонентов Java EE, включая EJB.

1.3.1. Сущности и Java Persistence API

В EJB 3.1 был учтен переход JPA 2 от EJB 3 API к отдельной спецификации Java EE. Но JPA имеет множество специализированных точек интеграции с EJB, по-

тому что эти две спецификации связаны теснее некуда. Здесь мы рассмотрим лишь некоторые аспекты JPA, потому что этой технологии посвящена отдельная глава.

Технология хранения данных гарантирует автоматическое сохранение Java-объектов в реляционной базе данных, такой как Oracle, SQL server или DB2. Управление хранимыми объектами осуществляется средствами JPA. Этот механизм автоматически сохраняет Java-объекты с применением объектно-реляционного отображения (Object-Relational Mapping, ORM). ORM – это процесс отображения полей Java-объектов в таблицы базы данных, определяемые с применением конфигурационных файлов или аннотаций. JPA освобождает от необходимости писать сложный, низкоуровневый код, использующий JDBC для сохранения объектов в базе данных.

Фреймворк ORM прозрачно осуществляет сохранение, используя метаданные, определяющие порядок отображения объектов в таблицы. ORM – это далеко не новое понятие и существует уже достаточно давно. Старейшим фреймворком ORM является, пожалуй, Oracle TopLink. Еще одним популярным среди разработчиков является открытый фреймворк JBoss Hibernate. Так как спецификация JPA определяет стандарт фреймворков ORM для платформы Java, вы можете использовать в качестве основы для JPA любые известные реализации ORM, такие как JBoss Hibernate, Oracle TopLink или Apache OpenJPA.

JPA – это решение не только для серверных приложений. Поддержка сохранности данных также часто востребована в настольных приложениях на основе Swing. Это обстоятельство послужило толчком к принятию решения превратить JPA 2 в отдельную библиотеку, которую можно было бы использовать за пределами контейнера EJB 3. Во многом подобно JDBC, одной из основных целей JPA является обеспечить универсальный механизм хранения данных для любых приложений на Java.

Сущности

Сущности (entities) – это Java-объекты, хранимые в базе данных. Если сеансовые компоненты являются «глаголами» системы, то сущности – это «существительные». Типичными примерами могут служить сущности, такие как `Employee` (Служащий), `User` (Пользователь) или `Item` (Элемент). Сущности – это объектно-ориентированные представления прикладных данных, хранимых в базе данных. Сущности сохраняются в случае аварии сервера или его остановки. Метаданные ORM определяют, как объекты отображаются в базу данных. Пример таких метаданных будет представлен в следующей главе. Сущности JPA поддерживают все реляционные и объектно-ориентированные возможности, включая отношения между сущностями, наследование и полиморфизм.

Диспетчер сущностей

Сущности сообщают провайдеру JPA, как они должны отображаться в базу данных, но не занимаются сохранением самих себя. Интерфейс `EntityManager`

предусматривает методы чтения метаданных ORM для сущности и выполнения операций по сохранению. `EntityManager` знает, как добавлять сущности в базу данных, обновлять хранящиеся сущности, а также удалять и извлекать их из базы данных.

Язык запросов Java Persistence Query Language

JPA поддерживает специализированный SQL-подобный язык запросов Java Persistence Query Language (JPQL) для поиска сущностей, хранящихся в базе данных. С надежным и гибким API, таким как JPQL, вы ничего не потеряете, выбрав автоматизированный механизм хранения данных вместо вручную написанного кода, использующего JDBC. Кроме того, JPA поддерживает и язык SQL для тех редких случаев, когда он может пригодиться.

1.3.2. Контексты и внедрение зависимостей для Java EE

В Java EE 5 имелась поддержка простейшей формы внедрения зависимостей, которую можно было бы использовать в EJB. Она называлась *внедрением ресурсов* (*resource injection*) и позволяла внедрять контейнеры ресурсов, такие как источники данных, очереди, ресурсы JPA и контейнеры EJB, с использованием аннотаций `@EJB`, `@Resource` и `@PersistenceContext`. Эти ресурсы могли внедряться в сервлеты, управляемые компоненты JSF (JSF backing beans) и EJB. Проблема состояла в том, что поддержка была весьма ограниченной в своих возможностях. Например, она не позволяла внедрять EJB в Struts или JUnit; объекты доступа к данным (Data Access Objects, DAO), кроме тех, что поддерживаются самой реализацией EJB; или вспомогательные классы в EJB.

Мощным решением данной проблемы является механизм CDI. Он предоставляет EJB (и всем остальным API и компонентам в среде Java EE) лучшие универсальные службы внедрения зависимостей и управления контекстом. В число функций CDI входят: внедрение зависимостей, автоматическое управление контекстом, ограничение области видимости, спецификаторы, именование компонентов, продюсеры, регистрация/поиск, стереотипы, интерцепторы, декораторы и события. В отличие от многих других более старых решений внедрения зависимостей, CDI является компактным и футуристическим механизмом, управляемым с помощью аннотаций и полностью поддерживающим типы компонентов. Подробнее о CDI будет рассказываться в главе 12.

1.4. Реализации EJB

Любому классу, реализованному на языке Java, для работы нужна виртуальная машина (Java Virtual Machine, JVM). Точно так же (как мы узнали в разделе 1.3) для работы сеансовым компонентам и компонентам MDB необходим контейнер EJB. В этом разделе мы познакомим вас в общих чертах с несколькими реализациями контейнеров EJB.

Контейнер можно считать продолжением базовой идеи JVM. Так же как JVM прозрачно управляет памятью, контейнер прозрачно обеспечивает компоненты EJB службами, такими как транзакции, поддержка безопасности, удаленные взаимодействия и веб-службы. С определенными допущениями контейнер можно даже считать этакой виртуальной машиной JVM, накачанной стероидами с целью обеспечить работу EJB. Согласно спецификации EJB 3 контейнер предоставляет службы, применимые только к сеансовым компонентам и MDB. Операция добавления компонента EJB 3 в контейнер называется *развертыванием* (deployment). Как только компонент EJB будет благополучно развернут в контейнере, он готов к использованию вашими приложениями.

В мире Java контейнеры не ограничены царством EJB 3. Возможно вы уже знакомы с веб-контейнером, позволяющим создавать веб-приложения с использованием технологий Java, таких как сервлеты, JSP и JSF. Контейнер Java EE – это сервер приложений с поддержкой EJB 3, веб-контейнеров и других Java EE API и служб. Примерами реализаций контейнеров Java EE могут служить серверы приложений: Oracle WebLogic, GlassFish, IBM WebSphere, JBoss и Caucho Resin.

1.4.1. Серверы приложений

Серверы приложений – традиционная среда для развертывания EJB. Серверы приложений по сути являются контейнерами Java EE, включающими поддержку всех Java EE API, а также средства администрирования, развертывания, мониторинга, масштабирования, балансировки нагрузки, безопасности и так далее. В дополнение к поддержке технологий, связанных с Java EE, некоторые серверы приложений могут также функционировать как промышленные HTTP-серверы. Другие поддерживают модульную архитектуру с применением таких технологий, как OSGi. С выходом спецификации Java EE 6, серверы приложений могут принимать легковесную форму *Web Profile*. Web Profile – это сокращенное подмножество Java EE API, специально предназначенное для веб-приложений. Web Profile API включает JSF 2.2, CDI 1.1, EJB 3.2 Lite (обсуждается в разделе 1.4.2), JPA 2.1, JTA 1.2 и механизм проверки компонентов. К моменту написания этих строк, серверы приложений GlassFish и Resin уже поддерживали Java EE 7 Web Profile. Обратите внимание, что реализации Java EE 7 Web Profile могут свободно добавлять дополнительные API. Например, Resin добавляет JMS, а также значительную часть EJB API, включая механизмы обмена сообщениями, удаленных взаимодействий и планирования (но не имеющие обратной совместимости с EJB 2). На рис. 1.4 приводится сравнение Web Profile с полноценной платформой Java EE.

Профиль Web Profile определяет все необходимое для создания современных веб-приложений. В настоящее время веб-приложения редко пишутся, что называется «с нуля», на основе низкоуровневых сервлетов, а чаще основываются на JSF и используют различные технологии EE.

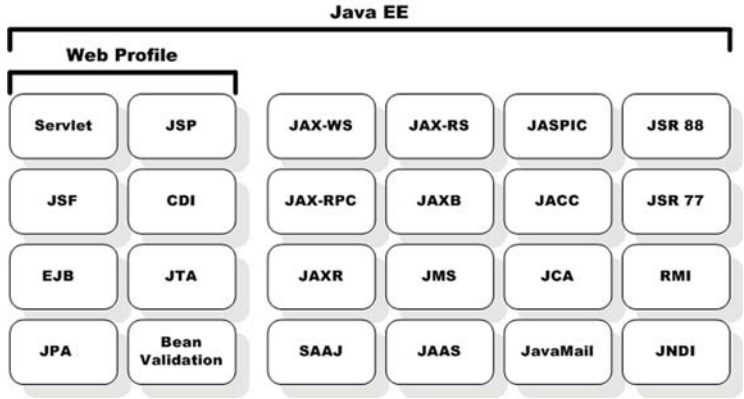

Рис. 1.4. Java EE Web Profile в сравнении с полноценной платформой Java EE

1.4.2. EJB Lite

По аналогии с профилем Java EE 7 Web Profile, существует также облегченная версия EJB 3.2, которая называется EJB 3.2 Lite. Как и Web Profile, версия EJB Lite предназначена для создания веб-приложений и точно так же, как в случае в Web Profile, производители реализаций EJB 3.2 Lite API вольны включать в них любые особенности EJB по своему желанию. С практической точки зрения важно отметить, что из EJB 3.2 Lite полностью исключена поддержка обратной совместимости с EJB 2. Благодаря этому контейнеры EJB получаются более легковесными, так как нет необходимости включать в них устаревшие API, в довесок к легковесной модели EJB. А так как из EJB 3.2 Lite также исключена поддержка компонентов MDB и удаленных взаимодействий, серверы могут быть еще более легковесными, если, конечно же, эти особенности вам не нужны. Для справки в табл. 1.2 проводится сравнение основных возможностей EJB и EJB Lite.

Таблица 1.2. Сравнение возможностей EJB и EJB Lite

Возможность	EJB Lite	EJB
Компоненты без сохранения состояния	✓	✓
Компоненты с сохранением состояния	✓	✓
Компоненты-одиночки	✓	✓
Компоненты, управляемые сообщениями		✓
Компоненты без интерфейсов	✓	✓
Компоненты с локальными интерфейсами	✓	✓
Компоненты с удаленными интерфейсами		✓
Интерфейсы веб-служб		✓
Асинхронный вызов	✓	✓

Таблица 1.2. (окончание)

Возможность	EJB Lite	EJB
Интерцепторы	✓	✓
Декларативная поддержка безопасности	✓	✓
Декларативные транзакции	✓	✓
Программное управление транзакциями	✓	✓
Служба таймеров	✓	✓
Поддержка EJB 2.x		✓
Поддержка CORBA		✓

1.4.3. Встраиваемые контейнеры

Традиционные серверы приложений выполняются в виде отдельных процессов, в которых вы можете развертывать свои приложения. Встраиваемые контейнеры EJB, напротив, можно запускать программно, с использованием Java API, внутри своих приложений. Эта особенность имеет большое значение для модульного тестирования с использованием фреймворка JUnit, а также для использования возможностей EJB 3 из командной строки или в приложениях на основе Swing. В момент запуска встроенный контейнер сканирует пути к библиотекам классов в материнском приложении и автоматически развертывает первую найденную реализацию EJB. На рис. 1.5 представлена архитектура встраиваемого контейнера EJB 3.

Рис. 1.5. Встраиваемые контейнеры EJB 3.1 выполняются непосредственно внутри Java SE и предоставляют все службы EJB, такие как транзакции, безопасность и обмен сообщениями

Встраиваемые контейнеры существуют уже достаточно давно, например: OpenEJB, EasyBeans и Embedded JBoss. Для их работы достаточно наличия облегченной версии EJB Lite, но большинство встраиваемых контейнеров способны поддерживать все возможности. Например, встраиваемые версии GlassFish, JBoss и Resin поддерживают все возможности, доступные в сервере приложений. Подробнее встраиваемые контейнеры будут рассматриваться в главе 15, когда мы будем говорить о тестировании EJB 3.

1.4.4. Использование EJB 3 в Tomcat

Apache Tomcat – популярный и легковесный контейнер сервлетов – не поддерживает EJB 3, потому что в отличие от серверов приложений, контейнеры сервлетов

не требуют этого. Но вы легко сможете использовать EJB 3 в Tomcat в виде встраиваемых контейнеров. Так, например, проект Apache OpenEJB разрабатывался специально для поддержки EJB 3 в Tomcat. Как показано на рис. 1.6, в Tomcat можно также использовать механизм CDI, посредством Apache OpenWebBeans. OpenWebBeans и OpenEJB являются родственными проектами и бесшовно интегрируются друг с другом. При желании с их помощью можно использовать большинство возможностей Java EE 7 API в Tomcat.

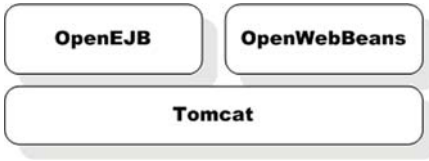

Рис. 1.6. Поддержку EJB и CDI в Tomcat можно организовать с помощью OpenEJB и OpenWebBeans

1.5. Превосходные инновации

Начиная с этого момента мы начнем погружаться в практические аспекты и посмотрим, как выглядит блистательный мир EJB 3 в программном коде. Попутно мы будем отмечать отличительные особенности EJB 3.

1.5.1. Пример «Hello User»

Демонстрация примеров «Hello World» в книгах превратилась в непреложное правило с самого первого их появления в книге «The C Programming Language» Брайана Кернигана (Brian Kernighan) и Денниса Ритчи (Dennis Ritchie) (Prentice Hall PTR, 1988)[3]. Примеры вида «Hello World» завоевали большую популярность и не без оснований – они отлично подходят для представления технологий максимально простым и понятным способом. Весь программный код для этой книги был написан с прицелом на использование системы сборки Maven, и те, кто пользуется средой разработки Eclipse, сочтут весьма удобным плагин m2eclipse для Eclipse, обеспечивающий интеграцию с Maven.

В 2004 году, один из авторов данной книги, Дебу Панда, написал статью для TheServerSide.com, в которой заметил, что с выходом EJB 3 станет возможным написать пример «Hello World», уместив его всего в несколько строк кода. Любой искушенный разработчик EJB 2 скажет вам, что подобное невозможно в EJB 2, так как такой пример должен будет включать собственный (домашний) интерфейс (home interface), интерфейс компонента, класс компонента и дескриптор развертывания. А теперь давайте посмотрим, прав ли был Дебу в своих предсказаниях.

Листинг 1.1. Сеансовый компонент HelloUser

```
package ejb3inaction.example;
import javax.ejb.Stateless;
```

[3] Брайан У. Керниган, Деннис М. Ритчи «Язык программирования С», Вильямс, 2013, ISBN: 978-5-8459-1874-1. – *Прим. перев.*

```
@Stateless // ❶ Аннотация для сеансового компонента без сохранения состояния
public class HelloUserBean implements HelloUser { // ❷ Простой Java-объект
    public String sayHello(String name) {
        return String.format("Hello %s welcome to EJB 3.1!",name);
    }
}
```

В листинге 1.1 приводится законченный, действующий компонент EJB! Класс компонента – это обычный Java-класс ❷, без каких-либо интерфейсов. Поддержка компонентов без интерфейса (no-interface view) появилась в EJB 3.1. До этого EJB требовала указывать интерфейс, чтобы определить видимые методы. Отсутствие интерфейса в объявлении класса по сути означает, что для вызова извне будут доступны все общедоступные (public) методы компонента. Это упрощает код, но требует с большим вниманием относиться к выбору области видимости для методов. Например, метод `exposeAllTheCompanysDirtySecrets()`[4] наверняка следует сделать скрытым (private). Забавная строка `@Stateless` в листинге 1.1 – это аннотация ❶, которая преобразует простой Java-объект (POJO) в полноценный сеансовый компонент EJB без сохранения состояния. Фактически, аннотации несут в себе настроечную информацию в «форме комментария», которая должна быть добавлена в код.

EJB 3 позволяет создавать компоненты EJB на основе простых Java-объектов, ничего не ведающих о службах платформы. Вы можете включать в эти объекты аннотации, добавляющие службы, такие как поддержка удаленных взаимодействий или веб-служб, и методы для вызова механизмом управления жизненным циклом компонента.

Чтобы запустить этот компонент EJB, его следует развернуть в контейнере EJB. Если у вас появилось желание опробовать данный пример, загрузите архив *actionbazaar-snapshot-2.zip* с сайта https://code.google.com/p/action-bazaar/, извлеките папку *chapter1* с проектом, подготовленным для сборки, и установите его с помощью Maven и с использованием встраиваемого сервера GlassFish.

В этой книге мы исследуем массу программного кода – какие-то примеры будут столь же простыми, как этот. Вы можете запустить пример «hello» как веб-службу, для чего достаточно добавить аннотацию `@WebService`. С помощью аннотации `@Inject` можно внедрить ресурс, например вспомогательный компонент, который будет переводить текст сообщения на другие языки. У вас есть свои мысли о том, чего бы вам хотелось получить от EJB? Продолжайте читать и мы расскажем вам, как этого добиться.

1.5.2. Аннотации и XML

До аннотаций (появившихся в Java SE 5), единственным средством определения конфигурации приложений оставались файлы XML, если не считать такие инструменты, как XDoclet, завоевавших популярность во многих более или менее прогрессивных реализациях EJB 2.

[4] Имя метода можно перевести так: «открыть все неприглядные секреты компании» – *Прим. перев.*

Использование файлов XML влечет за собой множество проблем. Формат XML чересчур многословен, трудно читаем и в нем очень легко допустить ошибку. Кроме того, XML никак не поддерживает одну из сильнейших сторон Java – строгий контроль типов. Наконец, конфигурационные файлы XML часто получаются монолитными и они отделяют информацию о настройках от программного кода, использующего ее, что усложняет сопровождение. Все вместе эти проблемы называют «Адом XML» и аннотации были созданы специально, чтобы ослабить эти проблемы.

EJB 3 стала первой распространенной технологией на основе языка Java, проложившей путь к применению аннотаций. С тех пор по ее стопам пошли многие другие инструменты, такие как JPA, JSF, Servlets, JAX-WS, JAX-RS, JUnit, Seam, Guice и Spring.

Как можно заключить из примера в листинге 1.1, аннотации являются по сути параметрами настройки, присваивающие фрагментам кода, таким как объявление класса или метода, определенные атрибуты. Когда контейнер EJB обнаруживает эти атрибуты, он добавляет соответствующие им службы. Данный прием называют *декларативным программированием*, когда разработчик указывает, что должно быть сделано, а система скрытно добавляет необходимый код.

Аннотации в EJB 3 существенно упрощают разработку и тестирование приложений. Разработчик может декларативно добавлять службы в компоненты EJB по мере необходимости. Как показано на рис. 1.7, аннотации преобразуют простой Java-объект в компонент EJB, как это делает, например, аннотация `@Stateless` в примере.

Рис. 1.7. Простые Java-объекты могут превращаться в компоненты EJB с помощью аннотаций

Несмотря на все недостатки, файлы XML все же имеют некоторые преимущества. Они позволяют быстро увидеть, как организованы компоненты. С их помощью можно изменять настройки для каждого случая развертывания в отдельности или конфигурировать компоненты, исходный код которых недоступен для изменения. Кроме того, настройки, которые слабо связанные с программным кодом, трудно выразить с помощью аннотаций. Примером могут служить настройки номера порта или URL, местоположения некоторого файла, и так далее. Поэтому вам полезно будет узнать, что EJB 3 поддерживает конфигурационные файлы XML. Более того, XML-файлы можно использовать для переопределения настроек, выраженных в виде аннотаций. Если только вы не испытываете особой любви к файлам XML, мы рекомендуем начинать с аннотаций и добавлять переопределяющие настройки в файлах XML, только когда это действительно необходимо.

1.5.3. Значения по умолчанию и явные настройки

В отличие от многих других фреймворков, таких как Spring, в EJB используется иной подход к определению поведения по умолчанию. В Spring, например, если вы

чего-то не попросите, то вы это не получите. Вам придется явно запрашивать все необходимые особенности для своих Spring-компонентов. В дополнение к упрощению настройки с применением аннотаций, EJB 3 также уменьшает общий объем конфигураций, подставляя наиболее рациональные значения по умолчанию. Например, компонент «Hello World» автоматически получает поддержку выполнения в многопоточной среде, включения в пул и транзакций без каких-либо указаний с вашей стороны. Аналогично, если вам потребуется добавить поддержку отложенных и/или асинхронных вычислений, удаленных взаимодействий или веб-служб, вам достаточно будет добавить несколько аннотаций. Здесь нет ничего, что требовалось бы понимать, явно включать или настраивать – все включено по умолчанию. То же относится к JPA и CDI. Рационализм использования умолчаний проявляется особенно ярко, когда дело доходит до организации автоматического сохранения компонентов с помощью JPA.

1.5.4. Внедрение зависимостей и поиск в JNDI

Версия EJB 3 была полностью переработана для обеспечения поддержки внедрения зависимостей. Благодаря чему теперь имеется возможность внедрять компоненты EJB в компоненты Java EE, а компоненты Java EE в компоненты EJB. Это особенно верно при использовании CDI совместно с EJB 3. Например, если потребуется получить доступ к компоненту `HelloUser`, представленному в листинге 1.1, из другого компонента EJB, сервлета или компонента JSF, это можно реализовать как показано ниже:

```
@EJB // ❶ Внедрение компонента EJB
private HelloUserBean helloUser;

void hello(){
    helloUser.sayHello(“Curious George”);
}
```

Разве не здорово?! Аннотация `@EJB` ❶ прозрачно «внедряет» компонент `HelloUserBean` в аннотированную переменную. Аннотация `@EJB` проверяет тип и имя компонента, и отыскивает его с помощью механизма JNDI. Все компоненты EJB автоматически регистрируются в JNDI в момент развертывания. Обратите внимание, что при необходимости все можно использовать поиск в JNDI. Например, динамический поиск компонента можно выполнить так:

```
Context context = new InitialContext();
HelloUserBean helloUser = (HelloUserBean)
    context.lookup(“java:module/HelloUserBean”);
helloUser.sayHello(“Curious George”);
```

Подробнее о внедрении компонентов EJB и их поиске будет рассказываться в главе 5.

1.5.5. CDI и механизм внедрения в EJB

Механизм внедрения EJB имеет более давнюю историю, чем CDI. Из этого естественно следует, что механизм CDI обладает более совершенными характеристиками, чем механизм внедрения в EJB. Что еще более важно, CDI можно использовать для внедрения практически чего угодно. Механизм внедрения EJB, напротив, можно применять только к объектам, хранящимся в JNDI, таким как компоненты EJB, а также к некоторым другим объектам, управляемым контейнером, таким как контексты EJB. Механизм CDI обеспечивает гораздо более строгий контроль типов, чем EJB. Вообще говоря, CDI является надмножеством механизма внедрения EJB. Например, ниже показано, как с помощью CDI можно внедрить компонент EJB:

```
@Inject // Внедрение компонента EJB с помощью CDI
private HelloUserBean helloUser;

void hello(){
    helloUser.sayHello("Curious George");
}
```

Может показаться, что механизм CDI следовало бы использовать для осуществления всех внедрений в окружении Java EE, но в настоящее время он имеет некоторые ограничения. Несмотря на то, что CDI способен находить компонент EJB по его типу, он не может работать с удаленными компонентами EJB. Механизм внедрения EJB (`@EJB`) различает локальные и удаленные компоненты и возвращает соответствующий тип. Поэтому механизм CDI следует использовать для внедрения, только если это возможно.

1.5.6. Тестируемость компонентов POJO

Так как все компоненты EJB в действительности являются обычными Java-объектами, они легко поддаются модульному тестированию с помощью JUnit. В процессе тестирования можно даже использовать CDI для внедрения компонентов EJB непосредственно в модульные тесты, связывания их с фиктивными объектами и так далее. Благодаря поддержке встраиваемых контейнеров, средствами JUnit можно даже выполнить полное интеграционное тестирование компонентов EJB. Некоторые проекты, такие как Arquillian, специально созданы с целью интеграции JUnit со встраиваемыми контейнерами. А в листинге 1.2 показано, как Arquillian позволяет внедрять компоненты EJB в тесты JUnit.

Листинг 1.2. Модульное тестирование компонентов EJB 3 с помощью Arquillian

```
package ejb3inaction.example;

import javax.ejb.EJB;
import org.jboss.arquillian.api.Deployment;
import org.jboss.arquillian.junit.Arquillian;
import org.jboss.shrinkwrap.api.Archive;
```

```
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.spec.JavaArchive;
import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(Arquillian.class)       // Запуск JUnit с помощью Arquillian
public class HelloUserBeanTest {
    @EJB
    private HelloUser helloUser; // Внедрение тестируемого компонента

    @Deployment
    public static Archive<?> createDeployment() {
        return ShrinkWrap.create(JavaArchive.class, "foo.jar")
        .addClasses(HelloUserBean.class);
    }

    @Test
    public void testSayHello() {
        // Собственно тестирование компонента EJB
        String helloMessage = helloUser.sayHello("Curious George");
        Assert.assertEquals(
            "Hello Curious George welcome to EJB 3.1!",helloMessage);
    }
}
```

Тестированию компонентов EJB посвящена глава 15.

1.6. Новшества в EJB 3.2

Целью версии 3.2 является дальнейшее развитие спецификации EJB в направлении создания полного решения, отвечающего всем корпоративным потребностям, и усовершенствование архитектуры EJB в направлении снижения ее сложности для разработчика. В этом разделе мы затронем некоторые новшества, появившиеся в EJB 3.2.

1.6.1. Поддержка EJB 2 теперь является необязательной

В спецификации EJB 3.2 поддержка EJB 2 была объявлена необязательной. Это означает, что серверы приложений, полностью совместимые с Java EE 7, больше не обязаны поддерживать компоненты EJB 2. Также необязательной была объявлена поддержка EJB QL и JAX-RPC.

1.6.2. Усовершенствования в компонентах, управляемых сообщениями

В EJB 3.2 компоненты MDB претерпели глобальную перестройку. Благодаря обновлениям в JMS 2.0 был упрощен прикладной интерфейс (API) а также по-

явилась поддержка интеграции с расширенными возможностями Java в других областях, такими как внедрение зависимостей с помощью CDI. Дополнительно к этому было сделано необязательным использование интерфейса `javax.jms.MessageListener`, что дает возможность создавать компоненты MDB с пустым интерфейсом слушателя (no-methods listener), который превращает общедоступные методы класса в методы слушателя. Ниже представлен краткий обзор упрощенного API для компонентов MDB.

Отправка сообщения:

```
@Inject @JMSConnectionFactory("jms/QueueConnectionFactory")
private JMSContext context; // Внедрение JMSContext

@Resource(lookup="jms/MessageQueue")
private Queue queue;        // Внедрение приемника

public void sendMessage(String txtToSend) {
    // Использование упрощенного API для отправки сообщения
    context.createProducer().send(queue, txtToSend);
}
```

Прием сообщения:

```
// Аннотируется для контейнера как MDB
// прослушивает "jms/BidQueue"
@MessageDriven(mappedName="jms/BidQueue")
// Класс, реализующий MessageListener для приема сообщений JMS
public class BidMdb implements MessageListener {
    @Resource
    // Внедряет MessageDrivenContext при необходимости
    private MessageDrivenContext mdc;

    // Метод, реализующий MessageListener и обрабатывающий сообщения
    public void onMessage(Message inMessage) {
        // обработка сообщения
    }
}
```

1.6.3. Усовершенствования в сеансовых компонентах с сохранением состояния

Сеансовые компоненты не претерпели больших изменений в EJB 3.2, по крайней мере, изменения оказались не настолько существенными, как в MDB. В сеансовые компоненты с сохранением состояния было внесено несколько усовершенствований, касающихся поддержки пассивации (passivation) и транзакций. Они перечислены ниже.

Отключение пассивации

До версии EJB 3.2 компоненты с сохранением состояния требовали реализации интерфейса `Serializable` во всех объектах, составляющих их, чтобы контейнер

EJB мог пассивировать (сохранять в дисковой памяти) их. Если какой-то объект в компоненте с сохранением состояния не поддерживал сериализацию, операция пассивации терпела неудачу и контейнер EJB мог уничтожить такой компонент, с потерей информации о его состоянии. Несмотря на то, что весьма желательно обеспечить поддержку сериализации в компонентах с сохранением состояния, иногда это невозможно. Чтобы предотвратить пассивацию таких компонентов, можно воспользоваться настройкой `passivationCapable`:

```
@Stateful(passivationCapable=false) // Препятствует пассивации компонента
public class BiddingCart {
}
```

Поддержка транзакций в методах обратного вызова, используемых механизмом управления жизненным циклом

Сеансовые компоненты с сохранением состояния и прежде поддерживали методы обратного вызова для использования механизмом управления жизненным циклом, но предыдущие спецификации EJB никак не регламентировали поддержку транзакций в этих методах. Поэтому в новой версии такая поддержка было добавлена явно, посредством аннотации `@TransactionalAttribute(REQUIRES_NEW)`. `REQUIRES_NEW` – это единственное допустимое значение:

```
@Stateful // Сеансовый компонент с сохранением состояния
public class BiddingCart {
    @PostConstruct
    // Метод lookupDefaults, вызываемый по событию @PostConstruct,
    // выполняется в рамках отдельной транзакции
    @TransactionAttribute(REQUIRES_NEW)
    public void lookupDefaults() { }
}
```

1.6.4. Упрощение локальных интерфейсов компонентов без сохранения состояния

В прежних версиях EJB, предшествовавших версии 3.2, если интерфейсы не были помечены как `@Local` или `@Remote`, реализующий компонент обязан был определять их. Ниже показано, как выглядит компонент и интерфейс в версиях EJB, предшествовавших 3.2:

```
public interface A {} // Интерфейсы A и B не определены,
public interface B {} // как @Local или @Remote

@Stateless
@Local({A.class, B.class}) // ❶ До версии 3.2 требовалось использовать @Local,
                           //   чтобы объявить оба интерфейса локальными
public class BidServicesBean implements A, B {}
```

Теперь EJB 3.2 поддерживает достаточно рациональные умолчания. Так по умолчанию, все интерфейсы, которые не объявлены как `@Local` или `@Remote`, автоматически становятся локальными. Благодаря чему можно опустить строку ❶ и записать этот же код, как показано ниже:

```
@Stateless
public class BidServicesBean implements A, B {}
```

1.6.5. Усовершенствования в TimerService API

В TimerService API была расширена область, где можно обращаться к таймерам. В версиях до EJB 3.2 объекты `Timer` и `TimerHandler` были доступны только компоненту, владеющему таймером. Это ограничение было устранено, и теперь в вашем распоряжении имеется новый метод `getAllTimers()`, возвращающий список всех активных таймеров в модуле EJB. Это дает возможность из любой точки в коде просматривать и изменять любые таймеры.

1.6.6. Усовершенствования в EJBContainer API

Для нужд EJB 3.2 была внесена пара изменений в EJBContainer API встраиваемых контейнеров. Во-первых, теперь API реализует интерфейс `AutoCloseable`, благодаря чему появилась возможность использовать инструкцию `try-with-resources`:

```
try (EJBContainer c = EJBContainer.createEJBContainer();) {
    // операции с контейнером
}
```

Во-вторых, объект `EJBContainer` встраиваемого контейнера требует наличия поддержки группы EJB Lite в EJB API. Подробнее о группах EJB API рассказывается в следующем разделе.

1.6.7. Группы EJB API

Поскольку технология EJB составляет основу для разработки корпоративных приложений на Java, она должна предоставлять большое число служб для удовлетворения прикладных потребностей. В их число входят службы поддержки транзакций, безопасности, удаленного доступа, синхронного и асинхронного выполнения, а также слежения за состоянием. Это далеко не полный список, его можно продолжать и продолжать. Но не все корпоративные приложения нуждаются во всех службах EJB. Чтобы облегчить жизнь разработчикам, в EJB 3.2 были определены группы EJB API. Эти группы четко ограничивают подмножества функций EJB. В спецификации EJB 3.2 определены следующие группы:

- EJB Lite;
- компоненты MDB;
- EJB 3.x Remote;
- службы таймеров механизма сохранения EJB;

- конечные точки JAX-WS Web Service;
- встраиваемый контейнер EJB (необязательно);
- EJB 2.x API;
- сущности механизма сохранения (необязательно);
- конечные точки JAX-RPC Web Service (необязательно).

За исключением некоторых групп, которые являются необязательными, полный контейнер EJB обязан реализовать все группы. Наиболее важной из них является группа EJB Lite. Она содержит минимально возможное число особенностей EJB, которых, впрочем, вполне достаточно для удовлетворения потребностей в транзакциях и безопасности в большинстве случаев. Это делает реализацию EJB Lite идеальной для встраивания в контейнер сервлетов, такой как Tomcat, чтобы привнести в него некоторые корпоративные качества, или для использования в приложении для Android на планшетном компьютере для обработки данных.

Теперь, когда мы познакомились с некоторыми новыми возможностями EJB 3.2, давайте сравним технологию EJB с другими фреймворками, имеющимися на рынке, которые также имеют целью предоставить решения для разработки корпоративных приложений на Java.

1.7. В заключение

Теперь вы должны иметь неплохое представление о том, что такое EJB 3, какие возможности она несет и почему следует задуматься об ее использовании в разработке серверных приложений. Мы коротко рассказали вам о новых особенностях EJB 3, включая следующие важные моменты:

- компоненты EJB 3 – это простые Java-объекты (POJO), настраиваемые с помощью аннотаций;
- доступ к компонентам EJB из клиентских приложений и модульных тестов сильно упростился, благодаря поддержке внедрения зависимостей;
- EJB предоставляет полный комплект мощных и масштабируемых служб, что называется «из коробки».

Мы также показали, как выглядит программный код, использующий функции EJB 3. Получив эти знания, вам наверняка не терпится увидеть еще больше примеров. Мы удовлетворим ваше любопытство, хотя бы частично, в следующей главе. Готовьтесь, вас ждет головокружительный тур по EJB 3 API, в ходе которого мы покажем вам, насколько простым может быть код.

ГЛАВА 2.
Первая проба EJB

Эта глава охватывает следующие темы:

- приложение ActionBazaar;
- сеансовые компоненты с сохранением и без сохранения состояния в приложении ActionBazaar;
- интеграция CDI и EJB 3;
- сохранение объектов в JPA 2.

В эпоху глобализации изучение технологий с книгой на коленях и клавиатурой под руками, попутно решая практические задачи, стало нормой. Посмотрим правде в глаза – где-то в глубине души вы наверняка предпочли бы пройти подобное «крещение огнем», чем снова и снова тащиться по старым, проторенным дорогам. Эта глава – для храбрых первооткрывателей, живущих внутри нас, желающих заглянуть за горизонт, в новый мир EJB 3.

В первой главе вы осмотрели ландшафт EJB 3 с высоты 6000 метров, из окна сверхзвукового лайнера. Мы дали определение EJB, описали службы и общую структуру EJB 3, а также рассказали, как EJB 3 связана с CDI и JPA 2. В этой главе мы пересядем в маленький разведывательный самолет и пролетим намного ниже. Здесь мы бегло рассмотрим пример решения практической задачи с использованием EJB 3, JPA 2 и CDI. В этом примере будут использоваться компоненты EJB 3 нескольких типов, многоуровневая архитектура и некоторые службы, представленные в главе 1. Вы сами убедитесь, насколько проста и удобна EJB 3 и как быстро можно включить ее в работу.

Если вы не большой любитель обозревать окрестности с высоты, не пугайтесь. Представьте себе эту главу, как первый день на новой работе, когда вы пожимаете руки незнакомцам за соседними столами. В следующих главах вы поближе сойдетесь с новыми коллегами, узнаете об их пристрастиях, предубеждениях и причудах, и научитесь обходить их недостатки. А пока от вас требуется только запомнить их имена.

Опробование примеров кода

С этого момента мы предлагаем не пренебрегать возможностью исследования примеров кода. Вы можете получить полный комплект примеров, загрузив zip-файл со страницы www.manning.com/panda2. Мы также настоятельно рекомендуем установить среду разработки по своему выбору для опробования кода. В этом случае вы сможете следовать за нами, вносить в код свои изменения и запускать его внутри контейнера.

В этой главе мы приступим к решению задачи, которое продолжится в остальных главах, – созданию приложения ActionBazaar. Это воображаемая система, на примере которой мы будем пояснять различные аспекты. В некотором смысле эта книга является учебным примером разработки приложения ActionBazaar с использованием EJB 3. А теперь коротко пройдемся по приложению ActionBazaar, чтобы понять, о чем пойдет речь.

2.1. Введение в приложение ActionBazaar

ActionBazaar – это простой интернет-аукцион, по образу и подобию eBay. Продавцы сдувают пыль с сокровищ, хранящихся у них в подвалах, делают несколько размытых фотографий и посылают их в приложение ActionBazaar. Нетерпеливые покупатели входят в соревновательный раж и перебивают цену друг друга в попытке приобрести сокровища, изображенные на нечетких фотографиях и снабженных описаниями с орфографическими ошибками. Победители платят деньги. Продавцы высылают товар. И все счастливы, ну или почти все.

Как бы нам ни хотелось присвоить себе лавры первооткрывателей, тем не менее, мы должны признать, что впервые идея приложения ActionBazaar была выдвинута в книге «Hibernate in Action» Кристиана Бауэра (Christian Bauer) и Гэвина Кинга (Gavin King) (Manning, 2004), в виде примера приложения CaveatEmptor. Книга «Hibernate in Action» в основном посвящена разработке уровня хранения данных с применением фреймворка Hibernate объектно-реляционного отображения. Позднее эта идея была использована Патриком Лайтбоди (Patrick Lightbody) и Джейсоном Каррера (Jason Carreira) в книге «WebWork in Action» (Manning, 2005) при обсуждении открытого фреймворка уровня представления. Нам показалось, что было бы неплохо использовать эту же идею и в нашей книге, «EJB 3 в действии».

В этом разделе мы представим вам приложение ActionBazaar. Сначала мы определим ограниченную архитектуру ActionBazaar, а затем будем наращивать ее с применением EJB 3. В оставшейся части главы, следующей за этим разделом, мы займемся исследованием некоторых важных функций этих технологий на примерах из приложения ActionBazaar, познакомимся с некоторыми типами компонентов EJB и посмотрим, как они используются в комплексе с CDI и JPA 2.

Итак, начнем со знакомства с требованиями и архитектурой примера.

2.1.1. Архитектура

Для начального знакомства с EJB 3 ограничимся в этой главе небольшим подмножеством функциональности ActionBazaar, начав с предложения цены и закончив

оформлением сделки с победителем. Это подмножество функциональности изображено на рис. 2.1.

Рис. 2.1. Цепочка операций, выполняемых приложением ActionBazaar, на основе которой мы займемся исследованием EJB 3. Претендент предлагает цену за желаемый товар, выигрывает, заказывает и немедленно получает подтверждение. В процессе подтверждения заказа пользователь вносит информацию о способе оплаты. После подтверждения получения денег продавец организует доставку покупки

Функции, перечисленные на рис. 2.1, представляют основу приложения ActionBazaar. Здесь отсутствуют такие основные функции, как подача объявления о продаже, просмотр списка объявлений и поиск нужного товара. Мы оставили эти функции для обсуждения в дальнейшем. Их реализация включает разработку модели всей предметной области, которая описывается в главе 9, где мы приступим к знакомству с особенностями моделирования предметных областей и механизма хранения данных JPA 2.

Цепочка операций, изображенная на рис. 2.1, начинается с того, что пользователь решил купить некоторый товар и предложить свою цену. Пользователь Дженни (Jenny) выбрала прекрасный подарок для дедушки на Рождество и быстро предлагает цену $5.00. Когда аукцион подходит к концу, побеждает самая высокая ставка. Дженни повезло, и никто не предложил цену выше, чем она, соответственно ее ставка $5.00 выигрывает. Как победителю, Дженни разрешают заказать товар у продавца Джо (Joe). В заказе указывается вся необходимая в таких делах информация – адрес доставки, расчет сумм, необходимых на упаковку и доставку, общая сумма, и так далее. Дженни убеждает свою маму оплатить заказ своей кредитной картой и указывает адрес дедушки, куда следует доставить покупку. Мало чем отличаясь от таких площадок электронных торгов, как Amazon.com и eBay, приложение ActionBazaar не заставляет пользователя ждать, пока пройдут все платежи, перед подтверждением заказа. Вместо этого заказ подтверждается сразу же после приема всех данных и запуска процесса расчетов, который протекает параллель-

но, в фоновом режиме. Дженни получает подтверждение о приеме заказа сразу после щелчка на кнопке **Order** (Заказать). И хотя Дженни этого не видит, процедура списания денег с кредитной карты ее мамы начнется немедленно, в фоновом режиме, сразу после приема подтверждения. Когда процесс расчетов завершится, обеим участникам сделки, Дженни и Джо, будут отправлены электронные письма с уведомлением. После получения уведомления о переводе денег, Джо доставит подарок дедушке Дженни к означенному моменту, как раз перед Рождеством!

В следующем разделе вы увидите, как с помощью EJB 3 можно реализовать компоненты, реализующие эту цепочку операций. Но, прежде чем приступить к изучению схемы решения в следующем разделе, попробуйте наглядно представить, как могли бы располагаться необходимые компоненты в многоуровневой архитектуре EJB. Какое место, по вашему мнению, могли бы занять в этой схеме сеансовые компоненты, CDI, сущности и JPA 2 API, учитывая знания, полученные в главе 1?

2.1.2. Решение на основе EJB 3

На рис. 2.2 показано, как с помощью EJB 3 можно реализовать сценарий работы ActionBazaar, описанный в предыдущем разделе, в традиционной четырехуровневой архитектуре и с применением модели предметной области.

Если внимательно изучить сценарий на рис. 2.2, можно заметить, что пользователь активизирует только две процедуры: добавление ставки и заказ товара в случае победы. Как вы уже наверняка догадались процедуры добавления ставки и заказа реализованы с помощью сеансовых компонентов (`BidService` и `OrderProcessor`) на уровне прикладной логики.

В ходе обеих процедур производится сохранение данных. Компоненту `BidService` нужно добавить в базу данных запись о сделанной ставке. Аналогично компоненту `OrderProcessor` нужно добавить запись с информацией о заказе. Все необходимые изменения в базе данных выполняются с помощью двух сущностей на уровне хранения, действующих под управлением JPA – `Bid` и `Order`. При этом `BidService` использует сущность `Bid`, а `OrderProcessor` – сущность `Order`. Посредством этих сущностей компоненты на уровне прикладной логики используют компоненты `BidDao` и `OrderDao` уровня хранения. Обратите внимание, что объекты DAO доступа к данным не обязательно должны быть компонентами EJB, потому что для решения стоящих перед ними задач не требуются службы, предоставляемые EJB. В действительности, как будет показано далее, объекты DAO являются простейшими Java-объектами, управляемыми механизмом CDI, и не используют никакие службы, кроме внедрения зависимостей. Напомним, что даже при том, что сущности JPA 2 содержат настройки ORM, сами они не участвуют в сохранении непосредственно. Как вы увидите в фактической реализации, для добавления, удаления, изменения и извлечения сущностей объекты DAO используют EntityManager API из JPA 2.

Если ваше мысленное представление достаточно близко совпадает с рис. 2.2, значит и код, который приводится далее, будет вам понятен, даже при том, что вы, возможно, не знакомы с особенностями EJB 3.

Рис. 2.2. Сценарий работы ActionBazaar реализован с помощью EJB 3. С позиции EJB 3 уровень представления выглядит как аморфное облако (в данном случае реализованное на основе JSF), генерирующее запросы к уровню прикладной логики. Компоненты уровня прикладной логики соответствуют различным операциям в сценарии – размещение ставки и оформление заказа для победившей ставки. Для сохранения информации в базе данных компоненты уровня прикладной логики используют сущности JPA, используя для этого объекты доступа к данным (DAO)

2.2. Реализация прикладной логики с применением EJB 3

Итак, приступим к исследованию решения, начав с уровня прикладной логики, как вы поступили бы в случае действующего приложения. Сеансовые компоненты EJB 3 идеально подходят для реализации процедур моделирования ставки и заказа. По умолчанию они поддерживают транзакции, многопоточную модель выполнения и возможность размещения в пулах – все характеристики, необходимые для постро-

ения прикладного уровня приложения. Сеансовые компоненты – самые простые, и вместе с тем самые универсальные элементы EJB 3, поэтому мы и начнем с них.

Напомним, что существует три разновидности сеансовых компонентов: с сохранением состояния, без сохранения состояния и компоненты-одиночки (singleton). В наших примерах мы будем использовать только сеансовые компоненты с сохранением и без сохранения состояния. Позднее нам также понадобятся компоненты, управляемые сообщениями (Message-Driven Beans, MDB). Для начала исследуем сеансовые компоненты без сохранения состояния, просто потому, что они проще.

2.2.1. Использование сеансовых компонентов без сохранения состояния

Сеансовые компоненты без сохранения состояния часто используются для моделирования операций, которые могут быть выполнены за один вызов метода, таких как добавление ставки в сценарии ActionBazaar. На практике сеансовые компоненты без сохранения состояния используются чаще других на уровне прикладной логики. Метод `addBid`, представленный в листинге 2.1, вызывается веб-уровнем приложения ActionBazaar, когда пользователь решает сделать ставку. Параметр метода, объект `Bid`, представляет саму ставку. Этот объект содержит имя претендента, информацию о товаре, для которого делается ставка, и сумма ставки. Как вы уже знаете, работа этого метода заключается в том, чтобы сохранить объект `Bid` в базе данных. Ближе к концу главы вы увидите, что в действительности объект `Bid` является сущностью JPA 2.

Листинг 2.1. Сеансовый компонент без сохранения состояния BidService

```
@Stateless // Пометить POJO как сеансовый компонент без сохранения состояния
public class DefaultBidService implements BidService {
    @Inject // Внедрить объект DAO
    private BidDao bidDao;
    ...
    public void addBid(Bid bid) {
        bidDao.addBid(bid);
    }
    ...
}

@Local // Пометить интерфейс как локальный
public interface BidService {
    ...
    public void addBid(Bid bid);
    ...
}
```

Первое, на что вы наверняка обратили внимание, насколько простым выглядит код. Класс `DefaultBidService` – это простой Java-объект (Plain Old Java Object, POJO) а интерфейс `BidService` – простой Java-интерфейс (Plain Old Java Interface, POJI). Интерфейс не содержит ничего такого, что вызывало бы сложности при его

реализации в наследующем классе. Единственно примечательной особенностью в листинге 2.1 являются три аннотации – `@Stateless`, `@Local` и `@Inject`.

- `@Stateless` – Аннотация `@Stateless` сообщает контейнеру EJB, что `DefaultBidService` является сеансовым компонентом без сохранения состояния. Встретив такую аннотацию, контейнер автоматически добавит в компонент поддержку многопоточной модели выполнения, транзакций и возможность размещения в пулах. Поддержка многопоточности и транзакций гарантируют возможность использования любых ресурсов, таких как база данных или очередь сообщений, без необходимости вручную писать код для обслуживания конкуренции или транзакций. Поддержка размещения в пулах гарантирует надежность даже при очень высоких нагрузках. При необходимости в компонент можно также добавить поддержку дополнительных служб EJB, таких как безопасность, планирование и интерцепторы.
- `@Local` – Аннотация `@Local`, отмечающая интерфейс `BidService` сообщает контейнеру, что реализация `BidService` может быть доступна локально, посредством интерфейса. Так как компоненты EJB и компоненты, использующие их, обычно находятся в одном и том же приложении, в этом есть определенный смысл. При желании аннотацию `@Local` можно опустить и интерфейс все равно будет интерпретироваться контейнером как локальный. Как вариант, интерфейс можно пометить аннотацией `@Remote` или `@WebService`. В случае применения аннотации `@Remote`, к компоненту будет предоставлен удаленный доступ через механизм Java Remote Method Invocation (RMI), идеально подходящий для организации удаленного доступа со стороны Java-клиентов. Если потребуется организовать удаленный доступ к компонентам EJB из клиентов, реализованных на других платформах, таких как Microsoft .NET или PHP, можно включить поддержку SOAP, применив к интерфейсу или классу компонента аннотацию `@WebService`. Отметьте также, что компоненты EJB вообще могут обходиться без каких-либо интерфейсов.
- `@Inject` – Как вы уже знаете, служба добавления ставки зависит от объекта DAO, осуществляющего сохранение данных. Аннотация `@Inject`, реализуемая механизмом CDI, внедряет объект DAO (не имеющий ничего общего с EJB) в переменную экземпляра `BidService`. Если вы пока не знакомы с механизмом внедрения зависимостей, можете представить себе аннотацию `@Inject`, как инструкцию, выполняющую нечто немного необычное, а если честно, то нечто магическое. Кто-то из вас может задаться вопросом: «А можно ли использовать приватную переменную `bidDao`, которая нигде не инициализируется?»! Если бы контейнер не позаботился об этой переменной, вы получили бы позорное исключение `java.lang.NullPointerException` при попытке вызвать метод `addBid` из листинга 2.1, потому что переменная bidDao содержала бы пустое значение. Интересно отметить, что внедрение зависимости можно представить себе как «нестандартный» способ инициализации переменных. Аннотация `@Inject` заставляет контейнер «иници-

ализировать» переменную `bidDao` соответствующей реализацией DAO до того, как произойдет первое обращение к ней.

Подробнее об отсутствии сохраняемого состояния

Так как результаты вызова метода `addBid` (новая ставка) сохраняются в базе данных, клиенту нет нужды беспокоиться о внутреннем состоянии компонента. Здесь нет никакой необходимости в сохранении состояния компонента, чтобы гарантировать сохранность значений его переменных между вызовами. Именно это свойство и называют «без сохранения состояния» при программировании серверных приложений.

Сеансовый компонент `BidService` может позволить себе роскошь не сохранять свое состояние, потому что операция добавления новой ставки выполняется в один шаг. Однако не все прикладные операции так просты. Когда операция выполняется в несколько этапов, может возникнуть необходимость в сохранении внутреннего состояния, чтобы не потерять связь между этапами, – это типичный прием реализации сложных операций. Сохранение состояния может пригодиться, например, когда пользователь при выполнении некоторого этапа определяет, какой этап будет следующим. Представьте себе мастера настройки, выполняющего свою работу в несколько этапов, опираясь на ответы пользователя. Ввод пользователя сохраняется на каждом этапе работы такого мастера и используется, чтобы определить, какой вопрос задать пользователю на следующем этапе. Сеансовые компоненты с сохранением состояния стараются сделать сохранение состояния серверного приложения максимально простым.

Это все, что мы хотели сказать о сеансовых компонентах без сохранения состояния и о службе добавления новой ставки. Теперь давайте обратим наше внимание на процедуру обработки заказа, где как раз требуется сохранять состояние. Немного ниже мы также посмотрим, как служба создания новой ставки используется механизмом JSF, и как выглядит реализация объектов DAO.

2.2.2 Использование сеансовых компонентов с сохранением состояния

В отличие от сеансовых компонентов без сохранения состояния, компоненты с сохранением состояния гарантируют восстановление внутреннего состояния компонента при следующем обращении клиента. Контейнер обеспечивает такую возможность, управляя сеансами и реализуя сохранение.

Управление сеансом

Во-первых, контейнер гарантирует клиенту возможность повторно обращаться к выделенному для него компоненту и вызывать разные его методы. Представьте себе такой компонент, как коммутационный телефонный узел, который будет вас соединять с одним и тем же представителем службы поддержки в пределах некоторого периода времени (такой период называется сеансом).

Во-вторых, контейнер гарантирует сохранность переменных экземпляра на протяжении всего сеанса, не требуя от разработчика писать какой-то дополнительный код для этого. Если продолжить аналогию со службой поддержки, контейнер гарантирует, что информация о вас и вся история ваших звонков в заданный период времени автоматически будет появляться на экране перед представителем службы поддержки при каждом вашем звонке. Операция оформления заказа в приложении ActionBazaar является наглядным примером применения сеансового компонента с сохранением состояния, потому что она выполняется в четыре этапа, каждый из которых грубо соответствует отдельному экрану, которые видит пользователь:

1. Выбор товара для заказа – Процедура оформления заказа начинается с щелчка пользователя на кнопке **Order Item** (Оформить заказ) на странице со списком выигравших ставок, после чего товар автоматически добавляется в заказ.
2. Определение информации о доставке, включая способ доставки, адрес, страховка и так далее. Информация о пользователе может храниться в истории предыдущих заказов, в том числе и некоторые умолчания. Пользователь может просмотреть историю и использовать информацию из предыдущих заказов. При этом на экране должна отображаться только информация, соответствующая выбранному товару (например, продавец может быть готов отправить товар только ограниченным числом способов и по ограниченному кругу адресов). После ввода информации о доставке автоматически рассчитывается ее стоимость.
3. Определение информации о порядке оплаты, такой как номер кредитной карты и адрес оплаты. Для данных о порядке оплаты также поддерживаются функции истории/умолчаний/фильтрации, как и для информации о доставке.
4. Подтверждение заказа после просмотра окончательной формы заказа, включающей общую стоимость.

Все эти этапы изображены на рис. 2.3. При использовании компонента с сохранением состояния, данные, вводимые пользователем на каждом этапе (а также внутренние данные, видеть которые пользователю совсем необязательно), могут накапливаться в переменных компонента до завершения процедуры.

Теперь, когда вы узнали все необходимое, давайте посмотрим, как это реализуется на практике.

Реализация решения

Следующий пример демонстрирует возможную реализацию процедуры оформления заказа в ActionBazaar в виде компонента `DefaultOrderProcessor`. Как видно из примера, `DefaultOrderProcessor` примерно отражает модель выполнения процедуры заказа. Методы в листинге приводятся в порядке, соответствующем последовательности их вызова из страниц JSF. Каждый из них реализует отдельный этап процесса оформления заказа. Методы `setBidder` и `setItem` вызываются уровнем представления в самом начале процедуры, когда пользователь щелкает на кнопке **Order** (Заказать). Клиентом, вероятнее всего, является теку-

щий зарегистрированный пользователь, а товаром – текущий выбранный товар. В методе `setBidder` компонент извлекает историю заказов с адресами доставки и информацией о платежах данного клиента, и сохраняет эти данные вместе с информацией о клиенте. В методе `setItem` выполняется фильтрация данных доставке и платежах, и оставляются только те данные, что могут быть применены к текущему товару. Сам товар также сохраняется в переменной экземпляра для использования на следующих этапах.

Рис. 2.3. Чтобы сделать процесс управляемым, приложение ActionBazaar разбивает его на несколько этапов. На первом этапе пользователь выбирает товар для заказа, на втором определяет адрес и способ доставки, на третьем указывает порядок оплаты. Завершается этот процесс обзором заключительной формы заказа и подтверждением

Следующее, что делает уровень JSF – запрашивает у пользователя адрес и способ доставки заказа. Для большего удобства, уровень представления предложит пользователю использовать любые допустимые сведения, указанные им в прошлом. Сохраненная информация о доставке извлекается с помощью метода `getShippingChoices`. Когда пользователь выберет элемент из истории или введет новые сведения, данные передаются обратно компоненту оформления заказа, через вызов метода `setShipping`. После установки информации о доставке выполняется сохранение истории клиента, если это необходимо. Компонент оформления заказа также немедленно рассчитает стоимость доставки и добавит ее во

внутреннюю переменную. Уровень JSF может получить эти сведения вызовом метода `getShipping`. Аналогично обрабатывается информация о порядке оплаты – с использованием методов `getBillingChoices` и `setBilling`.

В самом конце вызывается метод `placeOrder`, уже после того, как пользователь проверил заказ и щелкнул на кнопке **Confirm Order** (Подтвердить заказ). Метод `placeOrder` создает и заполняет фактический объект `Order` и пытается получить оплату с клиента, включая стоимость товара, его доставки, страховки и другие выплаты. Сделать это можно множеством способов, например, выполнить перевод средств с кредитной карты или со счета в банке. После попытки перевода денег компонент извещает покупателя и продавца о результатах. Если перевод состоялся, продавцу остается отправить товар по указанному адресу, указанным способом. Если попытка оплаты потерпела неудачу, клиент должен исправить, возможно, ошибочную информацию о способе оплаты и отправить ее вновь. Наконец, компонент сохраняет запись в базе данных, отражающую результат попытки оплаты, как показано в листинге 2.2.

Листинг 2.2. Сеансовый компонент с сохранением состояния OrderProcessor

```
@Stateful // ❶ Пометить POJO как компонент с сохранением состояния
public class DefaultOrderProcessor implements OrderProcessor {
    ...
    private Bidder bidder;                    // ❷ Определения
    private Item item;                        //    переменных
    private Shipping shipping;                //    экземпляра
    private List<Shipping> shippingChoices; //    с состоянием
    private Billing billing;                  //
    private List<Billing> billingChoices;     //

    public void setBidder(Bidder bidder) {
        this.bidder = bidder;
        this.shippingChoices = getShippingHistory(bidder);
        this.billingChoices = getBillingHistory(bidder);
    }

    public void setItem(Item item) {
        this.item = item;
        this.shippingChoices = filterShippingChoices(shippingChoices, item);
        this.billingChoices = filterBillingChoices(billingChoices, item);
    }

    public List<Shipping> getShippingChoices() {
        return shippingChoices;
    }

    public void setShipping(Shipping shipping) {
        this.shipping = shipping;
        updateShippingHistory(bidder, shipping);
        shipping.setCost(calculateShippingCost(shipping, item));
    }

    public Shipping getShipping() {
        return shipping;
```

```
    }

    public List<Billing> getBillingChoices() {
        return billingChoices;
    }

    public void setBilling(Billing billing) {
        this.billing = billing;
        updateBillingHistory(bidder, billing);
    }

    @Asynchronous // ❸ Объявляет метод асинхронным
    @Remove       // ❹ Объявляет метод заключительным
    public void placeOrder() {
        Order order = new Order();
        order.setBidder(bidder);
        order.setItem(item);
        order.setShipping(shipping);
        order.setBilling(billing);
        try {
            bill(order);
            notifyBillingSuccess(order);
            order.setStatus(OrderStatus.COMPLETE);
        } catch (BillingException be) {
            notifyBillingFailure(be, order);
            order.setStatus(OrderStatus.BILLING_FAILED);
        } finally {
            saveOrder(order);
        }
    }
    ...
}
```

Как видите, разработка компонентов с сохранением и без сохранения состояния отличается совсем немного. С точки зрения разработчика единственное отличие состоит в том, что класс `DefaultOrderProcessor` помечен аннотацией `@Stateful` вместо `@Stateless` ❶. Однако, как вы уже знаете, за кулисами различия выглядят гораздо более существенными и проявляются в том, как контейнер обслуживает взаимоотношения между клиентом и значениями, хранящимися в переменных экземпляра компонента ❷. Аннотация `@Stateful` также служит своеобразным сигналом для разработчика клиентской части, сообщающим, чего можно ожидать от компонента, если его поведение будет сложно определить из API и документации.

Аннотация `@Asynchronous` ❸ перед методом `placeOrder` обеспечивает возможность асинхронного выполнения метода. Это означает, что компонент будет возвращать управление клиенту немедленно, а метод продолжит выполнение в фоновом легковесном процессе. Это важная особенность в данном случае, потому что процесс оплаты потенциально может занять продолжительное время. Вместо того, чтобы заставлять пользователя ждать завершения процесса оплаты, благодаря аннотации `@Asynchronous` пользователь немедленно получит подтверждение о приеме заказа, а заключительный этап процесса оформления заказа сможет завершиться в фоновом режиме.

Обратите также внимание на аннотацию `@Remove` ❹ перед методом `placeOrder`. Хотя эта аннотация является необязательной, она играет важную роль в обеспечении высокой производительности в серверных приложениях. Аннотация `@Remove` отмечает окончание процедуры, моделируемой компонентом с сохранением состояния. В данном случае код сообщает контейнеру, что после вызова метода `placeOrder` нет необходимости продолжать поддерживать сеанс с клиентом. Если не сообщить контейнеру, что вызов этого метода означает окончание процедуры, контейнер будет ждать достаточно долго, пока не примет решение, что время истекло и сеанс можно безопасно закрыть. А так как гарантируется, что компоненты с сохранением состояния будут доступны клиенту на протяжении всего сеанса, это может означать хранение большого объема фактически ненужных данных в драгоценной памяти сервера в течение довольно длительного периода времени!

2.2.3. Модульное тестирование компонентов EJB 3

Появление возможности использовать EJB 3 в окружении Java SE стало одним из интереснейших событий в EJB 3.1. Как уже говорилось в главе 1, это достигается с помощью контейнеров EJB 3, которые могут встраиваться в любое окружение времени выполнения Java. Хотя такое было возможно и прежде, с помощью нестандартных встраиваемых контейнеров, таких как OpenEJB для Java EE 5, тем не менее, спецификация EJB 3.1 объявила поддержку таких контейнеров обязательной для всех реализаций.

Встроенные контейнеры особенно полезны в модульном тестировании компонентов EJB 3 с применением таких фреймворков, как JUnit и TestNG. Обеспечение надежности модульного тестирования компонентов EJB 3 является основной задачей таких проектов, как Arquillian. В листинге 2.3 показано, как легко протестировать сеансовый компонент `OrderProcessor` с помощью JUnit. Данный модульный тест имитирует последовательность действий клиента с помощью объектов-имитаций пользователя и товара.

Листинг 2.3. Клиент сеансового компонента с сохранением состояния

```
@RunWith(Arquillian.class) // ❶ Интегрировать Arquillian в JUnit
public class OrderProcessorTest {
    @Inject                // ❷ Внедрить экземпляр компонента
    private OrderProcessor orderProcessor;
    ...
    @Test                  // ❸ Выполняемый тест
    public void testOrderProcessor {
        // Имитация клиента
        Bidder bidder = (Bidder) userService.getUser(new Long(100));
        // Имитация товара
        Item item = itemService.getItem(new Long(200));
        orderProcessor.setBidder(bidder);
        orderProcessor.setItem(item);

        // Получить историю доставки для клиента
        List<Shipping> shippingChoices = orderProcessor.getShippingChoices();

        // Выбрать первый элемент списка
```

```
        orderProcessor.setShipping(shippingChoices.get(0));

        // Получить историю платежей для клиента
        List<Billing> billingChoices = orderProcessor.getBillingChoices();

        // Выбрать первый элемент списка
        orderProcessor.setBilling(billingChoices.get(0));

        // Завершить процедуру и сеанс
        orderProcessor.placeOrder();

        // Ждать некоторое время перед переносом обработки
        // заказа в отдельный процесс
    }
}
```

Аннотация `@RunWith` ❶ сообщает фреймворку JUnit, что он должен запустить Arquillian в ходе тестирования. Arquillian управляет жизненным циклом встраиваемого контейнера EJB – контейнер запускается перед началом тестирования и завершается по его окончании. Затем Arquillian развертывает тестируемые компоненты во встроенном контейнере – мы опустили код, осуществляющий развертывание, но вы можете увидеть его в загружаемых примерах. Arquillian также отвечает за внедрение компонента управления процедурой оформления заказа в тест ❷, а также всех его зависимостей.

Сам тест достаточно прост и понятен. Во-первых, он отыскивает объекты-имитации клиента и товара (используя пару других служб EJB). На языке модульного тестирования объекты, представляющие клиента и товар, называются частями испытуемого набора данных. Затем выполняется объекты-имитации делаются частью процесса оформления заказа. Тест также имитирует извлечение истории доставки и оплаты клиента и устанавливает необходимые параметры. в обоих случаях тест выбирает первые элементы из списков историй. Наконец, тест завершает процедуру, выполняя размещение заказа. Сеансовый компонент с сохранением состояния начинает свой жизненный цикл в момент внедрения в тест и завершает его сразу после окончания фонового процесса, запущенного вызовом метода `placeOrder`. На практике также можно было бы извлечь заказ, размещенный асинхронно, и проверить сохранение результатов в базе данных.

2.3. Использование CDI с компонентами EJB 3

Как отмечалось в главе 1, механизм CDI играет жизненно важную роль в обеспечении надежной поддержки внедрения зависимостей с помощью аннотаций во всех компонентах Java EE, включая EJB 3. В этом разделе мы покажем наиболее типичные способы использования механизма CDI с компонентами EJB 3 – как более надежной замены компонентов, управляемых механизмом JSF, и позволяющего добавлять в EJB компоненты, расположенные на других уровнях, отличных от уровня прикладной логики и не использующие службы EJB непосредственно.

2.3.1. Использование CDI с JSF 2 и EJB 3

Чтобы увидеть, как можно использовать CDI в качестве суперклея, соединяющего JSF и EJB 3, вернемся немного назад, к примеру реализации сеансового компонента без сохранения состояния `BidService`. Напомним, что `BidService` сохраняет ставки в базе данных. Очевидно, что метод `addBid` можно вызывать в странице, позволяющей клиенту сделать ставку в состязании за товар. Знакомые с особенностями сайтов-аукционов обнаружат их сходство со страницей, изображенной на рис. 2.4.

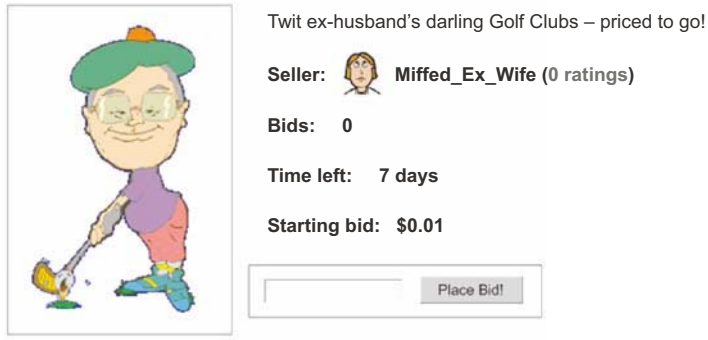

Рис. 2.4. Страница аукциона ActionBazaar

Большая часть страницы отведена под информацию о товаре, ставках, продавце и клиенте, например: название товара, его описание, наивысшая ставка, имя текущего клиента, информация о продавце и так далее. Наибольший интерес представляет текстовое поле ввода суммы ставки и кнопка размещения новой ставки. В листинге 2.4 показан код JSF 2, реализующий функциональность этих двух элементов страницы.

Листинг 2.4. JSF-страница добавления ставки

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets">
<h:body>
    ...
    <h:form id="bid-form">
       ...
       <!-- ❶ Выражения, осуществляющие связывание с компонентами,
            внедренными с помощью CDI -->
       <h:inputText id="input-bid" value="#{bid.amount}"/>
       ...
       <h:commandButton id="place-bid-button"
                        value="Place Bid!"
                        action="#{bidManager.placeBid}"/>
```

```
        ...
    </h:form>
    ...
</h:body>
</html>
```

Как можно видеть, оба элемента страницы, поле ввода `inputText` и кнопка `commandButton`, связаны с компонентами посредством выражений связывания ❶. Поле `inputText` связано с полем `amount` компонента `bid`. Это означает, что текстовое поле отображает значение связанного с ним поля компонента, а любое значение, введенное в текстовое поле, автоматически будет записано в поле компонента. Кнопка **Place Bid** (Сделать ставку) связана с методом `placeBid` компонента `bidManager`. Это означает, что связанный метод будет вызван автоматически в случае щелчка на кнопке. В листинге 2.5 проводится код, с которым связаны элементы JSF-страницы.

Листинг 2.5. Компонент BidManager

```
@Named // ❶ Присвоить имя компоненту BidManager при внедрении посредством CDI
@RequestScoped   // ❷ Область видимости компонента ограничена запросом
public class BidManager {
  @Inject        // ❸ Внедрение компонента BidService
  private BidService bidService;

  @Inject        // ❹ Внедрение текущего
  @LoggedIn      //    зарегистрированного пользователя
  private User user;

  @Inject        // ❺ Внедрение текущего
  @SelectedItem  //    выбранного товара
  private Item item;

  private Bid bid = Bid();

  @Produces      // ❻ Производит компонент,
  @Named         //    управляемый посредством CDI,
  @RequestScoped //    с именем bid
  public Bid getBid() {
      return bid;
  }

  public String placeBid() {
    bid.setBidder(user);
    bid.setItem(item);
    bidService.addBid(bid);

    return "bid_confirm.xhtml";
  }
}
```

Компонент `BidManager` не является компонентом EJB – это простой компонент, управляемый механизмом CDI. То есть, данному компоненту недоступны никакие

корпоративные службы EJB, такие как управление транзакциями, – только средства внедрения зависимостей, управления жизненным циклом и контекстом. Но в данном случае этого вполне достаточно, потому что `BidManager` выступает всего лишь в роли связующего звена между страницей JSF и уровнем служб EJB.

Сама реализация `BidManager` достаточно проста и понятна, даже несмотря на наличие разных аннотаций CDI. CDI-аннотация `@Named` ❶ присваивает компоненту `BidManager` имя при внедрении. По умолчанию компоненту присваивается имя, совпадающее с именем класса, в котором первый символ замещается символом нижнего регистра. В данном случае компонент `BidManager` получит имя `bidManager`. Такое именование компонентов необходимо для ссылки на них из JSF EL. Как было показано выше, метод `BidManager.placeBid` связывается с кнопкой **Place Bid** (Сделать ставку) посредством выражения связывания EL. CDI-аннотация `@RequestScoped` ❷ определяет область видимости компонента `BidManager`. Так как область видимости `BidManager` ограничивается рамками запроса, компонент будет создаваться при загрузке страницы JSF и уничтожаться после перехода на другую страницу.

В компонент `BidManager` внедряется несколько других компонентов, в виде зависимостей. Первый из них – сеансовый компонент без сохранения состояния `BidService` ❸, с которым мы уже познакомились. Внедрение компонентов, представляющих клиента ❹ и товар ❺ выглядят более интересными. Аннотации `@LoggedIn` и `@SelectedItem` – это CDI-квалификаторы, определяемые пользователем. Подробнее о квалификаторах CDI рассказывается в главе 12. А пока просто запомните, что квалификаторы – это метаданные, определяемые пользователем, используемые для определения конкретной характеристики внедряемого компонента. Например, в случае внедрения объекта, представляющего пользователя, в `BidManager` определяется, что компонент должен представлять зарегистрированного (logged-in) пользователя. Программный код предполагает, что CDI имеет ссылку на подходящий экземпляр класса `User` в доступной области видимости. Наиболее вероятно, что компонент `User`, соответствующий зарегистрировавшемуся пользователю, находится в области видимости сеанса и был помещен туда в ходе процедуры регистрации. Аналогично в реализации `BidManager` используется квалификатор `@SelectedItem`, указывающий, что он зависит от текущего товара, выбранного пользователем. Выбранный компонент `Item` был, вероятно, помещен в область видимости запроса, когда пользователь щелкнул ссылку для просмотра информации о товаре.

Также интерес для нас представляют аннотации `@Produces`, `@RequestScoped` и `@Named` ❻ перед методом `getBid`. Как можно догадаться, аннотация `@Produces` является ключевой. Она сообщает механизму CDI, что метод `getBid` возвращает объект `Bid`, которым он должен управлять. Аннотации `@Named` и `@RequestScoped` перед методом `getBid` сообщают механизму CDI, что возвращаемому объекту `Bid` должно быть присвоено имя `bid` и для него должна быть установлена область видимости запроса. Благодаря присвоению имени объекту `Bid`, на него можно сослаться из EL, как если бы он находился внутри страницы JSF. Мы еще вернемся к объекту `Bid` в следующих разделах, а пока просто имейте в виду, что вся информация о ставке

хранится в сущности JPA 2, включая величину ставки, установленную в текстовом поле на странице JSF. Так как `BidManager` создает и хранит ссылку на объект `Bid`, он автоматически получает доступ ко всем данным, введенным в странице JSF.

Когда в ответ на щелчок на кнопке в странице JSF вызывается метод `placeBid`, компонент `BidManager` использует всю имеющуюся у него информацию и заполняет объект ставки `Bid`. Затем с помощью сеансового компонента без сохранения состояния `BidService` сохраняет ставку в базе данных. В заключение пользователь переадресуется на страницу, где он сможет подтвердить ставку.

Это все, что мы на данный момент хотели рассказать об особенностях применения механизма CDI в компонентах EJB 3 уровня представления. Механизм CDI можно использовать в EJB 3 множеством других способов, в том числе и на уровне хранения данных, для реализации объектов DAO доступа к данным. Давайте посмотрим, как это делается.

2.3.2. Использование CDI с EJB 3 и JPA 2

В системах на основе Java EE 7, уровень DAO часто реализуется в виде компонентов EJB 3. В этом есть определенный смысл, если DAO отмечен атрибутом, требующим применения транзакций (подробнее об атрибутах управления транзакциями рассказывается в главе 6). Благодаря этому повышается надежность выполнения клиентских операций, так как объекты DAO часто используют ресурсы, требующие применения механизма транзакций, такие как базы данных. Но для маленьких приложений, где уровни служб и объектов DAO зачастую разрабатывается одними и теми же разработчиками, такая надежность нужна не всегда. Существует возможность разрабатывать объекты DAO, использующие простые компоненты, управляемые механизмом CDI, и внедрять их в компоненты EJB на уровне служб. Чтобы посмотреть, как может выглядеть такая реализация, давайте вернемся к сеансовому компоненту без сохранения состояния `BidService`, представленному в листинге 2.6.

Листинг 2.6. Сеансовый компонент без сохранения состояния BidService

```
@Stateless // Пометить POJO как сеансовый компонент без сохранения состояния
public class DefaultBidService implements BidService {
    @Inject // Внедрить объект DAO
    private BidDao bidDao;
    ...
    public void addBid(Bid bid) {
        bidDao.addBid(bid);
    }
    ...
}

@Local // Пометить интерфейс как локальный
public interface BidService {
    ...
    public void addBid(Bid bid);
    ...
}
```

Объект BidDao внедряемый в компонент EJB, является компонентом, который управляется механизмом CDI посредством интерфейса. Объект опирается на поддержку в EJB транзакций и многопоточных контекстов, и не требует наличия каких-либо служб, отличных от простой поддержки внедрения зависимостей, предлагаемой механизмом CDI. Любые другие службы, такие как поддержка безопасности и асинхронная обработка, также вероятно лучше использовать на уровне служб, а не на уровне хранения данных. В листинге 2.7 приводится реализация объекта DAO.

Листинг 2.7. Компонент BidDao, управляемый механизмом CDI

```
public class DefaultBidDao implements BidDao {
    @PersistenceContext  // Внедрить диспетчер сущностей JPA 2
    private EntityManager entityManager;
    ...
    public void addBid(Bid bid) {
        entityManager.persist(bid);
    }
    ...
}

public interface BidDao {
    ...
    public void addBid(Bid bid);
    ...
}
```

Объект доступа к данным очень прост. Некоторые из вас могут быть удивлены, что ни класс, ни интерфейс не снабжаются никакими аннотациями. Это вполне нормальная ситуация в мире CDI, потому что компоненты, управляемые механизмом CDI, в действительности являются самыми обычными Java-объектами. Когда CDI обнаруживает аннотацию @Inject в компоненте BidService, он отыскивает объект, реализующий интерфейс BidDao и внедряет его. Важно отметить, что объект BidDao является «безымянным» – он не отмечен аннотацией @Named. Ее отсутствие объясняется отсутствие ссылок на объект DAO по имени. Для объекта DAO также не определяется область видимости. Механизм CDI предполагает, что компонент принадлежит области видимости по умолчанию. Если не указано иное, CDI создает совершенно новый экземпляр компонента и внедряет его в зависящий компонент, действуя практически так же, как оператор «new». Отметьте также, что сам BidDao тоже может внедрять другие компоненты в свои переменные экземпляра. В примере выше CDI внедряет диспетчер сущностей JPA 2 в объект DAO, который используется для сохранения сущности Bid в базе данных. Подробнее о JPA 2 рассказывается в следующем разделе.

Особенности CDI, описанные в этом разделе – это лишь вершина айсберга. Механизм CDI обладает обширным массивом других функциональных возможностей, таких как стереотипы, события, интерцепторы и декораторы. С некоторыми из них мы познакомимся в главе 12. А пока обратим наше внимание на последний элемент мозаики EJB 3: JPA 2.

2.4. Использование JPA 2 с EJB 3

Механизм JPA 2 де-факто является основным решением по хранению данных на платформе Java EE и тесно связан с EJB 3. В Java EE 5, механизм JPA был частью EJB 3. Если у вас есть опыт использования Hibernate, TopLink или JDO, вы будете чувствовать себя как дома, используя JPA 2. Большинство из этих инструментов в настоящее время поддерживает стандарт JPA 2.

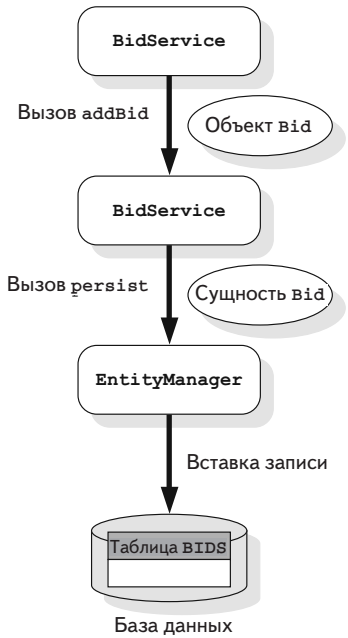

Рис. 2.5. Компонент BidService вызывает метод addBid компонента BidDao и передает ему объект Bid. Компонент BidDao вызывает метод persist компонента EntityManager для сохранения сущности Bid в базе данных. После подтверждения транзакции можно увидеть обнаружить появление соответствующей записи в таблице BIDS

Как вы вскоре увидите сами, интерфейс `EntityManager` определяет API для выполнения операций сохранения, а сущности JPA определяют порядок отображения данных в реляционную базу данных. Несмотря на то, что JPA скрывает значительную долю сложностей, связанных с сохранением данных, тем не менее, применение объектно-реляционных отображений является далеко не простой темой, и мы посвятим вопросам применения JPA целых три главы – 9, 10 и 11.

Практически на каждом этапе работы приложения ActionBazaar осуществляется сохранение информации в базе данных с применением механизма JPA 2. В настоящий момент обзор этих действий не является для нас ни необходимым, ни интересным. Поэтому мы лучше посмотрим, как выглядят взаимодействия с JPA 2, вновь вернувшись к реализации `BidDao`. Чтобы вспомнить, о чем идет речь, взгляните еще раз на рис. 2.5, где изображены различные компоненты, взаимодействующие друг с другом при создании новой ставки в приложении ActionBazaar. Сначала мы посмотрим, как сущность `Bid` отображается в базу данных, а затем подробно обсудим порядок использования диспетчера сущностей JPA 2 в компоненте `BidDao`.

2.4.1. Отображение сущностей JPA 2 в базу данных

К настоящему моменты вы уже видели, как используется объект `Bid` на разных уровнях приложения, и узнали, что этот объект является сущностью JPA 2. Мы еще не демонстрировали фактическую реализацию сущности `Bid`, и теперь самое время сделать это, чтобы было от чего отталкиваться при обсуждении особенностей отображения сущностей JPA 2 в базу данных. Итак, в листинге 2.8 приводится исходный код сущности `Bid`.

Листинг 2.8. Сущность Bid

```
@Entity                      // ❶ Пометить POJO как сущность
@Table(name="BIDS")          // ❷ Таблица для отображения
public class Bid {
    private Long id;
    private Item item;
    private Bidder bidder;
    private Double amount;

    @Id                      // ❸ Определить поле-идентификатор ID
    @GeneratedValue          // ❹ Значение поля генерируется автоматически
    @Column(name="BID_ID")   // ❺ Определить имя поля для отображения
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @ManyToOne               // ❻ Определить отношение "многие-к-одному"
    // ❼ Определить отображение отношения
    @JoinColumn(name="BIDDER_ID", referencedColumnName="USER_ID")
    public Bidder getBidder() {
        return bidder;
    }

    public void setBidder(Bidder bidder) {
        this.bidder = bidder;
    }

    @ManyToOne               // ❽ Определить отношение "многие-к-одному"
    // ❾ Определить отображение отношения
    @JoinColumn(name = "ITEM_ID", referencedColumnName = "ITEM_ID")
    public Item getItem() {
        return item;
    }

    public void setItem(Item item) {
        this.item = item;
    }

    @Column(name="BID_AMOUNT") // ❿ Определить имя поля для отображения
```

```
    public Double getAmount() {
        return amount;
    }

    public void setAmount(Double amount) {
        this.amount = amount;
    }
}
```

Возможно, взглянув на листинг 2.8, кто-то уже догадался, как в действительности осуществляется объектно-реляционное отображение в JPA 2, даже если он не знаком с инструментами ORM, такими как Hibernate. Вглядитесь в аннотации, отражающие реляционные понятия, такие как таблицы, столбцы (поля), внешние и первичные ключи.

Аннотация `@Entity` отмечает тот факт, что класс `Bid` представляет сущность JPA ❶. Аннотация `@Table` сообщает JPA, что сущность `Bid` отображается в таблицу `BIDS` ❷. Аналогично, аннотации `@Column` (❺ и ❿) определяют, в какие поля таблицы `BIDS` должны отображаться отмеченные ими свойства `Bid`. Аннотация `@Id` играет особую роль. Она отмечает свойство `Id` как первичный ключ сущности `Bid` ❸. Первичный ключ уникально идентифицирует экземпляр сущности, по аналогии с записями в базе данных. Аннотация `@GeneratedValue` ❹ указывает, что первичный ключ должен генерироваться автоматически, в момент сохранения сущности в базу данных. Сущность `Bid` связана отношениями с некоторыми другими сущностями JPA, храня прямые ссылки на объекты, такие как `Bidder` и `Item`. Механизм JPA поддерживает подобные косвенные отношения между объектами и обеспечивает их отображение в базу данных. В нашем примере именно это и реализуют аннотации `@ManyToOne` (❻ и ❽) и `@JoinColumn` (❼ и ❾). В обоих случаях аннотация `@ManyToOne` указывает на природу отношений между сущностями. В терминах реляционных баз данных это означает, что таблица `BIDS` хранит внешние ключи, связывающие ее с таблицами, где хранятся сущности `Item` и `Bidder`. Аннотация `@JoinColumn` определяет фактическую реализацию связи по внешнему ключу. В случае отношения «ставка-товар», внешний ключ хранится в таблице `BIDS` с именем `ITEM_ID`, и ссылается на ключ с именем `ITEM_ID`, который, вероятно, является первичным ключом таблицы `ITEMS`. В случае отношения «ставка-клиент», внешний ключ с именем `BIDDER_ID` ссылается на поле `USER_ID`. Теперь давайте посмотрим, как с помощью компонента `BidDao` выполняется сохранение сущности `Bid` в базе данных.

2.4.2. Использование EntityManager

Возможно вы обратили внимание, что сущность `Bid` не имеет метода для сохранения самой себя в базе данных. Эту работу берет на себя диспетчер сущностей JPA `EntityManager` который, опираясь на настройки объектно-реляционного отображения, предоставляет сущностям прикладной интерфейс с операциями сохранения. Класс `EntityManager` знает, как сохранять сущности в базе данных в виде реляционных записей, как читать данные из базы и превращать их в сущнос-

ти, как изменять данные, хранимые в базе, и как удалять эти данные. Диспетчер `EntityManager` имеет методы, соответствующие полному комплекту CRUD-операций (Create (Создать), Read (Читать), Update (Изменить), Delete (Удалить)), а также поддерживает надежный язык запросов Java Persistence Query Language (JPQL).

Давайте более внимательно посмотрим на листинг 2.9, где демонстрируется, как работают некоторые из операций, поддерживаемых диспетчером `EntityManager`.

Листинг 2.9. Использование операций диспетчера EntityManager в BidDao

```
public class DefaultBidDao implements BidDao {
    @PersistenceContext              // ❶ Внедряет EntityManager
    private EntityManager entityManager;

    public void addBid(Bid bid) {
        entityManager.persist(bid);  // ❷ Сохранить экземпляр сущности
    }

    public Bid getBid(Long id) {
        return entityManager.find(Bid.class, id); // Извлечь экземпляр
    }

    public void updateBid(Bid bid) {
        entityManager.merge(bid);    // Изменить экземпляр
    }

    public void deleteBid(Bid bid) {
        entityManager.remove(bid);   // Удалить экземпляр
    }
}
```

Настоящее волшебство здесь заключено в интерфейсе `EntityManager`. Интересно отметить, что интерфейс `EntityManager` играет роль переводчика между объектно-ориентированным и реляционным мирами. Диспетчер читает аннотации объектно-реляционного отображения, такие как `@Table` и `@Column`, указанные в исходном коде сущности `Bid`, и по ним определяет, как сохранять ее в базе данных. Диспетчер `EntityManager` внедряется в компонент `DefaultBidDao` посредством аннотации `@PersistenceContext` ❶.

Чтобы сохранить данные в базе, метод `addBid` вызывает метод `persist` диспетчера `EntityManager` ❷. К моменту, когда метод `persist` вернет управление, базе данных будет передана примерно следующая инструкция SQL, выполняющая добавление записи с информацией о ставке:

```
INSERT INTO BIDS (BID_ID, BIDDER_ID, BID_AMOUNT, ITEM_ID)
VALUES (52, 60, 200.50, 100)
```

Возможно вам будет интересно вернуться к листингу 2.8 и посмотреть, какие аннотации с настройками объектно-реляционного отображения в сущности `Bid` использует `EntityManager` для создания этой инструкции SQL. Напомним, что аннотация `@Table` указывает имя таблицы `BIDS`, куда должна быть добавлена запись,

а каждая из аннотаций `@Column` и `@JoinColumn` определяют имена полей в таблице `BIDS`, соответствующие полям сущности `Bid`. Например, свойство `Id` отображается в поле `BIDS.BID_ID`, свойство `amount` отображается в поле `BIDS.BID_AMOUNT`, и так далее. Как уже говорилось выше, аннотации `@Id` и `@GeneratedValue` создают в таблице `BIDS` поле первичного ключа `BID_ID`, значение для которого должно быть автоматически сгенерировано провайдером JPA перед выполнением инструкции `INSERT` (значение `52` в примере инструкции SQL выше). Именно для реализации такого процесса преобразования сущности в поля таблицы и задумывались механизмы объектно-реляционного отображения и JPA.

Подобным же образом первичный ключ используется методом `find` для извлечения сущности, методом `merge` – для изменения сущности, и метод `remove` – для ее удаления.

На этом мы заканчиваем краткое введение в Java Persistence API и эту обзорную главу. Теперь вы должны представлять, насколько простой, эффективной и надежной является EJB 3.

2.5. В заключение

Как уже отмечалось во введении, цель этой главы состояла не в том, чтобы дать вам «таблетку знания» EJB 3, а просто показать, чего можно ожидать от новой версии платформы Java Enterprise.

В этой главе было начато создание приложения ActionBazaar, главной темы этой книги. На примере сценария работы этого приложения мы показали вам некоторые функциональные возможности EJB 3, включая сеансовые компоненты с сохранением и без сохранения состояния, CDI, JSF 2 и JPA 2. Познакомили вас с некоторыми основными понятиями, такими как аннотации, внедрение зависимостей и объектно-реляционное отображение.

Мы попробовали использовать сеансовый компонент без сохранения состояния для реализации прикладной логики добавления новой ставки в аукционную систему. А затем показали сеансовый компонент с сохранением состояния, инкапсулирующий логику процедуры оформления заказа. Вы увидели, как можно тестировать компоненты EJB 3 с помощью JUnit, и как JSF 2, CDI и EJB 3 вместе обеспечивают бесшовные взаимодействия разных уровней приложения. В заключение мы исследовали порядок сохранения ставок в базе данных и использовали EntityManager API для управления сущностями в базе данных.

В следующей главе мы опустимся еще ниже и займемся более детальным изучением сеансовых компонентов.

Часть II

КОМПОНЕНТЫ EJB

В этой части книги вы узнаете, как использовать компоненты EJB для реализации прикладной логики. Сначала в главе 3 мы погрузимся в детали сеансовых компонентов и обозначим некоторые наиболее удачные приемы их использования. Затем в главе 4 будет дано краткое введение в особенности обмена сообщениями (включая приемы передачи и приема сообщений), механизм JMS и поближе познакомимся с компонентами MDB. В главе 5 мы перейдем к изучению более сложных тем, таких как контексты EJB, применение JNDI для поиска компонентов EJB и других ресурсов, ресурсы и механизм внедрения EJB, основы интерцепторов AOP и контейнеры клиентских приложений. В главе 6 мы обсудим поддержку транзакций и безопасности с точки зрения разработчика, включая предпосылки к использованию транзакций, а также порядок применения групп и ролей для обеспечения безопасности. В главе 7 мы расскажем об основах службы таймеров EJB Timer Service, где познакомим вас с разными типами таймеров. В главе 8 мы займемся экспортированием компонентов Enterprise Java Beans в виде веб-служб с применением SOAP и REST.

ГЛАВА 3. Реализация прикладной логики с помощью сеансовых компонентов

Эта глава охватывает следующие темы:

- сеансовые компоненты без сохранения состояния;
- сеансовые компоненты с сохранением состояния;
- компоненты-одиночки (singleton);
- асинхронные компоненты.

Прикладная логика составляет основу любого корпоративного приложения. В идеальном мире разработчики должны основное свое внимание уделять реализации именно прикладной логики, тогда как такие задачи, как представление, хранение и интеграция по сути являются задачами оформительскими. С этой точки зрения наиболее важной частью технологии EJB составляют сеансовые компоненты, потому что цель всей их жизни заключается в моделировании бизнес-процессов.

Если сравнить приложение с колесницей, несущей греко-римского воина в бой, тогда сеансовые компоненты можно сравнить с возницей. Сеансовые компоненты используют данные и системные ресурсы (колесница и лошади) для достижения целей пользователя (воина), применяя прикладную логику (навыки управления колесницей). По этой и множеству других причин сеансовые компоненты, особенно сеансовые компоненты без сохранения состояния, пользовали сумасшедшей популярностью в EJB 2, даже несмотря на все проблемы. Спецификация EJB 3 сделала компоненты этого типа более простым в использовании и добавила в них массу новых возможностей.

В главе 1 мы получили некоторое представление о сеансовых компонентах. В главе 2 были продемонстрированы простые примеры их использования. В этой главе мы обсудим сеансовые компоненты более подробно, сосредоточившись на

их целях, различиях между типами, особенностях их разработки, а также рассмотрим некоторые дополнительные особенности сеансовых компонентов, включая выполнение асинхронных операций и управление конкуренцией с применением компонентов-одиночек.

Эту главу мы начнем с исследования некоторых базовых понятий сеансовых компонентов, а затем перейдем к обсуждению некоторых фундаментальных характеристик. Потом рассмотрим каждый их тип – с сохранением состоянии, без сохранения состояния и одиночки (singleton) – и познакомимся с поддержкой асинхронных операций, доступной во всех разновидностях компонентов.

3.1. Знакомство с сеансовыми компонентами

Типичное корпоративное приложение реализует множество прикладных операций и процедур. Например, приложение ActionBazaar включает такие процедуры, как создание учетной записи пользователя, добавление товара в аукционный лист, добавление ставки, оформление заказа и многие другие. Логика работы каждой процедуры реализуется в виде отдельного сеансового компонента. Но, чтобы лучше понять суть сеансовых компонентов, сначала необходимо разобраться с понятием сеанса.

В основе сеансовых компонентов лежит идея обработки каждого запроса клиента в рамках отдельной логической процедуры, выполняемой в пределах сеанса. Так что же такое сеанс? Пример сеанса можно подсмотреть в Microsoft Remote Desktop. С помощью Remote Desktop можно устанавливать соединения с удаленными компьютерами и управлять ими. Вы получаете изображение рабочего стола на удаленном компьютере, как если бы сидели за его монитором. Вы можете использовать диски удаленного компьютера, запускать приложения и, если компьютер находится в отдельной сети, обращаться к внешним ресурсам, доступным только в этой сети. Завершив необходимые операции, вы отключаетесь, завершая тем самым сеанс связи. В этом случае ваш локальный компьютер является клиентом, а удаленный компьютер – сервером. Проще говоря, *сеанс* – это соединение между клиентом и сервером, которое длится конечный промежуток времени.

Не все сеансы равны. Продолжительность сеансов может отличаться: одни сеансы могут очень короткими, другие очень долгими. А теперь оставим пример с удаленным компьютером и посмотрим, как люди ведут беседы. Беседа двух человек является своеобразной формой сеанса. Короткая беседа может состоять из единственного вопроса и ответа на него, например: «Который час?», «Полдень.». Примером длинной беседы может служить разговор двух друзей, живо обсуждающих баскетбольный матч. Короткая беседа в нашем примере завершилась ответом о текущем времени. Эти две беседы можно классифицировать, как без сохранения состояния и с сохранением состояния, соответственно. Здесь прослеживается явная аналогия с соответствующими типами сеансовых компонентов, обсуждаемыми далее.

Сеансы без сохранения и с сохранением состояния можно увидеть в разработке ПО повсюду. В приложении ActionBazaar некоторые процедуры сохраняют свое промежуточное состояние, другие – нет. Например, регистрация нового клиента выполняется в несколько этапов, в ходе которых человек выбирает имя пользователя, пароль и создает учетную запись. Процедура добавления новой ставки, напротив, не требует сохранения состояния. Запрос на выполнение этой процедуры включает в себя идентификатор пользователя, номер товара и сумму ставки. Сервер отвечает на такой запрос либо признаком успеха, либо признаком ошибки. Все это выполняется за один этап.

Помимо сеансовых компонентов с сохранением и без сохранения состояния существует еще и третий тип – *компоненты-одиночки* (singleton). Компонент-одиночку можно сравнить с проводником в поезде. Проводник знает стоимость проезда, пункты остановок, время прибытия и график движения. Пассажиры могут задавать вопросы проводнику, но только по одному за раз. Проводник, в свою очередь, может проверять билеты и отвечать на вопросы. Обычно в вагоне имеется только один проводник и все пассажиры в этом вагоне могут обращаться к нему. Так выглядит аналогия компонента-одиночки. В приложении существует только один компонент-одиночка.

Независимо от типа, все сеансовые компоненты оптимально подходят для обращения к ним всех клиентов приложения. Клиентом может быть все, что угодно: компонент веб-приложения (сервлет, страница JSF и так далее), приложение командной строки, другое приложение EJB и приложение с графическим интерфейсом на основе JavaFX/Swing. Клиентом может быть даже приложение Microsoft .NET, пользующееся веб-службой, или приложение для iOS или Android. Сеансовые компоненты обладают чрезвычайной мощью и могут использоваться для создания масштабируемых серверных систем, обслуживающих всех этих клиентов.

Сейчас у многих из вас должен возникнуть вопрос: «Что же делает сеансовые компоненты такими особенными?». В конце концов, каждый из трех типов сеансовых компонентов выглядит достаточно простым, чтобы его можно было использовать без применения какого-либо контейнера, разве не так? Нет, не так. Понятие сеансового компонента проистекает непосредственно из технологии EJB. В главе 1 упоминалось, что помимо простого моделирования прикладных процедур, контейнер предоставляет сеансовым компонентам множество важных служб, таких как внедрение зависимостей, управление жизненным циклом, поддержка многопоточности, транзакции, обеспечение безопасности и поддержка пулов. Следующий раздел поможет вам прояснить, почему следует использовать сеансовые компоненты, и когда.

3.1.1. Когда следует использовать сеансовые компоненты

Сеансовые компоненты – это нечто большее, чем простая абстракция, позволяющая разделить прикладную логику на отдельные процедуры. Контейнер управляет сеансовыми компонентами и предоставляет им множество жизненно важных

служб. В число этих служб входят внедрение зависимостей, управление транзакциями, обеспечение безопасности, удаленные взаимодействия, отложенные вычисления (таймеры) и интерцепторы. При правильном использовании сеансовые компоненты образуют основу масштабируемых и надежных приложений. Безусловно, есть риск злоупотреблений сеансовыми компонентами – если все прикладные операции оформить в виде сеансовых компонентов, это повлечет за собой падение производительности из-за перегрузки контейнера. Впрочем, перегрузка контейнера сама по себе – не самое страшное. В отсутствие контейнера вам пришлось бы реализовать не только прикладную логику, но и все необходимые службы, предоставляемые контейнером, что наверняка еще больше ухудшило бы производительность и надежность. Если вы будете использовать службы EJB только для решения задач, для которых они предназначены, тогда нагрузка на контейнер будет не так уж и велика.

По умолчанию все сеансовые компоненты поддерживают транзакции и многопоточную модель выполнения. Кроме того, многие контейнеры организуют сеансовые компоненты без сохранения состояния в пулы для большей масштабируемости (подробнее эта технология будет рассматриваться в разделе 3.2). Поэтому прикладные компоненты, пользующиеся этими службами, должны быть сеансовыми компонентами. Первыми кандидатами на реализацию в таком качестве, являются компоненты, так или иначе использующие серверные ресурсы, такие как соединения с базами данных и механизм многопоточного. Сеансовые компоненты также могут служить механизмами доставки служб контейнера, таких как безопасность, отложенные вычисления, асинхронная обработка, удаленные взаимодействия и веб-службы. Компоненты, нуждающиеся в этих службах, должны быть сеансовыми компонентами (обычно эти службы применяются на уровне прикладной логики).

Знать, когда следует использовать сеансовые компоненты, важно, но не менее важно знать, когда их использовать не следует. Создание классов вспомогательных сеансовых компонентов не особенно полезно. Вспомогательный метод форматирования телефонных номеров или парсинга исходных данных, разделенных символами табуляции не получит никакой выгоды от служб, предлагаемых контейнером EJB. Вспомогательные методы не пользуются службами безопасности, транзакциями или механизмом удаленных взаимодействий. Создание классов вспомогательных сеансовых компонентов может увеличить нагрузку на контейнер, не давая при этом никаких выгод. Хотя EJB предоставляет возможность внедрения зависимостей, управления жизненным циклом и интерцепторы, все то же самое могут получить простые компоненты, управляемые механизмом CDI. Поэтому вспомогательные API должны по возможности использовать CDI, а не EJB. То же самое можно сказать о классах объектов DAO и репозиториев. Хотя все эти объекты пользуются транзакциями и поддержкой многопоточного выполнения, они не обязаны быть сеансовыми компонентами, потому что скорее всего будут использоваться посредством прикладного уровня EJB. Как результат, реализация объектов DAOs в виде сеансовых компонентов просто добавит лишнюю нагрузку на контейнер.

Как и в Java EE 6, вполне возможно использовать компоненты EJB непосредственно, в виде JSF-компонентов. Однако, хотя это и возможно, и даже полезно для быстрого создания прототипов, в общем случае такого подхода следует избегать. Сеансовые компоненты не должны использоваться для навигации по страницам или обрабатывать параметры запросов, поступающие в виде веб-форм. Все эти задачи с большим успехом можно решать с помощью компонентов JSF или простых компонентов, управляемых механизмом CDI. Включение этой логики в сеансовый компонент выталкивает его на уровень представления, а смешивание логики работы пользовательского интерфейса с прикладной логикой ухудшает читаемость кода и осложняет его сопровождение.

Примечание. *Мы будем использовать JDBC в наших примерах только ради простоты, так как мы еще ничего не рассказывали о EJB 3 Persistence API (JPA). Мы не собираемся исходить из предположения, что вы уже хорошо знаете механизм ORM. Кроме того, использование JDBC поможет нам показать, как осуществляется внедрение зависимостей для ресурсов и управление жизненным циклом сеансовых компонентов без сохранения состояния. Однако в общем случае следует избегать использования JDBC и отдавать предпочтение JPA, как более мощному и удобному инструменту.*

3.1.2. Состояние компонента и типы сеансовых компонентов

Как уже говорилось выше, состояние – это фундаментальная характеристика классификации сеансовых компонентов на три разновидности. Достаточно близкой аналогией управления состоянием сеансовых компонентов является цикл «запрос/ответ» работы веб-сервера по протоколу HTTP, не поддерживающему соединения и сохранение состояния. Веб-браузер запрашивает страницу, а сервер отправляет ее и закрывает соединение. Конкретное наполнение страницы может отличаться для разных запросов, но браузер не предполагает, что веб-сервер будет хранить какую-либо информацию о клиенте, запросе или сеансе. Однако это не означает, что веб-сервер не может хранить информацию о состоянии для своих внутренних нужд. Например, веб-сервер может поддерживать открытым соединение с удаленным файловым сервером, хранить в памяти кэш файлов, и так далее.

От сеансовых компонентов без сохранения состояния так же не ожидается, что они будут хранить информацию о состоянии клиента. Более того, клиент не может даже надеяться на взаимодействие с одним и тем же сеансовым компонентом в последовательности вызовов, но все операции с сеансовыми компонентами без сохранения состояния выполняются атомарно. Независимо от наличия поддержки пулов, клиенты почти гарантированно при каждом обращении будут взаимодействовать с другим сеансовым компонентом. Но даже при том, что сеансовый компонент не хранит информацию о каком-то конкретном клиенте, он может и обычно задействует ресурсы многократного использования, такие как соединения с базой данных. Сеансовые компоненты без сохранения состояния идеально подходят для реализации служб, имеющих атомарный характер и действующих по принципу «обработал и забыл». В эту категорию попадает огромное число служб

корпоративных приложений, поэтому наверняка большая часть ваших сеансовых компонентов будут без сохранения состояния. Если вернуться к нашему приложению ActionBazaar, класс `BidService` является первым кандидатом на реализацию в виде сеансового компонента без сохранения состояния. Как показано на рис. 3.1, он предоставляет API для добавления, получения и удаления ставок (все эти операции могут быть выполнены за единственный вызов метода).

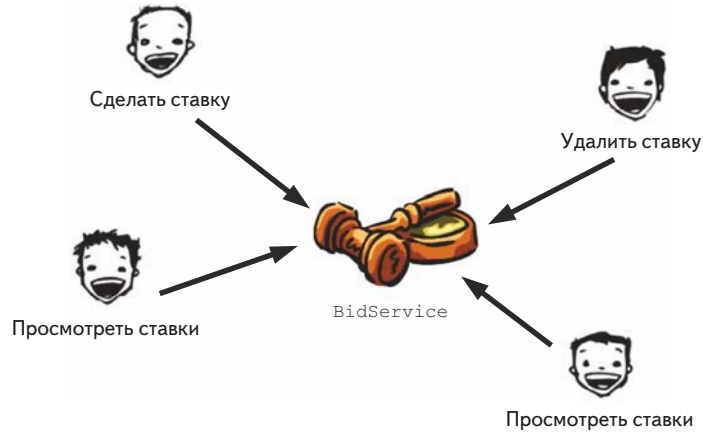

Рис. 3.1. Класс BidService не имеет состояния

Отличной аналогией сеансовых компонентов с сохранением состояния может служить онлайн-игра в пасьянс, изображенная на рис. 3.2. На протяжении игры сервер хранит состояние игрока и игры в своей памяти. Каждая игра фактически является сеансом, поддерживаемым с помощью постоянного соединения между клиентом (игроком) и сервером. Одновременно в игру может играть несколько игроков, но каждый будет играть в свою игру, не мешая другим, то есть каждый получит свой экземпляр игры с собственным состоянием.

Рис. 3.2. Игра в пасьянс может служить аналогом сеансов с сохранением состояния

Экземпляры сеансовых компонентов с сохранением состояния аналогично выделяются для каждого клиента в отдельности. Эти экземпляры создаются при первом обращении клиента, и недоступны другим клиентам, хранят информацию о состоянии в своих переменных экземпляра и уничтожаются, когда клиент закрывает сеанс. Сеансовые компоненты с сохранением состояния удобны для

моделирования процедур, выполняющихся в несколько этапов и необходимо сохранять состояние при переходах между этапами. В приложении ActionBazaar на роль сеансовых компонентов с сохранением состояния идеально подходят службы создания учетных записей продавцов и покупателей. Эти службы выполняют многоэтапные процедуры, состоящие из таких операций, как ввод имени учетной записи, биографической информации, географического местоположения, информации о порядке оплаты и так далее. Процедура может также быть отменена на любом этапе и тем самым привести к закрытию сеанса. В большинстве приложений службы с сохранением состояния используются довольно редко, а ситуации, когда сеансовые компоненты с сохранением состояния действительно необходимы, встречаются еще реже. Подробнее об этом мы поговорим в разделе 3.2.

Если онлайн-игра в пасьянс служит хорошей аналогией сеансовых компонентов с сохранением состояния, то многопользовательские онлайн-игры, особенно ролевые, с большим числом участников, как показано на рис. 3.3, могут служить превосходной аналогией компонентов-одиночек. Многопользовательские игры поддерживают одновременное участие в игре множества игроков и хранят состояние, общее для игры в целом. Существует только один экземпляр игры, к которой подключаются участники. По сути, все клиенты используют один и тот же сеанс.

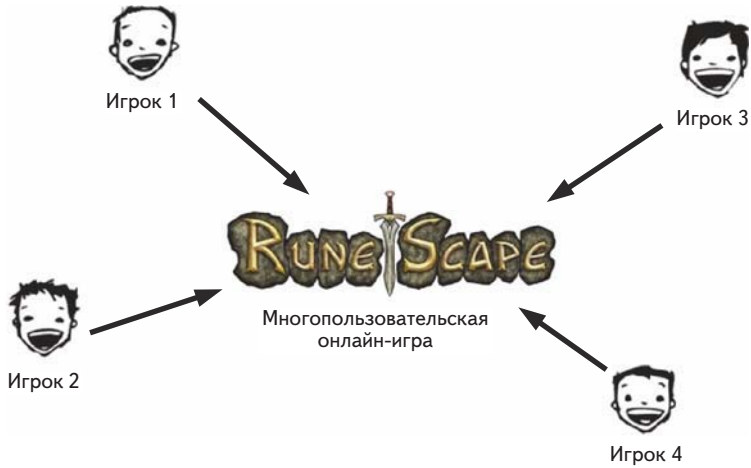

Рис. 3.3. Многопользовательская игра, как, например, RuneScape, поддерживающая общее состояние, как это делают компоненты-одиночки EJB

Аналогично существует только один экземпляр сеансового компонента-одиночки (singleton), который совместно используется всеми клиентами. Сама идея была выдвинута Бандой Четырех[1] в шаблоне проектирования «Одиночка». Только,

[1] Эрих Гамма (Erich Gamma), Ричард Хелм (Richard Helm), Ральф Джонсон (Ralph Johnson), Джон Влиссидс (John Vlissides) – коллектив авторов книги «Design Patterns: Elements of Reusable Object-Oriented Software», Addison-Wesley Professional, 1994, ISBN-10: 0-20163-361-2, ISBN-13: 978-0-20163-361-0 («Приемы объектно-ориентированного проектирования. Паттерны проектирования», Питер, 2001, ISBN: 5-272-00355-1). – *Прим. перев.*

вместо приватного конструктора и фабричного метода, создающего объект-одиночку, создание экземпляра осуществляется контейнером, и контейнер же гарантирует наличие единственного экземпляра. Контейнер также помогает обслуживать параллельные запросы к компоненту, поступающие от множества клиентов. Все клиенты совместно используют одно и то же внутреннее состояние компонента-одиночки, а компонент-одиночка, как предполагается, сохраняет и передает изменения всем остальным клиентам. Компоненты-одиночки предназначены для моделирования служб, использующих общее состояние, которое хранится в кэше. В приложении ActionBazaar на роль компонента-одиночки могла бы претендовать служба рассылки предупреждений, таких как предупреждения о мошенничестве. Компоненты-одиночки относительно редко используются в корпоративных приложениях, но их значение трудно переоценить, когда они действительно нужны.

Теперь, когда вы получили представление о состояниях и типах сеансовых компонентов, самое время поближе познакомится с каждым из типов. Так как на практике чаще используюися сеансовые компоненты без сохранения состояния, начнем с них.

3.2. Сеансовые компоненты без сохранения состояния

Как уже отмечалось, сеансовые компоненты без сохранения состояния не поддерживают какую-либо информацию о состоянии беседы, а задача, стоящая перед службой, должна быть решена за один вызов метода. Это, конечно же, не говорит о том, что сеансовые компоненты без сохранения состояния должны состоять только из одного метода. Напротив, такие компоненты обычно реализуют несколько родственных служб. Среди сеансовых компонентов, компоненты без сохранения состояния имеют самую высокую производительность.

В этом разделе вы узнаете больше о приемах разработки сеансовых компонентов без сохранения состояния. Примете участие в реализации некоторых ключевых элементов приложения ActionBazaar с их использованием. Это поможет вам лучше запомнить ключевые особенности сеансовых компонентов без сохранения состояния. Здесь вы также увидите некоторый практический код, узнаете, как пользоваться аннотацией `@Stateless`, а также познакомитесь с разными типами прикладных интерфейсов и механизмом управления жизненным циклом компонентов.

3.2.1. Когда следует использовать сеансовые компоненты без сохранения состояния

Очевидным случаем использования сеансовых компонентов без сохранения состояния являются прикладные службы, выполняемые за одно действие. Есть также несколько других сценариев, где имеет смысл применять такие компоненты. Часто на уровне хранения данных не требуется использовать компоненты EJB. Но иногда на этом уровне может быть желательно использовать поддержку пулов, без-

опасности или транзакций. Как будет рассказываться в следующем разделе, пулы являются важным фактором масштабируемости и могут использоваться не только службами прикладного уровня, но и компонентами на уровне хранения. Для многих приложений, создаваемых с применением приемов быстрой разработки (Rapid Application Development, RAD), необходимость в уровне служб не является такой насущной, потому что уровень пользовательского интерфейса (уровень представления) имеет непосредственный доступ к хранилищам данных. В этом случае объекты доступа к данным могут быть оформлены как сеансовые компоненты без сохранения состояния, чтобы получить преимущества поддержки многопоточного выполнения и транзакций. Как также упоминалось выше, сеансовые компоненты без сохранения состояния могут использоваться непосредственно, в качестве JSF-компонентов с помощью CDI.

Подобно другим компонентам EJB, сеансовые компоненты без сохранения состояния должны использоваться экономно. Например, бессмысленно использовать сеансовые компоненты для реализации утилит. Кроме того, не используйте компоненты без сохранения состояния там, где необходима поддержка сохранения состояния. Теоретически возможно имитировать поведение компонентов-одиночек с помощью статических полей в сеансовых компонентах без сохранения состояния. Но такой подход имеет две проблемы. Во-первых, даже при том, что отдельные компоненты без сохранения состояния поддерживают работу в многопоточной среде, наличие статических полей делает такое их применение небезопасным, потому что компоненты без сохранения состояния в этой ситуации могут использоваться множеством клиентов одновременно. Компоненты-одиночки, напротив, поддерживают механизмы управления одновременным доступом и потому могут безопасно использоваться несколькими клиентами сразу. Вторая проблема возникает в кластерном окружении. В кластере, сеансовый компонент без сохранения состояния на разных серверах может иметь разные значения статических полей.

3.2.2. Организация компонентов в пулы

Механизм организации пулов в EJB достаточно сложен, плохо описан в документации и вообще относится к малопонятным темам, поэтому он заслуживает некоторых пояснений с нашей стороны, и особенно это относится к организации пулов сеансовых компонентов без сохранения состояния. Основная идея состоит в том, чтобы повторно использовать предопределенное число экземпляров компонента для обслуживания входящих запросов. Когда поступает очередной запрос к компоненту, контейнер создает его. Когда выполнение метода компонента завершается, компонент помещается в пул. То есть, компонент может либо обрабатывать запрос, либо находиться в режиме ожидания. Пулы имеют верхнюю границу. Если число параллельных запросов к сеансовому компоненту без сохранения состояния превысит размер пула, приложение оказывается не в состоянии обработать их немедленно и помещает запросы в очередь. Как только в пуле появляется свободный экземпляр, ему передается запрос из очереди. Ни один экземпляр не может

находиться в режиме ожидания «вечно» – по истечении некоторого периода бездействия они автоматически удаляются. Пулы могут также иметь нижнюю границу, определяющую минимальное число экземпляров.

Поддержка пулов дает несколько важных преимуществ. Наличие минимального числа компонентов в пуле гарантирует наличие объектов, готовых к обработке запроса, по его прибытии. Это уменьшает общее время обслуживания запроса за счет устранения задержек, вызванных необходимостью создания компонентов. Повторное использование экземпляров компонентов, когда это возможно, вместо легкомысленного их уничтожения, позволяет уменьшить число операций по созданию и уничтожению объектов в приложении, экономя тем самым время, затрачиваемое на сборку мусора. Наконец, при определении верхней границы, пул начинает действовать как эффективный механизм ограничения полосы пропускания. Без такого механизма система может оказаться перегруженной в случае внезапного взрывного увеличения числа конкурирующих запросов. Регулирование полосы пропускания гарантирует предсказуемость поведения сервера даже при очень тяжелых нагрузках. Динамические пулы, регулирование полосы пропускания и очереди запросов являются важнейшими элементами событийно-ориентированной ступенчатой архитектуры (Staged Event-Driven Architecture, SEDA), широко используемой для организации современных, надежных, масштабируемых интернет-служб.

Поддержка пулов не является стандартной особенностью, утвержденной в спецификации EJB. Как и поддержка кластеров, поддержка пулов остается необязательной возможностью, но поддерживаемой большинством серверов приложений. Контейнеры состязаются между собой в масштабируемости и других особенностях, положительно влияющих на производительность, поэтому почти все контейнеры предоставляют те или иные параметры настройки поддержки пулов и их значения по умолчанию. Например, ниже перечислены некоторые настройки механизма пулов, доступные в GlassFish.

- *Начальный и минимальный размеры пула* – минимальное число компонентов в пуле. Значение по умолчанию: 0.
- *Максимальный размер пула* – максимальное число компонентов, которое может присутствовать в пуле. Значение по умолчанию: 32.
- *Квант изменения размера пула* – число компонентов, удаляемых из пула при его простое. Значение по умолчанию: 8.
- *Максимальное время простоя пула* – максимальное число секунд, которое компонент может находиться в состоянии ожидания, не участвуя в обработке запросов, прежде чем он будет передан механизму сборки мусора. Значение по умолчанию: 600.

Это лишь часть настроек механизма пулов – сервер поддерживает еще множество параметров. Каждое приложение индивидуально и очень важно иметь возможность настройки пулов под конкретные потребности.

В современных виртуальных машинах JVM конструирование объектов является очень недорогой операцией. Большинство современных реализаций JVM так-

же поддерживают сборку мусора по поколениям, что увеличивает эффективность уничтожения короткоживущих объектов. Сборка мусора по поколениям используется по умолчанию в HotSpot VM. Все это означает, что обычно не требуется указывать минимальный размер пула. Нет ничего страшного в создании экземпляров «на лету» и их уничтожении по истечении некоторого времени бездействия (когда компонент не будет задействован повторно для обработки очередного запроса). Именно поэтому минимальный размер пула в современных серверах приложений, таких как GlassFish, WebLogic 12c and JBoss 7 по умолчанию устанавливается равным нулю. Но, если в вашей реализации JVM не используется или не поддерживается сборка мусора по поколениям, вы определенно должны настроить минимальный размер пула (например, чтобы уменьшить нагрузку на сборщика мусора или уменьшить паузы, вызванные его работой). В таких ситуациях кэширование и повторное использование объектов может способствовать существенному ускорению сборки мусора. Пулы небольших размеров следует также использовать для тяжеловесных объектов, требующих большого времени на создание или использующих ресурсы, которые не могут храниться в пулах, такие как TCP-соединения.

Настройка максимального размера пула практически всегда служит хорошей защитой от ситуации нехватки ресурсов, вызванной внезапным появлением большого числа запросов. Обычно значение максимального размера пула устанавливают в диапазоне от 10 до 30. Как правило, максимальный размер пула следует устанавливать равным наибольшему ожидаемому числу одновременно обслуживаемых пользователей при нормальном функционировании. Слишком маленькое значение может сделать приложение неотзывчивым, и запросы слишком долго будут ждать обработки в очереди, тогда как слишком большое значение влечет за собой риск исчерпания ресурсов при тяжелых нагрузках. (Аналогично следует настроить размер пула базы данных, в соответствии с ожидаемым числом обслуживаемых пользователей или пропускную способность сервера баз данных.) Быстрые службы должны иметь более низкий верхний предел пула, а медленные службы – более высокий. Если ваш сервер имеет по-настоящему высокую пропускную способность и вам требуется обеспечить одновременное обслуживание очень большого числа пользователей, можно попробовать вообще снять верхнее ограничение на размеры пулов.

Очень важно правильно настроить пулы для своих конкретных условий. Большинство серверов приложений позволяют отслеживать состояние пулов компонентов во время выполнения, чтобы помочь вам в выборе настроек. Теперь, когда вы узнали, когда следует использовать сеансовые компоненты без сохранения состояния и как настраивать их пулы, давайте посмотрим некоторые примеры и исследуем их механику.

3.2.3. Пример BidService

Оформление ставки является очень важной функциональной особенностью приложения ActionBazaar. Пользователи могут просматривать текущие ставки и де-

лать свои. Администраторы ActionBazaar и сотрудники отдела обслуживания клиентов могут просматривать и удалять ставки, в зависимости от обстоятельств. Все эти действия схематически изображены на рис. 3.4.

Рис. 3.4. Некоторые операции со ставками, поддерживаемыми приложением ActionBazaar. Клиенты могут просматривать текущие ставки и делать свои, а администраторы могут удалять ставки при необходимости. Все эти операции могут быть реализованы в одном сеансовом компоненте без сохранения состояния

Так как все эти операции, связанные со ставками, очень просты и выполняются за одно обращение, для их реализации можно использовать сеансовый компонент без сохранения состояния. В листинге 3.1 представлен компонент `DefaultBidService` с методами, осуществляющими добавление, извлечение и удаление ставок. По сути, это расширенная версия простого компонента `PlaceBid`, который мы видели ранее. Полный код примера доступен по адресу: http://code.google.com/p/action-bazaar/.

> **Примечание.** *Мы будем использовать JDBC в наших примерах только ради простоты, так как мы еще ничего не рассказывали о JPA. Мы не собираемся исходить из предположения, что вы уже хорошо знаете механизм ORM. Кроме того, использование JDBC поможет нам показать, как осуществляется внедрение зависимостей для ресурсов и управление жизненным циклом сеансовых компонентов без сохранения состояния. Однако в общем случае следует избегать использования JDBC и отдавать предпочтение JPA, как более мощному и удобному инструменту.*

Листинг 3.1. Пример сеансового компонента без сохранения состояния

```
@Stateless(name = "BidService") // ❶ Пометить как сеансовый компонент
                                //    без сохранения состояния
public class DefaultBidService implements BidService {
```

```
  private Connection connection;

  @Resource(name = "jdbc/ActionBazaarDB") // ❷ Внедрить источник данных
  private DataSource dataSource;

  @PostConstruct                  // ❸ Вызывать по событию PostConstruct
  public void initialize() {
    try {
      connection = dataSource.getConnection();
    } catch (SQLException sqle) {
      sqle.printStackTrace();
    }
  }

  public void addBid(Bid bid) {
    Long bidId = getBidId();
    try {
      Statement statement = connection.createStatement();
      statement.execute("INSERT INTO BIDS "
        + " (BID_ID, BIDDER, ITEM_ID, AMOUNT) VALUES( "
        + bidId
        + ", "
        + bid.getBidder().getUserId()
        + ", "
        + bid.getItem().getItemId()
        + ", "
        + bid.getBidPrice() + ")");
    } catch (Exception sqle) {
      sqle.printStackTrace();
    }
  }

  private Long getBidId() {
    ...Код, генерирующий уникальный ключ...
  }

  public void cancelBid(Bid bid) {
    ...
  }

  public List<Bid> getBids(Item item) {
    ...
  }

  @PreDestroy                     // ❹ Вызывать по событию PreDestroy
  public void cleanup() {
    try {
      connection.close();
      connection = null;
    } catch (SQLException sqle) {
      sqle.printStackTrace();
    }
  }
}
...
```

```
@Remote                              // ❺ Интерфейс удаленных взаимодействий
public interface BidService {
  void addBid(Bid bid);
  void cancelBid(Bid bid);
  List<Bid> getBids(Item item);
}
```

Как вы уже знаете, аннотация `@Stateless` превращает POJO в сеансовый компонент без сохранения состояния ❶. Класс `DefaultBidService` реализует интерфейс `BidService`, помеченный аннотацией `@Remote` ❺. Аннотация `@Resource` используется для внедрения источника данных JDBC ❷. Аннотации `@PostConstruct` ❸ и `@PreDestroy` ❹ отмечают методы обратного вызова, которые управляют JDBC-соединением с базой данных, полученным в результате внедрения источника данных. На случай, если потребуется обслуживание удаленных клиентов, с помощью аннотации `@Remote` ❺ определяется интерфейс удаленных взаимодействий. В следующем разделе мы приступим к исследованию особенностей сеансовых компонентов EJB без сохранения состояния на примере этого кода и начнем с аннотации `@Stateless`.

3.2.4. Применение аннотации @Stateless

Аннотация `@Stateless` превращает простой Java-объект `DefaultBidService` в сеансовый компонент без сохранения состояния. Хотите верьте, хотите нет, но кроме того, что она сообщает контейнеру о назначении POJO, эта аннотация ничего больше не делает. Аннотация `@Stateless` имеет следующее определение:

```
@Target(value = {ElementType.TYPE})
@Retention(value = RetentionPolicy.RUNTIME)
public @interface Stateless {
    public String name() default "";
    public String mappedName() default "";
    public String description() default "";
}
```

Параметр `name` определяет имя компонента. Контейнеры используют этот параметр для включения компонента EJB в глобальное дерево JNDI. JNDI по сути является реестром, ресурсов, управляемым сервером приложений. Все компоненты EJB автоматически включаются в JNDI после обнаружения их контейнером. Подробнее об именовании компонентов EJB и дереве JNDI мы поговорим в главе 5. С параметром `name` мы еще раз встретимся в главе 14, при обсуждении развертывания дескрипторов. В листинге 3.1 имя компонента (параметр `name`) определено как `BidService`. Из объявления аннотации видно, что параметр `name` является необязательным. Вы можете опустить его, сократив код до:

```
@Stateless
public class DefaultBidService implements BidService {
```

Если параметр `name` отсутствует, контейнер присвоит компоненту неквалифицированное имя класса. В данном случае контейнер присвоил бы имя

DefaultBidService. Поле mappedName используется некоторыми производителями как имя компонента. Вы также можете указать его при определении своих компонентов EJB – некоторые контейнеры используют это имя для включения компонента в JNDI. Но в общем случае использовать поле mappedName нежелательно.

3.2.5. Прикладные интерфейсы компонентов

Клиенты могут использовать сеансовые компоненты тремя разными способами. Первый: через локальный интерфейс, внутри той же виртуальной машины JVM. Второй: через удаленный интерфейс с использованием механизма RMI. Третий: посредством веб-службы SOAP или REST. К одному и тому же компоненту можно обратиться любым из этих способов.

Поддержка этих трех методов доступа к компоненту включается с помощью аннотаций. Каждая аннотация должна быть помещена перед интерфейсом, реализуемым компонентом. Рассмотрим эти аннотации поближе.

Локальный интерфейс

Локальный интерфейс предназначен для использования клиентами, находящимися в том же экземпляре контейнера (JVM). Определение локального прикладного интерфейса осуществляется с помощью аннотации @Local. Ниже приводится одно из возможных определений локального интерфейса для класса DefaultBidService из листинга 3.1:

```
@Local
public interface BidLocalService {
    void addBid(Bid bid);
    void cancelBid(Bid bid);
    List<Bid> getBids(Item item);
}
```

Локальные интерфейсы являются самыми простыми в определении и использовании. Они также являются наиболее распространенным типом интерфейсов EJB и типом по умолчанию. То есть, вы можете опустить аннотацию @Local, но интерфейс все равно будет интерпретироваться как локальный. Отметьте, что в EJB 3.1 больше не требуется определять локальные интерфейсы. Более того, интерфейсы вообще можно не определять; в этом случае вы получите доступ к компоненту непосредственно через реализацию класса. Например, компонент BidService можно переопределить, убрав все интерфейсы, как показано ниже:

```
@Stateless
public class BidService {
  ...
  @PostConstruct
  public void initialize() {
    ...
  }

  public void addBid(Bid bid) {
```

```
    ...
  }

  public void cancelBid(Bid bid) {
    ...
  }

  public List<Bid> getBids(Item item) {
    ...
  }

  @PreDestroy
  public void cleanup() {
    ...
  }
}
```

В этом случае доступ к службе ставок можно получить путем внедрения экземпляра `BidService` по конкретному классу, а не через интерфейс. Обратите внимание, что если локальный интерфейс не объявлен явно, доступными будут все методы компонента.

Удаленный интерфейс

Клиенты, находящиеся за пределами виртуальной машины JVM, где выполняется экземпляр контейнера, должны использовать некоторую разновидность *удаленного интерфейса*. Если клиент так же написан на Java, наиболее логичным выглядит выбор RMI – удаленного интерфейса доступа к компонентам EJB. RMI – высокоэффективный API, основанный на использовании протокола TCP/IP и осуществляющий обмен данными в компактном двоичном виде, автоматизирующий большую часть операций, которые необходимо выполнить для вызова метода Java-объекта по сети. В EJB 3 имеется поддержка доступа к компонентам через механизм RMI, реализованная в виде аннотации `@Remote`. Прикладной интерфейс `BidService` в нашем примере использует эту аннотацию, чтобы сделать компонент доступным удаленно:

```
@Remote
public interface BidService extends Remote {
  ...
}
```

Удаленный прикладной интерфейс может наследовать `java.rmi.Remote`, как это сделано в нашем примере, хотя, строго говоря, это не обязательно. От удаленных прикладных методов не требуется, чтобы они возбуждали исключение `java.rmi.RemoteException`, если только прикладной интерфейс не наследует `java.rmi.Remote`. Удаленные прикладные интерфейсы должны выполнять одно обязательное требование: все параметры и возвращаемые значения методов удаленного интерфейса *должны* поддерживать интерфейс `Serializable`. Это обусловлено тем, что с помощью RMI по сети могут передаваться только объекты, реализующие интерфейс `Serializable`.

Пример кода для этой главы включает поддержку удаленных клиентов. В состав примеров входит простой интерфейс JavaFX, вызывающий методы удаленного компонента.

Интерфейс конечной точки веб-службы

Помимо локального и удаленного интерфейсов, сеансовые компоненты могут также поддерживать интерфейсы веб-служб. В Java EE 7 поддерживается две различные технологии организации веб-служб: SOAP, посредством JAX-WS, и REST, посредством JAX-RS. В обоих случаях соответствующие аннотации могут быть помещены либо перед отдельным интерфейсом, либо непосредственно перед классом компонента.

Ниже приводится версия REST-службы для работы со ставками. Класс определяет сеансовый компонент без сохранения состояния и делегирует выполнение фактических операций внедренному экземпляру `BidService`:

```
@Stateless
// Пометить сеансовый компонент без сохранения состояния как
// экспортируемый через REST с корневым URI /bid
@Path("/bid")
public class BidRestService {
  @EJB
  private BidService bidService;
  ...
  // При получении запроса HTTP GET будет вызван метод getBid
  @GET
  // Метод getBid должен вернуть объект ставки в формате XML
  @Produces("text/xml")
  // Идентификатор ставки извлекается из параметра HTTP-запроса
  public Bid getBid(@QueryParam("id") Long id) {
    return bidService.getBid(id);
  }
  ...
  // При получении запроса HTTP DELETE будет вызван метод deleteBid
  @DELETE
  public void deleteBid(@QueryParam("id") Long id) {
    Bid bid = bidService.getBid(id);
    bidService.deleteBid(bid);
  }
}
```

Веб-служба REST вызывается обращением по адресу http://<hostname>:<port>/actionbazaar/rest/bid. Параметры запроса передаются по мере необходимости. Например, адрес URL для извлечения информации о ставке имеет вид: http://<hostname>:<port>/actionbazaar/rest/bid?id=1010.

Как будет показано в главе 8, необходимость создания адаптера REST обусловлена тем, что методы HTTP плохо отображаются в вызовы методов службы на Java. Технология SOAP, с другой стороны, лучше поддерживает отображение запросов в вызовы методов, поэтому существующие компоненты EJB обычно легко могут экспортироваться посредством JAX WS.

Для организации традиционных веб-служб на основе SOAP, в состав Java EE 6 входит JAX-WS. Как и JAX-RS, включение механизма JAX-WS выполняется с помощью аннотаций. Чтобы экспортировать существующий компонент в виде веб-службы, достаточно просто создать новый экземпляр и добавить аннотацию `@javax.jws.WebService`. Следующий фрагмент демонстрирует это на примере `BidService`:

```
@WebService
public interface BidSoapService {
  List<Bid> getBids(Item item);
}
```

Как видно из этого фрагмента, имеется возможность выборочно скрывать методы, которые не должны быть доступны в веб-службе. Данный интерфейс исключает методы `cancelBid` и `addBid`; данные методы не будут доступны через веб-службу SOAP. Они все еще доступны через локальный и удаленный интерфейсы. Аннотация `@WebService` не накладывает никаких особых ограничений ни на интерфейс, ни на реализацию компонента. Более подробно веб-службы EJB будут обсуждаться в главе 8.

3.2.6. События жизненного цикла

Сеансовые компоненты без сохранения состояния имеют очень простой жизненный цикл – они либо существуют, либо нет. Этот жизненный цикл изображен на рис. 3.5. Сразу после создания компонент помещается в пул, в ожидании появления запросов от клиента. В конечном итоге компонент уничтожается, либо когда нагрузка на сервер уменьшается, либо когда завершается само приложение. При этом контейнер выполняет следующие действия:

1. Создает экземпляр компонента вызовом конструктора по умолчанию.
2. Внедряет ресурсы, такие как провайдеры JPA и соединения с базами данных.
3. Помещает экземпляр компонента в управляемый пул (если поддерживается).
4. Когда от клиента поступает запрос, извлекает простаивающий компонент из пула. В этот момент контейнер может создавать дополнительные компоненты для обработки запросов. Если контейнер не поддерживает пулы, он просто создает экземпляр компонента.
5. Вызывает запрошенный метод посредством прикладного интерфейса.
6. Когда метод вернет управление, компонент помещается обратно в пул (если поддерживается). Если контейнер не поддерживает пулы, он просто уничтожает компонент.
7. При необходимости контейнер удаляет из пула ненужные компоненты.

Как отмечалось выше, сеансовые компоненты без сохранения состояния обычно отличаются очень высокой производительностью. При использовании пулов, относительно небольшое число экземпляров компонента может своевременно обрабатывать значительное количество параллельных запросов.

Рис. 3.5. Курица или яйцо – в течение жизни сеансовый компонент без сохранения состояния проходит три стадии: отсутствие, ожидание и обработка запроса. Как результат, компонент может реагировать только на два события, соответствующие моментам создания и уничтожения компонента

При более тщательном исследовании можно заметить, что организация жизненного цикла компонентов гарантирует доступность каждого экземпляра только для одного запроса в каждый момент времени. Именно поэтому сеансовые компоненты без сохранения состояния полностью поддерживают многопоточную модель выполнения и вам не нужно беспокоиться о синхронизации, даже при работе в многопоточном серверном окружении!

Сеансовые компоненты без сохранения состояния поддерживают два метода обработки событий жизненного цикла с помощью следующих аннотаций:

- `@PostConstruct` – метод, отмеченный этой аннотацией, будет вызываться сразу после создания экземпляра и внедрения всех ресурсов;
- `@PreDestroy` – метод, отмеченный этой аннотацией, будет вызываться непосредственно перед уничтожением экземпляра.

Если потребуется, этими аннотациями можно пометить несколько методов. В листинге 3.1 используются обе аннотации, `@PostConstruct` и `@PreDestroy`, где они отмечают методы инициализации и освобождения ресурсов, соответственно. Обычно эти методы используются для инициализации и освобождения внедренных ресурсов, используемых прикладными методами. Именно этот подход и реализован в листинге 3.1 – методы открывают и закрывают соединение с базой данных, используя внедренный источник данных JDBC. У кого-то может возникнуть вопрос: почему бы не использовать для этих целей конструктор и метод `finalize`. Дело в том, что когда вызывается конструктор, контейнер еще не выполнил внед-

рение ресурсов, поэтому все ссылки на них будут пустыми. Что же касается метода `finalize`, его рекомендуется использовать только для освобождения низкоуровневых ресурсов, таких как ссылки JNI. И уж точно не следует использовать его для закрытия соединений с базами данных. Метод `finalize` вызывается намного позже того, как компонент покинет пул, уже в процессе сборки мусора.

Напомним, что метод `addBid` в листинге 3.1 добавляет новую ставку, сделанную пользователем. Метод создает инструкцию `java.sql.Statement`, используя соединение JDBC, и с ее помощью добавляет запись в таблицу `BIDS`. Объект соединения JDBC, используемый для создания инструкции, является классическим примером тяжеловесного ресурса. Он имеет дорогостоящую процедуру открытия и по возможности должен использоваться для выполнения не одного запроса. Так как он удерживает множество собственных низкоуровневых ресурсов, важно своевременно закрывать его.

Источник данных JDBC, из которого создается соединение в листинге 3.1, внедряется с помощью аннотации `@Resource`. Подробнее о внедрении ресурсов с помощью аннотации `@Resource` рассказывается в главе 5, а пока это все, что вам нужно знать. Давайте поближе рассмотрим методы обратного вызова из листинга 3.1.

Событие PostConstruct

После внедрения всех ресурсов, контейнер проверяет наличие в классе компонента методов с аннотацией `@PostConstruct`. Если такие методы имеются, он вызывает их перед тем, как передать экземпляр компонента в работу. В данном случае аннотацией `@PostConstruct` отмечен метод `initialize`:

```
@PostConstruct
public void initialize () {
  ...
  connection = dataSource.getConnection();
  ...
}
```

В этом методе из внедренного источника данных создается экземпляр `java.sql.Connection` и сохраняется в переменной экземпляра, которая используется методом `addBid`.

Событие PreDestroy

В некоторый момент контейнер может решить, что экземпляр компонента должен быть уничтожен. Метод обратного вызова с аннотацией `@PreDestroy` дает компоненту возможность освободить занятые им ресурсы перед уничтожением. В методе `cleanup` осуществляется освобождение ресурса соединения с базой данных перед уничтожения компонента контейнером:

```
@PreDestroy
public void cleanup() {
  ...
  connection.close();
```

```
  connection = null;
  ...
}
```

3.2.7. Эффективное использование сеансовых компонентов без сохранения состояния

Подобно другим технологиям, EJB и сеансовые компоненты без сохранения состояния имеют наиболее удачные приемы, которых следует придерживаться, и неудачные, которых желательно избегать. В этом разделе мы расскажем о некоторых таких приемах.

Настраивайте пул под свои нужды

Как говорилось в разделе 3.2.2, правильная настройка пула играет важную роль для извлечения максимальной выгоды из сеансовых компонентов без сохранения состояния. В общем случае большинство серверов приложений имеют вполне приемлемые настройки по умолчанию, но их корректировка действительно может существенно увеличить производительность приложения, особенно при больших нагрузках. Не следует также пренебрегать возможностью мониторинга пулов EJB, поддерживаемой большинством серверов приложений. Пользуясь результатами мониторинга, можно определить некоторые характерные особенности работы приложения.

Не злоупотребляйте поддержкой удаленного доступа

Напомним, что по умолчанию интерфейсы определяются как локальные. Даже при том, что поддержка удаленных интерфейсов может быть бесценной, когда это действительно необходимо, она может давать значительное снижение производительности при бездумном использовании. Проблема в том, что если по ошибке использовать аннотацию `@Remote` вместо `@Local`, локальные компоненты EJB будут внедряться аннотацией `@EJB` как удаленные. Именно поэтому очень важно избегать аннотации `@Remote`, если она не нужна.

Используйте интерфейсы

Технически, начиная с версии EJB 3.1, можно избежать использования интерфейсов, но мы рекомендуем использовать их в любом случае, если только вы не разрабатываете прототип или не используете сеансовые компоненты в JSF. При разработке служб лучше стремиться к образованию как можно более слабых связей, применяя интерфейсы, так как такой подход упрощает тестирование и увеличивает гибкость.

Особое внимание уделяйте проектированию удаленных интерфейсов

При проектировании удаленных интерфейсов включайте в них только те методы, которые действительно предназначены для удаленного использования. Мож-

но иметь локальный и удаленный интерфейсы, экспортирующие совершенно разные наборы методов.

Удаленные объекты не должны передаваться по ссылке

Параметры и возвращаемые значения удаленных методов должны поддерживать сериализацию. Кроме того, передача объектов по сети осуществляется их копированием. Это означает, что не следует ожидать обновления локальной ссылки при передаче объекта удаленному методу, который изменяет этот объект. Например, если удаленному сеансовому компоненту без сохранения состояния передать список объектов, вы не увидите изменения в списке, произведенные этим компонентом, если только он не будет возвращен удаленным методом. Такой прием может сработать при локальном доступе к компоненту, выполняющемся в том же приложении и в той же виртуальной машине JVM, но он точно не будет работать с удаленными компонентами.

Избегайте слишком мелкого дробления удаленных вызовов

Удаленный вызов является весьма дорогостоящей операцией – локальные компоненты могут позволить себе мелкое дробление вызовов, тогда как удаленные должны выполнять максимальный объем работы в одном вызове. При проектировании удаленных служб желательно избегать многократных обращений к серверу для получения разных элементов информации в рамках обслуживания одного запроса, потому что каждое такое обращение влечет за собой накладные расходы, сумма которых может оказаться неприемлемой. Например, если в пользовательском интерфейсе (User Interface, UI) желательно предусмотреть возможность отмены сразу нескольких ставок, нет смысла вызывать метод `BidService.cancel()` в цикле – это было бы очень неэффективно. Вместо этого следует добавить дополнительный метод, который будет принимать список ставок для отмены.

3.3. Сеансовые компоненты с сохранением состояния

Напомним, что в отличие от сеансовых компонентов без сохранения состояния, компоненты с сохранением состояния поддерживают хранение промежуточного состояния между вызовами методов. Поддержка состояния диалога открывает новые возможности, такие как выполнение продолжительных операций с базами данных. С программной точки зрения сеансовые компоненты с сохранением состояния мало чем отличаются от своих собратьев без сохранения состояния. Единственное существенное отличие заключается в обслуживании их жизненных циклов контейнером. Контейнер гарантирует, что каждый вызов метода будет передаваться одному и тому же экземпляру компонента, хоть удаленному, хоть локальному. Достигается это за счет закулисных манипуляций. На рис. 3.6 схематически показано, как это осуществляется.

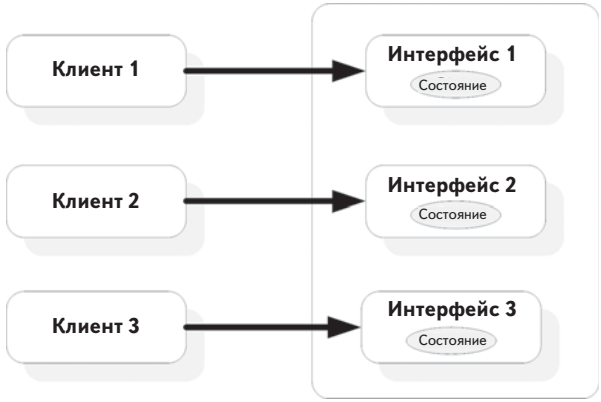

Рис. 3.6. За каждым клиентом резервируется свой экземпляр компонента. Благодаря этому компонент получает возможность хранить информацию, пока сеанс не будет завершен клиентом или пока не истечет предельное время ожидания

Помимо гарантии использования одного и того же экземпляра компонента для всех вызовов от данного клиента, контейнер также гарантирует доступность компонента только из одного потока выполнения.

Это означает, что вам нет нужды беспокоиться о синхронизации – к компоненту может обращаться множество потоков выполнения, но контейнер гарантирует его сохранность в непротиворечивом состоянии. Однако при такой организации теряется некоторая гибкость – только один поток выполнения в каждый конкретный момент времени, без каких-либо исключений.

Связь «один-к-одному» между клиентом и экземпляром компонента обеспечивает нормальный диалоговый режим. Но поддержание такой связи имеет свою цену. Экземпляры компонентов не могут сохраняться в пуле и повторно использоваться другими клиентами. Вместо этого экземпляр должен постоянно храниться в памяти и ожидать других запросов от клиента, пока тот не закроет сеанс. Как результат, экземпляры сеансовых компонентов с сохранением состояния, удерживаемые большим числом одновременно обслуживаемых пользователей, могут занимать значительные объемы памяти. Однако и для этой ситуации существует прием оптимизации, который называется *пассивацией* (passivation). Мы обсудим его в разделе 3.3.2.

3.3.1. Когда следует использовать сеансовые компоненты с сохранением состояния

Сеансовые компоненты с сохранением состояния идеально подходят для реализации процедур, состоящих из нескольких этапов. Чтобы понять, что это означает, представьте мастер создания нового конверта в Microsoft Word. Этот мастер проведет вас через процесс создания нового конверта, позволив попутно выбрать стиль оформления, ввести адрес получателя и отправителя. В реальной жизни подобные процедуры часто намного сложнее, включают множество дополнительных этапов и выполняют запросы к базам данных.

Многоэтапные процедуры не всегда требуют использования сеансовых компонентов с сохранением состояния. Например, для управления состоянием на уровне веб-служб можно использовать простые компоненты, управляемые механизмом CDI. Как будет рассказываться в главе 12, механизмы CDI и JSF обладают богатейшей поддержкой контекстов, отсутствующей в EJB. Как правило, компоненты с сохранением состояния следует использовать, только если процедура реализует некоторую прикладную логику, не укладывающуюся в рамки веб-уровня. Подобно любым другим компонентам EJB, сеансовые компоненты с сохранением состояния имеют несколько важных свойств. Эти компоненты поддерживают работу в многопоточной среде выполнения и транзакции, а также пользуются средствами безопасности, предоставляемыми контейнером. Эти компоненты также могут быть доступны через RMI, SOAP и REST. Наконец, контейнер автоматически управляет их жизненным циклом – компоненты с сохранением состояния могут пассивироваться при долгом простаивании или уничтожаться по истечении предельного времени ожидания. Показаниями к использованию сеансовых компонентов с сохранением состояния могут служить многие предпосылки, что и к использованию компонентов без сохранения состояния.

3.3.2. Пассивация компонентов

Одно из больших преимуществ сеансовых компонентов с сохранением состояния заключается в поддержке контейнером возможности их архивирования, когда они долго не используются. Причин длительного бездействия компонентов может быть масса. Например, пользователь может отлучиться по делам или переключиться на работу с другим приложением. Причины могут быть самыми разными, но очевидно одно – сервер должен заметить длительное бездействие компонента и предпринять шаги к временному освобождению занимаемых им ресурсов. Для освобождения памяти контейнер использует прием пассивации.

Под пассивацией подразумевается перемещение экземпляра компонента из оперативной памяти на диск. Эта задача решается контейнером путем сериализации всего экземпляра компонента. Активация – это процедура, обратная пассивации и она выполняется, когда появляется необходимость вновь задействовать экземпляр компонента, как показано на рис. 3.7. Активация выполняется посредством извлечения экземпляра из хранилища с последующей его десериализацией. Как следствие, все переменные экземпляра в компоненте должны быть либо простыми значениями Java, либо объектами, реализующими интерфейс `java.io.Serializable`, либо должны быть помечены как перемещаемые (`transient`).

Как будет показано в разделе 3.3.7, компоненты с сохранением состояния предусматривают точки входа для выполнения операций непосредственно перед пассивацией и сразу после активации. Используя эти точки входа можно сохранять данные в базе и выполнять другие заключительные операции. Обратите внимание, что подобно поддержке пулов, пассивация никак не регламентируется спецификацией EJB. Но большинство серверов приложений поддерживают ее. Как и в

случае спулами, есть возможность осуществлять мониторинг и настройку пассивации компонентов с сохранением состояния.

Рис. 3.7. Пассивация и активация – важнейшие приемы оптимизации, применяемые к сеансовым компонентам с сохранением состояния

3.3.3. Сеансовые компоненты с сохранением состояния в кластере

Хотя спецификация EJB и не требует этого, большинство серверов приложений все же поддерживают выполнение сеансовых компонентов с сохранением состояния в кластерах. Это означает, что состояние компонента копируется на все серверы, образующие кластер. Даже если на сервере, где в настоящее время находится экземпляр компонента, произойдет авария, механизмы кластеризации и балансировки нагрузки прозрачно переадресуют вас на другой доступный сервер в кластере, где будет находиться готовый к использованию экземпляр компонента с действительным текущим состоянием. Объединение в кластеры часто используется для организации критически важных систем, чтобы обеспечить максимальную надежность.

3.3.4. Пример реализации создания учетной записи

Процесс создания учетной записи пользователя приложения ActionBazaar слишком сложен, чтобы его можно было реализовать в виде единственной операции. Поэтому данная процедура разбита на несколько этапов. На каждом этапе пользователь вводит очередную порцию данных. Например, пользователь сначала может ввести имя и пароль; затем биографическую информацию, как, например, настоящее имя, адрес

и контактную информацию; затем информацию о способах оплаты, например, через кредитную карту или счет в банке; и так далее. В конце процедуры создается новая учетная запись и процесс завершается. Эта процедура изображена на рис. 3.8.

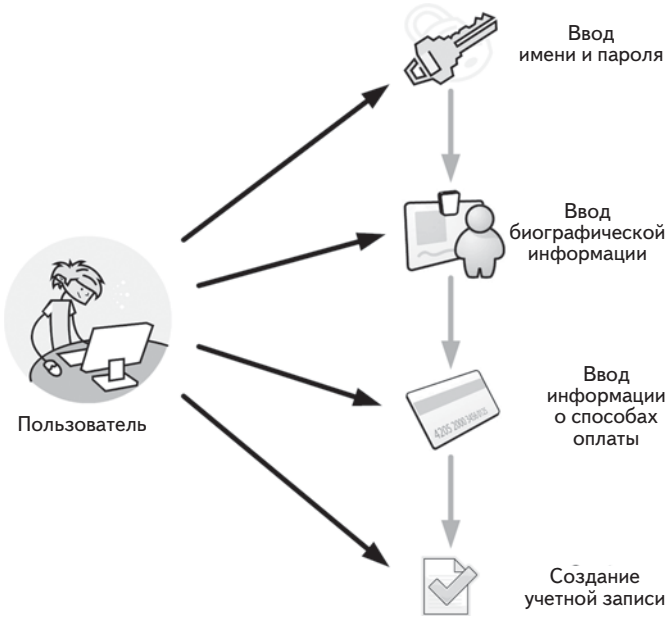

Рис. 3.8. Процесс создания учетной записи пользователя приложения ActionBazaar делится на несколько этапов. Каждый этап – это вызов отдельного метода, а сохранение состояния между вызовами обеспечивает сеансовый компонент с сохранением состояния

Каждый этап реализован в виде отдельного метода компонента `BidderAccountCreator`, представленного в листинге 3.2. Данные, собранные на каждом этапе, накапливаются в переменных экземпляра сеансового компонента с сохранением состояния. Вызов любого из методов, `cancelAccountCreation` или `createAccount`, завершает процедуру. Метод `createAccount` создает учетную запись в базе данных и завершает процедуру. Метод `cancelAccountCreation`, напротив, просто завершает процедуру и ничего не сохраняет в базе данных. Метод `cancelAccountCreation` может быть вызван в любой момент. Полный код примера можно найти на сайте книги.

Листинг 3.2. Пример сеансового компонента с сохранением состояния

```
// ❶ Пометить POJO как сеансовый компонент с сохранением состояния
@Stateful(name="BidderAccountCreator")
public class DefaultBidderAccountCreator implements BidderAccountCreator {
  @Resource(name = "jdbc/ActionBazaarDataSource")
  private DataSource dataSource;
  private Connection connection;
  private LoginInfo loginInfo;                  // ❷ Переменные
  private BiographicalInfo biographicalInfo; //    экземпляра
  private BillingInfo billingInfo;              //    компонента

  @PostConstruct // ❸ Обрабатывает событие PostConstruct
```

```
@PostActivate  // ❹ Обрабатывает событие PostActivate
public void openConnection() {
  try {
    connection = dataSource.getConnection();
  } catch (SQLException sqle) {
    sqle.printStackTrace();
  }
}

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

public void addBiographicalInfo(BiographicalInfo biographicalInfo)
    throws WorkflowOrderViolationException {
  if (loginInfo == null) {
    throw new WorkflowOrderViolationException(
      "Login info must be set before biographical info");
  }
  this.biographicalInfo = biographicalInfo;
}

public void addBillingInfo(BillingInfo billingInfo)
    throws WorkflowOrderViolationException {
  if (biographicalInfo == null) {
    throw new WorkflowOrderViolationException(
      "Biographical info must be set before billing info");
  }
  this.billingInfo = billingInfo;
}

@PrePassivate  // ❺ Обрабатывает событие PrePassivate
@PreDestroy    // ❻ Обрабатывает событие PreDestroy
public void cleanup() {
  try {
    connection.close();
    connection = null;
  } catch (SQLException sqle) {
    sqle.printStackTrace();
  }
}

@Remove                                  // ❼ Вызов
public void cancelAccountCreation() {    //   этого
  loginInfo = null;                      //   метода
  biographicalInfo = null;               //   вызывает
  billingInfo = null;                    //   уничтожение
}                                        //   компонента

@Remove
public void createAccount() {
  try {
    Statement statement = connection.createStatement();
    String sql = "INSERT INTO BIDDERS("
```

```
                        ...
                        + "username, "
                        ...
                        + "first_name, "
                        ...
                        + "credit_card_type"
                        ...
                    + ") VALUES ("
                        ...
                        + "'" + loginInfo.getUsername() + "', "
                        ...
                        + "'" + biographicalInfo.getFirstName() + "', "
                        ...
                        + "'" + billingInfo.getCreditCardType() + "'"
                        ...
                    + ")";
      statement.execute(sql);
      statement.close();
    } catch (SQLException sqle) {
      sqle.printStackTrace();
    }
  }
}

public interface BidderAccountCreator {
  void addLoginInfo(LoginInfo loginInfo);
  void addBiographicalInfo(BiographicalInfo biographicalInfo)
    throws WorkflowOrderViolationException;
  void addBillingInfo(BillingInfo billingInfo)
    throws WorkflowOrderViolationException;
  void cancelAccountCreation();
  void createAccount();
}
```

Как упоминалось выше, кого-то может удивить, сколько общего имеют сеансовые компоненты с сохранением состояния со своими собратьями без сохранения состояния.

> **Примечание.** *Мы будем использовать JDBC в наших примерах только ради простоты. Окончательная версия приложения будет использовать JPA, но пока мы не хотим вносить сумбур в ваши умы кодом, использующим этот механизм. подробнее о JPA будет рассказываться в главе 9.*

Аннотация `@Stateful` перед объявлением класса `DefaultBidderAccountCreator` превращает POJO в сеансовый компонент с сохранением состояния ❶. Кроме имени, во всем остальном данная аннотация действует точно так же, как `@Stateless`, и мы коротко обсудим ее в следующем разделе. Компонент реализует прикладной интерфейс `BidderAccountCreator`. Точно так же, как в листинге 3.1, с помощью аннотации `@Resource` в компонент внедряется источник данных JDBC. оба метода, отмеченные аннотациями `@PostConstruct` ❸ и `@PostActivate` ❹, подготавливают компонент к использованию открытого соединения с базой данных через внед-

ренный источник данных. Методы, отмеченные аннотациями `@PrePassivate` ❺ и `@PreDestroy` ❻, напротив, закрывают соединение.

Переменные экземпляра – `loginInfo`, `biographicalInfo` и `billingInfo` – используются для хранения состояния диалога с клиентом между вызовами методов ❷. Каждый из прикладных методов реализует отдельный этап процедуры создания учетной записи и постепенно заполняют переменные экземпляра. В листинге это не показано, но на каждом этапе может выполняться достаточно сложная логика, в том числе и обращения к базе данных (например, для проверки). Вся процедура завершается, когда клиент вызывает любой из методов, отмеченных аннотацией `@Remove` ❼.

3.3.5. Применение аннотации @Stateful

Аннотация `@Stateful` превращает простой Java-объект `DefaultBidderAccountCreator` в сеансовый компонент с сохранением состояния. Кроме того, что она сообщает контейнеру о назначении POJO, эта аннотация почти ничего больше не делает. Аннотация `@Stateful` имеет следующее определение в спецификации:

```
@Target(value = {ElementType.TYPE})
@Retention(value = RetentionPolicy.RUNTIME)
public @interface Stateful {
    public String name() default “”;
    public String mappedName() default “”;
    public String description() default “”;
}
```

Как видите, определение аннотации `@Stateful` в точности совпадает с определением аннотации `@Stateless`. Параметр `name` определяет имя компонента. В листинге 3.2 с его помощью компоненту присваивается имя `BidderAccountCreator`. Из объявления аннотации видно, что параметр `name` является необязательным. Вы можете опустить его, сократив код до:

```
@Stateful
public class DefaultBidderAccountCreator implements BidderAccountCreator {
```

Если параметр `name` отсутствует, контейнер присвоит компоненту имя класса. В данном случае контейнер присвоил бы имя `DefaultBidderAccountCreator`. Поле `mappedName` используется некоторыми производителями как имя компонента. Вы также можете указать его при определении своих компонентов EJB – некоторые контейнеры, такие как GlassFish, используют это имя для включения компонента в JNDI. Фактически это поле с именами EJB осталось в наследство от прежних версий, и было стандартизовано в Java EE 6 (имена EJB мы обсудим а главе 5). Как уже отмечалось, `DefaultBidderAccountCreator` реализует прикладной интерфейс `BidderAccountCreator`. Давайте рассмотрим особенности прикладных интерфейсов сеансовых компонентов с сохранением состояния и сравним их с интерфейсами компонентов без сохранения состояния.

3.3.6. Прикладные интерфейсы компонентов

Прикладные интерфейсы сеансовых компонентов с сохранением состояния решают практически те же проблемы, что и прикладные интерфейсы компонентов без сохранения состояния, с двумя исключениями. Сеансовые компоненты с сохранением состояния так же поддерживают локальные и удаленные взаимодействия с помощью аннотаций `@Local` и `@Remote`. Но они не поддерживают интерфейс конечных точек веб-служб. Это означает, что сеансовые компоненты с сохранением состояния не могут экспортироваться с использованием JAX-RS или JAX-WS. Это обусловлено самой природой веб-служб, которые не хранят информацию о своем состоянии.

Прикладной интерфейс всегда должен включать метод с аннотацией `@Remove`. Допускается применять эту аннотацию к нескольким методам. Она сообщает контейнеру, что клиент завершил работу с компонентом. В следующих разделах мы детальнее рассмотрим этапы жизненного цикла, и тогда вам станет ясно, почему всегда следует предусматривать метод, отмеченный аннотацией `@Remove`.

3.3.7. События жизненного цикла

Сеансовые компоненты с сохранением состояния имеют более сложный жизненный цикл, чем компоненты без сохранения состояния. Основных отличия два: сеансовые компоненты с сохранением состояния связаны с определенным клиентом и они могут пассивироваться. Рассмотрим для начала рис. 3.9, где изображен жизненный цикл. Контейнер выполняет следующие действия:

1. В начале нового сеанса контейнер всегда создает новый экземпляр компонента, используя для этого конструктор по умолчанию.
2. После вызова конструктора контейнер внедряет ресурсы, такие как контексты JPA, источники данных и другие компоненты.
3. Экземпляр хранится в памяти, в ожидании обращений.
4. Клиент вызывает прикладной метод через прикладной интерфейс.
5. Контейнер ожидает очередных запросов на вызовы методов и выполняет их.
6. Если клиент достаточно долго бездействует, контейнер пассивирует экземпляр компонента (если эта операция поддерживается). Компонент сериализуется и сохраняется на диск.
7. Если клиент возобновит работу и обратится к пассивированному компоненту, экземпляр будет активирован (прочитан с диска в память).
8. Если клиент продолжает долго бездействовать, компонент уничтожается.
9. Если клиент затребовал удалить экземпляр компонента, находящийся в этот момент в пассивированном состоянии, он будет активирован и затем уничтожен и утилизирован сборщиком мусора.

Подобно сеансовым компонентам без сохранения состояния, компоненты с сохранением состояния поддерживают несколько методов обработки событий жизненного цикла: два метода, как и в компонентах без сохранения состояния,

обрабатывают события создания и уничтожения компонента, и два других метода обрабатывают события, связанные с пассивацией/активацией. Эти методы объявляются с помощью следующих аннотаций:

- `@PostConstruct` – метод, отмеченный этой аннотацией, будет вызываться сразу после вызова конструктора по умолчанию и внедрения всех ресурсов;
- `@PrePassivate` – метод, отмеченный этой аннотацией, будет вызываться перед пассивацией компонента, то есть, непосредственно перед его сериализацией и сохранением на диск;
- `@PostActivate` – метод, отмеченный этой аннотацией, будет вызываться сразу после того, как компонент будет прочитан в память, но перед вызовом любых прикладных методов;
- `@PreDestroy` – метод, отмеченный этой аннотацией, будет вызываться после истечения таймаута или явного вызова клиентом одного из методов с аннотацией `@Remove`; после этого экземпляр будет передан сборщику мусора для утилизации.

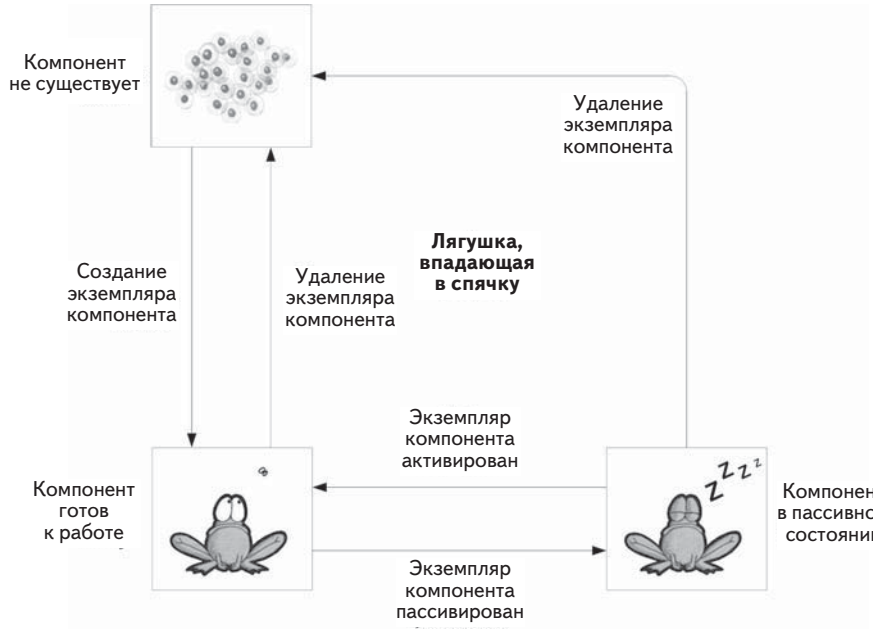

Рис. 3.9. Жизненный цикл сеансового компонента с сохранением состояния. Из-за того, что компонент хранит информацию клиента, его нельзя поместить в пул. Зато его можно пассивировать после некоторого периода простоя и активировать, когда клиент возобновит работу с ним

Если потребуется, этими аннотациями можно пометить несколько методов – вы не ограничены одной аннотацией каждого вида на класс. Метод обработки события `PrePassivate` дает компоненту возможность подготовиться к сериализации, напри-

мер, скопировать несериализуемые переменные в сериализуемые, удалить из переменных ненужные данные для экономии дискового пространства. Часто на этом этапе подготовки выполняется освобождение тяжеловесных ресурсов, таких как открытые соединения с базами данных и другие сетевые соединения, которые не могут быть сериализованы. Перед пассивацией (сериализацией) добропорядочный компонент должен закрыть такие ресурсы и явно очистить соответствующие ссылки.

С точки зрения экземпляра компонента между пассивацией и уничтожением не так много отличий. На практике очень часто аннотации `@PrePassivate` и `@PreDestroy` применяются к одному и тому же методу. То же самое в значительной степени относится и к аннотациям `@PostConstruct` и `@PostActivate`. В случае этих двух событий, обработчики впервые или повторно приобретают тяжеловесные ресурсы. Листинг 3.2 является отличным примером в этом отношении: объект `java.sql.Connection` не поддается сериализации и потому должен создаваться заново в момент активации.

Важно отметить, что вы не должны восстанавливать соединения с внедренными ресурсами. В момент активации контейнер сам повторно внедрит все необходимые ресурсы. Также автоматически будут восстановлены ссылки на любые другие внедренные сеансовые компоненты с сохранением состояния. В методах-обработчиках следует создавать или уничтожать только ресурсы, которыми вы управляете вручную.

Вдобавок к этим методам можно отметить один или более методов аннотацией `@Remove`. Она сообщает контейнеру, что после выполнения этого метода компонент должен быть уничтожен – клиент больше не нуждается в нем и не будет обращаться к нему в дальнейшем. Если же клиент попытается обратиться к компоненту после вызова такого метода, будет возбуждено исключение. В случае с компонентом `BidderAccountCreator`, аннотацией `@Remove` можно пометить несколько методов. Один метод подтверждает создание учетной записи, другой – отменяет. Отказ от использования этой аннотации может отрицательно отразиться на производительности сервера. Проблема может не проявлять себя при небольшом числе клиентов, но с ростом их числа она станет очень важной, так как увеличится число людей, использующих сеансовые компоненты с сохранением состояния.

3.3.8. Эффективное использование сеансовых компонентов с сохранением состояния

Нет никаких сомнений, что сеансовые компоненты с сохранением состояния обеспечивают очень ценную возможность реализовать логику обработки данных в диалоговом режиме, если это действительно необходимо. Кроме того, в EJB 3 была добавлена поддержка дополнительных контекстов хранения, специально предназначенных для компонентов с сохранением состояния (обсуждаются в главе 10). Она существенно расширяет возможности компонентов. И тем не менее мы хотим сделать несколько замечаний, прежде чем вы приступите к использованию сеансовых компонентов с сохранением состояния. Во-первых, многие рекомендации, касающиеся сеансовых компонентов без сохранения состояния, применимы

и к компонентам с сохранением состояния. Дополнительно вы должны учитывать факторы, перечисленные ниже.

Внимательно выбирайте данные для сохранения в сеансе

Сеансовые компоненты с сохранением состояния могут превратиться в «пожирателей» ресурсов при неправильном их использовании. Так как информация о сеансе сохраняется контейнером в оперативной памяти, то при попытке одновременно обслужить тысяч пользователей с помощью компонентов с сохранением состояния можно столкнуться с проблемой исчерпания памяти или чрезмерного количества дисковых операций, связанных с пассивацией/активацией компонентов. Следовательно, необходимо внимательно разобраться, какие данные действительно необходимы для поддержания диалога, и постараться максимально уменьшить объем памяти, занимаемой компонентом. Например, будьте осторожны при сохранении объектов с очень глубоким деревом зависимостей, массивов байтов или массивов символов.

Когда сеансовые компоненты с сохранением состояния используются в кластерном окружении, информация о состоянии копируется между разными экземплярами контейнера EJB. Эти операции копирования отнимают некоторую долю полосы пропускания сети. Наличие больших объектов в компоненте может оказать существенное отрицательное влияние на производительность приложения из-за того, что контейнер будет тратить значительное время на отправку объектов другим экземплярам, чтобы обеспечить высокую надежность. Подробнее проблемы кластеризации мы рассмотрим в главе 15.

Настраивайте пассивацию

Правила пассивации обычно зависят от конкретной реализации. Неправильное использование политик пассивации (когда дается возможность настраивать пассивацию) может вызывать проблемы производительности. Например, Oracle Application Server пассивирует экземпляры компонентов, когда истекает максимальное время простоя, по достижении числа активных экземпляров компонентов некоторого порогового значения, а также по достижении верхней границы используемой памяти JVM. Обратитесь к документации для своего контейнера EJB и настройте соответствующие правила пассивирования. Например, если установить максимально возможное число активных экземпляров сеансовых компонентов с сохранением состояния равным 100, когда обычно одновременно обслуживается порядка 150 клиентов, контейнер постоянно будет выполнять пассивацию/активацию экземпляров, ухудшая общую производительность.

Своевременно удаляйте сеансовые компоненты с сохранением состояния

Вы можете существенно облегчить проблему нехватки памяти, явно удаляя ставшие ненужными экземпляры компонентов и не полагаясь, что контейнер уда-

лит их автоматически по истечении таймаута. Для этого каждый сеансовый компонент с сохранением состояния должен иметь хотя бы один метод, отмеченный аннотацией `@Remove`, который будет вызываться в конце многоходовой процедуры, реализуемой компонентом.

Теперь, после более близкого знакомства с сеансовыми компонентами с сохранением состояния, перейдем к знакомству с компонентами-одиночками.

3.4. Сеансовые компоненты-одиночки

Впервые поддержка компонентов-одиночек появилась в EJB 3.1. Как следует из названия, в течение всего времени выполнения приложения может существовать только один экземпляр такого компонента. То есть, все клиенты обращаются к одному и тому же компоненту, как показано на рис. 3.10. Компоненты-одиночки были добавлены с целью решить две архитектурные проблемы, долгое время преследовавшие приложения Java EE: инициализация сервера на запуске и обеспечение наличия единственного экземпляра компонента. До EJB 3.1 эти проблемы решались с применением «сервлетов запуска» или соединителей (connectors) JCA. В бесчисленном множестве приложений, в нарушение принципов спецификации Java EE, использовались статические переменные или неуправляемые объекты. В лучшем случае это были некрасивые решения, но существовало еще множество других, еще менее удачных решений.

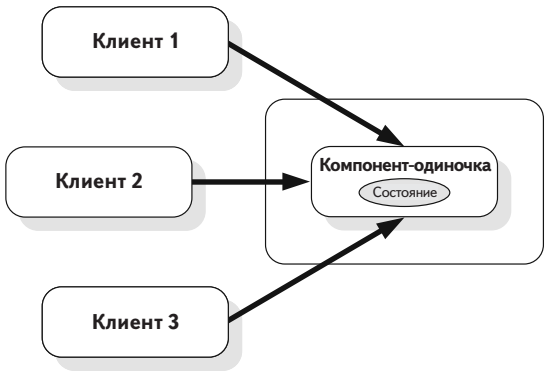

Рис. 3.10. Единственный экземпляр компонента совместно используется всеми клиентами

Компоненты-одиночки близко напоминают компоненты без сохранения состояния. Они поддерживают обработку тех же событий жизненного цикла, но имеют и свои, уникальные особенности, включая возможность управления параллельным доступом, а также возможность определять цепочки для создания компонентов-одиночек в определенной последовательности. Поддержка таких цепочек позволяет одним компонентам зависеть от других. Как уже упоминалось, компоненты-одиночки можно пометить, чтобы обеспечить их создание на этапе развертывания приложения. В этом случае запуск приложения не будет считаться выполненным, пока все отмеченные компоненты не будут успешно созданы. Так же, как компо-

ненты с сохранением состояния и без, компоненты-одиночки поддерживают внедрение, безопасность и транзакции.

3.4.1. Когда следует использовать сеансовые компоненты-одиночки

До появления спецификации EJB 3.1 большинство разработчиков приложений с веб-интерфейсом использовали «сервлеты запуска» в роли компонентов-одиночек. Сервлет настраивался на запуск как можно раньше через настройки в *web.xml*. Логика запуска помещалась в метод `init` сервлета. В этот метод включалась реализация типичных задач, таких как настройка журналирования, инициализация соединений с базами данных и кэширование часто используемых данных в памяти. Кэшированные данные затем помещались в контекст сеанса приложения или контейнера веб-приложения. Для синхронизации обновления кэшированных данных необходимо было предусматривать специальные конструкции, которые могли быть весьма сложными, особенно если приложение размещалось на нескольких серверах за механизмом балансировки нагрузки. Это было далеко не оптимальное решение.

Компоненты-одиночки позволяют отказаться от обходных решений, таких как «сервлеты запуска», POJO со статическими полями и другие варианты. Компоненты-одиночки используются, когда требуется иметь некоторое общее состояние, глобальное для всего приложения или для определенной задачи. Давайте познакомимся с ними поближе, а заодно выясним, в каких случаях не следует использовать компоненты-одиночки.

Задачи инициализации на запуске

Очень часто бывает необходимо выполнить пару операций на этапе развертывания приложения, прежде чем оно станет доступно внешним пользователям и другим приложениям. На запуске приложения может потребоваться проверить непротиворечивость данных в базе или убедиться в работоспособности некоторой внешней системы. Очень часто процедуры инициализации попадают в состояние гонки – хотя запуск LDAP можно настроить так, чтобы он начинался до запуска сервера GlassFish, сервер GlassFish может успеть развернуть приложение раньше, чем механизм LDAP будет готов принимать соединения. Компонент-одиночка, настроенный на создание во время инициализации приложения, мог бы в цикле осуществлять проверку готовности LDAP и тем самым гарантировать недоступность приложения, пока све внешние службы не запустятся и не будут готовы принимать запросы.

Централизованная информационная служба

Хотя создание централизованных пунктов обмена информацией противоречит принципам создания масштабируемых приложений, иногда все же они бывают необходимы. Примером может служить взаимодействие с устаревшей системой, ограничивающей число соединений с ней. Гораздо вероятнее ситуация, когда по-

требуется кэшировать некоторое значение и обеспечить доступ к нему множеству клиентов. Благодаря встроенной поддержке конкуренции можно обеспечить одновременное чтение значения несколькими клиентами и принудительную синхронизацию операций записи. Так, например, может понадобиться кэшировать значение в памяти, чтобы каждое обращение к веб-интерфейсу регистрировалось в базе данных.

Когда не нужно использовать компоненты-одиночки

Сеансовые компоненты-одиночки должны использоваться для решения задач, требующих наличия единственного компонента и доступа к службам контейнера. Например, компонент-одиночка не должен использоваться для реализации логики проверки телефонного номера. Эта задача не требует ни наличия единственного компонента, ни доступа к службам контейнера. Ее можно решить с помощью вспомогательного метода. Если компонент не хранит или не защищает свое состояние от параллельных обращений или изменений, он не должен быть реализован как сеансовый компонент-одиночка. Поскольку компонент-одиночка может превратиться в узкое место при неправильном использовании, в большинстве случаев предпочтительнее использовать сеансовые компоненты без сохранения состояния и применять компоненты-одиночки, только когда они действительно необходимы.

Никогда не используйте компоненты-одиночки для реализации служб без сохранения состояния – вместо этого пользуйтесь сеансовыми компонентами без сохранения состояния. В большинстве своем такие службы доступны не только для чтения и требуют взаимодействий с базой данных с применением API без поддержки многопоточной модели выполнения, таких как диспетчер сущностей JPA. Использование компонента-одиночки, обладающего такой поддержкой, в данном случае ухудшит масштабируемость, потому что он будет существовать в единственном экземпляре, и должен будет использоваться всеми клиентами. Теперь, после начального знакомства с компонентами-одиночками, перейдем к практическому примеру.

3.4.2. Пример реализации «товара дня» в ActionBazaar

Каждый день приложение ActionBazaar выбирает некоторый товар дня, как показано не рис. 3.11. Вскоре после полуночи сайт выбирает новый товар дня. Пару лет тому назад логика подобной операции могла находиться на веб-уровне, поскольку он являлся единственным интерфейсом к приложению. Но теперь ActionBazaar поддерживает клиентские приложения для iPhone и Android, использующие специализированную веб-службу и выделенный сайт для мобильных устройств. По этой причине логика была перенесена на прикладной уровень. Так как эта информация по сути является статической, нет никакого смысла вновь и вновь извлекать ее из базы данных при каждом посещении веб-сайта. Обращение к базе данных каждый раз создает ненужную нагрузку на саму базу данных и сеть. Один

из способов избежать этого заключается в том, чтобы кэшировать часто используемые данные.

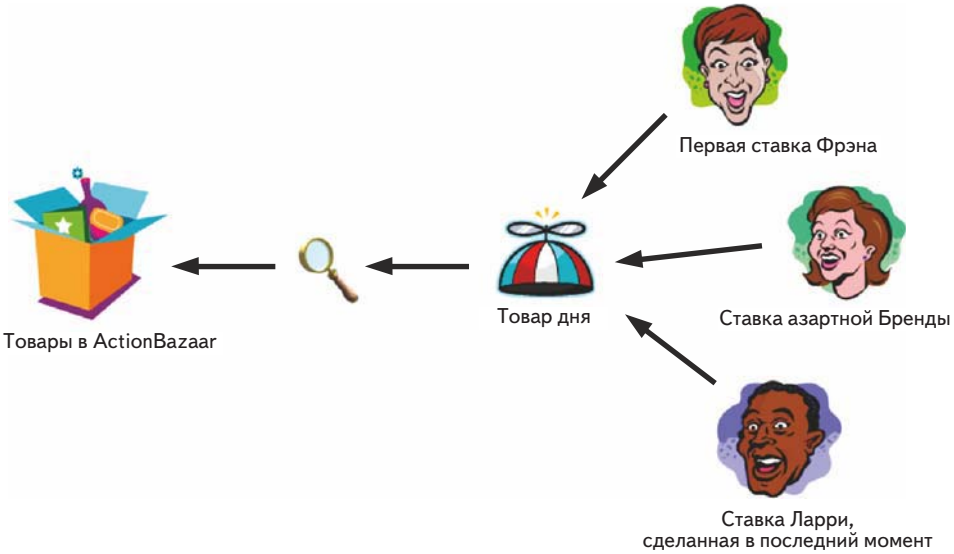

Рис. 3.11. Товар дня в ActionBazaar – отличный кандидат на оформление в виде компонента-одиночки

Каждый из клиентов получает информацию о товаре дня из компонента-одиночки. Этот компонент запоминает значение на запуске и затем обновляет его в полночь. В листинге 3.3 приводится реализация такого компонента. Здесь не показан компонент `SystemInitializer`, настраивающий журналирование и выполняющий другие операции по инициализации – его можно найти в zip-файле с исходным кодом примеров для этой главы.

Листинг 3.3. Пример сеансового компонента-одиночки

```
@Singleton // ❶ Пометить как сеансовый компонент-одиночку
@Startup   // ❷ Создает компонент на запуске
@DependsOn("SystemInitializer") // ❸ Определяет зависимость компонента
public class DefaultFeaturedItem implements FeaturedItem {
  private Connection connection;

  @Resource(name = "jdbc/ActionBazaarDataSource")
  private DataSource dataSource;

  private Item featuredItem;

  @PostConstruct // ❹ Этот метод вызывается сразу после создания
  public void init() {
    try {
      connection = dataSource.getConnection();
```

```
    } catch (SQLException sqle) {
      sqle.printStackTrace();
    }
    loadFeaturedItem();
  }

  // ❺ Запланировать обновление товара дня каждую полночь
  @Schedule(dayOfMonth="*",dayOfWeek="*",hour="0",minute="0",second="0")
  private void loadFeaturedItem() {
    featuredItem = ... загрузка информации о товаре из базы данных ...
  }

  @Override
  public Item getFeaturedItem() {
    return featuredItem;
  }
}
...
@Remote
public interface FeaturedItem {
  public Item getFeaturedItem();
}
```

Аннотация `@Singleton` превращает этот POJO в компонент-одиночку ❶. Он реализует прикладной интерфейс `FeaturedItem`. Аннотация `@Startup` сообщает контейнеру, что компонент должен быть создан на этапе запуска приложения ❷. При этом компонент не должен создаваться раньше, чем будет создан компонент-одиночка `SystemInitializer` ❸. Экземпляр `DefaultFeaturedItem` будет создан только после создания экземпляра `SystemInitializer`, затем контейнер выполнит внедрение ресурсов и вызовет метод `init`, отмеченный аннотацией `@PostConstruct` ❹. Контейнер EJB также создаст таймер, который будет срабатывать каждую полночь и вызывать метод, осуществляющий загрузку товара дня ❺. Таймеры EJB не являются особенностью компонентов-одиночек и будут обсуждаться в главе 5.

Как видите, сеансовые компоненты-одиночки очень похожи на компоненты без сохранения состояния. Единственное отличие лишь в том, что контейнер создаст единственный экземпляр компонента и будет использовать его для обслуживания всех клиентов. Давайте рассмотрим аннотацию `@Singleton` поближе.

3.4.3. Применение аннотации @Singleton

Аннотация `@Singleton` превращает простой Java-объект `DefaultFeaturedItem` в сеансовый компонент-одиночку. Кроме того, что она сообщает контейнеру о назначении POJO, эта аннотация почти ничего больше не делает. Аннотация `@Singleton` имеет следующее определение в спецификации:

```
@Target(value = {ElementType.TYPE})
@Retention(value = RetentionPolicy.RUNTIME)
public @interface Singleton {
    public String name() default "";
```

```
    public String mappedName() default "";
    public String description() default "";
}
```

Определение аннотации ничем не отличается от определения аннотаций `@Stateless` и `@Stateful`. Параметр `name` определяет имя компонента. Контейнеры используют этот параметр для включения компонента EJB в глобальное дерево JNDI. Параметр `name` является необязательным и если он отсутствует, в качестве имени по умолчанию будет использовано имя класса. Поле `mappedName` используется некоторыми производителями как имя компонента – некоторые контейнеры, такие как GlassFish, используют это имя для включения компонента в JNDI. Поле `description` служит для хранения описания компонента и может использоваться некоторыми инструментами и контейнерами.

3.4.4. Управление конкуренцией в компоненте-одиночке

Так как к компоненту-одиночке может обратиться сразу несколько клиентов, возникает проблема разграничения доступа. Существует две разновидности поддержки конкуренции в компонентах-одиночках: на уровне контейнера и на уровне компонента. По умолчанию (если не указано иное) управление конкуренцией осуществляется на уровне контейнера. В этом случае синхронизация вызовов методов выполняется самим контейнером. Аннотации, предназначенные для этой цели, позволяют помечать методы как доступные для чтения или записи, а также определять таймаут для методов, которые способны блокировать выполнение. При управлении конкуренцией на уровне компонента вы сами можете решать, какие средства языка Java (такие как `synchronized`, `volatile` или `java.util.concurrent`) использовать.

Выбор способа синхронизации осуществляется с помощью аннотаций. Если не указана ни одна из аннотаций, предполагается использование механизмов синхронизации на уровне контейнера. Именно такой подход избран в листинге 3.3. Ниже перечислены возможные комбинации аннотаций и параметров для каждого способа управления конкуренцией:

- `@ConcurrencyManagement(ConcurrencyManagementType.BEAN)` – управление конкуренцией осуществляется на уровне компонента;
- `@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)` – управление конкуренцией осуществляется на уровне контейнера.

Аннотация `@ConcurrencyManagement` должна помещаться в определение класса компонента, а не в прикладной интерфейс:

```
@Target(value = {ElementType.TYPE})
@Retention(value = RetentionPolicy.RUNTIME)
  public @interface ConcurrencyManagement {
  public ConcurrencyManagementType value() default
    ConcurrencyManagementType.CONTAINER;
}
```

Управление конкуренцией на уровне контейнера

По умолчанию используется управление конкуренцией на уровне контейнера – доступ ко всем методам компонента упорядочивается посредством блокировок для записи. В случае с компонентом `DefaultFeaturedItem` из листинга 3.3, в каждый конкретный момент времени метод `getFeaturedItem` может выполняться только в контексте одного потока. Очевидно, что это не совсем то, что нам требуется: если на сайт одновременно зайдет тысяча клиентов, они будут вынуждены ждать, чтобы увидеть товар дня. Это не самое лучшее решение с точки зрения производительности, не говоря уже о продажах. В спецификации EJB определяется две аннотации, с помощью которых можно сообщить контейнеру о желаемом поведении блокировки:

- `@Lock(LockType.READ)` – метод, отмеченный этой аннотацией будет одновременно доступен множеству клиентов, пока кто-то не приобретет блокировку для записи;
- `@Lock(LockType.WRITE)` – при вызове метода будет устанавливаться блокировка для записи, поэтому метод всегда будет выполняться только в контексте одного потока.

Если не указана ни одна из аннотаций, по умолчанию применяется `@Lock(LockType.WRITE)`. Эти аннотации можно применять к методам и в прикладном интерфейсе, и в самом классе компонента. Их действие можно переопределить в XML-файле с настройками развертывания, хотя изменение правил наложения блокировок на этапе развертывания – не лучшее решение. Так как вызов метода компонента-одиночки может привести к блокировке вызывающего кода, предусмотрен механизм разблокировки по таймауту, чтобы исключить возможность блокировки навечно. По истечении таймаута возбуждается исключение `javax.ejb.ConcurrentAccessTimeoutException`. Это исключение времени выполнения, соответственно вызывающий код не сможет перехватить его. Для определения величины таймаута служит аннотация `@AccessTimeout`. Она определена, как показано ниже:

```
@Target(value = {ElementType.METHOD, ElementType.TYPE})
@Retention(value = RetentionPolicy.RUNTIME)
public @interface AccessTimeout {
    public long value();
    public TimeUnit unit() default TimeUnit.MILLISECONDS;
}
```

Как видно из определения, аннотация позволяет указать продолжительность таймаута (`value`) и единицу измерения времени. По умолчанию время измеряется в миллисекундах (1 секунда = 1 000 миллисекунд). Можно использовать следующие единицы измерения:

- наносекунды;
- микросекунды;
- миллисекунды;
- секунды;

- минуты;
- часы;
- дни.

В большинстве приложений продолжительность таймаута задается в миллисекундах или в секундах, и в очень редких случаях – в минутах. Системный таймер (особенно это относится к Windows) может оказаться не в состоянии измерять время с точностью до наносекунд, а измерение таймаутов в часах и днях будет, пожалуй, чересчур экстремальным вариантом. Давайте изменим наш прикладной интерфейс для `DefaultFeaturedItem` и добавим в него таймаут, длительностью в одну минуту:

```
public interface FeaturedItem {
  @Lock(LockType.READ)
  @AccessTimeout(value=1, unit=TimeUnit.MINUTES)
  public Item getFeaturedItem();
}
```

Управление конкуренцией на уровне компонента

Выбирая способ управления конкуренцией на уровне компонента, все хлопоты, связанные с этим, вы берете на себя. При этом вы можете использовать примитивы управления конкуренцией, имеющиеся в языке Java, такие как `synchronized`, `volatile`, `wait`, `notify` и так далее, или легендарный API `java.util.concurrent`, созданный Дугом Леа (Doug Lea). Чтобы пойти этим путем, нужно хорошо знать особенности параллельного программирования и конструкции управления конкуренцией в Java (этой теме посвящено множество замечательных книг). Разрешение проблем, связанных с конкуренцией, довольно сложная задача, но разрешение тех же проблем на уровне контейнера еще сложнее. На практике предпочтение управлению конкуренцией на уровне компонента следует отдавать, только если необходимо организовать более тонкое разделение, чем позволяют механизмы на уровне контейнера, что случается очень редко. В листинге 3.4 демонстрируется реализация синхронизации доступа к `DefaultFeaturedItem`, когда только один поток может обратиться к компоненту. Очевидно, что этот код будет иметь проблемы с производительностью, поэтому данное решение не следует использовать на практике.

Листинг 3.4. Управление конкуренцией на уровне компонента

```
@Singleton
@Startup
@DependsOn("SystemInitializer")
// Управление конкуренцией осуществляется на уровне компонента
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class DefaultFeaturedItem implements FeaturedItem {
  private Connection connection;

  @Resource(name = "jdbc/ActionBazaarDataSource")
```

```
  private DataSource dataSource;

  private Item featuredItem;

  @PostConstruct
  // Ограничить доступность компонента единственным потоком
  private synchronized void init() {
    try {
      connection = dataSource.getConnection();
    } catch (SQLException sqle) {
      sqle.printStackTrace();
    }

    loadFeaturedItem();
  }

  @Schedule(dayOfMonth="*",dayOfWeek="*",hour="0",minute="0",second="0")
  private synchronized Item loadFeaturedItem() {
    ...
  }

  @Override
  public synchronized Item getFeaturedItem() {
    return featuredItem;
  }
}
```

Теперь, когда вы получили представление о некоторых особенностях поддержки конкуренции, перейдем к исследованию прикладных интерфейсов компонентов-одиночек.

3.4.5. Прикладной интерфейс компонента

Сеансовые компоненты-одиночки поддерживает те же возможности для прикладных интерфейсов, что и сеансовые компоненты без сохранения состояния. Можно определять удаленные и локальные интерфейсы, а также интерфейсы веб-служб (SOAP и REST) с помощью аннотаций `@Local`, `@Remote` или `@WebService`, соответственно. Все три типа интерфейсов рассматриваются в разделе 3.2.5.

Единственное существенное отличие между прикладными интерфейсами компонентов-одиночек и сеансовых компонентов без сохранения состояния состоит в возможности применения следующих аннотаций к объявлениям методов в интерфейсах:

- `@Lock(LockType.READ);`
- `@Lock(LockType.WRITE);`
- `@AccessTimeout`.

Все три эти аннотации производят эффект, только при управлении конкуренцией на уровне контейнера. В противном случае они не действуют. Вполне возможно в интерфейсе веб-службы определить иное поведение механизма управления конкуренцией, отличное от поведения локального интерфейса (хотя на практике

такая возможность используется редко). Как и в случае с другими разновидностями сеансовых компонентов – с сохранением и без сохранения состояния – нет необходимости явно объявлять локальный интерфейс.

3.4.6. События жизненного цикла

Сеансовые компоненты-одиночки имеют самый простой жизненный цикл среди всех компонентов. После создания они не уничтожаются до завершения приложения. Сложности в их жизненном цикле возникают, только когда между ними имеются зависимости. В процессе развертывания приложения контейнер выполняет обход всех компонентов-одиночек и проверяет их зависимости. Порядок определения зависимостей мы обсудим чуть ниже. Кроме того, компонент может быть помечен как создаваемый автоматически в ходе развертывания приложения. В этом случае прикладная логика компонента выполняется во время запуска приложения. Взгляните для начала на рис. 3.12, где изображен жизненный цикл компонента-одиночки.

Рис. 3.12. Жизненный цикл сеансовых компонентов-одиночек

1. Контейнер создает новый экземпляр компонента-одиночки на этапе запуска приложения, если этот компонент помечен аннотацией `@Startup` или указан в виде зависимости (`@DependsOn`) другого компонента. Обычно создание компонентов-одиночек откладывается до момента первого обращения к ним.
2. Когда конструктор компонента завершится, контейнер внедряет ресурсы, такие как контексты JPA, источники данных и другие компоненты.

3. Вызывается метод, отмеченный аннотацией `@PostConstruct`. Компонент остается недоступным, пока этот метод не завершится успехом.
4. Экземпляр компонента хранится в памяти, ожидая обращений к его методам.
5. Клиент вызывает прикладной метод посредством прикладного интерфейса.
6. Контейнер информируется, что он должен прекратить работу.
7. Перед завершением он вызывает метод компонента, отмеченный аннотацией `@PreDestroy`.

Сеансовые компоненты-одиночки поддерживают те же методы обработки событий жизненного цикла, что и сеансовые компоненты без сохранения состояния. Благодаря им можно организовать выполнение некоторых операций сразу после внедрения ресурсов, но до того, как компонент станет доступен для вызова прикладных методов, и непосредственно перед тем, как компонент будет передан сборщику мусора. Эти методы перечислены ниже:

- `@PostConstruct` – метод, отмеченный этой аннотацией, будет вызываться сразу после создания экземпляра и внедрения всех ресурсов;
- `@PreDestroy` – метод, отмеченный этой аннотацией, будет вызываться непосредственно перед уничтожением экземпляра и перед

Самое важное, что следует помнить об этих методах, – они вызываются только один раз. Компонент-одиночка, как следует из названия, создается и уничтожается только однажды.

Компонент-одиночку можно пометить аннотацией `@DependsOn`. Эта аннотация определяет зависимость одного компонента от другого. Когда создается экземпляр компонента, предварительно создаются все его зависимости. Как предполагается, когда один компонент зависит от другого, этот другой компонент должен выполнить какие-то настройки окружения. Ниже приводится определение аннотации `@DependsOn`:

```
@Target(value = {ElementType.TYPE})
@Retention(value = RetentionPolicy.RUNTIME)
public @interface DependsOn {
  public String[] value();
}
```

Эта аннотация помещается перед объявлением класса компонента и принимает одно или более имен компонентов-одиночек, которые должны быть созданы перед этим компонентом. Порядок следования зависимостей в списке не определяет порядок создания компонентов. То есть, если компонент отмечен аннотацией `@DependsOn(A,B)`, не следует надеяться, что компонент `A` будет создан перед компонентом `B`. Для этого компонент `B` сам должен быть отмечен аннотацией `@DependsOn` с именем компонента `A`. Аннотация `@DependsOn` тесно связана с аннотацией `@Startup`.

3.4.7. Аннотация @Startup

Одной из ключевых особенностей компонентов-одиночек является возможность их автоматического запуска в процессе развертывания приложения. Для этого

компонент-одиночка должен быть отмечен аннотацией `@Startup`. Эта аннотация определена, как показано ниже:

```
@Target(value = {ElementType.TYPE})
@Retention(value = RetentionPolicy.RUNTIME)
  public @interface Startup {
}
```

Сама аннотация не имеет параметров. Она просто устанавливает признак, который будет проверен контейнером. Когда контейнер обнаружит компонент, отмеченный этой аннотацией, он сначала создаст все его зависимости. Эта особенность влечет за собой несколько следствий, которые мы обсудим в следующем разделе.

3.4.8. Эффективное использование сеансовых компонентов-одиночек

Сеансовые компоненты-одиночки стали долгожданным нововведением в EJB 3.1. Но прежде, чем вы начнете их использовать, необходимо сделать несколько важных замечаний. Во-первых, поскольку существует только один экземпляр, это обстоятельство может отрицательно сказываться на производительности при неправильном использовании. Во-вторых, использование управления конкуренцией на уровне контейнера не освобождает от необходимости настраивать ее поведение. Так же как сборщик мусора не освобождает от необходимости управлять памятью (оперативная память не безгранична), механизм управления конкуренцией на уровне контейнера не является панацеей от всех проблем. В-третьих, не следует злоупотреблять возможностью создания экземпляров компонентов-одиночек на запуске приложения. И, наконец, обработка исключений в компонентах-одиночках имеет свои отличительные особености.

Выбор типа конкуренции

Выбор способа управления конкуренцией является одним из важнейших решений, которое необходимо принять при создании нового компонента-одиночки. Управление конкуренцией на уровне контейнера является наиболее предпочтительным для подавляющего большинства компонентов. В этом случае задача синхронизации полностью возлагается на контейнер. Сделав выбор в пользу контейнера, вы получаете возможность отмечать методы аннотациями `@Lock(LockType.READ)` и `@Lock(LockType.WRITE)`. Блокировка типа `READ` включает возможность одновременного доступа к компоненту из нескольких потоков выполнения, тогда как блокировка типа `WRITE` обеспечивает исключительный доступ. При выборе управления конкуренцией на уровне компонента, вам придется вручную управлять доступом посредством конструкций языка Java, таких как `synchronized`, `volatile` и других. Данный подход имеет смысл применять, только если действительно необходимо организовать более тонкое управление блокировками. Однако, если данные хранятся в объекте `HashMap`, обладающем поддержкой конкуренции, вам не нужно задумываться о выборе механизма управления конкуренцией.

Настройка управления конкуренцией на уровне контейнера

Один из недостатков механизма управления конкуренцией на уровне контейнера состоит в том, что по умолчанию используются блокировки для записи, если явно не указано иное. Блокировка для записи ограничивает доступ к классу только одним методом в каждый конкретный момент времени. Каждый прикладной метод в классе фактически оказывается отмеченным ключевым словом `synchronized`. Это неблагоприятно сказывается на производительности компонента-одиночки – клиенты будут обслуживаться таким компонентом только по одному. Если обслуживание запроса занимает одну секунду, тогда тысячному клиенту придется ждать своей очереди примерно 17 минут. Время ожидания для каждого клиента будет разным – кому-то придется ждать одну секунду, кому-то значительно дольше. Поэтому компонент-одиночка должен разделять операции записи (`@Lock(LockType.WRITE)`) и чтения (`@Lock(LockType.READ)`). Метод, осуществляющий запись, изменяет данные и потому должен выполняться под защитой исключительной блокировки. Метод, осуществляющий чтение, не требует такой строгой меры и может одновременно обслуживать множество клиентов. Если операция записи требует значительного времени, соответствующий метод должен быть отмечен аннотацией `@AccessTimeout`, чтобы избежать возможности блокирования кода на неопределенное время. Данная аннотация может также использоваться для поддержания определенного уровня обслуживания – если попытка вызова метода затягивается, возбуждается исключение, которое можно использовать как диагностический признак в разрешении проблем производительности.

Управление автозапуском компонентов-одиночек

Компоненты-одиночки с автозапуском можно использовать для автоматического выполнения прикладной логики на этапе запуска приложений и кэширования необходимых данных. Если отметить класс компонента-одиночки аннотацией `@Startup`, экземпляр этого компонента будет создаваться автоматически в ходе развертывания приложения. Следует особенно подчеркнуть, что экземпляр создается *во время развертывания приложения*. Контейнеры, такие как GlassFish, возбуждают исключение `java.lang.IllegalAccessException`, если компонент-одиночка пытается обратиться к сеансовому компоненту с сохранением или без сохранения состояния из метода, отмеченного аннотацией `@PostConstruct`. В этот момент другие компоненты могут быть еще не доступны, но вы можете пользоваться услугами JPA и выполнять операции с базой данных. Если в приложении имеется множество компонентов с автоматическим запуском, используйте аннотацию `@DependsOn` для управления последовательностью их создания. Без аннотации `@DependsOn` нет никаких гарантий, что требуемый компонент-одиночка будет создан, – компоненты создаются в случайном порядке. Мы также хотели бы предостеречь вас от создания циклических зависимостей.

Обработка исключений

Исключения, генерируемые компонентами-одиночками, интерпретируются несколько иначе, чем исключения, генерируемые другими сеансовыми компонентами. Исключение, сгенерированное в методе компонента-одиночки, отмеченном аннотацией `@PostConstruct`, вызывает уничтожение экземпляра. Если метод `@PostConstruct`, возбудивший исключение, принадлежит компоненту с автозапуском, сервер приложений может отказаться от развертывания приложения. Исключение в прикладном методе не приводит к таким катастрофическим последствиям для компонента – компонент будет уничтожен, только в случае завершения приложения. Это обстоятельство должно учитываться при разработке прикладной логики. Кроме того, так как компонент существует на протяжении всего времени жизни приложения, может потребоваться позаботиться о внешних ресурсах при их использовании. Если приложение может работать неделями, необходимо предусмотреть обработку вероятных ошибок. Если компонент открывает соединение с другой системой, в какой-то момент это соединение может оказаться разорванным – отнеситесь с особым вниманием к таймаутам.

3.5. Асинхронные сеансовые компоненты

В EJB 3.1 появилась поддержка асинхронных компонентов в виде аннотации `@Asynchronous`. Это не какой-то новый тип сеансовых компонентов, а скорее новая функциональная возможность, которую можно добавлять в любые сеансовые компоненты. Эта новая особенность позволяет методам прикладного интерфейса, вызываемым клиентами действовать асинхронно, в отдельных потоках выполнения. Прежде единственным механизмом параллельного выполнения в Java EE были компоненты MDB. Использование компонентов MDB и координация их действий с сеансовыми компонентами являло собой весьма тяжеловесное решение, неприменимое для простых задач и увеличивающее сложность.

Название «асинхронные сеансовые компоненты» является несколько неточным. Сами компоненты не являются асинхронными – асинхронными являются их методы. Аннотацией `@Asynchronous` можно отмечать отдельные методы или весь класс. Когда вызывается асинхронный метод или метод асинхронного класса, контейнер запускает отдельный поток выполнения.

3.5.1. Основы асинхронного вызова

Асинхронные методы действуют точно так же, как любые другие прикладные методы класса компонента – они могут принимать параметры, возвращать значения и возбуждать исключения. Значения, возвращаемые асинхронными методами, могут иметь тип `void` или `java.util.concurrent.Future<V>`. Вследствие этого имеется два варианта использования: «запустить и забыть» (`void`) или «запустить и проверить ответ позднее» (`Future<V>`). Только методы, возвращающие значение

типа `Future<V>`, могут декларировать исключения. Когда асинхронный метод возбуждает исключение, оно перехватывается и повторно возбуждается при вызове метода `get()` объекта `Future`. Оригинальное исключение при этом будет завернуто в `ExecutionException`. Если исключение возбуждается асинхронным методом не имеющим возвращаемого значения (`void`), оно будет потеряно и вызывающий код никогда не узнает о нем.

Важно отметить, что транзакции не распространяют свое действие на асинхронные методы – при вызове асинхронного метода запускается новая транзакция. Напротив, действие механизма поддержки безопасности распространяется и на асинхронные методы. Подробнее о транзакциях и безопасности рассказывается в главе 6.

Когда вызывается асинхронный метод, управление тут же возвращается клиенту, еще до того, как произойдет фактический вызов метода. Если метод не имеет возвращаемого значения (метод возвращает тип `void`), с точки зрения клиента операция будет выполнена по сценарию «запустил и забыл». Если требуется получить результат асинхронной операции или возникнет необходимость отменить ее, асинхронный метод должен возвращать объект `Future`. В обоих случаях выполнение метода протекает в отдельном потоке. Это открывает невероятно широкие возможности. В следующем разделе мы познакомим вас с основными правилами – когда и как следует использовать асинхронные операции в приложениях.

3.5.2. Когда следует использовать асинхронные сеансовые компоненты

Асинхронные сеансовые компоненты следует использовать только в двух ситуациях:

- когда операция выполняется продолжительное время и желательно иметь возможность запустить ее и продолжить заниматься другими делами, не зависящими от того, что произойдет в процессе выполнения этой операции;
- когда операция выполняется продолжительное время и желательно иметь возможность запустить ее, а позже проверить результат или отменить операцию на полпути;

До появления спецификации EJB 3.1 единственным способом организовать асинхронную обработку была многопоточная реализация клиента. Поскольку управление конкуренцией осуществлялось контроллером, не было никакой возможности известить компонент о необходимости прервать выполнение продолжительной операции. Напомним, что когда конкуренцией управляет контейнер, каждый метод синхронизируется с компонентом, а это означает, что в каждый конкретный момент времени только один поток может вызвать метод. С выходом EJB 3.1 появились дополнительные возможности.

Асинхронные сеансовые компоненты, как предполагалось, должны быть максимально легковесными, поэтому они не дают никаких гарантий надежности. Из этого следует, что если контейнер потерпит аварию в тот момент, когда выпол-

няется асинхронный метод, этот метод не сможет восстановиться после ошибки. Кроме того, асинхронные компоненты не являются слабо связанными, то есть клиент хранит прямую ссылку на используемый им асинхронный компонент. Если одновременно требуется высокая надежность и слабая связанность, а не просто возможность асинхронного выполнения операций, следует подумать о возможности применения компонентов, управляемых сообщениями (Message-Driven Bean, MDB).

3.5.3. Пример компонента ProcessOrder

Чтобы увидеть пример практического применения асинхронных компонентов, вернемся к приложению ActionBazaar. В нем имеются две задачи, связанные с оформления заказа, которые могут рассматриваться как кандидаты на асинхронное выполнение: отправка подтверждения по адресу электронной почты и выполнение оплаты с кредитной карты. Отправка подтверждения относится к классу задач «запустил и забыл», поэтому данная операция ничего не возвращает. Обработка кредитной карты, напротив, хотя и может потребовать некоторого времени, но ее результат должен быть известен, прежде чем продолжить работу. Если в ходе перевода денег возникнет ошибка, необходимо тут же остановить процедуру и известить клиента, что перевод не был выполнен. Обе эти асинхронные операции запускаются сеансовым компонентом без сохранения состояния `ProcessOrder`, представленным в листинге 3.5.

Листинг 3.5. ProcessOrder

```
@Stateless(name = "processOrder")
public class OrderService {

    private String CONFIRM_EMAIL = "actionbazaar/email/confirmation.xhtml";

    @EJB
    private EmailService emailService;

    @EJB
    private BillingService billingService;

    public void processOrder(Order order) {
        String content;
        try {
            URL resource = OrderService.class.getResource(CONFIRM_EMAIL);
            content = resource.getContent().toString();
            // Асинхронно отправить подтверждение на электронную почту
            emailService.sendEmail(order.getEmailAddress(),
                "Winning Bid Confirmation",content);
            // Асинхронно перевести деньги с кредитной карты
            Future<Boolean> success =
              billingService.debitCreditCard(order);
              // Выполнить другие операции, такие как запись в базу данных
              if(!success.get()) {
                  // обработать ошибку
```

```
            }
        } catch (IOException ex) {
            Logger.getLogger(OrderService.class.getName()).log(Level.SEVERE,
                                null, ex);
        }
    }
}
```

Отправку уведомления по электронной почте реализует класс `EmailService`. В текст уведомления включается номер заказа для слежения за ходом его выполнения, а также сведения о порядке оплаты и товаре. Отправка электронного письма может потребовать некоторого времени, потому что при этом необходимо установить соединение с почтовым сервером. Определение этого класса приводится в листинге 3.6.

Листинг 3.6. Асинхронная отправка уведомления по электронной почте

```
@Stateless(name = "emailService")
public class EmailService {

    Logger logger = Logger.getLogger(AuthenticateBean.class.getName());

    @Resource(name="config/emailSender")
    private String emailSender;

    @Resource(name = "mail/notification")
    private Session session;

    // Сообщить контейнеру, что метод должен выполняться в отдельном потоке
    @Asynchronous
    // Ничего не возвращает - "запустил и забыл"
    public void sendEmail(String emailAddress, String subject,
        String htmlMessage) {
          try {
              MimeMessage message = new MimeMessage(session);
              message.setFrom(new InternetAddress(emailSender));
              InternetAddress[] toAddress = new InternetAddress[] {
                  new InternetAddress(emailAddress)};
              message.setRecipients(Message.RecipientType.TO, toAddress);
              message.setSubject(subject);
              message.setContent(createHtmlMessage(htmlMessage));
              Transport.send(message);
          } catch (Throwable t) {
              logger.log(Level.SEVERE, null,t);
          }
    }
    ...
}
```

Класс `BillingService`, представленный в листинге 3.7, осуществляет перевод денег с кредитной карты. Аннотация `@Asynchronous` перед определением класса означает, что все его общедоступные методы будут выполняться в отдельном

потоке. В компонент внедряется объект `SessionContext`, благодаря чему вызов `success.cancel(true)` будет автоматически зарегистрирован в контексте сеанса. Подробнее о `SessionContext` рассказывается в главе 5.

Листинг 3.7. BillingService

```
@Stateless(name="billingService")
@Asynchronous // Пометить класс как асинхронный
public class BillingService {

    // Внедрить SessionContext для регистрации отмены операции
    @Resource
    private SessionContext sessionContext;

    // Возвращаемое значение типа Future<Boolean>
    public Future<Boolean> debitCreditCard(Order order) {
        boolean processed = false;
        // Проверить факт отмены операции
        if(sessionContext.wasCancelCalled()) {
            return null;
        }
        // Выполнить операцию с кредитной картой
        // Завернуть результат в реализацию Future
        return new AsyncResult<Boolean>(processed);
    }
}
```

Теперь, когда вы увидели простой пример использования аннотации `@Asynchronous`, познакомимся с ней поближе, а затем перейдем к исследованию интерфейса `Future`.

3.5.4. Применение аннотации @Asynchronous

Аннотация `@Asynchronous` является одной из самых простых, среди рассматривавшихся в этой главе. Она не имеет параметров и может применяться и к классу, и к отдельным методам. Если аннотация применяется к классу, все методы этого класса будут вызываться асинхронно. Применение к отдельным методам позволяет сделать асинхронными только эти методы. Данную аннотацию можно использовать с любыми компонентами, представленными в этой главе: с сохранением состояния, без сохранения состояния и одиночкам. Она имеет следующее определение:

```
@Target(value = {ElementType.METHOD, ElementType.TYPE})
@Retention(value = RetentionPolicy.RUNTIME)
public @interface Asynchronous {
}
```

Применение данной аннотации к классу демонстрируется в листинге 3.5, а к отдельным методам – в листинге 3.6. Аннотация, самая простая часть механики асинхронных вызовов – интерфейс `Future` выглядит намного сложнее.

3.5.5. Применение интерфейса Future

Если результат вызова метода представляет определенный интерес, сценарий «запустил и забыл» не подходит. Чтобы иметь возможность получить результат асинхронного метода, из него следует вернуть объект, реализующий интерфейс `java.util.concurrent.Future`. Интерфейс `Future` имеет следующие методы:

- `boolean cancel(boolean mayInterruptIfRunning)` – отменяет операцию;
- `V get()` – возвращает значение и блокируется, пока результат не станет доступен;
- `V get(long timeout, TimeUnit unit)` – возвращает результат или `null`, если результат оказался недоступен в течение указанного интервала времени;
- `boolean isCancelled()` – возвращает `true`, если операция была отменена;
- `boolean isDone()` – возвращает `true`, если метод завершился.

С помощью интерфейса `Future` можно выполнять следующие действия:

- отменять операцию;
- извлекать результат вычислений, возможно с определением таймаута, если нет возможности ждать результатов слишком долго;
- проверять – завершилась ли операция или была ли она отменена.

Класс `javax.ejb.AsyncResult` был добавлен в Java EE 6 для удобства обертывания результатов, чтобы исключить необходимость создавать собственную реализацию интерфейса `Future`. Асинхронный метод создает экземпляр `AsyncResult` в составе которого передает вызывающему коду полученное значение, как показано в листинге 3.7. Выполняя продолжительную операцию, компонент может проверить, была ли текущая операция отменена посредством интерфейса `Future`, обратившись к статическому методу `SessionContext.wasCancelCalled()`. Если операция была отменена, метод должен прервать работу на этом, потому что результат операции все равно будет проигнорирован. Класс `AsyncResult` прощает возврат асинхронных данных.

3.5.6. Эффективное использование асинхронных сеансовых компонентов

Аннотация `@Asynchronous` обладает большой мощностью и в то же время потенциально опасна. Бесконтрольное использование этой аннотации может ухудшить производительность. Потоки выполнения являются достаточно тяжеловесными ресурсами и потому их максимальное число ограничено. Это число может определяться операционной системой. Контейнер приложения тоже может накладывать свои ограничения, определяя максимальный размер пула потоков. В любом случае, достижение верхнего предела влечет негативные последствия. Таким образом, данная возможность должна использоваться только когда это действительно

необходимо. Обратите особое внимание на утечки потоков – если поток терпит ошибку при попытке завершиться, он может продолжать крутиться вхолостую до завершения приложения.

С асинхронными методами связаны еще две более важные проблемы: поддержка возможности отмены и обработка исключений.

Поддержка возможности отмены

Асинхронные операции, возвращающие результат, должны поддерживать возможность отмены. Это позволяет клиентам прервать выполнение операции на полпути, что особенно важно для продолжительных операций. Внутри асинхронного метода компонента можно вызвать метод `wasCancelCalled()` объекта `SessionContext`, чтобы определить, не отменили ли операцию клиент посредством объекта `Future`. Этот метод следует вызывать в цикле и перед вызовом блокирующих методов.

Обработка исключений

Если асинхронная операция не возвращает экземпляр `Future` или клиент никогда не вызывает метод `get()` объекта `Future`, то клиент не узнает, возникло ли исключение вовремя выполнения асинхронного метода. Такой упрощенный подход следует использовать, только когда успех или неуспех выполнения асинхронной операции не играет большой роли. В ситуациях, когда требуется знать, преуспела ли асинхронная операция, она должна возвращать объект `Future`, а вызывающий код должен проверять его и обрабатывать ошибки. Напомним, что при использовании подхода «запустил и забыл», единственный способ узнать о возникновении ошибки во время выполнения асинхронного метода – заглянуть в файл журнала сервера приложений.

3.6. В заключение

В этой главе мы исследовали различные типы сеансовых компонентов и различия между ними. Мы также познакомились с компонентами-одиночками и новой поддержкой асинхронных методов, добавленной в EJB 3.1. Как рассказывалось в этой главе, сеансовые компоненты без сохранения состояния имеют самый простой жизненный цикл и могут быть организованы в пулы. Экземпляры компонентов с сохранением состояния создаются для каждого клиента, по этой причине они могут оказывать существенное влияние на потребление ресурсов. Кроме того, пассивация и активация сеансовых компонентов с сохранением состояния может отрицательно сказываться на производительности при неумеренном их использовании. Для компонентов-одиночек создается только один экземпляр. Вам так же предоставляется возможность выбирать способ управления конкуренцией – с использованием встроенных механизмов контейнера или вручную (на уровне компонента). Но даже выбирая управление конкуренцией на уровне контейнера, вы все еще можете подсказывать ему, какие методы используются для чтения, а какие

для записи: по умолчанию предполагается, что все методы выполняют операцию записи. Ближе к концу главы вы познакомились с асинхронными сеансовыми компонентами и узнали, как аннотировать методы, чтобы они выполнялись в отдельном потоке. С помощью интерфейса `Future` вы можете откладывать получение результата на потом. Асинхронные компоненты не являются четвертым типом сеансовых компонентов – аннотации, управляющие асинхронным выполнением можно применять к любым сеансовым компонентам: без сохранения состояния, с сохранением состояния и одиночкам.

Вы познакомились с различными типами прикладных интерфейсов. Сеансовые компоненты без сохранения состояния поддерживают локальные, удаленные и веб-интерфейсы. Веб-интерфейсы можно экспортировать посредством JAX-RPC (устаревший механизм), JAX-WS (SOAP) и JAX-RS (REST). Мы показали, насколько внедрение зависимостей упрощает использование EJB и позволяет избежать сложностей с поиском в JNDI.

В данный момент вы знаете все основы, необходимые для разработки прикладной логики вашего приложения с использованием сеансовых компонентов с сохранением и без сохранения состояния, а также компонентов-одиночек. В следующей главе мы обсудим, как конструировать приложения, обменивающиеся сообщениями с помощью компонентов, управляемых сообщениями.

ГЛАВА 4. Обмен сообщениями и разработка компонентов MDB

Эта глава охватывает следующие темы:

- основы компонентов, управляемых сообщениями (MDB);
- отправка и прием сообщений;
- аннотация `@JMSConnectionFactory`;
- классы `JMSContext`, `JMSConsumer` и `JMSProducer`.

В этой главе мы поближе рассмотрим проблемы разработки компонентов, управляемых сообщениями (Message-Driven Bean, MDB). Мы дадим краткий обзор этой мощной концепции и покажем на примере приложения ActionBazaar, как можно использовать эти компоненты для решения практических задач. Сначала мы познакомим вас с базовыми понятиями обмена сообщениями и займемся исследованием основ Java Messaging Service (JMS) на примере создания компонента, посылающего сообщения. Затем мы перейдем к компонентам MDB и посмотрим, как EJB 3 упрощает задачу создания компонентов-приемников сообщений.

Очень важно получить понимание особенностей обмена сообщениями и JMS прежде, чем углубляться в изучение MDB и тому есть две причины. Во-первых, большинство компонентов MDB, которые вы увидите, в действительности являются улучшенными компонентами-приемниками JMS, реализующими интерфейсы JMS (такие как `javax.jms.MessageListener`) и использующими JMS API (такие как `javax.jms.Message`). Во-вторых, в большинстве решений на основе MDB в обмен сообщениями будет входить не только обработка входящих сообщений.

Если вы хорошо знакомы с приемами обмена сообщениями и JMS, можете сразу перейти к разделу, охватывающему компоненты MDB. Однако иногда все же полезно освежить свои знания.

4.1. Концепции обмена сообщениями

Когда мы говорим об обмене сообщениями в контексте Java EE, мы в действительности подразумеваем процесс отправки слабо связанных, асинхронных сообщений, которые передаются с применением надежных механизмов. Большинство взаимодействий между компонентами – например, между сеансовыми компонентами без сохранения состояния – являются синхронными и образующими тесные связи. Компонент-отправитель непосредственно вызывает метод компонента-получателя и ожидает завершения метода. Для успешного взаимодействия должны присутствовать обе стороны. В случае обмена сообщениями дело обстоит иначе: отправитель не знает, когда сообщение будет получено, и даже не может указать, какой компонент должен обработать то или иное сообщение. Зато механизм обмена сообщениями в Java EE обеспечивает надежную доставку и гарантирует, что сообщение не будет потеряно на пути от отправителя к получателю.

В качестве аналогии представьте себе телефонный звонок. Если на звонок отвечает человек – это синхронное взаимодействие. Если к телефону никто не подходит, вы можете оставить голосовое сообщение – это асинхронное взаимодействие. Ваше сообщение сохраняется и получатель может позднее прослушать его. Вы не знаете, когда человек прослушает сообщение, вы даже не знаете, кто первым его прослушает. Систему голосовой почты в данном контексте можно считать промежуточной службой, ориентированной на сообщения (Message-Oriented Middleware, MOM). Она выступает в роли посредника между отправителем и получателем сообщения, освобождая их от необходимости прямого присутствия при взаимодействии. В этом разделе мы коротко представим вам MOM, покажем, как можно использовать механизм обмена сообщениями в приложении ActionBazaar и исследуем три модели обмена сообщениями.

4.1.1. Промежуточное ПО передачи сообщений

MOM (Message-Oriented Middleware) – это программное обеспечение, гарантирующее надежную доставку сообщений между разными системами. Основное его назначение состоит в том, чтобы обеспечить интеграцию различных систем, часто оставшихся в наследство. Инфраструктура MOM может использоваться, например, для передачи сообщений между веб-интерфейсом и системой, написанной на языке COBOL и выполняющейся на большой ЭВМ. Когда выполняется отправка сообщения, программное обеспечение MOM сохраняет его в месте, известном отправителю и получателю. Отправитель сообщения называется *производителем* (producer), а местоположение, где будет храниться сообщение – *адресом* (destination). Позднее любой программный компонент, заинтересованный в получении сообщений, по указанному адресу сможет получить непрочитанные сообщения. Программные компоненты, принимающие сообщения, называются *потребителями* (consumers). На рис. 4.1 изображены различные компоненты MOM.

Системы MOM не являются чем-то новым – они существуют с начала 1980-х годов. Системы MOM создавались с целью упростить интеграцию и уменьшить

объем нестандартного кода, осуществляющего эту интеграцию. Современным системам все еще приходится взаимодействовать с другими системами – и старыми, и новыми. В настоящее время на рынке имеется множество различных реализаций MOM, и все современные серверы Java EE включают собственные реализации систем обмена сообщениями.

Рис. 4.1. Типичный сценарий работы MOM. Когда производитель отправляет сообщение промежуточному ПО, оно сохраняется и позднее выбирается потребителем

Чтобы получить еще более полное представление о том, как производится обмен сообщениями, исследуем этот вопрос на примере приложения ActionBazaar. Эту тему мы продолжим рассматривать на протяжении оставшейся части главы.

4.1.2. Обмен сообщениями в ActionBazaar

ActionBazaar не является законченной и самодостаточной системой. Задача этого приложения – управление торгами на аукционе, а другие задачи, такие как учет и доставка, решаются другими, специализированными приложениями. Компания ActionBazaar может решить приобрести и интегрировать в свою систему службы сторонних компаний, вместо того, чтобы создавать что-то свое. Например, ActionBazaar пользуется услугами Turtle Shipping Company, осуществляющей доставку товаров покупателям. Когда ставка клиента выигрывает на аукционе, через соединение «бизнес–бизнес» (business-to-business, B2B) в систему компании Turtle посылается запрос на доставку. Первоначально все службы других компаний были интегрированы синхронным способом. На рис. 4.2 показано, как выглядит эта интеграция и какими проблемами она страдает.

В синхронной модели пользователю приходилось ждать, пока запрос на доставку будет обработан внешней службой. Проблема в том, что соединение типа B2B часто оказывалось медленным и ненадежным. Кроме того, системы компании Turtle сами по себе медленнее и более нагруженные, в сравнении с серверами ActionBazaar. К тому же, некоторые системы Turtle в свою очередь используют услуги третьих систем. Как нетрудно догадаться, длительные задержки раздражали клиентов!

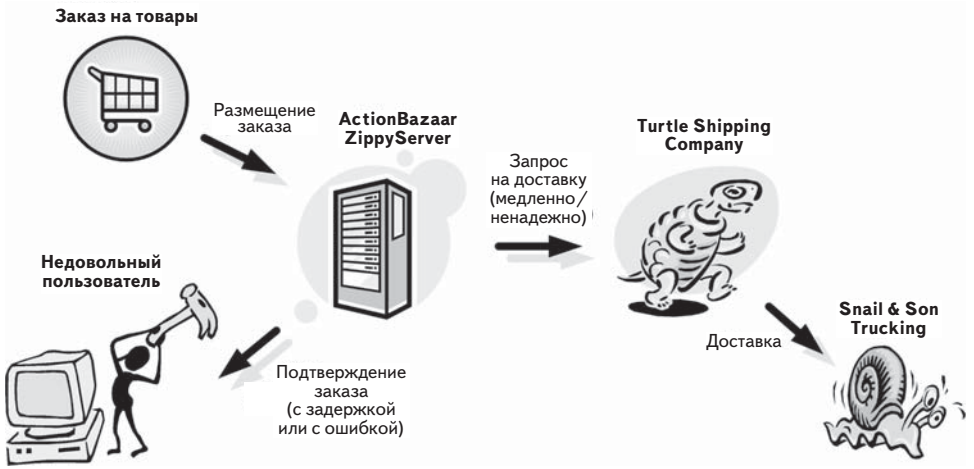

Рис. 4.2. Схема оформления заказа в ActionBazaar до внедрения MOM

Решение этой проблемы заключено в асинхронной интеграции со всеми внешними системами посредством механизма обмена асинхронными сообщениями. На рис. 4.3 показано, как выглядит эта интеграция. В этой модели между системами ActionBazaar и Turtle помещается MOM. Сообщение, содержащее запрос на доставку, отправляется системе MOM, где оно хранится, пока система Turtle не примет и не обработает его. И сразу после отправки сообщения, которая обычно выполняется почти мгновенно, пользователю высылается подтверждение об оформлении заказа.

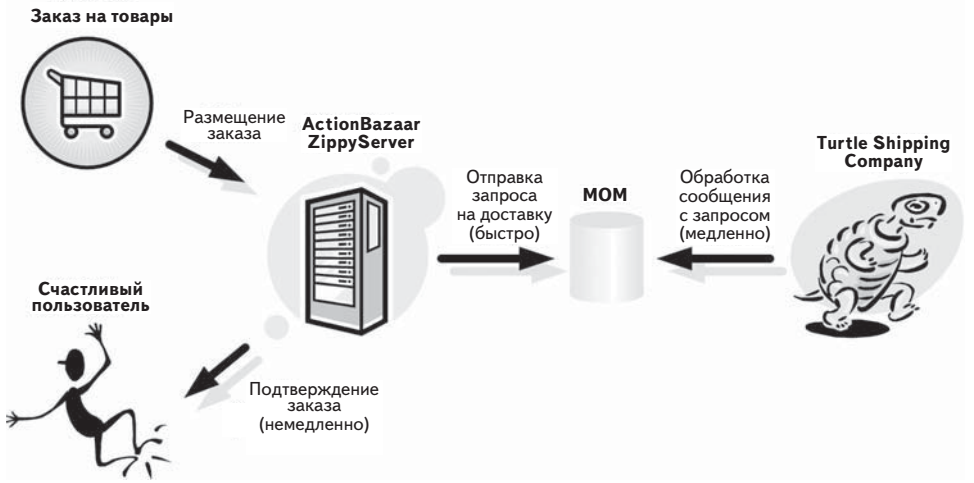

Рис. 4.3. Схема оформления заказа в ActionBazaar после внедрения MOM
Механизмы обмена сообщениями увеличивает скорость ответа клиенту и надежность обработки

В такой ситуации внедрение механизма обмена сообщениями увеличивает надежность, устраняя тесную связь между процедурой подтверждения заказа и фактической обработкой запроса на доставку. Надежность обусловлена тем, что в такой схеме не требуется, чтобы серверы ActionBazaar и Turtle работали одновременно. Кроме того, от серверов не ожидается, что они будут работать в одном темпе. Благодаря тому, что системы MOM хранят сообщения и повторно отправляют их в случае разрыва соединения на полпути, запросы на доставку не будут теряться, если в какой-то момент серверы компании Turtle окажутся недоступными. Кроме того, системы MOM повышают надежность, поддерживая транзакции и подтверждения о приеме и доставке сообщений.

Забегая вперед, заметим, что устранение тесной связи несет и другие преимущества. Например, К серверу MOM могут подключаться и другие компании, осуществляющие доставку, вместо или вместе с компанией Turtle. Это уменьшает зависимость ActionBazaar от единственной транспортной компании, и заказы могут обрабатываться любыми такими компаниями, доступными в данный момент.

4.1.3. Модели обмена сообщениями

Модель обмена сообщениями – это шаблон, определяющий порядок обмена сообщениями между отправителями и получателями. Механизм JMS поддерживает две стандартные модели: точка–точка (Point-To-Point, PTP) и издатель–подписчик (publish–subscribe). Приложение может использовать какую-то одну или обе модели сразу – в зависимости от того, как должен протекать обмен сообщениями. Обсудим каждую из моделей.

Точка–точка

В схеме точка–точка (PTP), единственное сообщение передается от единственного производителя (точка А) единственному потребителю (точка Б). В системе может существовать множество производителей и потребителей, но только один потребитель сможет обработать каждое конкретное сообщение. Адреса в схеме PTP называют *очередями* (*queues*). Производитель записывает сообщение в очередь, а потребитель извлекает его из очереди. Схема PTP не гарантирует доставку сообщений в каком-то определенном порядке – термин *очередь* здесь носит чисто символический характер. Если сообщение может быть прочитано множеством потенциальных потребителей, выбор потребителя производится почти что случайно, как показано на рис. 4.4.

Классическая аналогия «записки в бутылке» отлично описывает модель PTP. Потерпевший кораблекрушение (производитель) бросает в море бутылку с запиской. Море (очередь) переносит бутылку и выбрасывает на берег, где ее находит случайный прохожий (потребитель). Бутылка с запиской может быть «найдена» только однажды.

Передача запроса на доставку в приложении ActionBazaar как раз реализована с использованием модели PTP, потому что доставку товара клиенту может быть выполнена только одной транспортной компанией.

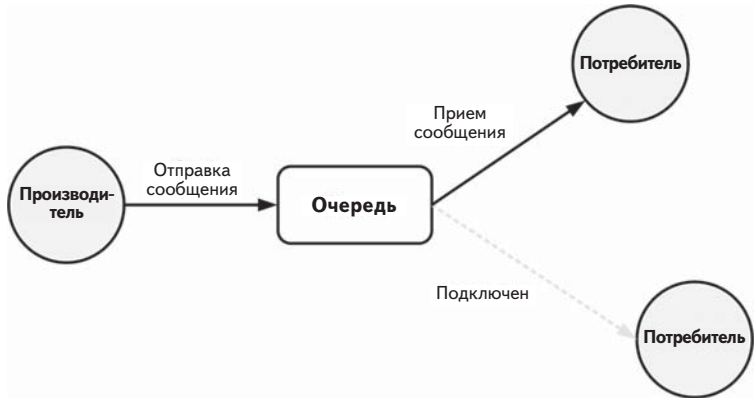

Рис. 4.4. Модель PTP с одним производителем и двумя потребителями

Издатель–подписчик

Модель издатель–подписчик напоминает традиционную службу доставки газет. Как показано на рис. 4.5, единственный производитель генерирует сообщение, которое может принять любое число потребителей, которые подключены в этот момент к адресу. Адреса сообщений в данной модели называют *темами* (topics), а потребители – *подписчиками*. Модель издатель–подписчик особенно хорошо подходит для рассылки широковещательных сообщений. Например, ее можно использовать для рассылки уведомлений об остановке системы на профилактику всем остальным поставщикам услуг, интегрированных в приложение ActionBazaar и готовых принять сообщение в данный момент.

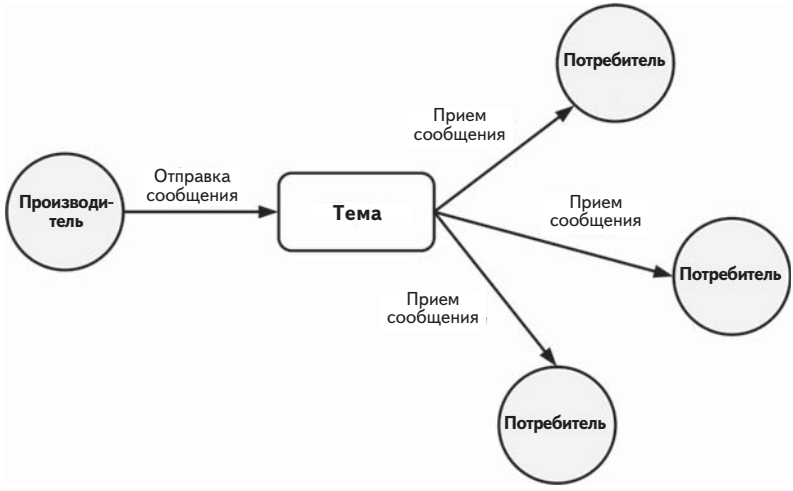

Рис. 4.5. Модель издатель–подписчик с одним производителем и тремя потребителями. Каждый подписавшийся на тему получит свою копию сообщения

Итак, вы получили начальное представление об обмене сообщениями и наверняка готовы погрузиться в программный код. В следующем разделе мы познакомимся с механизмом JMS и реализуем производителя, который будет использоваться в приложении ActionBazaar для отправки сообщений.

Модель запрос–ответ

Возможно у вас появится желание реализовать в приложении ActionBazaar прием подтверждения от службы Turtle, как только она извлечет сообщение из очереди.

В таких ситуациях удобно использовать модель запрос–ответ. В этой модели требуется включить в сообщение дополнительную информацию, чтобы потребитель знал, куда и как послать подтверждение. Такую модель часто называют многоуровневой, потому что обычно она реализуется поверх одной из двух моделей, описанных выше.

Например, в модели PTP отправитель указывает очередь, куда следует послать ответ (в JMS такая очередь называется очередью для ответа), а также уникальный идентификатор, общий для исходящего и входящего сообщений (в JMS называется согласующим идентификатором (correlation ID)). Получатель принимает исходное сообщение и отправляет ответ в очередь для ответов, копируя согласующий идентификатор. Отправитель принимает сообщение из очереди для ответов и по согласующему идентификатору определяет, в ответ на какое сообщение было получено подтверждение.

4.2. Введение в JMS

В этом разделе мы дадим краткий обзор JMS API на примере создания простого производителя сообщений. JMS – это обманчиво скромный API, открывающий доступ к очень мощной технологии. Прикладной интерфейс JMS API используется механизмом Java Database Connectivity (JDBC) для доступа к базам данных. JMS предоставляет стандартный способ доступа к системам MOM из программ на Java и служит великолепной альтернативой другим узкоспециализированным API.

Изучение программного кода – самый простой способ познакомиться с JMS. Далее мы займемся исследованием JMS на примере реализации кода для ActionBazaar, отправляющего сообщения с запросами на доставку. Как уже говорилось в разделе 4.1.2, когда пользователь размещает заказ, соответствующий запрос отправляется в очередь, совместно используемую системами ActionBazaar и Turtle. Код, представленный в листинге 4.1, осуществляет отправку сообщения и может быть частью метода простой службы, вызываемой приложением ActionBazaar. Вся необходимая информация о доставке – такая как номер товара, адрес доставки, способ доставки и страховая сумма – включена в сообщение, которое помещается в очередь `ShippingRequestQueue`.

Листинг 4.1. Фрагмент кода, осуществляющий отправку запроса из приложения ActionBazaar

```
@Inject // ❶ Простая аннотация CDI
// ❷ Используемая фабрика очередей
@JMSConnectionFactory("jms/QueueConnectionFactory")
```

```
private JMSContext context;                    // ❸ Упрощенный интерфейс JMS

@Resource(name="jms/ShippingRequestQueue") // ❹ Используемая
private Destination destination;           //   очередь

ShippingRequest shippingRequest = new ShippingRequest(); // ❺ Запрос
shippingRequest.setItem("item");                         //   ShippingRequest
shippingRequest.setShippingAddress("address");           //   посылаемый
shippingRequest.setShippingMethod("method");             //   в
shippingRequest.setInsuranceAmount(100.50);              //   сообщении

ObjectMessage om = context.createObjectMessage();  // ❻ JMSObjectMessage для
om.setObject(shippingRequest);                     // отправки ShippingRequest

JMSProducer producer = context.createProducer();   // ❼ JMSProducer,
producer.send(destination, om);                    //   выполняющий отправку
```

По мере описания каждого этапа, выполняемого этим кодом, мы познакомимся со значительной частью JMS API и и шаблонами использования. Для простоты из этого примера мы удалили код, осуществляющий обработку исключений.

Получение фабрики соединений и адреса

В JMS ресурсы сервера сообщений напоминают объекты `javax.sql.DataSource` в JDBC. Эти ресурсы хранятся в реестре JNDI, они создаются и настраиваются за пределами программного кода, обычно посредством конфигурационных файлов XML или административной консоли. В JMS имеется два основных типа ресурсов: `javax.jms.JMSContext` ❸ и `javax.jms.Destination` ❹, которые оба используются в листинге 4.1. Ссылка на объект `JMSContext` приобретается с помощью механизма внедрения зависимостей, с помощью CDI-аннотации `@Inject` ❶, а настройка `JMSContext` на соединение с фабрикой выполняется с помощью аннотации `@JMSConnectionFactory` ❷. Объект `JMSContext` обертывает `javax.jmx.ConnectionFactory` и `javax.jmx.Session` в один объект. Класс `JMSContext` был введен в состав EE 7 для упрощения JMS-операций и для ослабления ограничения «один-сеанс-на-соединение» в окружениях EE (это ограничение отсутствует в приложениях SE). Далее с помощью аннотации `@Resource` ❹ выполняется внедрение очереди для отправки сообщений, имя которой `ShippingRequestQueue` в точности соответствует назначению.

Этот пример кода взят из сервлета, использующего аннотации для внедрения зависимостей. Контейнер сервлета автоматически отыскивает ресурсы, зарегистрированные в JNDI, и внедряет их. Подробнее о внедрении зависимостей рассказывается в главе 5.

Подготовка сообщения

В этом примере требуется отправить компании Turtle сериализуемый Java-объект `ShippingRequest`. Для этого создается новый экземпляр объекта и заполняется необходимыми данными ❺. После этого требуется указать тип JMS-сообщения, чтобы отправить этот объект. Наиболее подходящим типом является `javax.jms.ObjectMessage`. С помощью `JMSContext` мы создаем объект `ObjectMessage`

и включаем в него объект `ShippingRequest` ❻. Теперь, когда сообщение готово к отправке, остается только отправить его.

Отправка сообщения

Отправка сообщения осуществляется с помощью `JMSProducer`. В процессе подготовки сообщения мы получили `JMSContext`, который предоставляет возможность создания объекта-производителя. Достаточно просто вызвать метод `createProducer` и отправить только что созданное сообщение ❼. Что может быть проще! Осталось решить еще один вопрос: следует ли после отправки сообщения освобождать использовавшиеся ресурсы?

Освобождение ресурсов

Самое замечательное в механизме внедрения зависимостей, поддерживаемом управляемым окружением, состоит в том, что контейнер автоматически решает многие важнейшие задачи. Освобождение ресурсов – одна из них. Предыдущий фрагмент кода взят из сервлета, соответственно `JMSContext` является ресурсом, управляемым контейнером. Это означает, что контейнер автоматически закроет ресурс. Однако `JMSContext` имеет метод `close`. При получении `JMSContext` из неуправляемого окружения – например, создаваемого с помощью `ConnectionFactory` в приложении Java SE – придется явно освободить ресурс вызовом метода `JMSContext.close()` или воспользоваться интерфейсом `AutoCloseable`, добавленным в Java SE 7.

Если все пройдет успешно, сообщение окажется в очереди. Прежде чем перейти к знакомству с реализацией потребителя, принимающего сообщения, давайте поближе рассмотрим объект `javax.jms.Message`.

4.2.1. Интерфейс JMS Message

Интерфейс `Message` стандартизует разные типы сообщений, которые могут передаваться между разными провайдерами JMS. Как показано на рис. 4.6, сообщение JMS состоит из следующих элементов: заголовок сообщения, свойства сообщения и тело сообщения. Все эти составляющие описываются в следующих разделах.

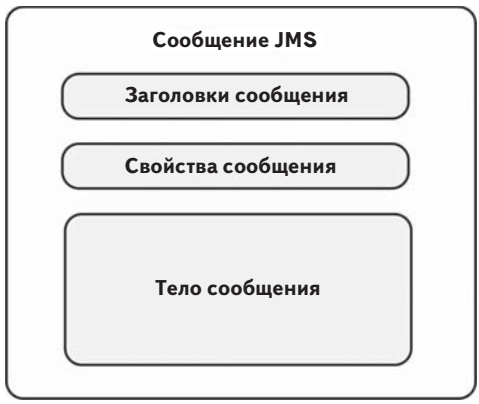

Рис. 4.6. Анатомия сообщения JMS

Отличной аналогией сообщений JMS может служить обычный почтовый конверт. Давайте посмотрим, в чем проявляется эта аналогия.

Заголовки сообщения

Заголовки – это пары имя/значение, общие для всех сообщений. Если следовать аналогии с конвертом, заголовки сообщения содержат стандартную информацию, которую можно найти на обычном конверте: адреса отправителя и получателя, почтовая марка и штемпель. Например, аналогом штемпеля на сообщении JMS является заголовок `JMSTimestamp`. Система MOM сохраняет в этом заголовке время отправки сообщения.

Ниже перечислены некоторые наиболее часто используемые заголовки JMS:

- `JMSCorrelationID`;
- `JMSReplyTo`;
- `JMSMessageID`.

Свойства сообщений

Свойства сообщений схожи с заголовками, но в отличие от последних не являются стандартными и должны явно устанавливаться приложением. Продолжая аналогию с почтовым конвертом: если вы решите написать на конверте «С праздником!», чтобы тем самым сообщить получателю, что письмо содержит поздравления, этот текст должен быть оформлен как свойство, а не заголовок. В приложении ActionBazaar, к примеру, запрос на доставку можно снабдить логическим свойством `boolean Fragile` и установить его в значение `true`, чтобы сообщить транспортной компании, что товар хрупкий и требует осторожного обращения:

```
Message.setBooleanProperty("Fragile", true);
```

Свойства могут иметь тип `boolean`, `byte`, `double`, `float`, `int`, `long`, `short`, `String` или `Object`.

Тело сообщения

Тело сообщения содержит содержимое конверта – это «полезный груз» сообщения. Содержимое тела сообщения определяет тип этого сообщения. В листинге 4.1 использовался тип `javax.jms.ObjectMessage`, потому что требовалось отправить Java-объект `ShippingRequest`. Можно также использовать типы `BytesMessage`, `MapMessage`, `StreamMessage` и `TextMessage`. Каждый из этих типов сообщений имеет свой интерфейс и шаблон использования. Не существует каких-то жестких правил, определяющих тип сообщения. Исследуйте все возможные варианты, прежде чем сделать выбор.

Мы только что закончили обзор наиболее важных составляющих механизма JMS, включая отправку сообщения, элементы сообщения и разные типы сообщений. Исчерпывающее знакомство с JMS, к сожалению, далеко выходит за рамки этой главы и книги. Дальнейшее исследование JMS вы сможете продолжить по адресу: http://docs.oracle.com/javaee/7/tutorial/doc/partmessaging.htm. Итак, поз-

накомившись поближе с сообщениями JMS, можно перейти к изучению кода, осуществляющего прием сообщений на сервере Turtle.

4.3. Использование компонентов MDB

Теперь займемся детальным исследованием компонентов, управляемых сообщениями (Message-Driven Bean, MDB). Сначала рассмотрим предпосылки к их использованию, а затем погрузимся в практику использования MDB. Попутно мы обсудим некоторые приемы, используемые при разработке компонентов MDB и поджидающие нас ловушки.

MDB – это компоненты EJB, предназначенные для приема асинхронных сообщений. Хотя компоненты MDB способны обрабатывать самые разные виды сообщений (см. врезку «Соединители JCA и обмен сообщениями» ниже), мы в первую очередь сосредоточимся на применении компонентов MDB для обработки сообщений JMS, потому что именно для этого они используются чаще всего. Кто-то из вас может задаться вопросом: зачем вообще развертывать компоненты EJB для решения задачи приема сообщений или почему следует использовать механизм JMS. Мы ответим на этот вопрос ниже. Мы создадим простое приложение-потребитель сообщений, использующее компоненты MDB, и покажем вам, как пользоваться аннотацией `@MessageDriven`. Вы также познакомитесь с интерфейсом `MessageListener`, с настройками активации и жизненным циклом MDB.

Соединители JCA и обмен сообщениями

Несмотря на то, что JMS является основным механизмом обмена сообщениями для MDB, в версии EJB 2.1 у него появились конкуренты. Благодаря архитектуре Java EE Connector Architecture (JCA), компоненты MDB могут принимать сообщения из любых корпоративных информационных систем (Enterprise Information System, EIS), таких как PeopleSoft или Oracle Manufacturing, а не только из систем MOM, поддерживающих JMS.

Представьте, что вам в наследство досталось приложение, в котором необходимо организовать передачу сообщений компонентам MDB. Добиться этого можно, реализовав JCA-совместимый адаптер/соединитель (adapter/connector), включающий поддержку *соглашений по внедрению* сообщений (*message inflow contract*). После развертывания ресурса адаптера или соединителя JCA в контейнере Java EE вы можете использовать соглашения по внедрению сообщений для доставки асинхронных сообщений в конечную точку внутри контейнера. Конечная точка (endpoint) JCA по сути то же самое, что и адрес JMS – она служит своеобразным прокси-сервером к MDB (потребителем сообщений, если говорить в терминах JMS). Как только сообщение достигнет конечной точки, контейнер активизирует зарегистрированные компоненты MDB, подключенные к этой конечной точке, и передает им сообщение.

Со своей стороны компонент MDB реализует интерфейс приемника (listener), соответствующий типу соединителя/адаптера JCA, передает параметры настройки активации соединителю JCA и регистрируется в нем как приемник сообщений (приемники сообщений и параметры настройки активации обсуждаются чуть ниже). JCA также позволяет интегрировать механизмы MOM с контейнерами Java EE, используя для этого JCA-совместимые соединители или адаптеры. За дополнительной информацией обращайтесь к документу «JCA JSR for Java EE 7» по адресу: http://jcp.org/en/jsr/detail?id=322.[1]

[1] Из материалов по теме JCA на русском языке можно порекомендовать цикл из трех статей: http://www.ibm.com/developerworks/ru/library/j-jca1/index.html. – *Прим. перев.*

4.3.1. Когда следует использовать обмен сообщениями и компоненты MDB

Обмен сообщениями и компоненты MDB – мощный инструмент, но, как и всякий инструмент, они имеют свою область применения. Компоненты MDB следует использовать, только если действительно необходима асинхронная обработка, слабая связанность *и* высокая надежность. То есть, если хотя бы одна из этих трех характеристик не является обязательной, вероятно не стоит использовать MDB.

Если вам просто нужно организовать асинхронную обработку, без предъявления требований к высокой надежности или слабой связанности, подумайте о возможности применения сеансовых компонентов с аннотацией `@Asynchronous`, как описано в главе 3. Аннотация `@Asynchronous` очевидно не обеспечивает слабой связанности, потому что из клиентского кода выполняется непосредственный вызов асинхронного сеансового компонента. Ненадежность асинхронных методов сеансовых компонентов не так очевидна, но она имеет место: если контейнер столкнется с ошибкой в середине асинхронного метода, сеансовый компонент будет утрачен. В случае с MDB, напротив, сообщение JMS не удаляется из промежуточного хранилища, пока компонент MDB не завершит его обработку. Если во время обработки в компоненте MDB возникнет исключение, сообщение может быть обработано повторно, когда контейнер будет готов к этому.

Аналогично, если требуется только слабая связанность, обратите свои взоры в сторону событий CDI. Как рассказывается в главе 12, события CDI позволяют эффективно отделять друг от друга потребителей и производителей сообщений с помощью событий на основе объектов. Подобно асинхронным сеансовым компонентам, шина событий CDI не является отказоустойчивой, к тому же она не является асинхронной. Обратите внимание, что при необходимости можно комбинировать аннотацию `@Asynchronous` с событиями CDI.

Когда необходимы все три составляющие – асинхронность, слабая связанность и надежность – компоненты MDB оказываются одним из лучших решений. Именно поэтому они так часто используются для интеграции систем. Давайте рассмотрим причины такого успеха.

4.3.2. Почему следует использовать MDB?

В большинстве корпоративных приложений требуется некоторая форма обмена сообщениями. В этом разделе мы рассмотрим некоторые причины, объясняющие необходимость включения поддержки сообщений в ваши приложения. Вы можете обнаружить, что восполнили эту потребность каким-то другим способом, но теперь, после знакомства с MDB, вы увидите, что они представляют намного более изящное решение.

Многопоточность

Вашему бизнес-приложению могут потребоваться многопоточные потребители сообщений, способные параллельно обрабатывать по нескольку сообщений и тем

самым обеспечить максимальную пропускную способность. Компоненты MDB помогут избежать лишних сложностей, потому что они изначально поддерживают многопоточность, без необходимости писать дополнительный код. Они управляют распределением входящих сообщений между множеством экземпляров компонентов (в пуле), которые сами не имеют специальной поддержки выполнения в многопоточной среде. Как только сообщение достигнет указанного адреса, из пула извлекается экземпляр MDB для его обработки, как показано на рис. 4.7. Этот прием называется *организацией компонентов MDB в пул* (MDB pooling) – мы познакомимся с ним далее в этой главе, когда будем обсуждать жизненный цикл компонентов MDB.

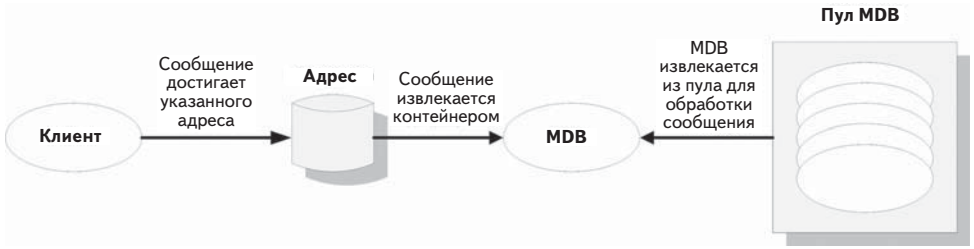

Рис. 4.7. Как только сообщение достигнет указанного адреса, контейнер извлечет его и передаст соответствующему экземпляру MDB из пула

Простота использования

Компоненты MDB избавляют от механического кодирования типовых операций с сообщениями JMS, таких как: получение фабрик соединений или адресов, создание соединений, открытие сеансов, создание потребителей и регистрация приемников. Как вы увидите в процессе реализации компонента MDB для компании Turtle, все эти задачи решаются автоматически, без вашего прямого участия. Использование в EJB 3 рациональных значений по умолчанию устраняет от необходимости писать большие конфигурационные файлы. В худшем случае вам придется указать конфигурационную информацию в простых аннотациях или в дескрипторе развертывания.

Надежность обработки сообщений

Как отмечалось выше, надежность является важнейшей характеристикой MDB. Все компоненты MDB по умолчанию пользуются транзакциями и подтверждают окончание обработки сообщений. Как следствие, сообщения не удаляются сервером сообщений, пока метод компонента MDB, вызванный для обработки сообщения, не завершится благополучно. Если в процессе обработки возникнет ошибка, будет выполнен откат транзакции и получение сообщения не будет подтверждено. Так как компоненты MDB пользуются транзакциями JTA, любые изменения в базе данных, произведенные в ходе обработки сообщения, также автоматически будут отменены. Как результат, неподтвержденное сообщение может быть повторно передано для

обработки другому компоненту MDB. В случае успешной обработки изменения в базе данных будут подтверждены, а сообщение будет удалено сервером сообщений, причем все эти операции будут выполнены атомарно, в рамках одной транзакции.

Запуск механизма приема сообщений

Чтобы запустить механизм приема сообщений из очереди с запросами на доставку, кто-то должен вызвать соответствующий метод в вашем коде. Не совсем понятно, как этого добиться в промышленном окружении. Очевидно, что возлагать запуск механизма на пользователя было бы нежелательно. В серверном окружении практически каждый способ вызова метода на запуске сервера сильно зависит от конкретных условий. То же верно и в отношении остановки механизма обработки сообщений вручную. С другой стороны, зарегистрированные компоненты MDB можно было бы активизировать или уничтожать с помощью контейнера, в момент запуска или остановки сервера.

Мы продолжим обсуждение этой проблемы ниже, когда приступим к исследованию действующего примера компонента MDB.

4.3.3. Разработка потребителя сообщений с применением MDB

Давайте теперь займемся созданием компонента MDB и реализуем сервер приема сообщений для транспортной компании Turtle. В листинге 4.2 приводится код реализации компонента MDB, который извлекает из очереди запросы на доставку и сохраняет каждый запрос в базе данных компании Turtle, в таблице `SHIPPING_REQUEST`. Чтобы упростить пример, для сохранения данных в базе в нем используется Java Persistence API (JPA). Подробнее механизм JPA мы начнем обсуждать в главе 9. Применение JPA также поможет нам продемонстрировать жизненный цикл MDB и особенности управления транзакциями: если при обработке сообщения возникнет ошибка, будет выполнен откат транзакции, данные не сохранятся в базе и сообщение не будет удалено из очереди.

Листинг 4.2. Компонент обработки сообщений для компании Turtle

```
@MessageDriven(activationConfig = {                              // ❶ Настройка
  @ActivationConfigProperty(propertyName = "destinationType", //   очереди
    propertyValue = "javax.jms.Queue"),                          //   сообщений
  @ActivationConfigProperty(propertyName = "destinationLookup",//   в
    propertyValue = "jms/ShippingRequestQueue")                  //   компоненте
})

// ❷ Реализует интерфейс MessageListener для обработки сообщений JMS
public class TurtleShippingRequestMessageBean implements MessageListener{

  @PersistenceContext()          // Внедрение диспетчера EntityManager
  EntityManager entityManager;   // для сохранения данных в базе

  @Override
```

```
  public void onMessage(Message message){ // ❸ Реализация метода onMessage
    try {
      ObjectMessage om = (ObjectMessage) message; // ❹ Извлечение
      Object o = om.getObject();                  //    данных
      ActionBazaarShippingRequest sr =            //    отправленных из
          (ActionBazaarShippingRequest) o;        //    ActionBazaar

      Logger.getLogger(TurtleShippingRequestMessageBean.class.getName())
          .log(Level.INFO, String.format("Got message: %s", sr));

      // ❺ Преобразование данных из ActionBazaar в данные Turtle
      TurtleShippingRequest tr = new TurtleShippingRequest();
      tr.setInsuranceAmount(sr.getInsuranceAmount());
      tr.setItem(sr.getItem());
      tr.setShippingAddress(sr.getShippingAddress());
      tr.setShippingMethod(sr.getShippingMethod());

      // ❻ Сохранение запроса в базе данных Turtle
      entityManager.persist(tr);
    } catch (JMSException ex) {
      Logger.getLogger(TurtleShippingRequestMessageBean.class.getName())
          .log(Level.SEVERE, null, ex);
    }
  }
}
```

Аннотация `@MessageDriven` ❶ превращает данный объект в компонент MDB и определяет его настройки, включая тот факт, что он подключается к очереди `jms/ShippingRequestQueue`. Вспомните, как во врезке «Соединители JCA и обмен сообщениями» (выше) говорилось, что компоненты MDB могут принимать и другие типы уведомлений, отличные от сообщений JMS. Но в данном примере, как видно из листинга, компонент MDB реализует интерфейс `MessageListener` ❷. Для реализации интерфейса `MessageListener` необходимо определить метод `onMessage` ❸. Сам метод достаточно прост. Сначала он извлекает объект `ActionBazaarShippingRequest` из сообщения `ObjectMessage` ❹. Напомним, что объект `ActionBazaarShippingRequest` содержит данные, отправленные приложением ActionBazaar транспортной компании Turtle, составляющие запрос на доставку. Затем данные в объекте `ActionBazaarShippingRequest` преобразуются в объект `TurtleShippingRequest` ❺. Преобразование данных, как в данном примере, часто используется на практике с целью сохранить их в базе данных для последующей обработки. В заключение в работу включается `EntityManager` для сохранения данных в базу ❻. Напомним, что более подробное обсуждение диспетчера сущностей `EntityManager` и механизма JPA начнется в главе 9.

Далее мы займемся исследованием основных возможностей MDB, более детально проанализировав данный пример, и начнем мы с аннотации `@MessageDriven`.

4.3.4. Применение аннотации @MessageDriven

Компоненты MDB являются одними из самых простых компонентов EJB с точки зрения разработчика, и поддерживают самое маленькое чис-

ло аннотаций. Аннотация `@MessageDriven` и вкладываемая в нее аннотация `@ActivationConfigProperty` являются единственными, имеющими отношение к компонентам MDB. Аннотация `@MessageDriven` определена следующим образом:

```
@Target(value = {ElementType.TYPE})
@Retention(value = RetentionPolicy.RUNTIME)
public @interface MessageDriven {
  String name() default "";
  Class messageListenerInterface default Object.class;
  ActivationConfigProperty[] activationConfig() default {};
  String mappedName();
  String description();
}
```

Отметьте, что все три параметра аннотации являются необязательными. Минималисты могут сохранить аннотацию в коде максимально простой, как показано ниже, и добавлять настройки где-то в другом месте, например в дескрипторе развертывания (или просто использовать значения по умолчанию):

```
@MessageDriven
public class TurtleShippingRequestMessageBean
```

Первый параметр, `name`, определяет имя компонента MDB на случай, если потребуется явно связать его. Если аргумент `name` отсутствует, по умолчанию используется имя класса, в данном случае: `TurtleShippingRequestMessageBean`. Второй параметр, `messageListenerInterface`, определяет, какой интерфейс приемника сообщений реализует компонент MDB. Третий параметр, `activationConfig`, используется для определения настроек приемника конкретного типа. Рассмотрим подробнее два последних параметра.

4.3.5. Реализация интерфейса MessageListener

Контейнер использует интерфейс `MessageListener`, реализованный компонентом MDB, для регистрации этого компонента MDB с помощью провайдера JMS. После регистрации провайдер сможет передавать входящие сообщения компоненту, вызывая его методы. Параметр `messageListenerInterface` аннотации `@MessageDriven` – это единственный способ определить интерфейс `MessageListener`. Следующий фрагмент демонстрирует, как это делается:

```
@MessageDriven(
  messageListenerInterface="javax.jms.MessageListener")
public class ShippingRequestProcessor {
```

Но на практике этот параметр используется редко, а интерфейс определяется с помощью ключевого слова `interface`, как это сделано в примере из листинга 4.2 ❷.

Интерфейс `MessageListener` можно также объявить в дескрипторе развертывания и убрать упоминание о нем из программного кода:

```
<ejb-jar...>
  <enterprise-beans>
```

```
    <message-driven>
      ...
      <messaging-type> javax.jms.MessageListener</messaging-type>
    </message-driven>
  </enterprise-beans>
</ejb-jar>
```

Выбор того или иного варианта в значительной степени определяется личными предпочтениями. В следующем разделе мы рассмотрим порядок настройки интерфейса `MessageListener`.

4.3.6. Использование параметра ActivationConfigProperty

Параметр `ActivationConfigProperty` аннотации `@MessageDriven` дает возможность включать дополнительные настройки для провайдера в виде массива экземпляров `ActivationConfigProperty`. Интерфейс `ActivationConfigProperty` определен, как показано ниже:

```
@Target(value = {})
@Retention(value = RetentionPolicy.RUNTIME)
public @interface ActivationConfigProperty {
  String propertyName();
  String propertyValue();
}
```

Каждый экземпляр представляет собой пару имя/значение, используемую провайдером для настройки компонента MDB. Проще всего объяснить, как действуют эти настройки, на простом примере. Ниже мы определяем две наиболее типичные настройки активации JMS: `destinationType` и `destinationLookup`:

```
@MessageDriven(activationConfig = {
  @ActivationConfigProperty(propertyName = "destinationType",
    propertyValue = "javax.jms.Queue"),
  @ActivationConfigProperty(propertyName = "destinationLookup",
    propertyValue = "jms/ShippingRequestQueue")
})
```

Первая настройка, свойство `destinationType`, сообщает контейнеру, что этот компонент MDB принимает сообщения из очереди. Если бы вместо очереди компонент подключался к теме, в качестве значения следовало бы указать `javax.jms.Topic`. Настройка `destinationLookup` определяет адрес, с которого принимаются сообщения, с JNDI-именем `jms/ShippingRequestQueue`.

Существуют и другие параметры настройки для провайдера JMS, используемые при обмене сообщениями JMS. Все эти параметры перечислены в табл. 4.1.

Важно помнить, что эти настройки определены в спецификации JMS 2.0 для потребителей сообщений JMS. Если вы создаете компоненты MDB для потребления разнотипных сообщений, настройки активации, специфичные для одного провайдера, могут быть несовместимы с другим провайдером. Далее мы поближе познакомимся с некоторыми из этих настроек активации.

Таблица 4.1. Настройки активации для JMS-провайдера, стандартизованные в спецификации JMS 2.0

Настройка активации	Описание
`destinationLookup`	Имя адреса – `javax.jms.Queue` или `javax.jms.Topic` – в реестре JNDI.
`connectionFactoryLookup`	Имя фабрики соединений – `javax.jms.ConnectionFactory`, `javax.jms.QueueConnectionFactory` или `javax.jms.TopicConnectionFactory` – в реестре JNDI.
`acknowledgeMode`	`AUTO_ACKNOWLEDGE` (по умолчанию) или `DUPS_OK_ACKNOWLEDGE`.
`messageSelector`	Селектор для фильтрации принимаемых сообщений.
`destinationType`	`javax.jms.Queue` или `javax.jms.Topic`.
`subscriptionDurability`	Применяется только при установленной настройке `destinationType=" javax.jms.Topic"`. Может принимать только два значения: `Durable` или `NonDurable` (по умолчанию).
`clientId`	Определяет идентификатор клиента, используемый при подключении к провайдеру.
`subscriptionName`	Применяется только при установленной настройке `destinationType=" javax.jms.Topic"`. Имя долговременной (`Durable`) или краткосрочной (`NonDurable`) подписки.

Режим подтверждения

Сообщения остаются в очереди JMS, пока потребитель не подтвердит их получение. Существует множество «режимов» подтверждения сообщений. По умолчанию используется режим `Auto-acknowledge`, который подразумевает автоматическое подтверждение сообщений. Именно такой режим будет выбран в примере выше, потому что в нем отсутствует явное определение иного значения настройки `acknowledgeMode`. Можно также установить режим подтверждения `Dups-ok-acknowledge`, как показано ниже:

```
@ActivationConfigProperty {
    propertyName="acknowledgeMode"
    propertyValue="Dups-ok-acknowledge"
```

Длительность подписки

Если компонент MDB принимает сообщения из темы, можно указать, будет ли эта подписка долговременной или краткосрочной.

Напомним, что в схеме «издатель–подписчик» каждое сообщение передается всем потребителям, подписанным в данный момент на тему. Такая схема очень похожа на широковещательную рассылку сообщений, когда никто, кто не подключен к теме в момент появления сообщения, не получит его копию. Исключением из этого правила являются подписчики, оформившие *продолжительную подписку* (durable subscription).

Если потребитель оформил продолжительную подписку на тему, все сообщения, отправленные в эту тему, гарантированно будут доставлены ему. Если такой подписчик не был подключен к теме в момент доставки сообщения, система MOM будет хранить копию, пока подписчик не подключится и не получит сообщение. Ниже показано, как оформить продолжительную подписку:

```
MessageConsumer orderSubscriber = session.createDurableSubscriber(
    orderTopic, "OrderProcessor");
```

Здесь создается потребитель с продолжительной подпиской на тему `javax.jms.Topic orderTopic` и идентификатором подписчика `OrderProcessor`. С этого момента все сообщения в тему будут храниться, пока потребитель с идентификатором подписчика `OrderProcessor` не примет их. Аннулировать подписку можно следующим способом:

```
session.unsubscribe("OrderProcessor");
```

Если необходимо оформить продолжительную подписку для компонента MDB, его настройки активации `ActivationConfigProperty` должны выглядеть, как показано ниже:

```
@ActivationConfigProperty(
    propertyName="destinationType",
    propertyValue="javax.jms.Topic"),
@ActivationConfigProperty(
    propertyName="subscriptionDurability",
    propertyValue="Durable"
```

Чтобы оформить краткосрочную подписку, параметру настройки `subscriptionDurability` следует присвоить значение `NonDurable`. Это же значение устанавливается по умолчанию, если данная настройка не указана.

Селектор сообщений

Параметр `messageSelector` позволяет определять селекторы сообщений в компонентах MDB, действующие параллельно низкоуровневым селекторам JMS. В нашем примере потребитель получает все сообщения, поступающие на указанный адрес. При желании вы можете организовать фильтрацию сообщений с помощью *селектора сообщений* (message selector) – критерии, указанные в селекторе, применяются к заголовкам и свойствам сообщений и определяют желаемые характеристики. Например, можно указать, что компонент MDB будет обрабатывать только запросы на доставку хрупких товаров, как показано ниже:

```
@ActivationConfigProperty
    propertyName="messageSelector"
    propertyValue="Fragile is TRUE")
```

Как вы наверняка заметили, синтаксис селекторов практически идентичен предложению `WHERE` в языке SQL, только в роли имен столбцов таблиц в базе данных здесь выступают имена заголовков и свойств сообщений. Выражения в селекторах могут быть сколько угодно сложными. Они могут включать литералы,

идентификаторы, пробелы, выражения, стандартные скобки, логические операторы и операторы сравнения, арифметические операторы и сравнение с пустым значением (null). В табл. 4.2 перечислены наиболее часто используемые элементы селекторов.

Таблица 4.2. Наиболее часто используемые элементы селекторов

Элемент	Описание	Пример

Литералы	Это могут быть строки, точные или приблизительные числовые значения и логические значения.	`BidManagerMDB`
		`100`
		`TRUE`

Идентификаторы	Это могут быть имена свойств и заголовков сообщений; регистр символов имеет значение.	`RECIPIENT`
		`NumOfBids`
		`Fragile`
		`JMSTimestamp`

Пробельные символы	Те же, которые считаются пробельными в языке Java: пробел, табуляция, перевод формата и символы завершения строк.	
Операторы сравнения	Такие операторы, как: `=`, `>`, `>=`, `<=`, `<>`.	`RECIPIENT='BidManagerMDB'` `NumOfBids>=100`
Логические операторы	Поддерживаются все три логических оператора: `NOT`, `AND` и `OR`.	`RECIPIENT='BidManagerMDB'` `AND NumOfBids>=100`
Сравнение с пустым значением	Операции `IS NULL` и `IS NOT NULL`.	`Firstname IS NOT NULL`
Сравнение с логическими значениями	Операции `IS [NOT] TRUE` и `IS [NOT] FALSE`.	`Fragile is TRUE` `Fragile is FALSE`

4.3.7. События жизненного цикла

Как и сеансовые компоненты без сохранения состояния, о которых рассказывалось в главе 3, компоненты MDB имеют простой жизненный цикл (см. рис. 4.8). Для каждого компонента MDB контейнер:

- создает экземпляры MDB;
- внедряет ресурсы, включая контекст, управляемый сообщениями (message-driven context), о котором подробно рассказывается в главе 5;
- помещает экземпляры в управляемый пул (если поддерживается контейнером; в противном случае новые экземпляры создаются по мере необходимости);
- извлекает простаивающий компонент из пула по прибытии сообщения (в этот момент контейнер может увеличить размер пула);
- вызывает метод обработки сообщения – например, метод `onMessage`;
- когда метод `onMessage` завершится, экземпляр компонента помещается обратно в пул;
- по мере необходимости удаляет компоненты из пула (и уничтожает их).

Рис. 4.8. В течение жизни компонент MDB проходит три стадии: отсутствие, ожидание и обработка сообщения. Как результат, компонент может реагировать только на два события, соответствующие моментам создания и уничтожения компонента. Перехват этих событий можно организовать с помощью аннотаций @PostConstruct и @PreDestroy

В течение жизни компонента MDB возникает два события: `PostConstruct`, которое происходит сразу после создания компонента MDB и настройки всех внедренных в него ресурсов, и `PreDestroy`, которое происходит непосредственно перед извлечением компонентом из пула для последующего уничтожения. В методах-обработчиках этих событий допускается выполнять любые операции, но обычно эти методы используются для создания и освобождения дополнительных ресурсов. Например, представьте, что компонент MDB должен осуществлять запись в обычный файл. Для этого ему необходим объект `PrintWriter`. Вы легко можете определить свойство типа `PrintWriter` в классе компонента MDB:

```
PrintWriter printWriter;
```

Теперь необходимо инициализировать это свойство, что удобнее всего сделать в методе-обработчике события `PostConstruct`, как показано ниже:

```
@PostConstruct
void createPrintWriter() {
  File f = new File(new File(System.getProperty("java.io.tmpdir")),
      "ShippingMessages.txt");
  try {
```

```
    printWriter = new PrintWriter(f);
  } catch (Throwable t) {
    throw new RuntimeException(t);
  }
  Logger.getLogger(TurtleShippingRequestMessageBean.class.getName())
      .log(Level.INFO,
      String.format("Write to file: %s", f.getAbsolutePath()));
}
```

Сразу после создания экземпляра MDB, контейнер вызовет его метод `createPrintWriter`, который в свою очередь инициализирует свойство `printWriter`. Теперь свойство `printWriter` может использоваться методом `onMessage` или любым другим методом в компоненте MDB для записи в файл, пока компонент остается активным.

Когда контейнер решит, что можно избавиться от экземпляра MDB, в это время следует освободить ресурс файла. Сделать это удобнее всего в методе-обработчике события `PreDestroy`, который мог бы выглядеть как-то так:

```
@PreDestroy
void closePrintWriter() {
  printWriter.flush();
  printWriter.close();
}
```

Контейнер вызовет метод `closePrintWriter` непосредственно перед удалением экземпляра из пула. Этот метод вытолкнет содержимое буферов `PrintWriter` на диск и затем закроет файл.

Теперь у кого-то может возникнуть вопрос: безопасно ли осуществлять запись в файл подобным способом, учитывая, что контейнер может создавать множество экземпляров компонента MDB и помещать их в пул, чтобы обеспечить более высокую эффективность обработки сообщений? Как вы понимаете, в большинстве случаев ответ на этот вопрос: нет. Но это был всего лишь простой пример, демонстрирующий, как можно использовать события `PostConstruct` и `PreDestroy`.

Компоненты MDB отлично подходят для приема и обработки сообщений. Мы только что закончили знакомство с основами этой технологии. Но теперь возникает другой вопрос: а может ли компонент MDB отправлять сообщения? Ответ: да. Более того, вы будете делать это чаще, чем можете представить. Порядок отправки сообщений JMS с помощью компонентов MDB мы рассмотрим в следующем разделе.

4.3.8. Отправка сообщений JMS из компонентов MDB

По иронии, одна из задач, которую вам снова и снова придется выполнять в компонентах MDB – это отправка сообщений JMS. Например, представьте, что был получен неполный запрос на доставку и требуется из `ShippingRequestProcessor` послать соответствующее сообщение приложению ActionBazaar. Проще всего было бы послать уведомление в очередь ошибок, которая постоянно проверяется прило-

жением ActionBazaar. К счастью, вы уже знаете, как послать сообщение JMS (см. листинг 4.1). Для этого нужно внедрить очередь с именем `jms/ShippingErrorQueue` и фабрику соединений с именем `jms/QueueConnectionFactory`:

```
@Inject
@JMSConnectionFactory("jms/QueueConnectionFactory")
private JMSContext context;

@Resource(name="jms/ShippingErrorQueue")
private Destination errorQueue;
```

После этого система компании Turtle сможет посылать сообщения приложению ActionBazaar:

```
Message m = context.createTextMessage(
  String.format("Item in error %s",tr.getItem()));
JMSProducer producer = context.createProducer();
producer.send(destination, om);
```

Хотя мы явно не показали это в примере, тем не менее, компоненты MDB обладают еще одной особенностью: они способны управлять транзакциями MDB. Транзакции EJB детально будут рассматриваться в следующей главе, поэтому здесь мы познакомим вас лишь с самыми азами.

4.3.9. Управление транзакциями MDB

По умолчанию, перед вызовом метода `onMessage`, контейнер запускает транзакцию, а когда метод вернет управление – подтверждает ее, если транзакция не была помечена для отката через контекст, и в ходе выполнения метода `onMessage` не возникло необработанное исключение. Подробнее о транзакциях рассказывается в главе 6. Контейнер следует этому сценарию, если предполагает, что транзакция необходима. Определить это требование можно явно, как показано ниже:

```
@MessageDriven
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class TurtleShippingRequestMessageBean
```

При желании можно сообщить контейнеру, что компонент MDB не нуждается в транзакции. В этом случае вам придется вручную обеспечивать целостность данных в базе и подтверждать получение сообщения. Сообщения подтверждаются независимо от успешности их обработки. Мы не рекомендуем такой подход, но вы можете использовать его, как показано ниже:

```
@MessageDriven
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class TurtleShippingRequestMessageBean
```

Этим кратким обсуждением транзакций мы завершаем исследование основных особенностей компонентов MDB. В ходе этого исследования мы узнали, как с помощью компонентов MDB использовать мощный механизм обмена сообщениями без необходимости писать низкоуровневый код. Как было показано, компоненты

MDB автоматически поддерживают механизмы EJB, такие как поддержка выполнения в многопоточной среде, внедрение ресурсов, события жизненного цикла и транзакции, управляемые контейнером. вы познакомились с образцами программного кода и теперь вы можете использовать их как шаблоны для решения своих прикладных задач.

4.4. Приемы использования компонентов MDB

Подобно всем технологиям, компоненты MDB и асинхронные методы имеют свои ловушки и наиболее удачные приемы использования, которые следует знать и помнить. Это особенно верно в сложных системах, где обычно используются механизмы обмена сообщениями.

С особым вниманием выбирайте модель обмена сообщениями. Прежде чем погрузиться с головой в код, тщательно обдумайте свой выбор модели обмена сообщениями. В 9 случаях из 10 вы можете прийти к выводу, что модель PTP («точка–точка») лучше всего решает ваши проблемы. Но иногда предпочтительнее может оказаться модель «издатель–подписчик», особенно если требуется посылать широковещательные сообщения нескольким приемникам (например, извещение об остановке системы на профилактику). К счастью, большая часть кода, так или иначе связанного с сообщениями, не зависит от выбранной модели и вам следует стремиться следовать именно в этом направлении. По большей части, вопрос переключения между моделями должен сводиться лишь к изменению настроек.

Не злоупотребляйте компонентами MDB. Многие задачи могут с успехом решаться с помощью асинхронных методов сеансовых компонентов или событий CDI. Если вам просто нужно запустить еще одну задачу для выполнения некоторых операций, механизм сообщений JMS может оказаться чересчур тяжеловесным. Асинхронные методы в сравнении с ним выглядят намного проще. Управление потоками выполнения возьмет на себя контейнер, и вы сможете получать возвращаемые значения без нагромождения сложных последовательностей сообщений для обработки ошибок. Аналогично, если вам необходимо всего лишь ослабить связь между компонентами, используйте события CDI.

Не забывайте о преимуществах модульной архитектуры. Так как компоненты MDB очень похожи на сеансовые компоненты, вполне естественным кажется размещение прикладной логики непосредственно в методах обработки сообщений. Помните, что прикладная логика не должна переплетаться с операциями, имеющими отношение к обмену сообщениями. Наиболее удачным приемом (следование которому, впрочем, могло бы усложнить эту главу) считается размещение прикладной логики в сеансовых компонентах и вызывать ее из метода `onMessage`.

Не забывайте о фильтрах сообщений. Существует масса вполне обоснованных причин использования единственного адреса для самых разных целей. В подобных ситуациях с успехом можно использовать селекторы сообщений. Например, при использовании одной и той же очереди для запросов на доставку и уведомле-

ний об аннулировании заказов, на стороне клиента можно настроить установку свойства сообщения, идентифицирующего тип запроса, а на стороне потребителя – создать два отдельных компонента MDB с соответствующими селекторами, чтобы разделить обработку разнотипных запросов.

Иногда наоборот можно существенно увеличить производительность и упростить код, создав отдельные адреса для сообщений разных типов вместо использования селекторов. Например, использование разных очередей и компонентов MDB для запросов на доставку и уведомлений об отмене заказов могло бы ускорить передачу сообщений. В этом случае клиент мог бы отправлять сообщения разных типов в соответствующие очереди.

Внимательно подходите к выбору типа сообщения. Выбор типа сообщения не всегда очевиден, как кажется. Например, часто весьма привлекательной выглядит идея использовать в качестве сообщений строки в формате XML. Кроме всего прочего, такое решение позволяет добиться еще большего ослабления связей между системами. Например, сервер компании Turtle мог бы с успехом обрабатывать сообщения в формате XML, ничего не зная о внутреннем устройстве объекта `ActionBazaarShippingRequest`.

Однако размер сообщения в формате XML обычно многократно превосходит размер того же сообщения в двоичном формате, что может отрицательно сказаться на производительности MOM. При определенных условиях правильнее будет использовать двоичный формат, чтобы уменьшить нагрузку на систему MOM и снизить потребление памяти.

Остерегайтесь отравленных сообщений. Представьте, что вам было передано сообщение, «переварить» которое ваш компонент MDB не смог. Продемонстрируем это на примере: пусть принято сообщение, не являющееся объектом типа `ObjectMessage`. Как видно из следующего фрагмента кода, в этом случае попытка приведения типа в методе `onMessage` возбудит исключение `java.lang.ClassCastException`:

```
try {
  // Неправильный тип сообщения вызовет исключение
  // при попытке выполнить приведение типа
  ObjectMessage om = (ObjectMessage) message;
  Object o = om.getObject();
  ActionBazaarShippingRequest sr = (ActionBazaarShippingRequest) o;
  ...
} catch (JMSException ex) {
  Logger.getLogger(TurtleShippingRequestMessageBean.class.getName())
      .log(Level.SEVERE, null, ex);
}
```

Из-за того, что `onMessage` не может завершиться нормально, контейнер будет принудительно откатывать транзакцию и возвращать сообщение в очередь, вместо того, чтобы подтвердить его. Это большая проблема, потому что вы снова и снова будете извлекать одно и то же сообщение из очереди, сталкиваться с ошибкой и вновь возвращать его в очередь, и так до бесконечности! Сообщения, вызывающие такой эффект, называются *отравленными сообщениями* (poisoned messages).

К счастью многие системы MOM и контейнеры EJB имеют механизмы для обработки отравленных сообщений, включая подсчет возвратов и очереди для «мертвых» сообщений. Если настроить учет количества возвратов и очередь «мертвых» сообщений для адреса, куда направляются запросы на доставку, попытка доставить сообщение будет выполнена не более указанного числа раз. Когда счетчик попыток превысит установленное значение, сообщение будет помещено в специальную очередь для отравленных сообщений, которую называют *очередью мертвых сообщений* (dead-message queue). Единственная проблема – этот механизм не стандартизован и в некоторых реализациях может отсутствовать.

Не пренебрегайте настройкой размера пула MDB. Большинство контейнеров EJB позволяют определять максимальный размер пула для компонентов MDB конкретного типа. Фактически эта настройка управляет степенью конкуренции. Например, если в очередь одновременно поступило пять сообщений, а размер пула ограничен тремя компонентами, контейнер будет ждать, пока не обработаются первые три сообщения, прежде чем попытается создать дополнительные экземпляры. Однако это обоюдоострое оружие и требует особой осторожности. Если установить размер пула компонентов MDB слишком маленьким, сообщения будут обрабатываться медленно. В то же время, желательно установить разумные ограничения на размер пула MDB, чтобы не растрачивать вычислительные ресурсы системы на переключение между множеством параллельно выполняющихся экземпляров MDB. К сожалению, на момент написания этих строк, настройка размера пула MDB не была стандартизована и у разных производителей реализована по-разному.

Рассмотрите возможность использования несохраняемых сообщений, режима подтверждения `DUPS_OK_ACKNOWLEDGE` *и компонентов MDB без транзакций.* При желании можно указать, что JMS-сообщения не должны сохраняться. Это означает, что сервер сообщений не будет создавать резервные копии сообщений на диске. В результате этого, если на сервере сообщений произойдет авария, сообщения будут потеряны навсегда. Преимущество несохраняемых сообщений состоит в том, что они обрабатываются намного быстрее, но ценой потери надежности. Аналогично, определив режим подтверждения `DUPS_OK_ACKNOWLEDGE`, можно уменьшить накладные расходы и увеличить производительность. Тот же эффект наблюдается и при отказе от использования транзакций. Однако, при использовании приемов оптимизации следует проявлять осторожность и убедиться, что потеря надежности не окажет отрицательного влияния на прикладную сторону решаемой задачи.

4.5. В заключение

В этой главе мы охватили основные понятия, касающиеся обмена сообщениями, JMS и компонентов MDB. Обмен сообщениями – чрезвычайно мощная технология для корпоративных приложений, помогающая создавать слабосвязанные и надежные системы. Механизм JMS позволяет использовать системы MOM без необходимости взаимодействовать с низкоуровневыми API. Использование JMS API

для создания приложений-потребителей сообщений может оказаться достаточно трудоемкой задачей, а применение компонентов MDB дает простую и надежную возможность использовать MOM стандартизованным способом через Java EE.

В этой главе мы коснулись некоторых из основных особенностей EJB: JNDI и внедрение зависимостей, контексты EJB и транзакции. Более подробно эти особенности будут обсуждаться в последующих главах.

ГЛАВА 5. Контекст EJB времени выполнения, внедрение зависимостей и сквозная логика

Эта глава охватывает следующие темы:

- основы `EJBContext`;
- использование JNDI для поиска компонентов EJB и других ресурсов;
- аннотация `@EJB`;
- компоненты EJB в клиентских приложениях и встраиваемые контейнеры;
- основы интерцепторов AOP.

В двух предыдущих главах мы занимались разработкой сеансовых компонентов и компонентов, управляемых сообщениями (MDB). В этой главе мы познакомимся с некоторыми продвинутыми понятиями, касающимися сеансовых компонентов и компонентов MDB. Сначала мы рассмотрим, как контейнеры предоставляют доступ к своим службам и как обращаться к окружению времени выполнения. Затем мы исследуем более сложные приемы использования механизма внедрения зависимостей, реестра JNDI и интерцепторов EJB. По мере движения вперед вы увидите до какой степени EJB 3 освобождает вас от решения рутинных задач системного уровня, предоставляя чрезвычайно надежные и гибкие функциональные возможности.

5.1. Контекст EJB

Компоненты EJB вообще мало зависят от конкретной реализации контейнера. Это означает, что в идеальном случае компоненты EJB должны содержать лишь прикладную логику и никогда напрямую не обращаться к контейнеру или к его

службам. Но иногда в прикладном коде все же бывает необходимо явно вызывать службы контейнера. Например, может потребоваться вручную откатить транзакцию или создать таймер, чтобы запустить бизнес-процесс в некоторый момент времени в будущем. Для обслуживания всех этих ситуаций предназначен контекст EJB. Интерфейс `javax.ejb.EJBContext` – это потайная дверь в закулисный мир контейнера. В этом разделе расскажем вам об интерфейсе `EJBContext`, покажем, как с помощью механизма внедрения зависимостей получить доступ к контексту и как его использовать.

5.1.1. Основы контекста EJB

Как показано в листинге 5.1, интерфейс `EJBContext` открывает прямой доступ к таким службам, как транзакции, поддержка безопасности и таймеры, которые обычно настраиваются через конфигурацию и находятся под полным управлением контейнера.

Листинг 5.1. Интерфейс javax.ejb.EJBContext

```
public interface EJBContext {
    public Principal getCallerPrincipal();           // Поддержка
    public boolean isCallerInRole(String roleName); // безопасности

    public boolean getRollbackOnly();                // Управление
    public UserTransaction getUserTransaction();     // транзакциями
    public void setRollbackOnly();                   //

    public TimerService getTimerService();        // Доступ к службе таймеров

    public Object lookup(String name);             // Поиск в реестре JNDI
    public Map<String, Object> getContextData();   // Доступ к контекстным
                                                   // данным из интерцепторов

    public EJBHome getEJBHome();                     // Для совместимости
    public EJBLocalHome getEJBLocalHome();           // с EJB 2
}
```

Рассмотрим коротко, что делает каждый из этих методов (табл. 5.1). Более детальное исследование мы оставим на потом, когда будем рассматривать службы, доступ к которым обеспечивают эти методы. А пока просто отметьте для себя, какие службы доступны через контекст EJB и обратите внимание на сигнатуры методов.

Таблица 5.1. Интерфейс javax.ejb.EJBContext можно использовать для доступа к службам времени выполнения

Методы	Описание
`getCallerPrincipal` `isCallerInRole`	Эти методы могут пригодиться для обеспечения безопасности на уровне компонента. Мы обсудим их далее в главе 6, когда будем рассказывать о программной поддержке безопасности.

Таблица 5.1. (окончание)

Методы	Описание
`getEJBHome` `getEJBLocalHome`	Эти методы используются для получения «удаленного» и «локального» домашнего интерфейсов соответственно. Оба являются необязательными для контейнеров EJB 3 и практически не используются в компонентах для версий EJB выше 2.1. Мы будем рассматривать эти методы только в данном введении. Основное их назначение – поддержка обратной совместимости.
`getRollbackOnly` `setRollbackOnly`	Эти методы применяются для управления транзакциями на уровне контейнера EJB. Подробнее о транзакциях рассказывается в главе 6.
`getUserTransaction`	Этот метод используется для управления транзакциями на уровне компонента. Подробнее о транзакциях рассказывается в главе 6.
`getTimerService`	Этот метод используется для получения доступа к службе таймеров EJB. Подробнее о таймерах EJB рассказывается в главе 7.
`lookup`	Этот метод используется для получения ссылок на объекты, хранящиеся в реестре JNDI. При использовании механизма внедрения зависимостей, прибегать к непосредственному поиску в реестре JNDI практически не приходится. Но в редких случаях, когда механизм внедрения зависимостей не может использоваться, этот метод оказывается удобным решением проблемы. Подробнее эта тема будет обсуждаться далее в главе.
`getContextData`	Этот метод позволяет компонентам EJB получить доступ к контекстным данным из интерцепторов. Подробнее об интерцепторах рассказывается далее в этой главе.

5.1.2. Интерфейсы контекста EJB

Обе разновидности компонентов, сеансовые и MDB, имеют собственные подклассы, реализующие интерфейс `javax.ejb.EJBContext`. Как показано на рис. 5.1, подкласс сеансовых компонентов называется `javax.ejb.SessionContext`, а подкласс компонентов MDB – `javax.ejb.MessageDrivenContext`.

Каждый подкласс предназначен для конкретного окружения времени выполнения и типа компонента. Как результат, они или добавляют новые методы в суперкласс или делают недействительными методы, которые не должны использоваться компонентами того или иного типа.

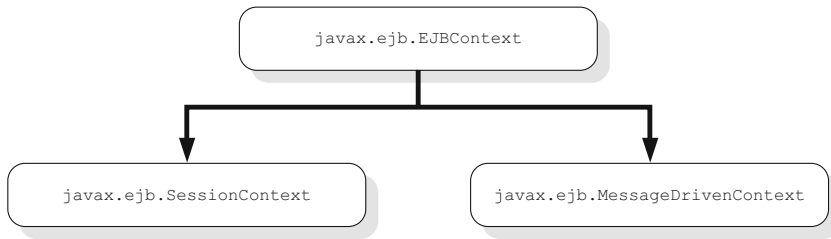

Рис. 5.1. Интерфейс EJBContext имеет подклассы для сеансовых компонентов и компонентов MDB

SessionContext

`SessionContext` – это реализация, добавляющая методы, специфичные для сеансовых компонентов. Эти методы перечислены в табл. 5.2. Хотя эти методы имеют свои области применения, на практике они используются достаточно редко.

Таблица 5.2. Дополнительные методы, добавляемые подклассом javax.ejb.SessionContext

Метод	Описание
`getBusinessObject`	Возвращает объект, как компонент без интерфейса или как один из прикладных интерфейсов (локальный или удаленный).
`getEJBLocalObject` `getEJBObject`	Возвращает локальный или удаленный объект для текущего экземпляра компонента. Может применяться только к компонентам EJB 2. При использовании в EJB 3 генерирует исключение.
`getInvokedBusinessInterface`	Возвращает интерфейс или компонент без интерфейса, использовавшийся для вызова прикладного метода.
`getMessageContext`	Если компонент доступен через веб-службу, этот метод возвращает `MessageContext`, связанный с этим запросом.
`wasCancelCalled`	Возвращает `true`, если пользователь потребовал отменить вызов асинхронного метода, выполняющегося продолжительное время.

Метод `wasCancelCalled()` был добавлен в EJB 3.1. Он предназначен для использования в асинхронных методах, как рассказывалось в главе 3. Обычно асинхронные методы возвращают объект `Future<V>`. Используя этот объект, пользователь может вызвать метод `Future<V>.cancel()`, чтобы прервать продолжительную асинхронную операцию. В реализациях таких операция считается хорошей практикой регулярно проверять значение, возвращаемое этим методом, чтобы определить, не должна ли асинхронная процедура прервать выполнение.

MessageDrivenContext

`MessageDrivenContext` – это реализация, специфичная для компонентов MDB. В отличие от `SessionContext`, данная реализация не добавляет новых методов – она переопределяет следующие методы так, что при вызове они возбуждают исключение: `isCallerInRole`, `getEJBHome` и `getEJBLocalHome`. Помните, что эти методы являются частью интерфейса `SessionContext`, но они не имеют смысла для компонентов MDB, потому что у них нет прикладного интерфейса и они никогда не вызываются клиентом непосредственно.

5.1.3. Доступ к контейнеру через контекст EJB

Получить доступ к `EJBContext` можно с помощью механизма внедрения зависимостей. Например, объект `SessionContext` можно внедрить в сеансовый компонент, как показано ниже:

```
@Stateless
public class DefaultBidService implements BidService {
  @Resource
  SessionContext context;
  ...
}
```

В этом фрагменте, обнаружив присутствие аннотации `@Resource` перед переменной `context`, контейнер определит, что компонент требует внедрения контекста контейнера. Подробнее об аннотации `@Resource` рассказывается далее в этой главе.

Во многом похожий на контекст сеанса, контекст `MessageDrivenContext` так же может быть внедрен в компонент MDB, как показано ниже:

```
@MessageDriven
public class OrderBillingProcessor {
@Resource
MessageDrivenContext context;
  ...
}
```

Примечание. *Не допускается внедрять* `MessageDrivenContext` *в сеансовые компоненты или* `SessionContext` *в компоненты MDB.*

Но пока достаточно разговоров о контексте EJB. Давайте теперь обратим наше внимание на один из важнейших механизмов EJB 3 – внедрение зависимостей (Dependency Injection, DI). Мы коротко рассмотрели DI в главе 2 и имели возможность наблюдать его действие в нескольких предыдущих главах. Теперь пришло время заняться более детальными исследованиями.

5.2. Использование EJB DI и JNDI

В Java EE 5 была добавлена аннотация `@EJB`, чтобы упростить компонентам, управляемым контейнером, получение ссылок на компоненты EJB. Следуя практике инверсии управления (inversion of control), контейнер использует механизм DI для связывания компонентов между собой. Хотя JNDI все еще используется за кулисами, эти тонкости скрыты от разработчика за ширмой аннотации `@EJB`. До выхода спецификации EE5, использовался только механизм JNDI. Главное отличие прежних версий состоит в том, что тогда за связывание компонентов отвечал исключительно разработчик, который вынужден был обращаться к реализации JNDI, используя шаблон проектирования локатора служб (service locator pattern), вместо того, чтобы доверить эту работу механизму внедрения зависимостей контейнера. В этом разделе мы сначала познакомимся с основами JNDI, чтобы получить представление о том, что происходит за кулисами. Затем займемся детальным изучением аннотации `@EJB`, посмотрим, когда и как использовать эту аннотацию, и как она связана с JNDI.

Инверсия управления

Инверсия управления (Inversion of control, IoC) – важный принцип объектно-ориентированного программирования. Согласно ему, объекты должны иметь как можно более слабые связи между собой за счет использования интерфейсов, а выбор конкретной реализации должен производиться уже во время выполнения. Шаблоны проектирования «Локатор служб» (Service Locator, SL) и «Внедрение зависимостей» (Dependency Injection) помогают следовать этому принципу.

Шаблон «Внедрение зависимостей»

Шаблон «Внедрение зависимостей» следует модели «вталкивания» (или «втискивания»). Зависимости объектов определяются некоторой конфигурацией (XML, аннотации и др.), а контейнер создает конкретные реализации этих зависимостей во время выполнения и втискивает их в объекты. В этом заключена суть технологии CDI (обсуждается далее в этой главе).

Шаблон «Локатор служб»

Шаблон SL следует модели «вытягивания». Объекты регистрируются внутри центрального хранилища с помощью специальной службы. А затем объекты самостоятельно, с помощью этой службы, «вытягивают» нужные им зависимости из центрального хранилища. JNDI как раз является реализацией такой службы и объекты могут использовать JNDI для вытягивания своих зависимостей.

5.2.1. Пример использования JNDI в EJB

По сути, JNDI – это аналог JDBC для служб имен и каталогов. Так же как JDBC предоставляет стандартный Java API для доступа к разнообразным базам данных, JNDI стандартизует доступ к службам имен и каталогов. Если вам когда-либо приходилось работать с файловой системой компьютера, значит вы уже знаете, что такое служба имен и каталогов. Файловая система компьютера является простейшим примером службы каталогов. С помощью файловой системы можно осуществлять поиск файлов и просматривать ее содержимое. Если прежде вам доводилось использовать службу Lightweight Directory Access Protocol (LDAP) или Microsoft Active Directory (AD), значит вы уже знакомы с еще более надежной службой имен и каталогов.

Как показано на рис. 5.2, JNDI представляет обобщенную абстракцию доступа к самым разным службам имен, таким как LDAP, Domain Naming System (DNS), Network Information Service (NIS), Novell Directory Services (NDS), Remote Method Invocation (RMI), Common Object Request Broker Architecture (CORBA), и так далее. Получив экземпляр контекста JNDI, его можно использовать для поиска ресурсов в любой службе имен, доступной этому контексту. За кулисами служба JNDI взаимодействует со всеми доступными ей службами имен, передавая им имена ресурсов, которые требуется найти и выясняя, где в действительности находится искомый ресурс.

Если в качестве аналогии представить JDBC, тогда инструкция `SELECT` на языке SQL – это имя ресурса, сервер базы данных – это служба имен, а JDBC – стандарт-

ный API. Получив инструкцию `SELECT`, сервер базы данных с ее помощью отыщет данные (фактический ресурс), которые в действительности могут находиться в связанной таблице или вообще на другом сервере базы данных.

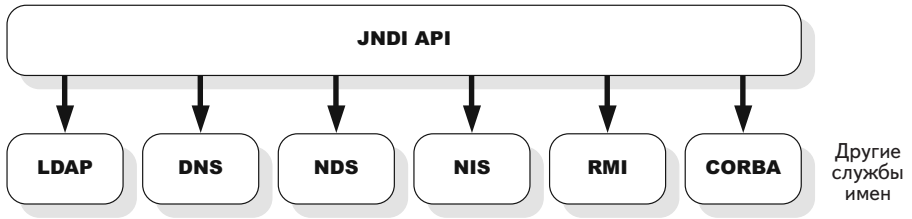

Рис. 5.2. JNDI предоставляет единый обобщенный API для доступа к различным службам имен, таким как LDAP, DNS, NDS, NIS, RMI и CORBA. Любая служба имен, реализующая интерфейс провайдера JNDI SPI, может быть бесшовно подключена к этому API

JNDI играет важную роль в Java EE, хотя и остается все время за кулисами. JNDI используется как центральное хранилище для ресурсов, управляемых контейнером. Как результат, все компоненты, управляемые контейнером, автоматически регистрируются в JNDI. Кроме того, типичный реестр JNDI в контейнере также хранит источники данных JDBC, очереди JMS, фабрики соединений JMS, диспетчеры сущностей JPA, фабрики диспетчеров сущностей JPA, сеансы JavaMail, и так далее. Всякий раз, когда у клиента (такого как компонент EJB) появляется необходимость задействовать управляемым ресурсом, он может воспользоваться услугами JNDI и найти требуемый ресурс по его уникальному имени. На рис. 5.3 показано, как может выглядеть типичное дерево JNDI для сервера приложений Java EE.

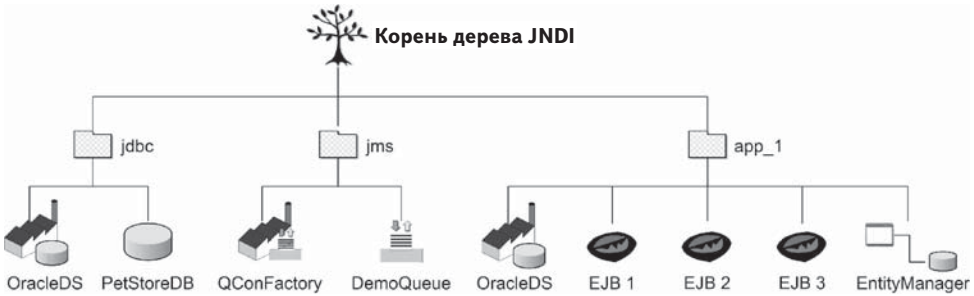

Рис. 5.3. Пример дерева JNDI для сервере приложений. Все глобальные ресурсы, такие как JDBC и JMS, связаны с корневым контекстом дерева JNDI. Каждое приложение имеет собственный контекст приложения и компоненты EJB, а также другие ресурсы, связанные с контекстом приложения

Как показано на рис. 5.3, ресурсы хранятся в дереве JNDI в виде иерархической структуры. Это означает, что поиск имен ресурсов в JNDI выполняется почти так же, как поиск имен файлов и каталогов в файловой системе. Иногда имена ресурсов могут начинаться с обозначения протокола, такого как `java:`, что очень

напоминает использование буквы диска, такого как C:\ для обозначения главного диска, или S:\, для обозначения подключенного сетевого диска. После получения ссылки на ресурс из контекста JNDI, вы сможете использовать его как обычный локальный ресурс.

Инициализация контекста JNDI

Чтобы воспользоваться ресурсом, хранящимся в контексте JNDI, клиент должен инициализировать контекст и найти ресурс. Несмотря на всю сложность самого механизма JNDI, пользоваться им достаточно просто. Это напоминает настройку драйвера JDBC перед подключением к базе данных.

Прежде всего, чтобы подключиться к службе имен или каталогов, необходимо получить библиотеки JNDI для этой службы. Это мало чем отличается от выбора правильного драйвера JDBC для подключения к базе данных. Если требуется подключиться к LDAP, DNS или файловой системе компьютера, необходимо получить провайдера для LDAP, DSN или файловой системы, соответственно.

При работе в окружении Java EE, сервер приложений уже имеет и загружает все необходимые библиотеки для доступа к окружению JNDI. В противном случае вам потребуется настроить свое приложение, указав, какие библиотеки JNDI оно должно использовать. Сделать это можно, например, создав объект Properties и передав его конструктору InitialContext:

```
Properties properties = new Properties();
properties.put(Context.INITIAL_CONTEXT_FACTORY
    ,"oracle.j2ee.rmi.RMIInitialContextFactory");
properties.put(Context.PROVIDER_URL,
    "ormi://192.168.0.6:23791/appendixa");
properties.put(Context.SECURITY_PRINCIPAL, "oc4jadmin");
properties.put(Context.SECURITY_CREDENTIALS, "welcome1");

Context context = new InitialContext(properties);
```

В этом примере выполняется настройка объекта Properties для доступа к дереву JNDI удаленного сервера приложений Oracle. Обратите внимание, что параметры подключения к JNDI зависят от конкретного производителя и данный пример не является универсальным, поэтому обращайтесь к документации с описанием своего сервера приложений, чтобы узнать, как организовать удаленное подключение.

Другой способ выполнить настройки – создать файл jndi.properties, положить его где-нибудь в пути CLASSPATH приложения, сохранить в этом файле те же самые пары имя/значение, что помещаются в объект Properties. После этого данный файл будет использоваться автоматически при создании контекста вызовом InitialContext:

```
Context context = new InitialContext();
```

В данном примере предполагается, что файл jndi.properties находится в пути CLASSPATH. Большинство серверов приложений автоматически создают

файл `jndi.properties` по умолчанию. Как результат, в большинстве ситуаций код инициализации контекста JNDI выглядит не сложнее создания простого объекта с помощью оператора `new` и конструктора по умолчанию. В табл. 5.3 перечислены наиболее часто используемые свойства JNDI, необходимые для подключения к удаленному провайдеру службы JNDI (SDI).

Таблица 5.3. Наиболее часто используемые свойства, необходимые для подключения к удаленной службе JNDI в окружении Java EE

Свойство	Описание	Пример значения
`java.naming.factory.initial`	Имя фабричного класса, который будет использоваться для создания контекста.	`oracle.j2ee.rmi.RMIInitialContextFactory`
`java.naming.provider.url`	Адрес URL удаленной службы JNDI.	`ormi://localhost:23791/chapter1`
`java.naming.security.principal`	Имя пользователя или другая идентификационная информации, позволяющая аутентифицировать вызывающую программу в службе JNDI.	`oc4jadmin`
`java.naming.security.credentials`	Пароль, используемый для аутентификации.	`welcome1`

Поиск ресурсов в JNDI

После подключения к провайдеру JNDI можно воспользоваться функциональными возможностями интерфейса `Context` и получить желаемый ресурс. Здесь мы сосредоточим все наше внимание на методе `lookup`, а другие предлагаем вам изучить самостоятельно. Этот метод описан в табл. 5.4.

Таблица 5.4. Метод lookup интерфейса Context

Метод	Описание
`Object lookup(String name)`	Возвращает ресурс с именем `name`, который следует привести к требуемому типу. Если в аргументе `name` передать пустую строку, возвращается новый экземпляр `Context`.

Итак, чтобы отыскать ресурс, необходимо знать его имя. Допустим, что компонент `BidService` зарегистрирован в JNDI под именем `"/ejb/bid/BidService"`. Чтобы найти его, можно выполнить поиск непосредственно:

```
Context context = new InitialContext();
BidService service = (BidService) context.lookup("/ejb/bid/BidService");
```

Или по цепочке:

```
Context newContext = new InitialContext();
Context bidContext = (Context) newContext.lookup("/ejb/bid/");
BidService service = (BidService) bidContext.lookup("BidService");
```

Несмотря на то, что эти примеры выглядят достаточно простыми, в действительности все гораздо сложнее. Поиск в JNDI – одна из основных составляющих сложности EJB 2.x. Во-первых, чтобы получить доступ к любому ресурсу, управляемому контейнером, его необходимо сначала найти, даже когда нужно получить доступ к источникам данных или компонентам EJB, из другого компонента EJB, находящегося в границах той же виртуальной машины JVM. Учитывая, что большинство компонентов EJB в приложении зависят от других компонентов и ресурсов, представьте, сколько повторяющихся строк кода, выполняющего поиск в JNDI, придется написать в среднем по размерам приложении! Хуже того, до версии Java EE 6, ресурсам в JNDI каждый сервер приложений присваивал собственные, нестандартизованные имена. Поэтому код поиска имен в JNDI был непереносим между серверами, а имена ресурсов были не всегда очевидны, что особенно характерно было для локальных ресурсов, в именах которых использовался таинственный префикс `java:comp/env/`.

В EJB 3 дела обстоят намного лучше, так как за исключением крайне редких случаев, вам вообще не придется выполнять поиск в JNDI непосредственно. Механика использования JNDI в EJB 3 скрыта за ширмой механизма внедрения зависимостей. Этот механизм является настолько высокой абстракцией, что вам даже не нужно знать, что где-то там в недрах производится поиск в JNDI. Подобная абстракция стала возможна после стандартизации имен в JNDI, что особенно верно для компонентов EJB, которые совместимы со всеми серверами EE. В следующем разделе мы познакомимся с этим стандартом и посмотрим, как присваиваются имена компонентам EJB.

5.2.2. Как присваиваются имена компонентам EJB

Чтобы обеспечить переносимость операций поиска компонентов EJB в JNDI между всеми серверами Java EE, был предложен следующий стандарт именования компонентов:

```
java:<пространство-имен>/[имя-приложения]/<имя-модуля>/
    <имя-компонента>[!полное-квалифицированное-имя-интерфейса],
```

где элементы `<пространство-имен>`, `<имя-модуля>` и `<имя-компонента>` являются обязательными всегда присутствуют в имени, а элементы `[имя-приложения]` и `[!полное-квалифицированное-имя-интерфейса]` считаются необязательными и могут отсутствовать. Давайте рассмотрим каждый элемент этого переносимого имени JNDI и посмотрим, как выбираются их значения.

<пространство-имен>

Среда имен сервера Java EE делится на четыре пространства имен, где каждое пространство имен представляет свою область видимости. Эти пространства имен перечислены в табл. 5.5.

Таблица 5.5. Пространства имен Java EE

Пространство имен	Описание
`java:comp`	Поиск в этом пространстве имен ограничивается областью видимости компонента. Например, каждый компонент EJB в JAR-файле получит собственное уникальное пространство имен `java:comp`. Это означает, что `AccountEJB` и `CartEJB` можно найти в `java:comp/LogLevel` и получить разные значения. Для обратной совместимости, это правило не относится к веб-модулю. Все компоненты, развертываемые из WAR-файла, попадают в одно общее пространство имен `java:comp`. Скорее всего вам редко придется пользоваться этим пространством имен. Основное его предназначение – сохранение обратной совместимости с версиями Java EE 6 и ниже, где `java:comp` было единственным стандартным пространством имен.
`java:module`	Поиск в этом пространстве имен ограничивается областью видимости модуля. Все компоненты в модуле попадут в одно пространство имен `java:module`. Файл EJB-JAR – это модуль, как и файл WAR. Для обратной совместимости, пространства имен `java:comp` и `java:module` интерпретируются в веб-модулях как одно пространство имен. Всегда, когда это возможно, вместо `java:comp` следует использовать `java:module`.
`java:app`	Поиск в этом пространстве имен ограничивается областью видимости приложения. Все компоненты из всех модулей в одном приложении попадут в общее пространство имен `java:app`. Архив EAR может служить примером приложения. Все WAR- и EJB-компоненты, развертываемые из EAR-архива попадут в это пространство имен.
`java:global`	Поиск в этом пространстве имен выполняется глобально. Сюда попадают все компоненты из всех модулей и всех приложений.

[имя-приложения]

Значение `[имя-приложения]` является необязательным и присутствует в именах только тех компонентов EJB, которые развертываются на сервере из EAR-архива. Если архив EAR не использовался, тогда значение `[имя-приложения]` отсутствует в переносимых именах JNDI компонентов EJB.

По умолчанию в качестве значения `[имя-приложения]` выбирается имя EAR-файла без расширения `.ear`. Переопределить это умолчание можно в файле `application.xml`.

<имя-модуля>

Значение `<имя-модуля>` является обязательным и всегда присутствует в имени ресурса. Это значение зависит от того, как развертываются модули, содержащие компоненты EJB.

Если компоненты развертываются из отдельных файлов EJB-JAR (JAR-файлы развертываются непосредственно), в качестве значения `<имя-модуля>` выбирается имя EJB-JAR файла без расширения `.jar`. Это умолчание можно переопределить с помощью элемента `module-name` в конфигурационном файле `META-INF/ejb-jar.xml`.

Если развертываются компоненты, являющиеся частью веб-модуля (WAR-файл), по умолчанию в качестве значения `<имя-модуля>` выбирается имя WAR-файла без расширения `.war`. Это умолчание можно переопределить с помощью элемента `module-name` в конфигурационном файле `WEB-INF/web.xml`.

Если развертываются компоненты, являющиеся частью приложения (EAR-файл), по умолчанию значение `<имя-модуля>` определяется в зависимости от того, являются ли компоненты EJB частью EJB-JAR или WAR в EAR. При развертывании компонентов из WAR-файла, выбор значения `<имя-модуля>` выполняется в соответствии с правилом для веб-модулей. Это правило можно переопределить в файле `WEB-INF/web.xml`. При развертывании компонентов из файлов EJB-JAR, значением `<имя-модуля>` становится полный путь к каталогу, где находится файл EJB-JAR внутри EAR, плюс имя файла EJB-JAR без расширения `.jar`. Это правило можно переопределить в файле `META-INF/ejb-jar.xml`.

<имя-компонента>

Значение `<имя-компонента>` является обязательным и всегда присутствует в имени ресурса. Для компонентов EJB, объявленных с применением аннотаций `@Stateless`, `@Stateful` или `@Singleton`, в качестве значения `<имя-компонента>` по умолчанию выбирается *неквалифицированное* имя класса компонента. Это умолчание можно переопределить с помощью атрибута `name()` аннотации. Для компонентов EJB, объявленных в файле `ejb-jar.xml`, значение `<имя-компонента>` определяется с помощью элемента `bean-name`.

[!полное-квалифицированное-имя-интерфейса]

Значение `[!полное-квалифицированное-имя-интерфейса]` является обязательным, и переносимые имена компонентов EJB с этим элементом всегда будут присутствовать в JNDI. Но сервер EE также требует, чтобы в JNDI присутствовало имя без этого значения. Такое «обрезанное» имя может пригодиться, когда доступ к компоненту осуществляется через единственный интерфейс (или если компонент вообще не имеет интерфейса). Прояснить все вышесказанное помогут следующие примеры. В качестве значений для `[!полное-квалифицированное-имя-интерфейса]` выбираются полные квалифицированные имена всех локальных, удаленных, локальных домашних (в EJB 2) или удаленных домашних (в EJB 2) интерфейсов, или полное квалифицированное имя класса компонента, если класс не реализует никакие интерфейсы.

Примеры

Следующий сеансовый компонент без сохранения состояния реализует единственный интерфейс:

```
package com.bazaar;
@Stateless
public class AccountBean implements Account { ... }
```

При развертывании из файла `accountejb.jar` он получит следующие имена в JNDI:

```
java:global/accountejb/AccountBean
java:global/accountejb/AccountBean!com.bazaar.Account

java:app/accountejb/AccountBean
java:app/accountejb/AccountBean!com.bazaar.Account

java:module/AccountBean
java:module/AccountBean!com.bazaar.Account
```

При развертывании из `accountejb.jar`, находящегося внутри `accountapp.ear`, он получит следующие имена в JNDI:

```
java:global/accountapp/accountejb/AccountBean
java:global/accountapp/accountejb/AccountBean!com.bazaar.Account

java:app/accountejb/AccountBean
java:app/accountejb/AccountBean!com.bazaar.Account

java:module/AccountBean
java:module/AccountBean!com.bazaar.Account
```

При развертывании из `accountweb.war`, он получит следующие имена в JNDI:

```
java:global/accountweb/AccountBean
java:global/accountweb/AccountBean!com.bazaar.Account

java:app/accountweb/AccountBean
java:app/accountweb/AccountBean!com.bazaar.Account

java:module/AccountBean
java:module/AccountBean!com.bazaar.Account
```

Теперь, когда вы знаете, как производится назначение переносимых JNDI-имен компонентам EJB, обратим свои взоры на аннотацию `@EJB`, устраняющую все сложности, связанные с поиском в JNDI, и позволяющую серверу EE внедрять компоненты по мере необходимости.

5.2.3. Внедрение зависимостей с применением @EJB

Аннотация `@EJB` была добавлена с целью поддержки внедрения компонентов EJB в клиентский код, без необходимости выполнять поиск в JNDI. Несмотря на то, что выбор переносимых JNDI-имен для компонентов EJB упрощает их поиск, сама необходимость поиска накладывает значительную долю ответственности на разработчика за внедрение необходимых зависимостей. Аннотация `@EJB` перекладывает эту ответственность на сервер EE, и контейнер EJB будет автоматически внедрять требуемые зависимости. В листинге 5.2 приводится определение аннотации `@EJB`.

Листинг 5.2. Аннотация javax.ejb.EJB

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
  public @interface EJB {
    String name() default "";
    Class beanInterface() default Object.class;
    String beanName() default "";
    String lookup() default "";
}
```

Все параметры аннотации `@EJB` являются необязательными, и в большинстве случаев сервер EE сможет внедрить компонент, зная только его тип. Параметры предназначены для разрешения более сложных ситуаций, встречающихся очень редко. Описание этих параметров приводится в табл. 5.6.

Таблица 5.6. Параметры аннотации @EJB

Параметр	Описание
`name`	Помимо выполнения операции внедрения, аннотация `@EJB` неявно создает привязку, ссылающуюся на внедренный компонент EJB в пространстве имен `java:comp`. Делается это в основном для обратной совместимости. Данный атрибут позволяет указать имя для неявной привязки. Этот параметр является эквивалентом элемента `<ejb-ref-name>` в дескрипторе развертывания для EJB 2.
`beanInterface`	Этот параметр помогает при необходимости сузить тип ссылок на компонент EJB и может быть локальным прикладным интерфейсом, удаленным прикладным интерфейсом или классом компонента, если он не реализует никаких интерфейсов.
`beanName`	Соответствует значению параметра `name` аннотаций `@Stateless` и `@Stateful` или тега `<ejb-name>` в дескрипторе развертывания в определении компонента. Подобно параметру `beanInterface` помогает сузить выбор компонента для внедрения, когда имеется несколько компонентов, реализующих один и тот же интерфейс.
`lookup`	Имя компонента, поиск которого фактически будет осуществляться в JNDI. Это наиболее часто используемый параметр.
`mappedName`	Используется некоторыми производителями как имя компонента.
`description`	Описание компонента.

5.2.4. Когда следует использовать внедрение зависимостей EJB

Было бы совсем неплохо использовать механизм внедрения зависимостей EJB всегда, когда это возможно. Но дело в том, что аннотация `@EJB` предназначалась для внедрения только сеансовых компонентов и только в управляемых окружениях. Эту аннотацию можно использовать внутри компонентов EJB; в коде, выполняющемся внутри контейнера клиентских приложений (Application-Client Container, ACC); или в компонентах, зарегистрированных в веб-контейнерах (в сервлетах или JSF-компонентах). Если компонент EJB понадобится в каком-то другом коде, следует использовать реестр JNDI и выполнять поиск непосредственно.

5.2.5. Аннотация @EJB в действии

Аннотация `@EJB` очень легко включается в работу. Кроме того, она является довольно мощным инструментам, благодаря рациональному выбору значений по умолчанию, и позволяет быстро внедрить компонент EJB с минимальными усилиями. Но она же является и весьма гибким инструментом, давая возможность обрабатывать уникальные ситуации, когда значений по умолчанию недостаточно и требуется указать дополнительные параметры внедрения. Ниже приводится несколько примеров, демонстрирующих использование аннотации `@EJB`.

Внедрение с параметрами по умолчанию

Представьте, что имеется следующий локальный интерфейс и реализующие его компонент:

```
@Local
public interface AccountLocal {
  public void Account getAccount(String accountId);
}
@Stateless(name="accountByDatabase")
public class AccountByDatabaseEjb implements AccountLocal {. . .}
```

Чтобы внедрить этот компонент в ресурс, управляемый контейнером, например в сервлет, аннотацию `@EJB` можно использовать без каких-либо дополнительных параметров, позволив контейнеру самому определить их значения:

```
@EJB
AccountLocal accountInfo;
```

Обнаружив аннотацию `@EJB` в этом примере, контейнер определит, что требуется внедрить компонент, реализующий интерфейс `AccountLocal`. Он выполнит поиск среди зарегистрированных компонентов и внедрит экземпляр `AccountByDatabaseEjb`, потому что этот компонент реализует требуемый интерфейс.

Внедрение с параметром beanName

Теперь представьте, что имеется несколько реализаций интерфейса `AccountLocal`. В предыдущем примере представлена реализация, взаимодействующая с базой данных; а теперь добавим к ней реализацию, использующую Active Directory:

```
@Stateless(name="accountByActiveDirectory")
public class AccountByActiveDirectoryEjb implements AccountLocal {. . .}
```

При наличии нескольких компонентов, реализующих один и тот же интерфейс `AccountLocal`, требуется дать аннотации `@EJB` небольшую подсказку, чтобы контейнер смог определить, какую из реализаций следует внедрить:

```
@EJB(beanName="accountByActiveDirectory")
AccountLocal accountInfo;
```

Параметр `beanName` в этом примере сузит круг поиска реализации для внедрения. Контейнер найдет компонент с именем `accountByActiveDirectory`, убедится, что он реализует интерфейс `AccountLocal` и внедрит экземпляр `AccountByActiveDirectoryEjb` в переменную `accountInfo`.

Внедрение с параметром beanInterface

Часто бывает желательно сделать один и тот же интерфейс и локальным, и удаленным. Организовать это можно так:

```
public interface AccountServices {
  public Account getAccount(String accountId);
}

@Local
public interface AccountLocal extends AccountServices {}

@Remote
public interface AccountRemote extends AccountServices {}

@Stateless
public class AccountEjb implements AccountLocal, AccountRemote {. . .}
```

С помощью параметра `beanInterface` аннотации `@EJB` можно сообщить контейнеру, какой интерфейс следует внедрить – локальный или удаленный:

```
@EJB(beanInterface="AccountLocal.class")
AccountServices accountServices;
```

Параметр `beanInterface` аннотации `@EJB` в этом примере сузит круг поиска реализации для внедрения. Контейнер найдет компонент с интерфейсом `AccountLocal.class`. При внедрении компонента, его тип будет приведен к типу `AccountServices`. Аннотация `@EJB` в данном случае сообщает контейнеру, что вы желаете использовать локальный компонент, но взаимодействовать с ним через интерфейс `AccountServices`. Это отличная возможность для тех, кто хочет совместить реализацию локальной и удаленной служб в одном классе.

Внедрение с параметром lookup

Использование параметра `lookup` позволяет устранить все неоднозначности, которые только могут возникнуть при внедрении EJB. Если указать этот параметр, контейнер не будет пытаться определить правильный компонент – вы явно сообщаете ему JNDI-имя компонента, который требуется внедрить:

```
@EJB(lookup="java:module/DefaultAccountService")
AccountServices accountServices;
```

Параметр `lookup` в этом примере сузит круг поиска реализации для внедрения. Вы можете указать любую действительную область видимости.

Аннотация `@EJB` имеет очень широкие возможности: она заставляет контейнер выполнить поиск и внедрение компонента за вас. Вообще говоря, аннотация

`@EJB` является частным случаем механизма внедрения зависимостей, поддерживая только внедрение компонентов EJB. Однако существуют и другие ресурсы, управляемые контейнером (такие как источники данных JDBC, очереди JMS и сеансы JavaMail), которые может потребоваться внедрить в код. Эти ресурсы можно внедрять с помощью аннотации `@Resource`, как описывается в следующем разделе.

Внедрение с использованием прокси-объектов

Что фактически внедряется при внедрении компонента EJB? Ответ: прокси-объект. Контейнер создает специальный промежуточный объект, который называют прокси-объектом (proxy object), и внедряет его, а не сам класс компонента. Делается это для того, чтобы контейнер мог правильно управлять всеми службами EJB, такими как транзакции и поддержка многопоточного выполнения. Кроме того, прокси-объект гарантирует правильность доступа к соответствующему компоненту. Например, представьте, что имеется следующий простой сеансовый компонент без сохранения состояния:

```
@Stateless
public class BidService {
}
```

Так как всем ссылкам на этот компонент контейнером будет назначена одна и та же идентичность, прокси-объект гарантирует корректность возвращаемого объекта. Вы можете использовать метод `equals()` для проверки, так ли это на самом деле:

```
@EJB
BidService bid1;
@EJB
BidService bid2;
. . .
bid1.equals(bid2); // вернет true
```

С другой стороны, пусть имеется следующий простой сеансовый компонент с сохранением состояния:

```
@Stateful
public class ShoppingCart {
}
```

В этом случае контейнер потребует присвоить различные идентичности разным экземплярам сеансового компонента с сохранением состояния. Прокси-объект обслужит это требование и метод `equals()` поможет вам убедиться в этом:

```
@EJB
ShoppingCart cart1;
@EJB
ShoppingCart cart2;
. . .
cart1.equals(cart2); // вернет false
```

5.2.6. Внедрение ресурсов с помощью аннотации @Resource

Аннотация `@Resource`, безусловно, самый универсальный механизм внедрения ресурсов в EJB 3. В большинстве случаев она используется для внедрения источ-

ников данных JDBC, ресурсов JMS и контекстов EJB. Но эта аннотация может также применяться для внедрения всего, что только имеется в реестре JNDI. Определение аннотации `@Resource` приводится в листинге 5.3.

Листинг 5.3. Аннотация javax.annotation.Resource

```
@Target({TYPE, FIELD, METHOD}) @Retention(RUNTIME)
  public @interface Resource {
    String name() default "";
    String lookup() default "";
    Class type() default java.lang.Object.class;
    AuthenticationType authenticationType()
        default AuthenticationType.CONTAINER;
    boolean shareable() default true;
}
```

Все параметры аннотации `@Resource` являются необязательными. Как и в случае с компонентами EJB, сервер EE обычно легко справляется с задачей внедрения ресурса, зная только его тип. Но так бывает не всегда, и в таких ситуациях параметры помогут сделать правильный выбор. Например, вам может потребоваться внедрить `DataSource`, но, если сервер EE настроен на работу с несколькими источниками данных, вам придется указать значение в параметре `name` или `lookup`, чтобы сузить круг поиска внедряемого ресурса. Описание этих параметров приводится в табл. 5.7.

Таблица 5.7. Параметры аннотации @Resource

Параметр	Описание
`name`	Помимо выполнения операции внедрения, аннотация `@Resource` неявно создает привязку, ссылающуюся на внедренный ресурс в пространстве имен `java:comp`. Делается это в основном для обратной совместимости. Данный атрибут позволяет указать имя для неявной привязки. Этот параметр является эквивалентом элемента `<res-ref-name>` в дескрипторе развертывания для EJB 2.
`lookup`	Имя ресурса, поиск которого фактически будет осуществляться в JNDI. Это наиболее часто используемый параметр.
`type`	Тип ресурса. Если аннотация `@Resource` применяется к полю, по умолчанию используется тип поля. Если аннотация применяется к методу записи (setter method), используется тип параметра метода.
`authenticationType`	Может иметь значение `AuthenticationType.CONTAINER` или `AuthenticationType.APPLICATION`. Используется только для ресурсов, являющихся фабриками соединений, таких как источники данных.
`shareable`	Определяет, может ли ресурс совместно использоваться сразу несколькими компонентами. Используется только для ресурсов, являющихся фабриками соединений, таких как источники данных.
`mappedName`	Используется некоторыми производителями как имя ресурса.
`description`	Описание ресурса.

Параметры аннотации `@Resource` используются точно так же, как и параметры аннотации `@EJB`.

5.2.7. Когда следует использовать внедрение ресурсов

Аннотация `@Resource` применяется для внедрения ресурсов, управляемых контейнером, в код, так же управляемый контейнером. Данная аннотация действует только внутри компонентов EJB, MDB, в коде, выполняющемся внутри контейнера клиентских приложений (ACC), или в компонентах, зарегистрированных в веб-контейнере (таком как сервлет или JSF-компонент). Спектр поддерживаемых ресурсов простирается от простых целочисленных значений, до сложнейших объектов `DataSource`: если ресурс управляется контейнером, его можно внедрить с помощью аннотации `@Resource`.

5.2.8. Аннотация @Resource в действии

Давайте поближе рассмотрим порядок внедрения таких ресурсов, как `DataSource`, JMS, `EJBContext`, JavaMail, таймеры и элементы окружения. Начнем с уже знакомого примера `DataSource`, чтобы объяснить основные возможности аннотации `@Resource`, прежде чем перейти к более сложным случаям.

Внедрение DataSource

Следующий фрагмент внедряет объект `DataSource` в компонент `DefaultBidService`:

```
@Stateless
public class DefaultBidService implements BidService {
  ...
  @Resource(lookup="java:global/jdbc/ActionBazaarDB")
  private DataSource dataSource;
```

Параметр `lookup` обеспечивает прямой поиск в JNDI и данный пример предполагает, что сервер EE настроен на использование объекта `DataSource`, связанного с этим местоположением. Если имя ресурса изменится или если на другом сервере EE объект `DataSource` будет связан с другим местоположением, попытка внедрения потерпит неудачу.

Внедрение ресурсов JMS

Вспомните обсуждение темы обмена сообщениями и использования компонентов MDB в главе 4. Если приложение использует механизм обмена сообщениями, ему наверняка понадобятся ресурсы JMS, такие как `javax.jms.Queue`, `javax.jms.Topic`, `javax.jms.QueueConnectionFactory` и `javax.jms.TopicConnectionFactory`. Подобно источникам данных JDBC, эти ресурсы хранятся в реестре JNDI сервера приложений и могут внедряться с помощью аннотации `@Resource`. Например, следующий фрагмент внедряет очередь `Queue` с именем `"java:global/jms/ActionBazaarQueue"` в поле `queue`:

```
@Resource(lookup="java:global/jms/ActionBazaarQueue")
private Queue queue;
```

Внедрение EJBContext

Выше мы уже знакомились с интерфейсами `EJBContext`, `SessionContext` и `MessageDrivenContext`. Механизм внедрения ресурсов часто используется для получения доступа к `EJBContext`. Следующий фрагмент, взятый из определения сеансового компонента `DefaultBidService`, внедряет контекст EJB в переменную экземпляра `context`:

```
@Resource
private SessionContext context;
```

Обратите внимание, что внедряемый контекст сеанса не хранится в JNDI. Фактически, было бы неправильным указать параметр `lookup` в данном случае, и сервер просто проигнорировал бы его. Когда контейнер обнаруживает, что аннотация `@Resource` применяется к переменной, которая должна хранить ссылку на контекст, он выясняет, какой из контекстов EJB действует для данного экземпляра компонента, производя поиск по типу данных `javax.ejb.SessionContext` переменной. Так как `DefaultBidService` – это сеансовый компонент, результат внедрения будет тот же, как если бы в качестве типа переменной был указан родительский класс, `EJBContext`. Следующий фрагмент все равно внедрит в переменную `context` соответствующий экземпляр `javax.ejb.SessionContext`, даже при том, что она объявлена с типом `javax.ejb.EJBContext`:

```
@Resource
private EJBContext context;
```

Такой способ внедрения может пригодиться в сеансовых компонентах, где не предполагается использовать методы, доступные через интерфейс `SessionContext`.

Внедрение элементов окружения

Если вы уже имеете опыт создания корпоративных приложений, то наверняка сталкивались с ситуациями, когда те или иные параметры приложения изменяются в зависимости от окружения и других причин (информация о сайте заказчика, версия продукта и так далее). Было бы слишком трудоемко хранить такую «полупостоянную» информацию в базе данных. Именно в таких ситуациях может пригодиться возможность внедрения элементов окружения.

Например, предположим, что в приложении ActionBazaar требуется устанавливать признак цензуры для некоторых стран. Если этот флаг установлен, ActionBazaar будет сверять объявления о продаже со списком запрещенных товаров в стране, где развернуто приложение. Внедрить элемент окружения можно, как показано ниже:

```
@Resource
private boolean censorship;
```

Элементы окружения определяются в дескрипторе развертывания и становятся доступны через JNDI. Признак цензуры в приложении ActionBazaar можно определить так:

```
<env-entry>
  <env-entry-name>censorship</env-entry-name>
  <env-entry-type>java.lang.Boolean</env-entry-type>
  <env-entry-value>true</env-entry-value>
</env-entry>
```

По сути элементы окружения играют роль прикладных констант и поддерживают небольшой диапазон типов данных. В частности, значениями тега `<env-entry-type>` могут быть Java-типы: `String`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Boolean`, `Double` и `Float`. Так как элементы окружения доступны через JNDI, они могут внедряться по именам. Как вы уже наверняка поняли, типы данных элементов окружения и переменных, куда производится внедрение, должны быть совместимыми. Иначе при попытке выполнить внедрение контейнер возбудит исключение. Отметьте, что если на основе внедрения зависимостей требуется организовать сложную конфигурацию, вам определенно следует посмотреть в сторону механизма CDI. Подробнее о CDI рассказывается в разделе 5.3.8.

Внедрение ресурсов JavaMail

В дополнение к источникам данных JDBC и ресурсам JMS, еще одним важнейшим ресурсом корпоративных приложений зачастую является ресурс JavaMail (`javax.mail.Session`). Сеансы JavaMail обеспечивают доступ к настройкам почтового сервера и хранятся в реестре JNDI сервера приложений. Объект `Session` можно внедрить в компонент EJB с помощью аннотации `@Resource` и использовать для отправки электронной почты. В приложении ActionBazaar эту возможность можно задействовать для отправки извещений клиентам, чьи ставки выиграли. Ниже представлен фрагмент кода, внедряющий объект `Session`:

```
@Resource(lookup="java:global/mail/ActionBazaar")
private javax.mail.Session mailSession;
```

Оставим определение настроек электронной почты в дескрипторе развертывания как упражнение. Пример непосредственного отображения аннотаций в дескрипторы можно найти в приложении А.

Внедрение службы таймеров

Служба таймеров контейнера дает компонентам EJB простую возможность планировать выполнения заданий (подробнее о таймерах рассказывается в главе 7). Внедрить службу таймеров в компонент EJB также можно с помощью аннотации `@Resource`:

```
@Resource
private javax.ejb.TimerService timerService;
```

Так же, как и контексты EJB, служба таймеров не зарегистрирована в JNDI, но контейнер вполне в состоянии определить, какой ресурс требуется внедрить, только исходя из типа переменной.

Однако, насколько бы мощный ни была аннотация `@Resource`, некоторые задачи оказываются ей не по плечу. Существуют ситуации, когда приходится явно выполнять поиск в JNDI. О некоторых из таких ситуаций мы поговорим далее, а также покажем, как выполнять поиск программно.

Внедрение в поля и в параметры методов записи

В подавляющем большинстве случаев ресурсы и компоненты EJB внедряются в поля объектов. На языке DI это так и называется «внедрение в поля». Но помимо внедрения в поля поддерживается также возможность внедрения в параметры методов записи (при этом следует отметить, что EJB не поддерживает внедрение конструкторов, хотя механизм CDI имеет такую возможность). Чтобы увидеть, как это работает, преобразуем пример с источником данных и произведенм внедрение в параметр метода записи:

```
@Stateless
public class DefaultBidService implements BidService {
  ...
  private DataSource dataSource;
  ...
  @Resource(lookup="java:global/jdbc/ActionBazaarDB")
  public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
}
```

Так же как и при внедрении в поле, прежде чем экземпляр компонента будет передан в работу, контейнер обнаружит аннотацию `@Resource` перед методом `setDataSource`, отыщет источник данных в JNDI, используя параметр `lookup`, и оформит вызовы метода `setDataSource`, подставив в параметр полученный источник данных.

Использовать или не использовать прием внедрения в параметр метода записи в значительной степени является вопросом личных предпочтений. Несмотря на то, что внедрение в параметр метода выглядит немного более трудоемким, все же он имеет свои преимущества. Во-первых, такой подход упрощает модульное тестирование за счет вызова общедоступного метода из фреймворка тестирования, такого как JUnit. Во-вторых, при использовании этого приема упрощается размещение кода инициализации в методе, если это необходимо.

В данном случае, соединение с базой данных можно открыть непосредственно в методе `setDataSource`, сразу после внедрения, как показано ниже:

```
private Connection connection;
...
@Resource(lookup="java:global/jdbc/ActionBazaarDB")
public void setDataSource(DataSource dataSource) {
  this.connection = dataSource.getConnection();
}
```

5.2.9. Поиск ресурсов и компонентов EJB в JNDI

Для внедрения ресурсов можно использовать аннотации `@EJB` и `@Resource`, но иногда все же приходится вручную выполнять поиск в JNDI. Существует два спо-

соба выполнить поиск программно – либо с помощью `EJBContext`, либо с помощью `InitialContext`. Далее мы рассмотрим оба.

Поиск с помощью EJBContext

Выше уже говорилось о возможности найти любой объект, хранящийся в JNDI, с помощью метода `EJBContext.lookup()` (включая и ссылки на сеансовые компоненты). Этот же прием можно использовать и для решения более сложных задач, например, для организации динамической смены ресурсов во время выполнения. Прием внедрения зависимостей, несмотря на все преимущества, вынуждает использовать статическую конфигурацию, которую нельзя изменить программно.

Использование возможности поиска для получения разных ресурсов на разных этапах выполнения напоминает динамическое конструирование инструкций SQL. Разница лишь в том, что вместо запроса к базе данных выполняется поиск в JNDI. Все, что для этого необходимо сделать, фактически сводится к динамическому конструированию имени искомого ресурса. Как результат, программы, управляемые данными и/или вводом пользователя, получают возможность определять зависимости в процессе выполнения, а не на этапе развертывания. Следующий фрагмент демонстрирует метод `EJBContext.lookup()` в действии:

```
@Stateless
public class DefaultDiscountService implements DiscountService {
  @Resource
  private SessionContext sessionContext;
  ...
  DiscountRateService discountRateService
      = (DiscountRateService) sessionContext.lookup(
            "java:app/ejb/HolidayDiscountRateService");
  ...
  long discount = discountRateService.calculateDiscount(...);
```

Этот пример демонстрирует, как найти `DiscountRateService` с помощью `SessionContext`. Примечательно, что строка с JNDI-именем искомого ресурса является частью кода и, соответственно, легко может быть изменена, если потребуется найти другие реализации `DiscountRateService`, опираясь на некоторые бизнес-правила. В данном примере искомый компонент EJB должен отображаться в пространство имен `java:app` с именем `/ejb/HolidayDiscountRateService`.

Поиск с помощью InitialContext

Несмотря на относительное удобство приема поиска с помощью `EJBContext`, проблема в том, что этот способ доступен, только внутри Java EE или контейнера клиентских приложений. Когда речь заходит о внедрении POJO, находящихся за пределами контейнера, у вас остается только самый простой способ поиска ссылок в JNDI – с использованием `InitialContext` механизма JNDI. Код, реализующий такой поиск, выглядит довольно шаблонно, зато он достаточно прост:

```
Context context = new InitialContext();
BidService bidService = (BidService)
```

```
        context.lookup("java:app/ejb/DefaultBidService");
...
bidService.addBid(bid);
```

Объект `InitialContext` можно создать в любом коде, имеющем доступ к JNDI API. Кроме того, этот объект можно использовать для доступа не только к локальному, но и к удаленному серверу JNDI. Обратите внимание, что хотя этот код выглядит довольно безобидно, его следует избегать по мере возможности. Такой механистический поиск в JNDI является одной из основных составляющих сложности EJB 2, особенно когда один и тот же фрагмент кода приходится повторять в приложении сотни раз.

5.2.10. Когда следует использовать поиск в JNDI

В большинстве случаев для внедрения ресурсов вполне достаточно аннотаций `@EJB` и `@Resource` и вам вообще не понадобится прибегать к непосредственному поиску в JNDI. Однако иногда без этого не обойтись.

Классы, не управляемые контейнером, вынуждены использовать поиск в JNDI. Аннотации `@EJB` и `@Resource` доступны только в классах, управляемых Java EE или контейнером клиентских приложений. В их число входят компоненты EJB, MDB, сервлеты и JSF-компоненты. Любые неуправляемые классы должны использовать поиск в JNDI через `InitialContext`, чтобы получить доступ к ресурсам сервера.

Доступ к ресурсам не всегда статичен, из-за чего не всегда возможно использовать аннотацию `@EJB` или `@Resource` для внедрения ресурса в объект. Если выбор внедряемого ресурса производится во время выполнения, вам придется использовать поиск в JNDI. Если подобное динамическое внедрение потребуется организовать в управляемом ресурсе, управляемом контейнером, таком как компонент EJB, можно внедрить `EJBContext` в компонент (с помощью аннотации `@Resource`, конечно же) и затем вызывать `EJBContext.lookup()` для динамического поиска требуемых ресурсов. Если динамическое внедрение необходимо выполнить в неуправляемом ресурсе, таком как вспомогательный класс, следует использовать `InitialContext`.

5.2.11. Контейнеры клиентских приложений

К настоящему моменту вы познакомились с внедрением компонентов EJB и ресурсов в приложениях, развертываемых и выполняющихся внутри сервера EE. А что если доступ к компонентам EJB и другим управляемым ресурсам потребуется получить из приложения SE, выполняющегося за пределами сервера EE? Именно для разрешения подобных проблем существует контейнер клиентских приложений (Application-Client Container, ACC).

ACC – скрытое сокровище в мире EE. Это мини-контейнер Java EE, который можно запустить из командной строки. Его можно рассматривать, как улучшенную виртуальную машину Java (JVM) с некоторыми встроенными механизмами Java EE. Внутри ACC можно запускать любые клиентские приложения Java SE, такие

как приложения на основе Swing, как если бы это была обычная виртуальная машина JVM. Достоинство контейнера клиентских приложений в том, что он распознает и обрабатывает большинство аннотаций Java EE, таких как `@EJB`. Кроме всего прочего ACC способен находить и внедрять компоненты EJB, находящиеся на удаленных серверах, взаимодействовать с удаленными компонентами EJB посредством RMI, поддерживает аутентификацию, может выполнять авторизацию, публикацию и подписку на ресурсы JMS, и так далее. Но в полном своем блеске ACC предстает, когда возникает необходимость использовать компоненты EJB в приложениях SE или внедрить ресурсы в POJO во время модульного тестирования.

Внутри ACC можно запустить любой Java-класс, обладающий методом `main`. Как правило, вам нужно упаковать свое приложение в файл JAR и определить `MainClass` в `META-INF/MANIFEST`. Дополнительно файл JAR может содержать дескриптор развертывания (`META-INF/applicationclient.xml`) и файл `jndi.properties`, определяющий параметры подключения с удаленным контейнером EJB. В качестве небольшого примера: если бы в EE-приложении ActionBazaar имелась удаленная EJB-служба для создания ставок, SE-приложение могло бы легко внедрить ее в статическое свойство, как показано ниже:

```
@EJB
private static BidService remoteBidService;

public class BidServiceClient {
  public static void main(String[] args) {
        System.out.println("result = " + remoteBidService.addBid(bid));
  }
```

Процедура запуска приложения SE в контейнере ACC зависит от конкретных особенностей сервера EE. Давайте посмотрим, как сделать это в GlassFish. Допустим, что приложение SE упаковано в JAR-файл с именем `bidservice-client.jar`. Запустить его в контейнере ACC под управлением сервера приложений Sun Microsystems GlassFish можно следующим образом:

```
appclient -client bidservice-client.jar
```

Подробности, касающиеся использования ACC, ищите в документации к своему серверу EE.

5.2.12. Встраиваемые контейнеры

Спецификация Java EE определяет стандартную среду выполнения корпоративных приложений внутри сервера EE, но не все приложения создаются для работы в этой среде. Тем не менее, в большинстве приложений будут востребованы такие типичные услуги, как DI и транзакции, которые предоставляют серверы EE. В таких ситуациях с успехом можно использовать встраиваемые контейнеры. *Встраиваемый контейнер* – это контейнер EJB, размещаемый в памяти, внутри которого можно выполнять приложения SE (или модульные тесты) под управлением собственной виртуальной машины JVM. В результате приложение SE получает в свое распоряжение большую часть возможностей, предоставляемых компонентами EJB.

Кто-то может подумать, что ACC и встраиваемые контейнеры суть одно и то же, однако это далеко не так. Назначение контейнера ACC – обеспечить подключение приложения SE к удаленному серверу EE, чтобы открыть доступ к его ресурсам. То есть, все ресурсы обрабатываются как удаленные. Для приложения во встраиваемом контейнере все ресурсы являются локальными, потому что приложение SE размещает их в собственной памяти контейнера EJB, который превращается для них в небольшой сервер EE.

Создание встроенного контейнера

Класс `javax.ejb.embeddable.EJBContainer` – это абстрактный класс, который может быть реализован сервером EE, хотя это и не является обязательным требованием. Создание экземпляра возлагается на статический метод:

```
EJBContainer ec = EJBContainer.createEJBContainer();
```

Приложение SE может затем использовать объект `EJBContainer` для взаимодействий с контейнером EJB, например, оно может выполнять поиск и получать экземпляры компонентов EJB. В табл. 5.8 перечислены некоторые методы класса `EJBContainer`.

Таблица 5.8. Методы класса EJBContainer

Метод	Описание
`close()`	Завершает работу встроенного контейнера EJB.
`createEJBContainer()`	Создает экземпляр `EJBContainer` с параметрами по умолчанию.
`createEJBContainer(Map<?,?>)`	Создает экземпляр `EJBContainer`, определяя значения некоторых параметров.
`getContext()`	Возвращает `javax.naming.Context`.

Регистрация компонентов EJB

При создании контейнера вызовом метода `createEJBContainer()` автоматически будет выполнен поиск компонентов EJB в пути поиска классов приложения (classpath). Если не требуется сканировать все каталоги в пути поиска классов, или по каким-то причинам необходимо загрузить только определенные компоненты EJB, используйте метод `createEJBContainer(java.util.Map<?,?>)` и включите в ассоциативный массив имена загружаемых модулей. Например, при следующих параметрах будут загружены только компоненты EJB в файлах `bidservice.jar` и `accountservice.jar`; компоненты в других модулях будут игнорироваться.

```
Properties props = new Properties();
props.setProperty(EJBContainer.MODULES,
    new String[]{"bidservice","accountsevice"});
EJBContainer ec = EJBContainer.createEJBContainer(props);
```

При обнаружении компонента EJB встроенный контейнер зарегистрирует его в JNDI, определив для него переносимое глобальное имя JNDI, как это делает пол-

ноценный сервер EE. Например, допустим, что в пути поиска классов присутствует `bidservice.jar`, имеющий интерфейс `@Local com.bazaar.BidServiceLocal` и его реализацию `@Stateless com.bazaar.BidServiceEjb`. Встроенный контейнер зарегистрирует его под именами:

```
java:global/bidservice/BidServiceEjb
java:global/bidservice/BidServiceEjb!com.bazaar.BidServiceLocal
```

Встраиваемые контейнеры поддерживают компоненты EJB с интерфейсами `@Local` или без интерфейсов, а компоненты `@Remote` – нет.

Поиск компонентов EJB

Поиск компонентов EJB в JNDI выполняется точно так же, как и внутри полноценных контейнеров. Поиск классов, не управляемых встроенным контейнером, осуществляется с помощью объекта `Context`, возвращаемого методом `EJBContainer.getContext()`. Например, получить компонент `BidServiceLocal` можно так:

```
Context ctx = ec.getContext();
BidServiceLocal bs1 = ctx.lookup("java:global/bidservice/BidServiceEjb");
```

И, конечно же, внедрение компонентов EJB, управляемых внедренным контейнером, можно выполнять с помощью аннотаций `@EJB` и `@Resource`.

Завершение

По окончании использования встроенного контейнера, вызывайте `ec.close()`, чтобы завершить его работу и освободить ресурсы. Завершение работы контейнера не означает, что следует завершить и само приложение SE – завершить работу встроенного контейнера может потребоваться по самым разным причинам. Закрыв контейнер, приложение может запустить новый, если это потребуется.

Встроенный контейнер может быть так же дополнен поддержкой интерфейса `AutoCloseable`. Благодаря чему его можно использовать в инструкции `try-with-resource` и быть уверенными, что контейнер завершится автоматически. Ниже показано, как можно использовать инструкцию `try-with-resource`:

```
try (EJBContainer ec = EJBContainer.createEJBContainer())
{
    // сделать нечто со встроенным котейнером
} catch (Throwable t) {
    t.printStackTrace();
}
```

5.2.13. Эффективный поиск и внедрение компонентов EJB

Аннотация `@EJB` – самый простой, быстрый и безопасный способ связать ваше приложение с экземплярами компонентов. В большинстве случаев контейнер EJB может сам определить, какой компонент требуется создать и внедрить, по его клас-

су или интерфейсу. Но когда имеется несколько компонентов, реализующих один и тот же интерфейс, и программа ссылается на компоненты по этому интерфейсу, без поиска не обойтись. Механизм внедрения CDI имеет более удобное решение, которое мы рассмотрим далее.

5.2.14. Механизмы внедрения EJB и CDI

Аннотация `@EJB` является самым простым, удобным и достаточно мощным инструментом сервера EE для внедрения компонентов EJB в управляемые ресурсы. Аннотация `@EJB` стала большим шагом в направлении упрощения разработки корпоративных приложений и ввела поддержку DI в контейнер EJB. Но аннотация `@EJB` может внедрять только компоненты EJB и только в управляемые ресурсы, такие как другие компоненты EJB, JSF-компоненты и сервлеты. Механизм CDI, напротив, обладает намного более широкими возможностями и его аннотация `@Inject` способна внедрять практически все что угодно и куда угодно, в том числе и компоненты EJB. Но, если аннотация `@Inject` так хороша и способна внедрять компоненты EJB, зачем тогда вообще использовать аннотацию `@EJB`?

В простых случаях аннотации `@EJB` и `@Inject` можно считать взаимозаменяемыми. Допустим, что у нас имеется компонент `SimpleBidService`, являющийся простейшим сеансовым компонентом EJB без сохранения состояния из приложения ActionBazaar:

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

В данном случае аннотации `@EJB` и `@Inject` будут действовать совершенно одинаково и с одинаковыми результатами:

```
@Inject
SimpleBidService bidService;

@EJB
SimpleBidService bidService;
```

Но с усложнением примера, в механизме CDI начинают проявляться дополнительные проблемы. Например, допустим, что `BidService` – это интерфейс, имеющий множество реализаций:

```
public interface BidService { ... }

@Stateless(name="defaultBids")
public class DefaultBidService implements BidService { ... }

@Stateless(name="clearanceBids")
public class ClearanceBidService implements BidService { ... }
```

Аннотация `@EJB` позволяет легко справиться с этой проблемой несколькими способами, основным из которых является использование параметра `beanName`:

```
@EJB(beanName="clearanceBids")
BidService clearanceBidService;
```

Аннотация `@Inject` действует немного иначе и требует дополнительных усилий, чтобы сузить выбор внедряемой реализации `BidService`. Она требует создания класса-производителя в соответствии с шаблоном фабрики:

```
public class BidServiceProducer {
  @Produces
  @EJB(beanName="defaultBids")
  @DefaultBids
  BidService defaultBids;

  @Produces
  @EJB(beanName="clearanceBids")
  @ClearanceBids
  BidService clearanceBids;
}
```

Теперь аннотацию `@Inject` можно объединить с квалификаторами, используемыми в `BidServiceProducer`, для внедрения нужного экземпляра компонента `BidService`:

```
@Inject @ClearanceBids
BidService clearanceBidService;
```

В конечном счете, чтобы внедрить компонент EJB, кроме самых простых случаев, даже при использовании механизма CDI, приходится опираться на аннотацию `@EJB` в классах-производителях (producer classes), чтобы гарантировать создание правильного экземпляра компонента EJB.

5.3. AOP в мире EJB: интерцепторы

Приходилось ли вам сталкиваться с ситуацией, когда требования изменяются к концу работы над проектом и вам предлагается добавить в компоненты EJB приложения какую-нибудь общую для всех особенность, такую как журналирование или аудит? Добавление подобных особенностей в каждый компонент может потребовать слишком много времени, к тому же большой объем типового кода, который потребуется включить в Java-классы, усложнит сопровождение приложения. Но все не так плохо: подобные задачи можно решать с помощью интерцепторов EJB 3. В этом разделе мы расскажем, как создать простой интерцептор, осуществляющий журналирование. Мы также покажем, как этот интерцептор можно сделать интерцептором по умолчанию, выполняемым всякий раз, когда вызывается метод компонента. В этом разделе вы узнаете как действуют интерцепторы.

5.3.1. Что такое AOP?

Вполне возможно, что вы уже сталкивались с термином *аспектно-ориентированное программирование* (Aspect-Oriented Programming, AOP). Основная идея AOP

состоит в том, что типовой код, не связанный с решением прикладных задач и повторяющийся от компонента к компоненту, должен рассматриваться как часть поддерживающей инфраструктуры и не должен смешиваться с прикладной логикой. Часто для обозначения такой функциональности используют термин *сквозная функциональность* (crosscutting concerns) – функциональность, которая насквозь пронизывает прикладную логику.

Сквозная функциональность

Часто в качестве примера сквозной функциональности демонстрируется реализация журналирования, например, с целью отладки. Воспользуемся приложением ActionBazaar в качестве основы и допустим, что нам потребовалось обеспечить журналирование вызова каждого метода в системе. Обычно подобное требование означает необходимость добавления инструкций журналирования в начало каждого отдельного метода в системе, которые выводили бы в журнал строку вида: «вход в метод XX»! Эти инструкции, не имеющие ничего общего с прикладной логикой, пришлось бы разбросать по всему приложению. Другими распространенными примерами сквозной функциональности являются учет, профилирование и сбор статистической информации.

Поддержка аспектно-ориентированного программирования позволяет выделять сквозную функциональность в отдельные модули. Журналирование, учет, профилирование и сбор статистической информации – все эти задачи можно было бы реализовать в виде отдельных модулей. Затем эти модули можно было бы применять к соответствующим «сквозным» фрагментам прикладного кода, таким как начало или конец каждого метода. В EJB 3 поддержка сквозной функциональности реализована как возможность перехвата вызовов прикладных методов и методов обработки событий жизненного цикла.

Итак, погрузимся в мир интерцепторов EJB 3. В этом разделе вы узнаете, что такое интерцепторы и как создаются интерцепторы для прикладных методов и методов обработки событий жизненного цикла.

5.3.2. Основы интерцепторов

Чтобы лучше понять суть интерцепторов EJB, давайте сначала коротко пройдемся по основным понятиям аспектно-ориентированного программирования (AOP).

Первый: *задача, функциональность* (crosscutting concern – сквозная функциональность). Этот термин обозначает задачу, которую требуется решить, или функциональность, которую нужно реализовать. Например: «Мне нужно добавить журналирование во все прикладные методы», – или: «Мне требуется обеспечить округление всех чисел до определенного числа знаков после запятой».

Второй: *совет* (*advice*). Совет – это фактический код, решающий поставленную задачу или реализующий требуемую функциональность. Например, если задача состоит в том, чтобы обеспечить регистрацию в базе данных вызовов любых методов, советом будет код, устанавливающий соединение с базой и выполняющий запрос для вставки данных.

Третий: *точка сопряжения* (*pointcut*). Точка сопряжения описывает точку в приложении, по достижении которой следует выполнить указанный *совет*. Типичными примерами точек сопряжения являются точки непосредственно перед вызовом методов и сразу после вызова.

Четвертый: *аспект* (aspect). Аспект – это комбинация точки сопряжения и совета. Поддержка аспектно-ориентированного программирования вплетает советы в соответствующие точки сопряжения, используя аспекты.

Интерцепторы EJB 3 являются наиболее общей формой интерцепторов – они представляют советы, окружающие точку сопряжения. Интерцептор вызывается в точке входа в метод и продолжает выполняться после возвращения из метода, благодаря чему интерцептор может исследовать возвращаемое значение метода. Интерцептор может также перехватить любое исключение, возникшее в методе. Интерцепторы можно применять к сеансовым компонентам и к компонентам, управляемым сообщениями.

Хотя интерцепторы EJB 3 обеспечивают достаточный объем возможностей для решения наиболее общих сквозных задач, они все же далеки по своим возможностям от полномасштабных пакетов поддержки AOP, таких как AspectJ. С другой стороны, интерцепторы EJB 3 проще в использовании. Теперь, когда вы познакомились с некоторыми основами интерцепторов, посмотрим, как ими пользоваться.

5.3.3. Когда следует использовать интерцепторы

Интерцепторы EJB 3 предназначены для использования с сеансовыми компонентами и компонентами MDB. Они представляют аспекты, окружающие вызовы методов, то есть с их помощью можно выполнить некоторые операции перед вызовом метода, исследовать возвращаемое значение метода и обработать любые исключения, генерируемые методом. Перед вызовом метода обычно выполняется сквозная логика, осуществляющая учет (журналирование, сбор статистической информации и так далее) или проверяющая и при необходимости изменяющая параметры, прежде чем они попадут в метод. После вызова метода, опять же обычно, выполняется сквозная логика, осуществляющая учет, а также проверяющая возвращаемое значение и изменяющая его перед передачей вызывающему коду, если это потребуется. Когда интерцептор перехватывает и обрабатывает исключения, генерируемые методами, снова может быть выполнена логика, осуществляющая учет, но интерцептор может также попытаться повторно вызвать метод или направить выполнение программы по другому пути. Теперь, когда вы знаете, для каких целей применяются интерцепторы, давайте посмотрим, как реализовать простейший интерцептор, осуществляющий журналирование.

5.3.4. Порядок реализации интерцепторов

Для реализации интерцептора EJB 3 требуется всего две аннотации. Первая: `@AroundInvoke`. Эта аннотация применяется к методу класса, который будет играть роль совета интерцептора. Не забывайте, что совет – это фактический код,

который будет вплетаться в вызовы прикладных методов сеансовых компонентов или компонентов MDB. В листинге 5.4 приводится короткий пример интерцептора. Подробно эта реализация обсуждается в разделе 5.3.6.

Листинг 5.4. Метод с аннотацией @AroundInvoke

```
@Interceptor
public class SayHelloInterceptor {
  @AroundInvoke
  public Object sayHello(InvocationContext ctx) throws Exception {
    System.out.println("Hello Interceptor!");
    return ctx.proceed();
  }
}
```

Вторая аннотация: `@Interceptors`. Эта аннотация применяется к классам компонентов EJB и MDB, и определяет порядок применения интерцепторов к методам классов. В листинге 5.5 показано применение аннотации `@Interceptors` к отдельным методам.

Листинг 5.5. Применение аннотации @Interceptors к методам компонента EJB

```
@Stateless
public class OrderBean {
  @Interceptors(SayHelloInterceptor.class)
  public Order findOrderById(String id) { return null; }
}
```

В листинге 5.6 показано применение аннотации `@Interceptors` ко всему классу компонента EJB.

Листинг 5.6. Применение аннотации @Interceptors ко всему классу компонента EJB

```
@Stateless
@Interceptors(SayHelloInterceptor.class)
public class OrderBean { }
```

Комбинация аннотаций `@AroundInvoke` и `@Interceptors` представляет собой мощный инструмент, упрощающий реализацию сквозной функциональности на основе интерцепторов EJB 3. Примеры выше дают лишь общее представление об особенностях их реализации. В разделе 5.3.6 мы представим более полный пример и детальнее остановимся на возможностях, имеющихся в EJB 3. Но прежде мы поговорим о дополнительных параметрах интерцепторов EJB 3.

5.3.5. Определение интерцепторов

Теперь, когда вы познакомились с аннотациями, связанными с интерцепторами, необходимо научиться пользоваться ими. Как и многие другие, аннотацию `@Interceptors` можно применять к отдельным методам и к целым классам. Поэтому далее мы покажем примеры обоих случаев. Однако аннотации не всесильны.

Иногда они не в состоянии справиться с возложенной на них ролью. Это относится и к аннотации `@Interceptors` – мы покажем пример, когда вам придется отказаться от аннотаций и использовать файл `ejb-jar.xml`.

Интерцепторы уровня методов и классов

Аннотация `@Interceptors` дает возможность указать один или более классов-интерцепторов для метода или класса. В листинге 5.5 единственный интерцептор подключается к методу `findOrderById()`. В этом примере перед каждым вызовом метода `findOrderById()` компонента `OrderBean` будет вызываться метод `sayHello()` класса `SayHelloInterceptor`:

```
@Interceptors(SayHelloInterceptor.class)
public Order findOrderById(String id) { ... }
```

В листинге 5.6 единственный интерцептор подключается к классу `OrderBean`. Когда интерцептор подключается к классу, он вызывается при вызове любого метода целевого класса. В данном примере метод `sayHello()` класса `SayHelloInterceptor` будет вызываться перед вызовом любого метода класса `OrderBean`:

```
@Stateless
@Interceptors(SayHelloInterceptor.class)
public class OrderBean { ... }
```

С помощью аннотации `@Interceptors` можно подключить сразу несколько интерцепторов к классу или к методу. Для этого достаточно передать аннотации список классов-интерцепторов, перечисленных через запятую. Например, следующий фрагмент подключает к классу `OrderBean` два интерцептора:

```
@Stateless
@Interceptors({SayHelloInterceptor.class, SayGoodByeInterceptor.class})
public class OrderBean { ... }
```

Интерцептор по умолчанию

Помимо интерцепторов уровня класса или метода можно также определять, так называемые, интерцепторы по умолчанию (default interceptor). Интерцептор по умолчанию – это глобальный механизм, который подключается ко всем методам всех компонентов в модуле.

Важно понимать, что область видимости интерцепторов по умолчанию ограничена модулями, в которых они определены. Если интерцептор определяется в файле EJB-JAR, он будет применяться *только* к компонентам в этом файле. Если интерцептор определен в файле JAR, входящем в состав библиотеки или файла WAR, он будет применяться только к компонентам в этом файле JAR, а не ко всему файлу WAR.

Определить интерцептор по умолчанию для модуля можно только в файле `ejb-jar.xml`. Никаких специальных аннотаций для таких интерцепторов не су-

ществует. Рассмотрим пример определения интерцептора по умолчанию для модуля EJB-JAR. В листинге 5.7 приводится фрагмент файла конфигурации с определениями двух интерцепторов для модуля ActionBazaar.

Листинг 5.7. Определение интерцепторов по умолчанию

```
<ejb-jar...>
  <!-- Объявление классов-интерцепторов -->
  <interceptors>
    <interceptor>
      <interceptor-class>com.bazaar.DefaultInterceptor1</interceptor-class>
    </interceptor>
    <interceptor>
      <interceptor-class>com.bazaar.DefaultInterceptor2</interceptor-class>
    </interceptor>
  </interceptors>

  <assembly-descriptor>
    <interceptor-binding>
      <!-- Символ * указывает, что интерцепторы подключаются
           ко всем компонентам в модуле -->
      <ejb-name>*</ejb-name>
      <!-- Список всех интерцепторов, применяемых
           ко всем компонентам EJB в модуле;
           порядок следования интерцепторов определяет
           порядок их выполнения -->
      <interceptor-class>
        com.bazaar.DefaultInterceptor2
      </interceptor-class>
      <interceptor-class>
        com.bazaar.DefaultInterceptor1
      </interceptor-class>
    </interceptor-binding>
  </assembly-descriptor>
</ejb-jar>
```

Порядок выполнения интерцепторов

У вас наверняка созрел интересный вопрос: «Если одновременно применить интерцепторы по умолчанию, уровня класса и уровня метода (что вполне допустимо), в каком порядке они будут вызываться?».

Интерцепторы вызываются в порядке сужения области их действия. То есть, первыми будут вызваны интерцепторы по умолчанию, затем интерцепторы уровня класса (в порядке их следования в аннотации `@Interceptors`) и, наконец, интерцепторы уровня метода (так же в порядке их следования в аннотации `@Interceptors`). Таков порядок вызова интерцепторов, принятый по умолчанию.

Изменить этот порядок можно в файле `ejb-jar.xml`. В листинге 5.8 показано, как отменить действие интерцепторов по умолчанию и уровня класса для компонента `OrderBean`.

Листинг 5.8. Отмена действия интерцепторов по умолчанию и уровня класса

```
<ejb-jar...>
  <interceptors>
    <interceptor>
      <interceptor-class>com.bazaar.MyInterceptor</interceptor-class>
    </interceptor>
  </interceptors>
  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>OrderBean</ejb-name>
      <!-- ❶ Отменяет действие интерцепторов по умолчанию
           и уровня класса -->
      <interceptor-order>
        <!-- ❷ Полное квалифицированное имя класса-интерцептора -->
        <interceptor-class>com.bazaar.MyInterceptor</interceptor-class>
      </interceptor-order>
    </interceptor-binding>
  </assembly-descriptor>
</ejb-jar>
```

Тег `<interceptor-order>` ❶ позволяет отменить действие любых интерцепторов по умолчанию, а также интерцепторов уровня класса, указанных в аннотации `@Interceptors`. Вместо них в данном примере файла `ejb-jar.xml` настраивается применение интерцептора `MyInterceptor` ❷. Если с помощью аннотации `@Interceptors` в компоненте определяются также интерцепторы на уровне методов, в файле `ejb-jar.xml` так же можно отменить их действие и/или изменить порядок вызова. Как это делается, показано в листинге 5.9.

Листинг 5.9. Отмена действия интерцепторов по умолчанию, а также интерцепторов уровня класса и метода

```
<ejb-jar...>
  <interceptors>
    <interceptor>
      <interceptor-class>com.bazaar.MyInterceptor</interceptor-class>
    </interceptor>
  </interceptors>
  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>OrderBean</ejb-name>
      <interceptor-order>
        <interceptor-class>com.bazaar.MyInterceptor</interceptor-class>
      </interceptor-order>
      <!-- Отменяет действие интерцепторов по умолчанию,
           уровня класса и уровня метода -->
      <method>
        <method-name>placeOrder</method-name>
      </method>
    </interceptor-binding>
  </assembly-descriptor>
</ejb-jar>
```

Этот пример почти полностью повторяет предыдущий, разница заключается лишь в наличии нового тега <method>. В данном примере файла ejb-jar.xml отменяется действие всех интерцепторов по умолчанию, уровня класса и уровня метода для метода placeOrder(). Вместо них будет вызываться интерцептор MyInterceptor.

Определение интерцепторов с помощью аннотации @Interceptors и использование файла ejb-jar.xml для их настройки или отмены – это лишь часть истории. Иногда бывает необходимо отключить интерцепторы, о чем рассказывается далее.

Отключение интерцепторов

Иногда может потребоваться отключить интерцепторы. Существует две дополнительные аннотации, с помощью которых можно отключать интерцепторы по умолчанию или уровня класса. Применение аннотации @javax.interceptor.ExcludeDefaultInterceptors к классу или к методу отключает действие интерцептора по умолчанию для класса или метода, соответственно. Аналогично, применение аннотации @javax.interceptor.ExcludeClassInterceptors отключает действие интерцептора уровня класса для метода. В следующем примере демонстрируется возможность отключения действия интерцепторов по умолчанию и уровня класса для метода findOrderById(), но имейте в виду, что интерцептор SayHelloInterceptor продолжит действовать, так как он применяется на уровне данного метода:

```
@Interceptors(SayHelloInterceptor.class)
@ExcludeDefaultInterceptors
@ExcludeClassInterceptors
public Order findOrderById(String id) { ... }
```

Теперь, когда вы узнали, как определяются и применяются интерцепторы, обратим свое внимание на реализацию самих классов-интерцепторов и понаблюдаем за ними в действии.

5.3.6. Интерцепторы в действии

Давайте реализуем простой интерцептор журналирования для компонента BidServiceBean из главы 2. В листинге 5.10 представлен программный код интерцептора. Интерцептор подключается к методу addBid() и выводит сообщение в консоль при каждом вызове метода. В действующих приложениях подобный интерцептор можно использовать для вывода отладочной информации (например, посредством java.util.logging или Log4J).

Листинг 5.10. Интерцептор для прикладных методов компонентов EJB

```
@Stateless
public class BidServiceBean implements BidService {
    // ❶ Подключение интерцептора
    @Interceptors(ActionBazaarLogger.class)
```

```
    public void addBid(Bid bid) {
    }
}

public class ActionBazaarLogger {
    // ❷ Определение типа интерцептора
    @AroundInvoke
    public Object logMethodEntry(InvocationContext invocationContext)
            throws Exception {
        System.out.println("Entering method: "
            + invocationContext.getMethod().getName());
        return invocationContext.proceed();
    }
}
```

Рассмотрим сначала этот код в общих чертах, прежде чем приступить к детальному исследованию каждой его особенности (в следующих разделах). Интерцептор `ActionBazaarLogger` подключается к методу `addBid()` сеансового компонента без сохранения состояния `PlaceBidBean` с помощью аннотации `@Interceptors` ❶. Метод `logMethodEntry()` объекта `ActionBazaarLogger` отмечен аннотацией `@AroundInvoke` ❷. Все обращения к методу `addBid()` будут перехватываться объектом `ActionBazaarLogger` и перед методом `addBid()` будет вызываться метод `logMethodEntry()`, который выводит сообщение в системную консоль, используя `InvocationContext` для включения имени целевого метода в сообщение. В заключение вызывается метод `InvocationContext.proceed()`, сообщающий контейнеру, что вызов `addBid()` был выполнен без ошибок.

Интерцепторы могут быть реализованы либо непосредственно в классе компонента, либо в виде отдельных классов. Мы рекомендуем создавать интерцепторы в виде отдельных классов, потому что такой подход обеспечивает отделение сквозной функциональности от прикладной логики и позволяет применять интерцепторы к множеству компонентов. В конце концов, разве не в этом заключается смысл AOP?

Аспекты, окружающие вызовы

Важно отметить, что интерцептор всегда должен иметь единственный метод, действующий как окружающий (`@AroundInvoke`) вызов целевого метода. Такие окружающие методы не должны содержать прикладную логику, то есть, они не должны быть общедоступными методами в прикладных интерфейсах компонентов.

Окружающий метод автоматически вызывается контейнером, когда клиент обращается к целевому методу. В листинге 5.10 вызываемый метод отмечен аннотацией `@AroundInvoke`:

```
@AroundInvoke
public Object logMethodEntry(InvocationContext invocationContext)
        throws Exception {
    System.out.println("Entering method: "
        + invocationContext.getMethod().getName());
```

```
    return invocationContext.proceed();
}
```

Фактически это означает, что метод `logMethodEntry()` будет выполняться при каждом вызове интерцептора `ActionBazaarLogger`. Как можно заметить в коде примера, любой метод, отмеченный аннотацией `@AroundInvoke`, должен следовать следующему шаблону:

```
Object <METHOD>(InvocationContext) throws Exception
```

В качестве единственного параметра методу передается интерфейс `InvocationContext`, обладающий множеством возможностей, делающих механизм AOP необычайно гибким. Метод `logMethodEntry()` использует только два метода, экспортируемых интерфейсом. Вызов `getMethod().getName()` возвращает имя метода, обращение к которому было перехвачено, – в данном случае `"addBid"`.

Вызов метода `proceed()` чрезвычайно важен для работы интерцептора. Он сообщает контейнеру, что тот должен передать управление следующему интерцептору в цепочке или прикладному методу. Отказ от вызова метода `proceed()` приведет к тому, что ни прикладной метод, ни другие методы в последовательности интерцепторов (если таковые имеются) не будут вызваны.

Данную особенность с успехом можно использовать для реализации таких процедур, как обеспечение безопасности. Например, следующий метод интерцептора предотвратит вызов прикладного метода, если проверка безопасности окончилась неудачей:

```
@AroundInvoke
public Object validateSecurity(InvocationContext invocationContext)
        throws Exception {
    if (!validate(...)) {
        throw new SecurityException("Security cannot be validated. " +
            "The method invocation is being blocked.");
    }
    return invocationContext.proceed();
}
```

Интерфейс InvocationContext

InvocationContext В листинге 5.11 приводится определение интерфейса `javax.interceptor.InvocationContext`, где можно видеть еще несколько полезных методов.

Листинг 5.11. Интерфейс javax.interceptor.InvocationContext

```
public interface InvocationContext {
    public Object getTarget();
    public Method getMethod();
    public Object[] getParameters();
    public void setParameters(Object[]);
    public java.util.Map<String,Object> getContextData();
```

```
    public Object proceed() throws Exception;
}
```

Метод `getTarget()` возвращает экземпляр компонента, которому принадлежит перехваченный метод. Этот метод особенно удобно использовать для проверки текущего состояния компонента по его переменным экземпляра или методам доступа.

Метод `getMethod()` возвращает метод компонента, вызов которого был перехвачен интерцептором. Для методов, отмеченных аннотацией `@AroundInvoke`, это будет прикладной метод компонента; для методов, перехватывающих события жизненного цикла, `getMethod()` вернет `null`.

Метод `getParameters()` возвращает параметры, переданные перехваченному методу, в виде массива объектов. Метод `setParameters()`, напротив, позволяет изменять значения параметров во время выполнения перед передачей их прикладному методу. Эти два метода могут пригодиться для реализации интерцепторов, манипулирующих параметрами с целью изменить поведение компонента. Интерцептор в приложении ActionBazaar, прозрачно округляющий все денежные суммы до двух знаков после запятой во всех методах в приложении, как раз мог бы использовать с этой целью методы `getParameters()` и `setParameters()`.

В основе метода `InvocationContext.getContextData()` лежит тот факт, что все интерцепторы в цепочке для данного прикладного метода используют один и тот же объект контекста вызова. Как результат, прием добавления данных в объект `InvocationContext` можно использовать для организации взаимодействий интерцепторов. Например, допустим, что интерцептор поддержки безопасности сохраняет в контексте вызова результат проверки привилегий пользователя:

```
invocationContext.getContextData().put("MemberStatus", "Gold");
```

Как видите, данные в контексте вызова – это обычный ассоциативный массив, хранящий пары имя/значение. Другой интерцептор в цепочке может теперь извлечь эти данные и выполнить какие-то дополнительные операции, исходя из привилегий пользователя. Например, интерцептор калькулятора скидок может уменьшить плату за размещение объявления на сайте ActionBazaar для «Золотого» пользователя. Код, извлекающий уровень привилегий, мог бы выглядеть примерно так:

```
String memberStatus =
    (String) invocationContext.getContextData().get("MemberStatus");
```

Ниже представлен метод `DiscountVerifierInterceptor`, отмеченный аннотацией `@AroundInvoke`, использующий контекст вызова, а также большинство методов, перечисленных выше:

```
@AroundInvoke
public Object giveDiscount(InvocationContext context)
        throws Exception {
    System.out.println("*** DiscountVerifier Interceptor"
        + " invoked for " + context.getMethod().getName() + " ***");
```

```
    if (context.getMethod().getName().equals("chargePostingFee")
        && (((String) (context.getContextData().get("MemberStatus")))
            .equals("Gold"))) {
        Object[] parameters = context.getParameters();
        parameters[2] = new Double((Double) parameters[2] * 0.99);
        System.out.println(
            "**** DiscountVerifier Reducing Price by 1 percent ***");
            context.setParameters(parameters);
    }
    return context.proceed();
}
```

В интерцепторах прикладных методов можно возбуждать или обрабатывать исключения времени выполнения (runtime exceptions) или контролируемые исключения (checked exceptions). Если интерцептор возбуждает исключение до вызова метода `proceed`, остальные методы в цепочке интерцепторов, а так же прикладной метод вызываться не будут.

На этом мы заканчиваем обсуждать порядок вызова интерцепторов. Но помните, что компоненты EJB имеют не только прикладные методы – они также способны обрабатывать события жизненного цикла. Хотя это не совсем очевидно, но методы-обработчики таких событий также являются разновидностью методов-интерцепторов. Обработчики событий вызываются, когда компонент переходит от одного этапа жизненного цикла к другому. Хотя это и не было показано в примерах обработки событий жизненного цикла, тем не менее, иногда такие методы можно использовать с целью реализации сквозной функциональности (например, журналирования или профилирования), общей для множества компонентов. В таких ситуациях вы можете определить в классах-интерцепторах методы-обработчики событий жизненного цикла, вдобавок к методам-интерцепторам для прикладных методов. Давайте посмотрим, как это выглядит на практике.

Методы-обработчики событий жизненного цикла в классах-интерцепторах

Как известно, аннотации `@PostConstruct`, `@PrePassivate`, `@PostActivate` и `@PreDestroy` применяются к методам, обрабатывающим события жизненного цикла. При применении этих аннотаций к методам класса-интерцептора, они действуют точно так же. Обработчики событий в классе-интерцепторе называют *интерцепторами событий жизненного цикла*. Когда целевой компонент переходит от одного этапа к другому, вызываются аннотированные методы в классе-интерцепторе.

Следующий класс-интерцептор выводит сообщение, когда какой-нибудь компонент в приложении ActionBazaar выделяет или освобождает ресурсы после его создания или перед уничтожением:

```
public class ActionBazaarResourceLogger {

    @PostConstruct
    public void initialize(InvocationContext context) {
```

```
        System.out.println("Allocating resources for bean: "
            + context.getTarget());
        context.proceed();
    }

    @PreDestroy
    public void cleanup(InvocationContext context) {
        System.out.println("Releasing resources for bean: "
            + context.getTarget());
        context.proceed();
    }
}
```

Как демонстрирует этот пример, интерцепторы событий жизненного цикла не могут возбуждать контролируемые исключения (это бессмысленно, потому что отсутствует клиент, которому могло бы быть передано исключение для обработки).

Обратите внимание, что обработчики одних и тех же событий могут быть определены и в целевом компоненте, и в одном или более интерцепторах. В связи с этим особую важность приобретает неукоснительное использование метода `InvocationContext.proceed()` в интерцепторах событий жизненного цикла, как показано в примере выше. Это гарантирует вызов следующего интерцептора в цепочке или соответствующего метода-обработчика компонента. Нет никакой разницы между применением класса-интерцептора имеющего методы-обработчики событий и не имеющего их. Класс `ActionBazaarResourceLogger`, например, применяется к компонентам так:

```
@Interceptors({ActionBazaarResourceLogger.class})
public class PlaceBidBean { ... }
```

Безусловно, обычные методы-обработчики событий жизненного цикла в компонентах, реализующие управление ресурсами, на практике используются гораздо чаще, чем интерцепторы тех же событий, реализующие сквозную функциональность (такую как журналирование или профилирование). Тем не менее, иногда интерцепторы событий оказываются просто незаменимыми.

В табл. 5.9 проводится сравнение интерцепторов для прикладных методов и событий жизненного цикла.

Таблица 5.9. Различия между интерцепторами для прикладных методов и событий. Интерцепторы событий создаются для обработки событий жизненного цикла компонентов EJB. Интерцепторы прикладных методов ассоциируются с прикладными методами и автоматически вызываются, когда пользователь обращается к прикладным методам

Поддерживаемые возможности	Интерцепторы событий	Интерцепторы прикладных методов
Вызов	Вызывается, когда возникает соответствующее событие.	Вызывается, когда происходит обращение к прикладному методу.

Таблица 5.9. (окончание)

Поддерживаемые возможности	Интерцепторы событий	Интерцепторы прикладных методов
Местоположение	В отдельном классе-интерцепторе или в классе компонента.	В классе или в классе-интерцепторе.
Сигнатура метода	В отдельном классе-интерцепторе: `void<METHOD>(InvocationContext)` В классе компонента: `void<METHOD>()`	`Object <METHOD>(InvocationContext) throws Exception`
Аннотация	`@PreDestroy`, `@PostConstruct`, `@PrePassivate`, `@PostActivate`.	`@AroundInvoke`
Обработка исключений	Может возбуждать исключения времени выполнения, но не должен возбуждать контролируемые исключения. Если метод-интерцептор возбудит исключение, никакие другие методы-обработчики вызываться не будут.	Может возбуждать исключения времени выполнения. Может перехватывать и обрабатывать исключения времени выполнения. Если метод-интерцептор возбудит исключение, никакие другие методы-интерцепторы, как и сам прикладной метод, вызываться не будут.
Транзакции и безопасность	Контекст транзакций и системы безопасности не поддерживаются. Подробнее о транзакциях и безопасности рассказывается в главе 6.	Использует тот же контекст транзакций и системы безопасности, что и прикладной метод.

Очевидно, что интерцепторы играют чрезвычайно важную роль в EJB. Вероятно, возможности аспектно-ориентированного программирования в будущих версиях EJB будут расширяться и расширяться. Безусловно, интерцепторы имеют большой потенциал, чтобы превратиться в надежный способ расширения самой платформы EJB, по мере того, как производители будут предлагать все новые и новые службы на основе интерцепторов.

5.3.7. Эффективное использование интерцепторов

Чтобы эффективно использовать интерцепторы, в первую очередь следует помнить об их предназначении. Интерцепторы предназначены для реализации сквозной логики, выполняемой до и после вызова прикладного метода. Для большей эффективности старайтесь не выходить за рамки решения сквозных задач, таких как аудит, сбор статистической информации, журналирование, обработка ошибок и так далее. Хотя интерцепторы дают возможность исследовать и изменять возвращаемое значение перехваченного метода, старайтесь избегать этого. Изменение возвращаемого значения обычно связано с прикладными задачами, решение которых возлагается на сами компоненты.

Все, о чем рассказывалось до сих пор, касалось интерцепторов EJB. Несмотря на широту возможностей и удобство в использовании, интерцепторы EJB все же предназначены только для подключения к компонентам EJB. Далее мы сравним интерцепторы EJB с их более мощными собратьями – интерцепторами CDI.

5.3.8. Интерцепторы CDI и EJB

Интерцепторы EJB удобны, но поддерживают лишь самые основные возможности. Механизм CDI продвинулся в этом отношении намного дальше и реализует поддержку привязок интерцепторов, которые могут комбинироваться множеством разных способов, обеспечивая тем самым более широкие возможности их реализации. Далее мы рассмотрим несколько примеров таких привязок интерцепторов (interceptor bindings), и вы сразу увидите их преимущества перед простыми интерцепторами EJB.

Подключение интерцепторов

Допустим, что в приложении ActionBazaar реализована некоторая сквозная задача аудита и вам требуется включить ее в работу с использованием интерцептора CDI. Сначала необходимо создать привязку интерцептора. Привязка – это связь между интерцептором и компонентом EJB. Имя привязки должно отражать ее назначение. В данном примере мы рассматриваем задачу аудита, поэтому создадим привязку с именем `@Audited`:

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

Как видите, привязка интерцептора – это аннотация, объявленная с помощью аннотации `@javax.interceptor.InterceptorBinding`. Теперь привязку `@Audited` можно использовать в прикладном коде для подключения интерцептора к компонентам. Для этого привязку требуется применить и к интерцептору, и к компоненту. Сначала посмотрим, как привязка применяется к интерцептору.

Объявление привязок для интерцепторов

При создании класса-интерцептора CDI (*совета*, в мире AOP) необходимо объявить, что этот класс является интерцептором, а также определить, какие привязки интерцепторов будут связаны с этим классом. Продолжая наш пример, создадим класс `AuditInterceptor`:

```
@Audited @Interceptor
public class AuditInterceptor {
    @AroundInvoke
    public Object audit(InvocationContext context) throws Exception {
        System.out.print("Invoking: "
            + context.getMethod().getName());
        System.out.println(" with arguments: "
```

```
            + context.getParameters());
        return context.proceed();
    }
}
```

Аннотация `@Interceptor` объявляет этот класс интерцептором. Аннотация `@Audited` связывает этот интерцептор с привязкой. Все интерцепторы CDI должны быть связаны с какими-либо привязками. При использовании обычных интерцепторов EJB (таких как `ActionBazaarLogger` в листинге 5.10) применять эти аннотации не требуется. Далее можно видеть уже знакомую нам аннотацию `@AroundInvoke`, которая сообщает механизму CDI, что при вызове интерцептора должен вызывать данный метод. Теперь, когда у нас имеется готовая привязка, одним концом связанная с интерцептором, можно завершить подключение интерцептора к компоненту EJB.

Связывание интерцепторов с компонентами

Чтобы связать интерцептор с компонентом EJB, следует применить привязку интерцептора к классу компонента или к его методам. Допустим, что функцию аудита необходимо подключить к компоненту `BidService` в приложении ActionBazaar. Мы легко может включить эту функцию для всех методов компонента, отметив аннотацией весь класс:

```
@Stateless
@Audited
public class BidService {
    ...
}
```

или только для метода `addBid()`:

```
@Stateless
public class BidService {
    @Audited
    public void addBid(Bid bid) {
        bidDao.addBid(bid);
    }
    ...
}
```

В каждом случае необходимо использовать привязку интерцептора `@Audited`. Привязка `@Audited` объявлена, как поддерживающая любые целевые элементы (`@Target({TYPE, METHOD})`), поэтому ее можно применять и к классам, и к методам. Привязки можно объявлять, как применяемые только к классам, или только к методам. Так как привязка `@Audited` связана с классом `AuditInterceptor`, при вызове метода `addBid()` этот вызов будет перехвачен и управление будет передано методу `audit()`.

Это был простой пример, демонстрирующий использование интерцепторов CDI. Однако возможности интерцепторов CDI могут быть намного шире, благо-

даря включению в привязки других привязок интерцепторов и объявлению нескольких привязок для интерцепторов. Давайте познакомимся с этими возможностями поближе.

Beans.xml

Напомним, что при использовании CDI, файл JAR-архива с реализацией компонента должен содержаться в файле `META-INF/beans.xml`, чтобы механизм CDI смог найти JAR-файлы с компонентами. Кроме того, чтобы получить возможность использовать интерцепторы CDI, в файле `beans.xml` должны быть перечислены все активируемые интерцепторы. То есть, на заключительном шаге следует добавить `AuditInterceptor` в `bean.xml`:

```
<beans
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <interceptors>
      <class>com.bazaar.AuditInterceptor</class>
   </interceptors>
</beans>
```

Перечисление интерцепторов в `beans.xml` может показаться излишним, но это только кажется, потому что пример создания интерцептора `AuditInterceptor` прост и содержит только одну привязку. Наиболее сильной стороной интерцепторов CDI является возможность создавать новые привязки на основе существующих и применять более одной привязки к интерцепторам и компонентам. Добавьте в кучу интерцепторы EJB, и необходимость перечисления интерцепторов в `beans.xml` станет очевидной. Далее мы рассмотрим возможность создания множественных привязок интерцепторов.

Множественные привязки

При создании новой привязки интерцептора можно использовать уже имеющиеся, чтобы расширить возможности этой новой привязки. Представьте, что в приложении ActionBazaar имеется некоторый прикладной компонент EJB, требующий поддержки безопасности. Он должен быть доступен, только при соблюдении определенных условий. Чтобы реализовать эту задачу, было решено определить привязку интерцептора `@Secured`. Но, при обсуждении этой новой привязки кто-то заметил, что в довесок к поддержке безопасности обязательно следует также подключить логику, реализующую аудит. То есть, аудит может проводиться или не проводиться для прикладной логики, не имеющей требований к безопасности, но он всегда должен проводиться для прикладной логики, где безопасность является необходимым условием. Это прекрасный случай для использования привязки интерцептора. Давайте посмотрим, как могло бы выглядеть определение привязки `@Secured`:

```
@InterceptorBinding                    // ❶ Привязка интерцептора
@Target({TYPE, METHOD})
@Retention(RUNTIME)
@Audited                               // ❷ С дополнительной привязкой
public @interface Secured {}
```

Создание привязки интерцептора `@Secured` осуществляется с помощью `@InterceptorBinding` ❶. Затем, включением аннотации `@Audited` ❷ в определение `@Secured` добавляется еще одна привязка интерцептора. Это означает, что все интерцепторы, связанные с привязками `@Secured` и `@Audited` будут автоматически применяться к классам или методам, отмеченным привязкой `@Secured`. Например, взгляните на следующее определение `SecurityCheckInterceptor`:

```
@Secured @Interceptor
public class SecurityCheckInterceptor {
    @AroundInvoke
    public Object checkSecurity(InvocationContext context)
            throws Exception {
        // Здесь проверяется соответствие условиям безопасности
        // и если все в порядке, вызывается метод proceed
        return context.proceed();
    }
}
```

Теперь применим эту привязку интерцептора к методу `removeBid()` компонента `BidService` – к методу, который никогда не должен вызываться без соблюдения мер безопасности, а его вызовы обязательно должны регистрироваться на случай проверки в будущем:

```
@Stateless
public class BidService {
    ...
    @Secured
    public void removeBid(Bid bid) {
        bidDao.removeBid(bid);
    }
}
```

При обращении к методу `removeBid()` привязка интерцептора `@Secured` перехватит его. Привязка `@Secured` связана с `SecurityCheckInterceptor`, а привязка `@Audited` связана с `AuditInterceptor`. Так как в определении `@Secured` использовалась привязка `@Audited`, перед вызовом `removeBid()` будут вызваны оба интерцептора, `SecurityCheckInterceptor` и `AuditInterceptor`. Соответственно будут проверено соблюдение требований безопасности и проведен аудит прикладной логики.

Поскольку у нас имеется две привязки, связанные с двумя интерцепторами, возникает закономерный вопрос: какой из интерцепторов будет вызван первым? При использовании интерцепторов EJB порядок их вызова очевиден (см. раздел 5.3.5). А что можно сказать о порядке вызова интерцепторов CDI?

Ответ на этот вопрос кроется в файле `beans.xml`. Перечисляя интерцепторы, мы не только сообщаем механизму CDI, какие интерцепторы должны быть

активизированы, но и определяем порядок, в каком они должны вызываться. Для метода `removeBid()`, вероятно, лучше сначала было бы провести аудит вызовов этого метода, а затем проверить соблюдение требований безопасности. Указав `AuditInterceptor` первым, мы гарантируем, что он будет вызван перед `SecurityCheckInterceptor`:

```
<beans
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <interceptors>
      <class>com.bazaar.AuditInterceptor</class>
      <class>com.bazaar.SecurityCheckInterceptor</class>
   </interceptors>
</beans>
```

Если интерцепторы CDI смешиваются с интерцепторами EJB, сначала будут вызваны интерцепторы EJB, а затем интерцепторы CDI. То есть, порядок выполнения интерцепторов будет следующим:

1. Интерцепторы, перечисленные в аннотации `@Interceptors`.
2. Интерцепторы, перечисленные в `ejb-jar.xml`.
3. Интерцепторы CDI, перечисленные в `beans.xml`.

Вдобавок к возможности создания новых привязок интерцепторов с вложенными привязками, сами компоненты могут включать несколько привязок интерцепторов. Например, включить аудит, профилирование и сбор статистической информации для метода `addBid()` можно добавлением следующих аннотаций в класс `BidService`:

```
@Stateless
@Audited
public class BidService {
    @Profiled @Statistics
    public void addBid(Bid bid) {
        bidDao.addBid(bid);
    }
    ...
}
```

Так как аннотация `@Audited` применяется ко всему классу `BidService`, аудиту будет подвергнут каждый его метод. Кроме того, для вызовов метода `addBid()` будет собираться статистическая информация аннотацией `@Statistics` и выполняться профилирование аннотацией `@Profiled`. Порядок выполнения интерцепторов, как и прежде, определяется в файле `beans.xml`.

Связывание множества интерцепторов с компонентом выполняется достаточно просто. Каждая привязка связана с одним или несколькими интерцепторами, поэтому перед вызовом метода компонента произойдет целая серия вызовов интерцепторов. Сами интерцепторы, в свою очередь, могут быть связаны с несколькими

привязками, что фактически может повлиять на порядок их выполнения. Проще всего это объяснить на примере. Представьте, что прикладная логика ActionBazaar, кроме поддержки безопасности, должна быть подвергнута аудиту. Функции аудита, предоставляемой аннотацией `@Audited`, достаточно во многих случаях, но сейчас нам требуется нечто большее. Чтобы восполнить нехватку, создадим новый интерцептор:

```
@Secured @Audited @Interceptor
public class SecuredAuditedInterceptor {
    @AroundInvoke
    public Object extraAudit(InvocationContext context)
            throws Exception {
        ...
        return context.proceed();
    }
}
```

Этот интерцептор связан сразу с несколькими привязками – `@Secured` и `@Audited`. Это означает, что интерцептор будет вызываться, только при одновременном применении к методу привязок `@Secured` и `@Audited`. Если вернуться назад к методу `removeBid()`, можно увидеть, что обращение к нему не будет вызывать выполнение интерцептора `SecuredAuditedInterceptor`, потому что этот метод отмечен только одной аннотацией `@Secured`. Добавим к нему аннотацию `@Audited`:

```
@Stateless
public class BidService {
    ...
    @Secured @Audited
    public void removeBid(Bid bid) {
        bidDao.removeBid(bid);
    }
}
```

Теперь при обращении к `removeBid()` будет вызываться интерцептор `SecuredAuditedInterceptor`. Давайте переместим аннотацию `@Audited` на уровень класса:

```
@Stateless
@Audited
public class BidService {
    ...
    @Secured
    public void removeBid(Bid bid) {
        bidDao.removeBid(bid);
    }
}
```

В этом случае так же будет вызываться интерцептор `SecuredAuditedInterceptor`, потому что аннотация `@Audited` автоматически применяется ко всем методам в `BidService`, включая и `removeBid()`. А так как `removeBid()` дополнительно отме-

чен аннотацией `@Secured`, при обращении к нему будет вызываться интерцептор `SecuredAuditedInterceptor`.

5.4. В заключение

В этой главе мы познакомились с контекстом EJB, узнали, как использовать аннотации `@EJB` и `@Resource` для внедрения зависимостей, и когда следует использовать непосредственный поиск ресурсов и компонентов EJB в реестре JNDI. Мы познакомились с аспектно-ориентированным программированием и показали, как пользоваться интерцепторами EJB для реализации сквозной функциональности. В заключение мы сравнили интерцепторы EJB с более мощными интерцепторами CDI.

ГЛАВА 6.
Транзакции и безопасность

Эта глава охватывает следующие темы:

- основы использования транзакций в компонентах EJB;
- когда следует использовать транзакции;
- управление транзакциями на уровне контейнера и компонентов;
- основы аутентификации и авторизации в компонентах EJB;
- использование групп и ролей для поддержки безопасности.

Транзакции и безопасность – краеугольные камни, на которых строятся корпоративные приложения. С точки зрения разработки, механизмы поддержки транзакций и безопасности являются, пожалуй, наиболее сложными в реализации и практически не поддаются модификации после встраивания их в приложение. Оба механизма считаются системной сквозной функциональностью, пронизывающей прикладную логику и используемой ею. Спецификация EJB определяет особенности обоих механизмов и тем самым создает основу для создания надежных приложений, позволяя при этом программистам сосредоточиться на прикладных аспектах.

Для знакомых с основами JDBC заметим, что EJB создает дополнительный уровень поверх JDBC. Этот уровень вводит абстракции, которые иначе вам пришлось бы реализовать самим. JDBC – это механизм абстракций для организации взаимодействий с базами данных обобщенным способом с применением SQL; это не фреймворк. Создание масштабируемых приложений, использующих базы данных, требует гораздо большего, чем простая установка параметра автоматического подтверждения транзакций в значение `false`. Создание фреймворка управления транзакциями – далеко не тривиальная задача, и при его создании легко можно допустить ошибки в самых разных местах. В этой главе вы узнаете, как пользоваться транзакциями в EJB а также познакомитесь с приемами поддержки безопасности в своих приложениях.

Эта глава делится на две части. В первой из них рассказывается о транзакциях, а во второй – о безопасности. Знакомство с транзакциями мы начнем с изучения основ, а затем исследуем два подхода к управлению транзакциями. В число тем, которые мы рассмотрим в этой главе, входят: использование транзакций при работе с базами данных, двухфазное подтверждение транзакций при одновременной работе с несколькими базами данных, а также декларативное и программное управление безопасностью.

6.1. Знакомство с транзакциями

Транзакция – это группа заданий, которые должны быть выполнены атомарно. Если какое-либо из заданий в группе не удалось выполнить, все изменения, произведенные при выполнении любых других заданий, автоматически отменяются. То есть, в случае неудачи система возвращается в исходное состояние. Под заданием в контексте EJB понимается инструкция SQL или попытка обработать сообщение JMS. Чтобы было понятнее, приведем простой пример: когда вы переводите деньги с текущего счета на сберегательный, вам естественно хотелось бы, чтобы обе операции – списания и зачисления – были выполнены успешно. Если зачисление на сберегательный счет по какой-то причине потерпит неудачу (например, из-за отключения питания, ошибки жесткого диска или сети), списание средств с текущего счета должно быть отменено. Транзакции являются неотъемлемой частью нашей жизни, независимо от того, знаем мы об этом или нет. Никому не хотелось бы приехать в аэропорт, заплатив деньги и имея посадочный талон на руках, и обнаружить, что авиакомпания не забронировала для вас место в самолете.

Поддержка транзакция является составной частью архитектуры EJB. Как будет показано далее, при работе с компонентами EJB на выбор предоставляется две модели транзакций: программная и декларативная. Под программными понимаются *транзакции, управляемые на уровне компонентов* (Bean-Managed Transactions, BMT). Используя эту модель, разработчик должен явно запускать, подтверждать и откатывать транзакции. Декларативные транзакции – это транзакции, управляемые на уровне контейнера (Container-Managed Transactions, CMT). Управление подтверждением и откатом таких транзакций осуществляется через настройки, указанные разработчиком. Внутри одного приложения можно использовать сразу обе модели, однако в некоторых ситуациях, о которых мы расскажем в этой глав, они могут быть несовместимы. Если явно не указано иное, по умолчанию для компонентов используется модель CMT.

Все компоненты – сеансовые с сохранением и без сохранения состояния, одиночки и управляемые сообщениями – поддерживают обе модели BMT и CMT. Обратите внимание, что поддержка модели BMT распространяется также на сообщения JMS. Это стало возможным, благодаря реализации коннекторов Java Connector Architecture (JCA), позволяющей взаимодействовать с корпоративными информационными системами, унаследованными системами, и поддерживающей транзакции. Системы, такие как базы данных, очереди сообщений JMS и внешние системы, доступные через JCA, все они интерпретируются как ресурсы,

доступ к которым регулируется *диспетчером ресурсов*. Чтобы разобраться во всем этом, давайте сначала познакомимся с основными свойствами транзакций, а затем пройдем путь от Java SE до Java EE.

6.1.1. Основы транзакций

Транзакции – исключительно сложная тема и является предметом непрекращающихся исследований. Этой теме посвящены целые книги, и не без оснований. В этом разделе мы дадим краткий обзор основных понятий. Данного обзора вполне достаточно, чтобы предупредить вас об имеющихся опасностях, если вы только приступаете к знакомству с транзакциями, и заложить фундамент, опираясь на который вы сможете углубиться в изучение этой темы. Если вы уже знакомы с транзакциями, можете пропустить этот раздел или бегло пролистать его, чтобы освежить свои знания.

Выше мы определили транзакцию, как группу заданий, выполняемых атомарно, то есть как единая операция. Например, обновляя пять записей в базе данных – в одной таблице или в нескольких – может быть желательно, чтобы все операции либо преуспели, либо потерпели неудачу. Транзакции могут быть распределенными; при взаимодействии с двумя разными системами так же может быть желательно иметь похожую семантику. Например, если операция с кредитной картой в приложении ActionBazaar потерпит неудачу, необходимо также откатить запрос к системе управления товарами. А какими еще свойствами, кроме атомарности, обладают транзакции? Много-много лет тому назад Джим Грей (Jim Gray) придумал аббревиатуру ACID: Atomicity (атомарность), Consistency (согласованность), Isolation (изолированность) и Durability (надежность).

Атомарность

Как уже говорилось, транзакции имеют атомарную природу – они либо подтверждаются, либо откатываются. С точки зрения программирования – если заключить некоторый фрагмент кода в рамки транзакции, неожиданная ошибка в этом фрагменте приведет к отмене всех изменений, выполненных этим фрагментом. То есть система вернется в исходное состояние. Если код выполнится без ошибок, изменения будут сохранены.

Согласованность

Самое непростое из четырех свойств, потому что затрагивает не только программный код. Под согласованностью понимается допустимость состояния системы – и до, и после выполнения транзакции система должна находиться в согласованном состоянии, в соответствии с бизнес-правилами приложения. Разработчик должен гарантировать согласованность системы, используя для этого транзакции. Системы нижнего уровня, такие как базы данных, не имеют достаточно информации, чтобы знать, что соответствует понятию допустимого состояния. Успех или неудача транзакции не имеет значения – система всегда будет оставаться в допустимом состоянии.

В ходе выполнения транзакции система может находиться в недопустимом состоянии. В этом смысле транзакцию можно рассматривать как своеобразную песочницу – в ее границах вы временно защищены от действия правил. Но после подтверждения операций, выполненных в песочнице, система должна оказаться в состоянии, допустимом с точки зрения бизнес-правил. В контексте приложения ActionBazaar считается вполне допустимым списать деньги со счета клиента до снятия купленного им лота с торгов, пока транзакция не завершилась. В этом нет никакой опасности, потому что результаты работы кода не окажут воздействия на систему, пока транзакция не завершится успехом. Настройка ограничений в базе данных помогает обеспечить соответствие этой базы данных бизнес-правилам, но в ограничениях нельзя выразить все семантические правила.

Изоляция

Если в некоторый конкретный момент времени выполняется только одна транзакция, проблема изоляции просто не возникает. Однако в любой данной системе одновременно может выполняться множество транзакций, в рамках которых производятся операции с одними и теми же данными. Некоторые из этих операций могут изменять данные, тогда как другие могут просто извлекать их для отображения или анализа. Существует множество оттенков серого – например, транзакция может знать об изменениях, выполняемых в рамках другой транзакции, или действовать в полной изоляции. Степень изолированности транзакции влияет на производительность системы. Чем выше уровень изолированности, тем хуже производительность.

Во вселенной Java существует четыре уровня изолированности, перечисленные ниже.

- *Read uncommitted* (чтение неподтвержденных данных) – транзакция может читать неподтвержденные изменения, произведенные в других транзакциях. Это подобно извлечению программного кода, над которым работает программист в течение дня – неизвестно заранее скомпилируется ли этот код, и будет ли он работать.
- *Read committed* (чтение подтвержденных данных) – транзакция может читать только подтвержденные изменения, любые другие изменения, которые пока не завершены, будут недоступны. Здесь важно отметить, что если в рамках транзакции попытаться повторно прочитать одни и те же данные, нет никакой гарантии, что данные не изменятся.
- *Repeatable read* (повторяемость чтения) – в ходе транзакции чтение может выполняться многократно и каждый раз будет возвращаться один и тот же результат. Транзакция не увидит изменений, выполненных другими транзакциями, даже если они будут подтверждены до окончания данной транзакции. Фактически, транзакция с этим уровнем изоляции работает с собственной копией данных.
- *Serializable* (последовательный доступ) – только одна транзакция может работать с данными в каждый конкретный момент времени. Этот уровень

изоляции особенно ухудшает производительность, потому что исключает возможность параллельного доступа к данным.

С увеличением степени изоляции производительность падает. Это означает, что самая худшая производительность наблюдается при использовании уровня Serializable – в каждый конкретный момент времени может выполняться только одна транзакция. Выбор правильного уровня изоляции является весьма ответственным решением. В учет следует принимать не только производительность, но и правильность данных. Например, в приложении ActionBazaar не имеет смысла применять уровень Serializable для извлечения списка товаров с целью отображения на главной странице. К тому времени, как пользователь прочитает страницу, данные уже устареют. Использование самого строгого уровня ограничит масштабируемость приложения. Следует так же отметить, что поддержка уровней изоляции может отличаться дла разных баз данных и их версий.

Надежность

Под надежностью понимается сохранность изменений произведенных в рамках транзакции. Как только владелец транзакции получит подтверждение, что она успешно завершена, он может быть уверен, что любые изменения сохранятся даже в случае сбоя системы. Обычно это достигается за счет ведения журнала транзакциq – каждое изменение записывается в журнал и может быть «воспроизведено» для воссоздания состояния, предшествовавшего сбою. Например, представьте, что клиент был извещен о завершении транзакции, теперь данные в безопасности, даже если пропадет питание на сервере баз данных. В следующих двух разделах мы рассмотрим транзакции и их характеристики применительно к Java SE и EJB.

6.1.2. Транзакции в Java

Теперь вы имеете основные представления о транзакциях и их важности. Транзакции необходимы; без них переходные состояния были бы доступны другим пользователям и вызывали хаос в данных. В отсутствие поддержки транзакция вам пришлось бы вручную реализовать собственные механизмы блокировок, синхронизации и отката изменений. Реализация таких механизмов невероятно сложна, поэтому разработка системы транзакций может потребовать по-настоящему титанических усилий. К счастью практически все современные базы данных поддерживают ACID-совместимые транзакции. Давайте теперь перейдем от теоретических рассуждений о транзакциях к практике и посмотрим, как используются транзакции для доступа к базам данных из программ на языке Java.

Выполнение инструкций SQL и вызов хранимых процедур в базе данных производится с помощью JDBC. JDBC – это уровень абстракции доступа к базам данных, избавляющий от необходимости использовать разнородные API для разных баз данных. То есть, вы можете использовать один и тот же программный код для выполнения инструкций SQL в базе данных Oracle или DB2, хотя сам синтаксис SQL может отличаться. При использовании JDBC обычно сначала с помощью класса `java.sql.DriverManager` создается объект `java.sql.Connection`. Объ-

ект `Connection` представляет физическое соединение с базой данных. Он имеет множество методов, позволяющих обращаться к базе данных. В следующем фрагменте приводится пример выполнения последовательности инструкций SQL в базе данных ActionBazaar:

```
Class.forName("org.postgresql.Driver");
Connection con =
    DriverManager.getConnection("jdbc:postgresql:actionbazaar","user","pw");
Statement st = con.createStatement();
st.executeUpdate("insert into item ( item_id, bidstartdate, createddate ,
    initialprice , itemname ) values ( 0 , current_date, current_date,
    100.50 , 'Apple IIGS')");
st.executeUpdate("insert into item ( item_id, bidstartdate, createddate ,
    initialprice , itemname ) values ( 1 , current_date, current_date,
    100.50 , 'Apple IIE')");
st.executeUpdate("insert into item ( item_id, bidstartdate, createddate ,
    initialprice , itemname ) values ( 1 , current_date, current_date,
    100.50 , 'Apple IIC')");
```

Этот фрагмент имеет один существенный недостаток: результаты выполнения каждой инструкции SQL немедленно подтверждаются в базе данных. Поле `item_id` – это первичный ключ, то есть его значение должно быть уникальным. При попытке выполнить третью инструкцию произойдет ошибка, потому что запись с тем же значением `item_id` уже была добавлена и подтверждена. Чтобы обеспечить выполнение всех этих инструкций в рамках одной транзакции, необходимо отключить автоматическое подтверждение и явно подтверждать изменения или выполнять откат в случае ошибки. Ниже приводится улучшенная версия примера:

```
Class.forName("org.postgresql.Driver");
Connection con =
    DriverManager.getConnection("jdbc:postgresql:actionbazaar","user","pw");
con.setAutoCommit(false);
try {
  Statement st = con.createStatement();
  st.executeUpdate("insert into item ( item_id, bidstartdate, createddate ,
     initialprice , itemname ) values ( 0 , current_date, current_date,
     100.50 , 'Apple IIGS')");
  st.executeUpdate("insert into item ( item_id, bidstartdate, createddate ,
     initialprice , itemname ) values ( 1 , current_date, current_date,
     100.50 , 'Apple IIE')");
  st.executeUpdate("insert into item ( item_id, bidstartdate, createddate ,
     initialprice , itemname ) values ( 1 , current_date, current_date,
     100.50 , 'Apple IIC')");
  con.commit();
} catch (Throwable t) {
  con.rollback();
  t.printStackTrace();
}
```

На каком-то этапе своей карьеры практически каждый разработчик на Java писал нечто подобное. В этом коде осуществляется прямое управление соединением с базой данных. В нем очень легко ошибиться и неправильно обработать ошиб-

ку или не выполнить подтверждение. Если объект соединения передается между несколькими прикладными объектами для выполнения операций в рамках одной транзакции, ситуация становится практически неуправляемой. Помимо архитектурных недостатков, непосредственное управление соединением препятствует координации операций с множеством ресурсов, таких как ресурсы JMS.

6.1.3. Транзакции в EJB

Технология Java EE, компонентом которой является EJB, создавалась с целью упростить нам жизнь. Фрагменты кода в предыдущем разделе выглядят простыми, но эта простота обманчива. Столкнувшись с нагромождением понятий Java EE, кто-то может с тоской вспомнить старые добрые времена, когда все было намного проще. К настоящему времени в этой книге вы уже познакомились с JDBC и JPA. Попробовавшие загрузить исходные тексты примеров для книги и запустить их наверняка столкнулись с необходимостью настройки интерфейса `DataSource` на своем сервере приложений. Возможно вас интересует, как все эти аббревиатуры и технологии связаны между собой.

Давайте для начала познакомимся с определениями некоторых распространенных аббревиатур, которые будут встречаться вам далее, и узнаем, как они связаны с хранением данных и транзакциями.

- *JPA* (Java Persistence API) – это прикладной программный интерфейс, определяющий порядок отображения объектов Java в реляционные структуры, а также средства извлечения, сохранения и удаления этих объектов. Представьте простой Java-объект, описывающий человека. Интерфейс JPA предоставляет аннотации, с помощью которых можно описать правила отображения класса `Person` в сущность внутри базы данных. JPA также предоставляет классы и методы для доступа к экземплярам, хранимым в базе данных.
- *JDBC* (Java Database Connectivity) – это прикладной программный интерфейс, определяющий порядок доступа к базе данных. Он помогает писать универсальный программный код на Java, не зависящий от характерных особенностей баз данных. Это относительно низкоуровневый интерфейс – вы вынуждены будете использовать язык SQL, синтаксис которого для разных баз данных может отличаться. Извлекая данные из базы, вам придется преобразовывать их в в объекты, которые потом можно будет использовать в приложении.
- *JTA* (Java Transaction API) – это прикладной программный интерфейс управления транзакциями. Он опирается на модель *распределенной обработки транзакций* (Distributed Transaction Processing, DTP), созданной в Open Group. Если вам потребуется организовать программное управление транзакциями или реализовать сохранение данных на уровне компонентов, вам придется использовать интерфейсы JTA.
- *JTS* (Java Transaction Service) – это спецификация, определяющая особенности создания служб управления транзакциями. На основе JTS может

быть реализован механизм JTA. Как разработчику EJB, вам практически не придется касаться JTS. JTS определена в CORBA и является частью Object Services.

- `DataSource` – интерфейс, выступающий в качестве замены для `java.sql.DriverManager` (см. примеры кода в предыдущем разделе). Используется для получения соединений с базами данных и может быть зарегистрирован в JNDI.
- *XA* – адаптация спецификации Open Group DTP для Java. В своей практике вы будете встречаться с такими классами, как `javax.transaction.xa.XAResource`, `javax.transaction.xa.XAConnection` и другими. Помните, что спецификация DTP определяется в рамках JTA. Ресурсы и соединения механизма XA используются для поддержки транзакций, действие которых распространяется на несколько баз данных или ресурсов, таких как ресурсы JMS.

В EJB управление транзакциями осуществляется или программно, или с помощью аннотаций. Об этом подробнее будет рассказываться в следующих разделах. В случае программного управления транзакциями – то есть, когда сохранение данных реализовано на уровне компонентов – используются прикладные интерфейсы JTA. Соединение с ресурсами, такими как базы данных, внутри контейнера EJB осуществляется посредством объектов `DataSource`, и очень часто эти объекты поддерживают протокол XA. Протокол XA позволяет распространять действие транзакций сразу на несколько ресурсов, таких как базы данных и контейнеры. Объекты `DataSource` можно извлекать из JNDI программно или с помощью аннотации `@Inject`. При желании обмен данными между Java-объектами и базами данных можно реализовать с применением JPA – за кулисами механизм JPA будет использовать объект `DataSource` и транзакции EJB. Диспетчер транзакций реализует интерфейс JTS и предоставляет доступ к реализации JTA.

Настройка уровней изоляции не определяется стандартом Java EE и поэтому в каждом контейнере она может быть реализована по-разному. Чаще всего уровни изоляции определяются при настройке `DataSource`. Как результат, установленный уровень изоляции оказывается глобальным и применяется ко всем соединениям и транзакциям. Программным путем можно изменить уровень изоляции для соединения, полученного из `DataSource`, но это небезопасно. Во-первых, если в конечном итоге это соединение вернется в пул с измененными настройками, они будут воздействовать на другие транзакции. Во-вторых, изменение настроек, установленных контейнером, приведет к путанице и сбивающим с толку эффектам в работе приложения.

Но, если ограничиться настройками уровня изоляции транзакций в `DataSource`, какой уровень лучше использовать? Уровень, допускающий чтение неподтвержденных данных, небезопасен, потому что приложение будет видеть неподтвержденные изменения, выполненные в других транзакциях, которые могут быть недопустимыми. Это может привести к принятию неверных решений, неожиданным откатам и нарушению целостности базы данных. Уровень, обеспечивающий пов-

торяемость чтения, страдает эффектом чтения фантомных данных – например в ситуациях, когда повторный запрос мог бы вернуть другие данные из-за того, что в момент между первым и вторым запросами какая-то другая транзакция подтвердила изменения. Уровень изоляции, требующий последовательного доступа, ухудшает масштабируемость, потому что блокирует параллельный доступ к данным нескольких транзакций. Для ситуаций, когда уровень, ограничивающийся чтением только подтвержденных данных, неприемлем, можно предложить несколько вариантов. Один из них – создать отдельный источник данных (`DataSource`) для каждого уровня изоляции. Другой вариант – использовать столбцы с версиями и оптимистичную блокировку.

6.1.4. Когда следует использовать транзакции

Транзакции – составная часть EJB. По умолчанию компоненты EJB используют транзакции, управляемые на уровне контейнера, при этом для компонентов могут использоваться уже действующие транзакции или запускаться новые. Однако такое их поведение можно изменить дополнительными настройками. Как вы уже знаете, транзакции играют чрезвычайно важную роль в обеспечении целостности данных. И более уместным был бы вопрос «Когда не следует использовать транзакции?». Транзакции бессмысленно использовать при обращении к диспетчеру ресурсов, не поддерживающему транзакции. Такие диспетчеры ресурсов могут даже возбуждать исключения при попытке запустить транзакцию. Однако такие ситуации крайне редки.

Отключение транзакций при работе с диспетчером ресурсов, поддерживающим их, может привести к неожиданным последствиям. В случае с базой данных подключение может перейти в режим автоматического подтверждения или возбудить исключение. Даже простое извлечение данных из базы выполняется под управлением транзакций. Семантика происходящего в подобных ситуациях не имеет четкого определения и может отличаться в разных реализациях. Не отключайте транзакции, пытаясь увеличить производительность – они редко оказываются узким местом.

При использовании диспетчера ресурсов, который действительно не поддерживает транзакции, можно отметить класс компонента аннотацией, как показано ниже. Подробнее эти аннотации обсуждаются далее в этой главе. Если к моменту вызова метода этого компонента транзакция окажется запущенной, она будет приостановлена:

```
@TransactionManagement(TransactionManagementType.CONTAINER)
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class BidServiceBean implements BidService {
  ...
}
```

Итак, компоненты естественным образом работают с транзакциями. Отключайте их, только при работе с диспетчерами ресурсов не поддерживающими их. А теперь давайте посмотрим, как реализованы транзакции.

6.1.5. Как реализованы транзакции EJB

Вашей первичной задачей при разработке корпоративных приложений является правильное использование транзакций. Основу поддержки транзакций составляет механизм взаимодействий контейнера с диспетчерами ресурсов. В конечном счете все, что вы делаете в коде, транслируется в низкоуровневые операции с базой данных, такие как блокировка и разблокировка строк или таблиц в базе данных, ведение журнала транзакций, подтверждение транзакций применением записей в журнале или их откат путем отмены этих записей. Компонент, реализующий функции управления транзакциями при работе с определенным ресурсом, называется диспетчером ресурсов. Не забывайте, что ресурсом может быть не только система управления базами данных, такая как Oracle или PostgreSQL. Ресурсом может быть также сервер сообщений, такой как IBM MQSeries или корпоративная информационная система (Enterprise Information System, EIS), такая как PeopleSoft CRM.

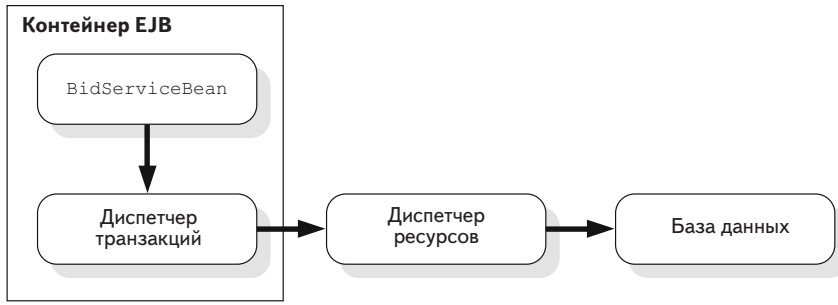

Рис. 6.1. Реализация транзакции – компоненты в базе данных

На рис. 6.1 представлена схема взаимоотношений прикладных компонентов с базами данных. Точно названный диспетчер транзакций осуществляет управление транзакциями. Он взаимодействует с одним или более диспетчерами ресурсов, которые выполняют операции с базами данных. В случае с базой данных Oracle, роль диспетчера ресурса играет драйвер JDBC, обеспечивающий связь с Oracle. Диспетчер транзакций может быть частью контейнера или отдельным процессом. Возможно, вы сталкивались с упоминанием подобной возможности в документации с описанием настройки провайдера JTA для Tomcat при использовании провайдера JPA.

Компоненты EJB взаимодействуют с диспетчером транзакций посредством JTA. Прикладной интерфейс JTA основан на модели распределенной обработки транзакций (DTP), созданной в Open Group. Первая версия JTA была разработана в 1999 году и с тех пор не претерпела существенных изменений. JTA поддерживает только синхронные взаимодействия между приложением и ресурсом (базой данных). Модель DTP подразделяет свой интерфейс на две части: TX, диспетчер транзакций, и XA, интерфейс между диспетчером транзакций и диспетчером ресурсов. Название XA часто служит префиксом в именах классов, таких как `javax`.

`transaction.xa.XAConnection`. Возможно вам доводилось слышать, как кто-то спрашивает – поддерживается ли тот или иной драйвер XA. Когда используется программное управление транзакциями (оно же BMT), запуск, подтверждение и откат транзакций осуществляется с помощью интерфейса `javax.transaction.UserTransaction`. Интерфейс `UserTransaction` является частью раздела TX модели DTP.

На данный момент у кого-то может возникнуть сомнения в необходимости диспетчера транзакций. Некоторые корпоративные приложения взаимодействуют только с одним ресурсом. Транзакции, предназначенные для взаимодействий с единственным ресурсом, называются *локальными транзакциями*. Но многие приложения используют множество ресурсов. Ситуации, когда приложению приходится работать с множеством баз данных и унаследованных систем, встречаются не так уж и редко. Именно в таких ситуациях диспетчер транзакций проявляются преимущества диспетчера транзакций – он координирует работу транзакций, действие которых распространяется на несколько ресурсов. Например, компоненту `OrderProcessorBean` в приложении ActionBazaar приходится обновлять записи в базе данных с информацией о товарах и запоминать сведения о кредитных картах в базе данных платежей, как показано на рис. 6.2.

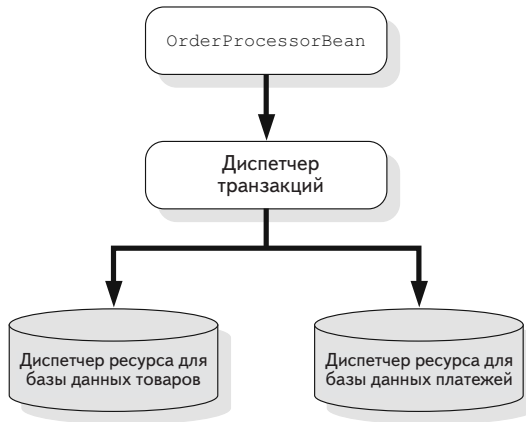

Рис. 6.2. Распределенное управление транзакциями. Прикладная программа делегирует операции с транзакциями диспетчеру транзакций, который координирует взаимодействия с диспетчерами ресурсов

Для приложения диспетчер транзакций – это внешний компонент, предоставляющий услуги управления транзакциями. Компонент `OrderProcessorBean` обращается к диспетчеру с запросами запустить, подтвердить или откатить транзакцию. Диспетчер транзакций координирует эти запросы между двумя диспетчерами ресурсов. Если попытка внести изменения в базу данных платежей потерпит неудачу, диспетчер транзакций обеспечит откат изменений в базе данных товаров. Координация операций с двумя или более ресурсами достигается за счет использования двухфазного подтверждения.

6.1.6. Двухфазное подтверждение

Протокол двухфазного подтверждения применяется в ситуациях, когда действие транзакции распространяется на несколько ресурсов. Как следует из названия, двухфазное подтверждение выполняется в два этапа. На первом этапе диспетчер транзакций опрашивает диспетчеров ресурсов на предмет готовности к подтверждению транзакции. Если все диспетчеры ресурсов отвечают утвердительно, диспетчер транзакций отправляет каждому из них сообщение с подтверждением транзакции, как показано на рис. 6.3. Если какой-то из диспетчеров ресурсов ответит отрицательно, транзакция откатывается. Каждый диспетчер ресурсов отвечает за ведение собственного журнала транзакций, чтобы в случае чего транзакция могла быть восстановлена.

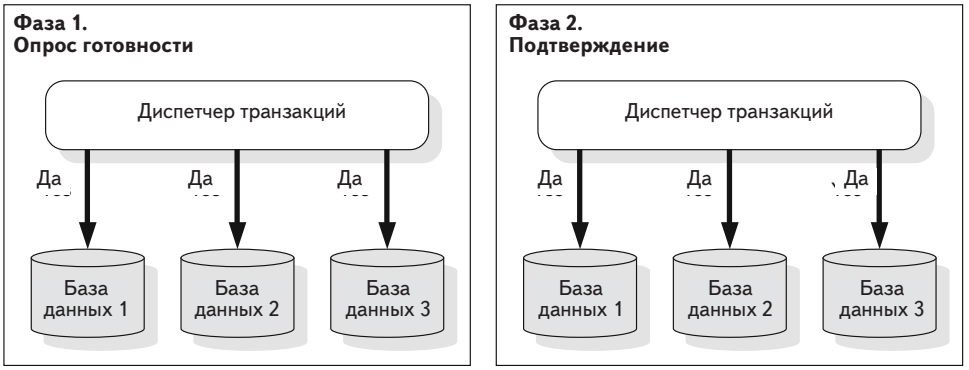

Рис. 6.3. Протокол двухфазного подтверждения. На этапе 1 каждая база данных сообщает диспетчеру транзакций о готовности сохранить изменения. На этапе 2 диспетчер транзакций сообщает каждой базе данных о подтверждении транзакции

Кого-то из вас наверняка обеспокоит вопрос: возможно ли, что на первом этапе диспетчер ресурса ответит утвердительно, а на втором потерпит неудачу. Да такое возможно и случается по разным причинам. Между моментом, когда будет получен утвердительный ответ, и моментом передачи команды на подтверждение, база данных может получить команду о завершении работы или может быть потеряно соединение по сети. Нередко так же возникает ситуация превышения таймаута из-за высокой нагрузки и в этом случае диспетчер транзакций в одностороннем порядке принимает решение откатить транзакцию. В этих случаях возбуждаются эвристические исключения.

Существует три разных эвристических исключения: `HeuristicCommitException`, `HeuristicMixedException` и `HeuristicRollbackException`. Исключение `HeuristicCommitException` возбуждается при отправке запроса на откат, когда было принято эвристическое решение подтвердить транзакцию. Исключение `HeuristicMixedException` возбуждается, когда часть диспетчеров ресурсов подтвердили транзакцию, а часть потерпели неудачу. В этой ситуации система оказы-

вается в недопустимом состоянии, требующем внешнего вмешательства. Исключение `HeuristicRollbackException` возбуждается, когда принято эвристическое решение и все ресурсы выполнили откат. В этом случае клиент должен повторить запрос. Приложение обязательно должно перехватывать и обрабатывать эти исключения.

Важно также помнить, что причиной неудачи может быть любая из сторон. Неудачу может потерпеть контейнер с диспетчером транзакций (например, если получена команда на остановку системы), оставив диспетчеров ресурсов в подвешенном состоянии. В этом случае каждый диспетчер ресурсов должен самостоятельно выполнить откат транзакции – они не должны блокировать доступ к ресурсам. Когда диспетчер транзакций возобновит работу, он попытается завершить транзакцию.

6.1.7. Производительность JTA

Интерфейс JTA проектировался для оптимальной работы независимо от количества используемых ресурсов. Существует множество оптимизаций, реализованных в каждом диспетчере транзакций, которые гарантируют, что JTA не станет узким местом в приложении. К их числу можно отнести однофазное подтверждение, оптимизацию на основе предположения о прерывании транзакции и оптимизацию при доступе только для чтения.

В предыдущем разделе мы рассказали о том, как реализован механизм двухфазного подтверждения транзакций. Двухфазное подтверждение необходимо в ситуациях, когда в транзакции участвуют два или более ресурсов. Но при работе с единственным ресурсом диспетчер транзакций использует однофазное подтверждение – он пропускает этап проверки готовности, который в данном случае оказывается бессмысленным.

Один или более ресурсов могут принять решение прервать транзакцию на этапе опроса. В этом случае бессмысленно опрашивать других диспетчеров ресурсов. Как только будет установлена необходимость отката транзакции, диспетчер транзакций немедленно известит об этом всех еще не опрошенных диспетчеров ресурсов. Не имеет смысла спрашивать у этих диспетчеров ресурсов готовность подтвердить/откатить транзакцию, если их ответ уже не имеет значения.

Ресурсы, доступные только для чтения и в которые не вносилось никаких изменений, не требуется включать в процесс подтверждения. Если диспетчер ресурса сообщит диспетчеру транзакций, что не имеет изменений, требующих подтверждения, диспетчер транзакций будет пропускать такой ресурс на этапе опроса, устраняя тем самым ненужные операции.

В дополнение к этим трем оптимизациям, в ходе выполнения обоих этапов двухфазного подтверждения диспетчер транзакций может обращаться к диспетчерам ресурсов параллельно. То есть, если в транзакции участвует три базы данных и опрос готовности к подтверждению каждой из них выполняется 10 мсек, тогда общая продолжительность этапа опроса займет только 10 мсек, а не 30, как в случае последовательного опроса.

6.2. Транзакции, управляемые контейнером

В модели CMT запуск, подтверждение и откат транзакций осуществляется контейнером. Границы декларативных транзакций всегда совпадают с началом и концом прикладных методов компонентов EJB. Точнее говоря, контейнер сначала запускает транзакцию JTA, затем вызывает метод и, в зависимости от результатов выполнения метода, подтверждает или откатывает транзакцию. Вам остается только с помощью аннотаций или дескриптора развертывания сообщить контейнеру, как он должен управлять транзакциями, и информировать диспетчера транзакций о ситуациях, когда требуется откатить транзакцию. Как упоминалось выше, контейнер предполагает, что модель CMT должна применяться ко всем прикладным методам компонентов.

В этом разделе мы поближе познакомимся с особенностями CMT. Здесь вы научитесь пользоваться аннотациями `@TransactionManagement` и `@TransactionAttribute`, а также узнаете, как откатывать транзакции с помощью `EJBContext` и как обрабатывать исключения.

6.2.1. Досрочное оформление заказов с применением модели CMT

Некоторые лоты в системе ActionBazaar предусматривают возможность «досрочного» приобретения. То есть, покупатель может купить товар за установленную продавцом цену, пока никто другой не успел сделать ставку. Благодаря этому продавцы и покупатели избавляются от необходимости ожидать окончания торгов. Как только пользователь щелкнет на кнопке **Snag-It** (Купить досрочно), приложение ActionBazaar прекратит прием новых ставок для этого лота, проверит кредитную карту покупателя, осуществит перевод денег и снимет лот с торгов. Самое важное: все эти операции выполняются в контексте транзакции. Очевидно, что покупатель был бы огорчен, если бы с него сняли деньги, а товар не доставили. Точно так же и продавец был бы недоволен, если бы он отправил товар, но не получил плату за него.

Для реализации досрочной покупки добавим метод `placeSnagItOrder` в сеансовый компонент `OrderManagerBean`. Код этого метода приводится в листинге 6.1. Метод сначала проверяет, имеются ли другие ставки для этого лота, и в случае их отсутствия проверяет кредитную карту клиента, осуществляет перевод денег и снимает лот с торгов. Чтобы максимально упростить код, мы опустили значительную часть реализации компонента.

Листинг 6.1. Реализация досрочного оформления заказа с использованием модели CMT

```
@Stateless(name = "BidManager")
// ❶ использовать модель CMT
@TransactionManagement(TransactionManagementType.CONTAINER)
public class BidManagerBean implements BidManager {
```

```
  @Inject
  private CreditCardManager creditCardManager;
  @Inject                          // ❷ Внедрить
  private SessionContext context; //    контекст EJB
  @Inject
  private CreditCardManager creditCardManager;
  // ❸ Определяет атрибут транзакции для метода
  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  @Override
  public void placeSnagItOrder(Item item, Bidder bidder, CreditCard card) {
    try {
      if(!hasBids(item)) {
        creditCardManager.validateCard(card);
        creditCardManager.chargeCreditCard(card,item.getInitialPrice());
        closeBid(item,bidder,item.getInitialPrice());
      }
    } catch (CreditProcessingException ce) {
      logger.log(Level.SEVERE,"An error ocurred processing the order.",ce);
      context.setRollbackOnly();  // ❹ Выполнить откат в случае исключения
    } catch (CreditCardSystemException ccse) {
      logger.log(Level.SEVERE,"Unable to validate credit card.",ccse);
      context.setRollbackOnly();  // ❹ Выполнить откат в случае исключения
    }
  }
}
```

Прежде всего сообщим контейнеру, что управление транзакциями для этого компонента возлагается на него ❶. Впрочем, для данного компонента делать это необязательно, так как использование модели CMT предполагается по умолчанию. Далее в компонент внедряется объект контекста EJB ❷. Метод `placeSnagItOrder` объявляется как требующий выполнения в рамках транзакции ❸. Если при попытке выполнить досрочную покупку возникнет исключение, мы, вызовом метода `setRollbackOnly` внедренного объекта `EJBContext`, сообщаем контейнеру, что необходимо откатить транзакцию ❹. Это целиком ваша ответственность произвести откат в случае ошибки. А теперь познакомимся поближе с аннотацией `TransactionManagement`.

6.2.2. Аннотация @TransactionManagement

Аннотация `@TransactionManagement` определяет, какая модель – CMT или BMT – должна использоваться для этого конкретного компонента. В данном случае аннотации передается значение `TransactionManagementType.CONTAINER`, которое означает, что управление транзакциями для компонента осуществляется контейнером. Если бы мы решили управлять транзакциями программно, мы указали бы значение `TransactionManagementType.BEAN`. Обратите внимание: хотя мы и включили данную аннотацию в пример, делать это было совсем необязательно, так как модель CMT используется по умолчанию. Когда мы перейдем к исследованию модели BMT, станет понятнее, почему по умолчанию выбирается модель CMT и обычно является более удачным выбором. Далее рассмотрим вторую аннотацию, имеющую отношение к транзакциям: `@TransactionAttribute`.

6.2.3. Аннотация @TransactionAttribute

Хотя при использовании модели CMT большую часть хлопот берет на себя контейнер, вам все еще необходимо явно сообщить ему как он должен управлять транзакциями. Чтобы понять, что под этим имеется в виду, представьте, что метод `placeSnagItOrder` мог бы вызываться из другого компонента, для которого уже была запущена транзакция. Какой сценарий действий предпочли бы вы: приостановить внешнюю транзакцию, присоединиться к ней или отменить? В нашем случае метод `placeSnagItOrder` вызывается из веб-слоя приложения, где транзакции не используются, поэтому очевидно, что должна быть запущена новая транзакция. Аннотация `@TransactionAttribute` дает возможность указать контейнеру, как он должен действовать в отношении транзакций.

Взглянув на код в листинге 6.1, можно заметить, что он вызывает метод `chargeCreditCard` компонента `CreditCardManager`. Этот метод представлен в листинге 6.2. Здесь можно видеть, что метод отмечен аннотацией, устанавливающей атрибут `MANDATORY` транзакции. Это означает, что вызов метода должен осуществляться только в рамках уже действующей транзакции – для него не может запускаться новая транзакция. Это вполне логично: разве можно снять деньги со счета в отрыве от другой операции, такой как обработка заказа? При вызове этого метода, он присоединится к транзакции, запущенной методом `placeSnagItOrder`. Если в `CreditCardManagerBean` возникнет ошибка, все изменения, произведенные в методе `placeSnagItOrder`, также будут отменены. Совершенно неважно, использует ли `CreditCardManagerBean` отдельную базу данных или выполняет операции через то же соединение JDBC. Отметьте также, что вам не требуется передавать объекты исключений, чтобы получить такое поведение, или писать какой-либо код, проверяющий наличие запущенной транзакции.

Листинг 6.2. Реализация метода, требующего наличия уже запущенной транзакции

```
@Stateless(name="CreditCardManager")
@TransactionManagement(TransactionManagementType.CONTAINER)
public class CreditCardManagerBean implements CreditCardManager {
  @TransactionAttribute(TransactionAttributeType.MANDATORY)
  public void chargeCreditCard(CreditCard creditCard, BigDecimal amount)
      throws CreditProcessingException {
    // снять деньги с кредитной карты...
  }
}
```

Аннотация `@TransactionAttribute` сообщает контейнеру, как он должен управлять транзакциями, в том числе: необходимость запуска новой транзакции, присоединения к существующей транзакции и так далее. Аннотация может применяться к отдельным методам или ко всему компоненту. Если аннотация применяется к компоненту, все прикладные методы этого компонента унаследуют атрибуты транзакции, указанные в аннотации. В листинге 6.1 мы указали в аннотации `@TransactionAttribute`, что для метода `placeSnagItOrder` действует атрибут `TransactionAttribute.REQUIRED`. Вообще эта аннотация может прини-

мать шесть разных значений типа `TransactionAttributeType`, которые перечислены в табл. 6.1.

Таблица 6.1. Влияние атрибутов транзакций на методы компонентов EJB

Атрибут	Транзакция запущена вызывающим кодом?	Влияние
`REQUIRED`	Нет	Контейнер запустит новую транзакцию.
	Да	Метод присоединится к запущенной транзакции.
`REQUIRES_NEW`	Нет	Контейнер запустит новую транзакцию.
	Да	Контейнер запустит новую транзакцию, а существующая транзакция будет приостановлена.
`SUPPORTS`	Нет	Транзакции не используются.
	Да	Метод присоединится к запущенной транзакции.
`MANDATORY`	Нет	Вызовет исключение `javax.ejb.EJBTransactionRequiredException`.
	Да	Метод присоединится к запущенной транзакции.
`NOT_SUPPORTED`	Нет	Транзакции не используются.
	Да	Существующая транзакция будет приостановлена, а метод продолжит работу вне транзакции.
`NEVER`	Нет	Транзакции не используются.
	Да	Вызовет исключение `javax.ejb.EJBException`.

Давайте рассмотрим, когда следует использовать каждый из атрибутов.

REQUIRED

Значение `REQUIRED` используется по умолчанию и имеет наиболее широкую область применения. Он указывает, что метод компонента EJB всегда должен вызываться в контексте транзакции. Если клиент, вызывающий метод, не запустит транзакцию, контейнер запустит ее автоматически перед вызовом метода и подтвердит по завершении метода. С другой стороны, если вызывающий код уже выполняется в контексте транзакции, метод будет выполнен в рамках этой транзакции. Если транзакция была запущена вызывающим кодом, и метод сообщил, что ее следует откатить, контейнер не только откатит транзакцию, но и возбудит исключение `javax.transaction.RollbackException`. Это позволит клиенту узнать, что запущенная им транзакция была отменена другим методом. Наш метод `placeSnagItOrder` вызывается из веб-слоя приложения, не поддерживающего транзакции. Соответственно, значение `REQUIRED` в аннотации `@TransactionAttribute` вынудит контейнер запустить новую транзакцию. Если все остальные методы сеансовых компонентов, вызываемые этим методом, также будут отмечены атрибутом `REQUIRED`, они будут выполняться в контексте уже запущенной транзакции. Такое поведение как нельзя лучше подходит для нашей

задачи, потому что нам необходимо, чтобы все сопутствующие операции выполнялись под «зонтиком» единой транзакции. Вообще говоря, атрибут REQUIRED следует использовать всегда, когда метод изменяет данные, но неизвестно, будет ли он вызываться клиентом, запускающим собственную транзакцию.

REQUIRES_NEW

Значение REQUIRES_NEW указывает, что контейнер всегда должен запускать новую транзакцию перед вызовом метода. Если вызывающий код уже запустил транзакцию, она будет приостановлена, пока метод не вернет управление. Это означает, что успех или неудача при выполнении новой транзакции никак не отразится на существующей.

С точки зрения клиента:

1. Его транзакция будет приостановлена.
2. Произойдет вызов метода.
3. Метод подтвердит или откатит свою транзакцию.
4. Выполнение клиентской транзакции продолжится после возврата из метода.

Значение REQUIRES_NEW имеет ограниченное применение. Его следует использовать, когда операции должны выполняться в контексте транзакции, но при этом откат этой транзакции не должен повлиять на клиента, и наоборот. Журналирование – отличный пример, где с успехом можно использовать этот атрибут: сообщение должно быть записано в любом случае, даже если родительская транзакция будет отменена. С другой стороны, Если попытка записать сообщение в журнал потерпит неудачу, это не должно повлиять на выполнение основной операции.

SUPPORTS

Значение SUPPORTS указывает, что метод имеет двойственное отношение к транзакциям. Если вызывающий код запустил транзакцию, метод присоединится к ней. Если же транзакция не была запущена, метод не будет запускать новую транзакцию. Обычно атрибут SUPPORTS используется для методов, выполняющих только операции чтения, такие как извлечение записей из таблицы в базе данных. В примере с досрочной покупкой этим значением отмечен метод, проверяющий наличие ставок (hasBids), потому что он не изменяет данных.

MANDATORY

Значение MANDATORY указывает, что метод должен вызываться в контексте уже действующей транзакции. Мы коротко коснулись этой темы при обсуждении компонента CreditCardManager. Перед вызовом такого метода контейнер проверит наличие действующей транзакции и при ее отсутствии возбудит исключение EJBTransactionRequiredException. То есть, если попытаться вызвать метод компонента CreditCardManager вне контекста действующей транзакции, будет возбуждено исключение.

NOT_SUPPORTED

Когда метод отмечен атрибутом `NOT_SUPPORTED`, он не может вызываться в контексте транзакции. Если перед вызовом метода вызывающий код запустит транзакцию, контейнер приостановит ее, вызовет метод и после завершения метода возобновит транзакцию. Этот атрибут обычно используется для поддержки провайдера JMS в режиме с автоматическим подтверждением транзакций. В этом случае получение сообщения подтверждается сразу же после успешной доставки, и у компонента MDB нет никакой возможности или явной необходимости выполнить откат факта доставки сообщения.

NEVER

В модели CMT значение `NEVER` означает, что метод никогда не должен вызываться в контексте транзакции. Если такая попытка будет произведена, контейнер возбудит исключение `javax.ejb.EJBException`. Это, пожалуй, самый редко используемый атрибут транзакций. Он может пригодиться, например, когда метод изменяет ресурс, не поддерживающий транзакции (такой как текстовый файл), и необходимо гарантировать, что клиент узнает о такой природе метода. Обратите внимание, что при этом не предполагается непосредственный доступ к файловой системе из компонента EJB.

Атрибуты транзакций и компоненты MDB

Как говорилось в главе 4, компоненты MDB поддерживают только часть из этих шести атрибутов – `REQUIRED` и `NOT_SUPPORTED`. Напомним, что клиенты никогда не вызывают компоненты MDB непосредственно – они вызываются в момент доставки сообщения компоненту из очереди. В этот момент не существует никаких транзакций, которые можно было бы приостановить или к которым можно было присоединиться. Соответственно атрибуты `REQUIRES_NEW`, `SUPPORTS` и `MANDATORY` не имеют смысла. Атрибут `NEVER` нелогичен, потому что нет необходимости в столь сильной защите от контейнера. Таким образом, двух оставшихся атрибутов оказывается вполне достаточно: атрибут `REQUIRED` используется, когда необходимо обеспечить выполнение операций в контексте транзакций, и `SUPPORTS` – когда в этом нет необходимости.

Теперь у вас есть понимание того, как объявлять компоненты, использующие модель управления транзакциями CMT и как разграничивать транзакции или присоединяться к ним. Успешное завершение метода, перед вызовом которого была запущена транзакция, приводит к ее подтверждению. А теперь посмотрим, как выполнять откат транзакций.

6.2.4. Откат транзакций в модели CMT

Иногда в ходе выполнения транзакции может возникнуть ситуация, требующая отмены изменений и возврата в исходное состояние. Откатить транзакцию может потребоваться из-за возникшей ошибки, ввода недействительного номера кредитной карты или в результате принятия некоторого решения. Модель CMT

дает компонентам возможность сообщить контейнеру о необходимости откатить транзакцию. Откат не выполняется немедленно – сначала устанавливается флаг, а в конце действия транзакции контейнер выполняет откат, руководствуясь этим флагом. Давайте вернемся к коду в листинге 6.1:

```
@Resource
private SessionContext context;
public void placeSnagItOrder(Item item, Bidder bidder, CreditCard card) {
  try {
    ...
  } catch (CreditProcessingException ce) {
    logger.log(Level.SEVERE,"An error ocurred processing the order.",ce);
    context.setRollbackOnly();
  }
  ...
}
```

Как видно в этом фрагменте, при обработке ошибки, связанной с номером кредитной карты, вызов метода `setRollbackOnly` объекта `javax.ejb.EJBContext` отмечает транзакцию для отката. Если не сделать этого, тогда все изменения в данных будут подтверждены. Важно помнить, что вся ответственность за вызов этого метода в случае исключений целиком возлагается на вас.

Метод `setRollbackOnly` объекта `EJBContext` может вызываться, только из методов, отмеченных атрибутом транзакции `REQUIRED`, `REQUIRED_NEW` или `MANDATORY`. Если транзакция не была запущена, как в случае с атрибутами `SUPPORTED` и `NEVER`, контейнер возбудит исключение `java.lang.IllegalStateException`.

Как отмечалось выше, когда транзакция отмечается как подлежащая отмене, она не завершается немедленно. То есть метод не завершается сразу же после вызова `setRollbackOnly` – он продолжает выполнение, как если бы ничего не произошло. Объект `EJBContext` предоставляет метод, с помощью которого можно узнать, не отмечена ли транзакция для отмены: `getRollbackOnly()`. Вызвав этот метод, можно определить – есть ли смысл продолжить выполнение продолжительной операции, результаты которой не будут подтверждены. Мы могли бы изменить код в листинге 6.1 так, чтобы он пропускал операцию закрытия лота в случае, если операция `chargeCreditCard` потерпит неудачу:

```
public void placeSnagItOrder(Item item, Bidder bidder, CreditCard card) {
  try {
    if(!hasBids(item)) {
      creditCardManager.chargeCreditCard(card,item.getInitialPrice());
      if(!context.getRollbackOnly()) {
        closeBid(item,bidder,item.getInitialPrice());
      }
    }
```

Примечание. *Методы* `setRollbackOnly` *и* `getRollbackOnly` *могут вызываться только при использовании модели CMT, и когда вызывающий метод отмечен атрибутом:* `REQUIRED`, `REQUIRES_NEW` *или* `MANDATORY`. *В противном случае контейнер будет возбуждать исключение* `IllegalStateException`.

Если обработка исключений, заключающаяся только в вызове `setRollbackOnly`, кажется утомительной, механической работой, вам приятно будет узнать, что в EJB 3 имеется более простое решение, основанное на аннотациях, о чем рассказывается далее.

6.2.5. Транзакции и обработка исключений

Исключения неизбежны и вы должны предусматривать их обработку. Очень часто исключения возникают в самые неожиданные моменты, например, во время демонстрации готовой системы высокому руководству. Так как предполагается, что исключения имеют нерегулярную природу, очень легко ошибиться при реализации их обработки или вообще не заметить их на этапе разработки. В листинге 6.1 предусматривается обработка двух исключений и в обоих случаях она заключается в откате транзакции. Взгляните еще раз на этот код:

```
public void placeSnagItOrder(Item item, Bidder bidder, CreditCard card) {
  try {
    if(!hasBids(item)) {
      creditCardManager.validateCard(card);
      creditCardManager.chargeCreditCard(card,item.getInitialPrice());
      closeBid(item,bidder,item.getInitialPrice());
    }
  } catch (CreditProcessingException ce) {
    logger.log(Level.SEVERE,"An error ocurred processing the order.",ce);
    context.setRollbackOnly();
  } catch (CreditCardSystemException ccse) {
    logger.log(Level.SEVERE,"Unable to validate credit card.",ccse);
    context.setRollbackOnly();
  }
}
```

Как видите, обработчики обоих исключений – `CreditProcessingException` и `CreditCardSystemException` – выполняют откат транзакции. Чтобы избежать необходимости снова и снова писать один и тот же шаблонный код, в EJB 3 была возможность управлять результатом транзакции в случае исключений с помощью аннотации `@javax.ejb.ApplicationException`. Лучше всего познакомиться с ней на простом примере. В листинге 6.3 приводится измененная версия метода `placeSnagItOrder`, отмеченного аннотацией `@ApplicationException`, осуществляющей откат транзакции в случае исключения.

Листинг 6.3. Использование аннотации @ApplicationException для отката транзакций в модели CMT

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void placeSnagItOrder(Item item, Bidder bidder, CreditCard card)
    // ❶ Объявление возбуждаемых исключений в инструкции throws
    throws CreditProcessingException, CreditCardSystemException {
  // ❷ Исключения могут возбуждаться в следующих вызовах методов
  if(!hasBids(item)) {
    creditCardManager.validateCard(card);
```

```
    creditCardManager.chargeCreditCard(card,item.getInitialPrice());
    closeBid(item, bidder, item.getInitialPrice());
  }
}
...
// ❸ Определение ApplicationException
@ApplicationException(rollback=true)
public class CreditProcessingException extends Exception {
...
// ❹ Отмечает RuntimeException как ApplicationException
@ApplicationException(rollback=true)
public class CreditCardSystemException extends RuntimeException {
```

Первое отличие от листинга 6.1, сразу бросающееся в глаза, – отсутствие блока `try/catch` и появление инструкции `throws` в объявлении метода ❶. Вообще считается хорошей практикой предусматривать в клиенте обработку прикладных исключений и генерировать соответствующие сообщения об ошибках. Методы, вызываемые здесь, могут возбуждать два исключения, перечисленные в инструкции `throws` ❷. Но самое важное, на что следует обратить внимание, – это две аннотации `@ApplicationException` перед классами исключений. Аннотация `@ApplicationException` ❸ идентифицирует контролируемые (checked) и неконтролируемые (unchecked) исключения Java как *прикладные исключения*.

> **Примечание.** *Прикладное исключение – это исключение, которое, как ожидается, будет обработано клиентом, использующим компонент EJB. В случае возбуждения такие исключения передаются непосредственно в вызывающий код. По умолчанию все контролируемые исключения, кроме* `java.rmi.RemoteException`*, считаются прикладными исключениями. Все исключения, наследующие класс* `java.rmi.RemoteException` *или* `java.lang.RuntimeException` *(неконтролируемые) считаются системными исключениями. В EJB не предполагается, что клиент будет обрабатывать системные исключения. Поэтому такие исключения не передаются клиенту, а заворачиваются в* `javax.ejb.EJBException`.

В листинге 6.3 аннотация `@ApplicationException` перед объявлением `CreditProcessingException` не изменяет поведение исключения – оно, как и прежде, передается клиенту. Но она изменяет поведение исключения `CreditCardSystemException` ❹, которое в ином случае было бы завернуто в `EJBException`, потому что интерпретируется как системное исключение. Применение аннотации `@ApplicationException` вынуждает интерпретировать это исключение как прикладное.

Возможно вы обратили внимание на атрибут `rollback` аннотации `@ApplicationException`. По умолчанию прикладные исключения не вызывают автоматический откат транзакций в модели CMT, потому что этот атрибут получает значение `false`, если явно не указано иное. Присвоив ему значение `true`, мы сообщаем контейнеру, что он должен откатить транзакцию, прежде чем передать исключение клиенту. Таким образом, в листинге 6.3 указывается, что оба исключения – `CreditProcessingException` и `CreditCardSystemException` – вызовут откат транзакции до того, как они будут переданы программе. Если контейнер по-

лучит необработанное системное исключение, такое как `ArrayIndexOutOfBounds` или `NullPointerException`, он также произведет откат транзакции. Но в этом случае контейнер посчитает, что компонент находится в недопустимом состоянии и уничтожит его.

6.2.6. Синхронизация с сеансом

Интерфейс синхронизации с сеансом позволяет сеансовым компонентам с сохранением состояния получать извещения о прохождении различных границ этапов транзакции. Данный интерфейс определен как `javax.ejb.SessionSynchronization`. Чтобы воспользоваться им, компонент должен реализовать методы интерфейса. Этот интерфейс определяет три метода:

- `void afterBegin()` – вызывается сразу после запуска новой транзакции контейнером, но перед вызовом прикладного метода;
- `void beforeCompletion()` – вызывается перед тем, как прикладной метод вернет управление, но до того, как контейнер завершит транзакцию;
- `void afterCompletion(boolean committed)` – вызывается сразу после завершения транзакции; флаг `boolean committed` позволяет определить, была ли подтверждена транзакция или произошел откат.

Первые два метода вызываются в контексте транзакции, что дает возможность читать или изменять данные из базы. Например, с помощью метода `afterBegin()` можно загружать кэшированные данные в компонент, а с помощью метода `beforeCompletion()` записывать их обратно.

6.2.7. Эффективное использование модели CMT

Модель CMT намного проще в использовании, в сравнении с моделью BMT. В отсутствие аннотаций и специфичных настроек в дескрипторе развертывания, компоненты автоматически будут использовать модель CMT и отмечаться атрибутом транзакций `REQUIRED`. Это наиболее «надежный» вариант. Приложение будет прекрасно работать без каких-то дополнительных усилий со стороны разработчика, если не будет возникать никаких прикладных исключений. Однако имеется несколько приемов, которые помогут эффективнее использовать модель CMT, гарантировать корректную работу приложения и упростить его сопровождение.

Чтобы упростить сопровождение кода, желательно определить логические границы транзакции и создать отдельный метод, осуществляющий операции в контексте этой транзакции. Этот метод должен решать две основные задачи: перехватывать прикладные исключения и устанавливать флаг, управляющий откатом транзакции. Обработка прикладных исключений должна быть сосредоточена в одном месте, а на «размазываться» по множеству компонентов, участвующих в транзакции. Когда прикладное исключение не вызывает откат транзакции, часто бывает сложно определить источник проблем, если обработка исключения разбросана по нескольким компонентам. Что касается откатов: логика обработки отката должна быть сосредоточена в одном месте, предпочтительно в мето-

де, играющем роль ворот в транзакцию. Чем шире разбросана логика обработки отката, тем сложнее читать, тестировать и сопровождать такой код. Кроме того, перед выполнением продолжительных операций обязательно вызывайте метод `getRollbackOnly()`, чтобы убедиться в их целесообразности. Бессмысленно тратить время на длительные вычисления, если результаты в конечном итоге будут просто отброшены.

При выборе атрибутов транзакций для применения к методам и классам отдавайте предпочтение наиболее строгим из них. Это уменьшит риск вызова метода с более строгими требованиями в окружении с более мягкими требованиями. Например, ничего хорошего не получится, если отметить класс атрибутом `SUPPORTS`, а затем добавить в него метод, требующий уровня поддержки транзакций `REQUIRED`, и забыть отметить его соответствующей аннотацией. Теоретически такой метод будет вызываться в режиме автоматического подтверждения транзакции. Лучше использовать транзакцию, когда она фактически не нужна, чем потерпеть фиаско, когда транзакция играет критически важную роль.

Правильная обработка прикладных исключений также имеет большое значение, потому что для таких исключений контейнер не выполняет откат транзакции автоматически. Если код может возбуждать прикладные исключения, требующие отмены транзакции, обязательно предусматривайте их обработку. По возможности старайтесь отмечать прикладные исключения аннотацией `@ApplicationException`. Это уменьшит риск появления ошибок, когда по забывчивости пропущен вызов метода `setRollbackOnly()`.

Наконец, пользуйтесь аннотацией `@TransactionAttribute`. Если метод всегда должен выполняться в контексте транзакции, но никогда не должен запускать ее, отмечайте его атрибутом `MANDATORY`. Если блок кода должен выполняться в контексте транзакции, но никогда не должен вызывать откат транзакции, запущенной вызывающим кодом, используйте атрибут `REQUIRES_NEW`.

Модель CMT с успехом может применяться для решения самого широкого круга задач, но иногда бывает необходимо явно управлять транзакциями. Организовать такое управление можно с помощью модели BMT.

6.3. Транзакции, управляемые компонентами

Сила модели CMT является и ее слабостью. При использовании этой модели границы транзакций принудительно устанавливаются в соответствии с границами прикладных методов, и вы не можете решать, когда запускать транзакции, когда подтверждать их или откатывать. Модель BMT, напротив, позволяет управлять всем этим программно, используя семантику, напоминающую модель транзакций JDBC. Но даже в этом случае контейнер поможет вам в создании фактической транзакции и позаботится о низкоуровневых тонкостях. Чтобы использовать модель BMT, необходимо иметь более полное представление о JTA API и в первую очередь об интерфейсе `javax.transaction.UserTransaction`, упомянутом ра-

нее. Далее мы вновь обратимся к примеру досрочного оформления заказов и реализуем его с применением модели BMT. Попутно мы поближе познакомимся с особенностями этой модели.

6.3.1. Досрочное оформление заказов с применением модели BMT

В листинге 6.4 приводится реализация примера досрочного оформления заказов с применением модели BMT. Основная прикладная логика осталась той же, что и в реализации с применением модели CMT (листинг 6.1). Она проверяет наличие ставок, проверяет кредитную карту, перечисляет деньги и снимает лот с торгов. В общем и целом код практически не изменился.

Листинг 6.4. Реализация досрочного оформления заказа с использованием модели BMT

```
@Stateless(name = "BidManager")
// ❶ использовать модель BMT
@TransactionManagement(TransactionManagementType.BEAN)
public class BidManagerBean implements BidManager {
  @Resource                                  // ❷ Внедрить
  private UserTransaction userTransaction; //   UserTransaction
  public void placeSnagItOrder(Item item, Bidder bidder, CreditCard card) {
    try {
      userTransaction.begin();               // ❸ Запустить транзакцию
      if(!hasBids(item)) {
        creditCardManager.validateCard(card);
        creditCardManager.chargeCreditCard(card,item.getInitialPrice());
        closeBid(item,bidder,item.getInitialPrice());
      }
      userTransaction.commit();              // ❹ Подтвердить транзакцию
    } catch (CreditProcessingException ce) {
      logger.log(Level.SEVERE,                          // ❺
        "An error ocurred processing the order.",ce); //  В случае
      context.setRollbackOnly();                        //  исключения
    } catch (CreditCardSystemException ccse) {          //  откатить
      logger.log(Level.SEVERE,                          //  транзакцию
        "Unable to validate credit card.",ccse);        //
      context.setRollbackOnly();                        //
    } catch (Exception e) {
      logger.log(Level.SEVERE,
        "An error ocurred processing the order.",e);
    }
  }
}
```

Обратите внимание на аннотацию `@TransactionManagement`, которой на этот раз вместо значения `TransactionManageMentType.CONTAINER` передано значение `TransactionManagementType.BEAN`, указывающее, что используется модель BMT ❶. Аннотация `TransactionAttribute` вообще отсутствует здесь, потому что она имеет смысл лишь в модели CMT. Далее внедряется объект `UserTransaction`,

реализующий интерфейс JTA ❷, посредством которого явно осуществляется запуск ❸, подтверждение ❹ и откат ❺ транзакции. Рамки транзакции теперь намного уже границ метода и включают только последовательность вызовов, которая действительно должна выполняться атомарно. В следующих разделах мы подробнее рассмотрим этот код, начав с получения ссылки на экземпляр `javax.transaction.UserTransaction`.

6.3.2. Получение экземпляра UserTransaction

Интерфейс `UserTransaction` включает в себя основные функциональные возможности, предоставляемые диспетчером транзакций Java EE. Механизм JTA содержит ряд других интерфейсов, используемых в разных ситуациях, но мы не будем охватывать их здесь, потому что в подавляющем большинстве случаев вам достаточно будет интерфейса `UserTransaction`. Как можно догадаться, интерфейс `UserTransaction` является слишком запутанным, чтобы создавать непосредственно, поэтому его проще получить с помощью контейнера. В листинге 6.4 используется самый простой способ – внедрение с помощью аннотации `@Resource`. Получить интерфейс можно еще парой способов: выполнить поиск в JNDI или воспользоваться объектом `EJBContext`.

Поиск в JNDI

Сервер приложений связывает `UserTransaction` с именем JNDI `java:comp/UserTransaction`. Поиск объекта в реестре JNDI можно выполнить вручную, как показано ниже:

```
Context context = new InitialContext();
UserTransaction userTransaction = (UserTransaction)context.lookup("java:comp/
   UserTransaction");
userTransaction.begin();
// Выполнить операции в контексте транзакции.
userTransaction.commit();
```

Этот способ обычно используется за пределами компонентов EJB – например, если возникает необходимость использовать транзакцию во вспомогательном или в неуправляемом классе внутри EJB, или в веб-слое, где внедрение зависимостей не поддерживается. Однако в такой ситуации можно использовать еще один подход, которым, впрочем, гораздо удобнее пользоваться в контексте EJB, где имеется доступ к широкому спектру абстракций.

EJBContext

Получить ссылку на `UserTransaction` можно также вызовом метода `getUserTransaction` объекта `EJBContext`. Этот прием может пригодиться при наличии ссылки на `SessionContext` или `MessageDrivenContext` для каких-то других целей, когда внедрение отдельного экземпляра интерфейса для доступа к транзакциям нежелательно. Обратите внимание, что метод `getUserTransaction` можно вызывать только когда используется модель BMT.

Его вызов в окружении, где действует модель CMT, может вызвать исключение `IllegalStateException`. Следующий фрагмент демонстрирует применение метода `getUserTransaction`:

```
@Resource
private SessionContext context;
...
UserTransaction userTransaction = context.getUserTransaction();
userTransaction.begin();
// Выполнить операции в контексте транзакции.
userTransaction.commit();
```

Важно также заметить, что попытка вызвать метод `EJBContext.getRollbackOnly()` или `setRollbackOnly()` в контексте модели BMT вызовет исключение `IllegalStateException`. Эти методы могут вызываться, только когда используется модель CMT. В следующем разделе мы посмотрим, как пользоваться интерфейсом `UserTransaction`.

6.3.3. Использование интерфейса UserTransaction

Вы уже видели наиболее часто используемые методы интерфейса `UserTransaction`: `begin`, `commit` и `rollback`. Однако этот интерфейс имеет еще ряд полезных методов, с которыми стоит познакомиться поближе. Взгляните, как выглядит полное определение интерфейса:

```
public interface UserTransaction {
  public void begin() throws NotSupportedException, SystemException;
  public void commit() throws RollbackException,
    HeuristicMixedException,HeuristicRollbackException, SecurityException,
    IllegalStateException, SystemException;
  public void rollback() throws IllegalStateException,
    SecurityException,SystemException;
  public void setRollbackOnly() throws IllegalStateException,
    SystemException;
  public int getStatus() throws SystemException;
  public void setTransactionTimeout(int seconds) throws SystemException;
}
```

Метод `begin` создает новую, низкоуровневую транзакцию и связывает ее с текущим потоком выполнения. Возможно вам будет интересно узнать, что произойдет, если вызвать метод `begin` дважды, прежде обратиться к методу `rollback` или `commit`. Может показаться, что таким способом можно запустить вложенную транзакцию. Однако это не так. В действительности второй вызов метода `begin` возбудит исключение `NotSupportedException`, потому что Java EE не поддерживает вложенные транзакции. Методы `commit` и `rollback`, в свою очередь, удаляют транзакцию, связанную с текущим потоком выполнения вызовом метода `begin`. При этом метод `commit` посылает диспетчеру транзакций сигнал «успеха», а метод `rollback` отменяет текущую транзакцию. Назначение метода `setRollbackOnly`

на первый взгляд непонятно. В конце концов, зачем предпринимать какие-то дополнительные шаги, чтобы пометить транзакцию, как подлежащую откату, когда ее можно откатить явно?

Чтобы понять это, представьте, что вам потребовалось вызвать метод, действующий в соответствии с моделью CMT, из компонента, выполняемого в контексте модели BMT, и этот метод производит длительные вычисления, предварительно проверяя состояние флага транзакции. Поскольку в такой ситуации действие транзакции BMT распространится на метод CMT, будет намного проще, особенно в больших методах, просто пометить транзакцию, как подлежащую откату вызовом `setRollbackOnly`, чем конструировать нагромождения из блоков `if-else`. Метод `getStatus` является более надежной версией метода `getRollbackOnly` из мира CMT. Вместо логического значения он возвращает целочисленный код, позволяющий более точно определить состояние текущей транзакции. Возможные состояния определяются интерфейсом `javax.transaction.Status` и перечислены в табл. 6.2.

Таблица 6.2. Возможные значения интерфейса `javax.transaction.Status interface`, определяющие коды состояния, возвращаемые методом `UserTransaction.getStatus`

Код состояния	Описание
`STATUS_ACTIVE`	Транзакция находится в активном состоянии.
`STATUS_MARKED_ROLLBACK`	Транзакция отмечена, как предназначенная для отката, возможно вызовом метода `setRollbackOnly`.
`STATUS_PREPARED`	Транзакция находится в состоянии готовности, когда все ресурсы готовы к подтверждению. (См. раздел 6.1.6. «Двухфазное подтверждение».)
`STATUS_COMMITTED`	Транзакция была подтверждена.
`STATUS_ROLLBACK`	Транзакция была отменена.
`STATUS_UNKNOWN`	Состояние транзакции неизвестно.
`STATUS_NO_TRANSACTION`	С текущим потоком не связана никакая транзакция.
`STATUS_PREPARING`	Транзакция готовится к подтверждению и ожидает ответа от подчиненных ресурсов. (См. раздел 6.1.6. «Двухфазное подтверждение».)
`STATUS_COMMITTING`	Транзакция находится в процессе подтверждения.
`STATUS_ROLLING_BACK`	Транзакция находится в процессе отмены.

Метод `setTransactionTimeout` устанавливает временной интервал (в секундах), в течение которого транзакция должна завершиться. Разные серверы приложений устанавливают разное время таймаута по умолчанию. Например, JBoss устанавливает таймаут по умолчанию равным 300 секундам, а Oracle Application Server 10g – 30 секундам. Этот метод может пригодиться для организации выполнения продолжительных транзакций. Однако, как правило, лучше переопределить настройки по умолчанию для всего сервера приложений с использовани-

ем специализированных интерфейсов. Однако сейчас вам, наверное, интереснее узнать, как устанавливать величину таймаута транзакций в модели CMT. Порядок настройки этого параметра определяется производителем и может осуществляться с помощью атрибута в дескрипторе развертывания или с применением аннотации.

6.3.4. Эффективное использование модели BMT

Компоненты EJB по умолчанию используют модель CMT управления транзакциями. Вообще не следует злоупотреблять применением модели BMT из-за увеличения объема кода, который придется писать, и более высокой сложности. Однако есть ряд вполне конкретных причин, оправдывающих применение модели BMT. Транзакции BMT необязательно должны начинаться и заканчиваться на границах единственного метода. В сеансовых компонентах с сохранением состояния иногда бывает желательно, чтобы действие транзакции простиралось на несколько вызовов методов, и тогда модель BMT оказывается как нельзя кстати. Правда в этом случае вы не сможете обращаться к компонентам, использующим модель CMT, и компонент с моделью BMT не может передать программный контекст другому компоненту. Кроме того, слишком легко можно проигнорировать исключения и не откатить или не подтвердить транзакцию. Это продемонстрировано в листинге 6.3, где последний обработчик исключения не вызывает метода `rollback`.

Одним из аргументов в пользу BMT является возможность точного управления границами транзакций, благодаря которой можно гарантировать сохранение данных, произведенных кодом, в самые кратчайшие сроки. Однако мы считаем, что подобной оптимизации следует избегать и лучше разбивать крупные методы на более мелкие и более специализированные. А теперь, после знакомства с транзакциями, обратим наше внимание на механизмы поддержки безопасности.

6.4. Безопасность EJB

Трудно переоценить важность поддержки безопасности в корпоративных приложениях. Скрыты ли они за стеной корпоративного брандмауэра или доступны через Веб, всегда найдутся желающие причинить вред. Безопасность обеспечивается двумя основными способами: предоставление доступа пользователям только к тем данным и операциям, на которые они имеют права, и предотвращение обхода защитных механизмов хакерами.

Поддержка безопасности должна осуществляться и в слое представления, и в слое прикладной логики. Отсутствие у пользователя доступа к странице еще не означает, что прикладная логика не должна соблюдать меры предосторожности. В мировой практике хакеры благополучно скомпрометировали множество вебслужб на основе технологии AJAX, доступ к которым предполагался только с ограниченного числа страниц и только зарегистрированными пользователями с определенными ролями. Одно из преимуществ EJB заключается в наличии мощной и гибкой модели безопасности. Используя эту модель, можно не только забло-

кировать доступ к странице, но и гарантировать ограничение доступа к методам компонентов EJB.

Подходы к поддержке безопасности в EJB в чем-то напоминают транзакции: вы можете управлять доступом декларативно или программно. Однако вам не придется выбирать между этими двумя моделями: вы можете свободно использовать декларативную модель в одних компонентах и программную в других. Итак, начнем наши исследования с обсуждения фундаментальных понятий – аутентификации и авторизации.

> **Примечание.** *Проект Open Web Application Security Project (OWASP) – это некоммерческая организация, основной задачей которой является поиск путей повышения безопасности в Веб. На сайте проекта можно найти массу превосходной документации, описывающей приемы обеспечения и тестирования безопасности приложений Java EE. Там же вы найдете пример приложения WebGoat, демонстрирующий, как не надо писать веб-приложения. Это приложение распространяется с открытыми исходными текстами и включает документацию, объясняющую различные ошибки в системе безопасности приложения, порядок их использования и исправления.*

6.4.1. Аутентификация и авторизация

Поддержка безопасности приложения заключается в двух основных функциях: аутентификации и авторизации. Аутентификация должна выполняться перед авторизацией, но, как будет показано, обе они являются необходимыми аспектами безопасности приложения.

Аутентификация

Аутентификация – это процедура проверки идентичности пользователя. Путем аутентификации пользователь доказывает свою подлинность. В реальном мире аутентификация может проводиться путем сличения внешности с фотографией на документе, подписи/почерка, отпечатков пальцев или с помощью генетической экспертизы. В царстве компьютеров аутентификация обычно заключается во вводе имени пользователя и пароля.

Авторизация

Авторизация – это процедура определения прав пользователя на доступ к ресурсам и операциям. Авторизации предшествует аутентификация – только после аутентификации пользователя можно установить круг его прав. В открытых системах аутентифицированный пользователь имеет доступ к любым ресурсам и операциям. Однако большинство систем ограничивают доступ к ресурсам, опираясь на идентичность пользователя. Даже при том, что в таких системах могут иметься ресурсы, доступные всем, доступ к большинству ресурсов предоставляется только узкому кругу пользователей. Аутентификация и авторизация тесно связаны с такими понятиями, как пользователи, группы и роли, о которых мы поговорим далее.

6.4.2. Пользователи, группы и роли

Пользователи, группы и роли – это три, тесно связанных между собой понятия, образующие основу поддержки безопасности в EJB. Мы уже упоминали понятие «пользователь», поэтому начнем с групп. Для простоты администрирования, пользователи делятся на группы, как показано на рис. 6.4. *Группы* – это способ логического разделения пользователей, помогающий приложению идентифицировать, кто к каким функциям должен иметь доступ – например, к функциям администрирования, поддержке клиентов и так далее. Все пользователи в группе Administrator могут проверять активность учетных записей и удалять их, а пользователи в группе Customer Service могут только изменять статус заказа. Прежде чем выполнить операцию, приложение проверяет принадлежность пользователя к соответствующей группе. Поддержка пользователей/групп в EJB построена по аналогии с поддержкой пользователей/групп в файловой системе Unix и упрощает управление списками доступа (access lists) к отдельным ресурсам.

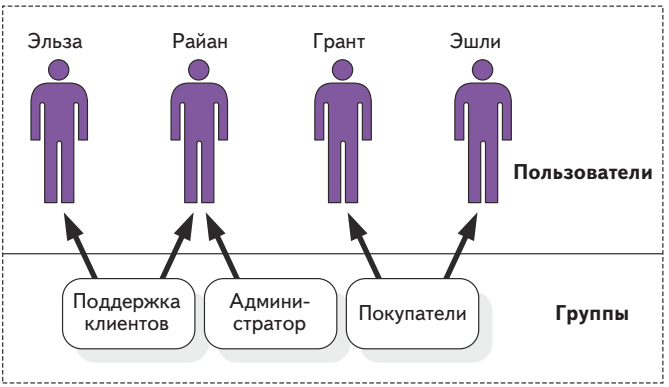

Рис. 6.4. Пользователи и группы

Понятие роли очень близко к понятию группы. *Роль* – это абстракция группы в контейнере приложения. Обычно в приложении определяется система соответствий между группами и ролями. Такое разделение позволяет писать приложения независимо от организации окружения, где выполняется развертывание. Приложение может предусматривать более тонкое деление прав доступа, чем поддерживается системой, или может быть приобретено у стороннего производителя, не знакомого с разделением ролей в вашей компании. То есть, эта абстракция позволяет отделить группы от фактических названий групп, используемых в производстве. Приложение может иметь группу Administrator, тогда как в каталоге LDAP компании аналогом ей является роль Department Head (начальник отдела, начальник службы). Конкретное оформление системы взаимосвязей между группами и ролями зависит от сервера приложений – многие контейнеры способны автоматически устанавливать связи при совпадении названий групп и ролей.

6.4.3. Как реализована поддержка безопасности в EJB

Поддержка безопасности в Java EE в значительной степени основана на службе Java Authentication and Authorization Service (JAAS). Служба JAAS отделяет систему аутентификации от приложения Java EE посредством четко определенного, подключаемого API. Приложение Java EE взаимодействует только с JAAS API – оно не несет ответственности за такие низкоуровневые тонкости, связанные с аутентификацией пользователя, как шифрование паролей или взаимодействие с внешней службой аутентификации, такой как Microsoft Active Directory или LDAP. Все это берет на себя модуль, созданный разработчиками контейнера, и доступный вам для настройки. В дополнение к аутентификации контейнер реализует еще и авторизацию в слое прикладной логики и в веб-слое EJB, так же с использованием JAAS.

JAAS API спроектирован так, что позволяет выполнять оба шага – аутентификацию и авторизацию – из любого слоя Java EE, включая веб-слой и слой компонентов EJB. Однако в реальности большинство приложений Java EE доступно через Веб и используют систему аутентификацию на всех уровнях, если не во всем сервере приложений. Механизм JAAS учитывает реалии жизни и использует информацию об аутентификации пользователя на всех уровнях Java EE. После аутентификации соответствующий контекст становится доступен всем уровням приложения, что избавляет от необходимости повторно выполнять аутентификацию. Контекст аутентификации доступен в виде объекта `Principal`, с которым мы уже познакомились в главе 5. На рис. 6.5 изображен типичный сценарий управления безопасностью в Java EE.

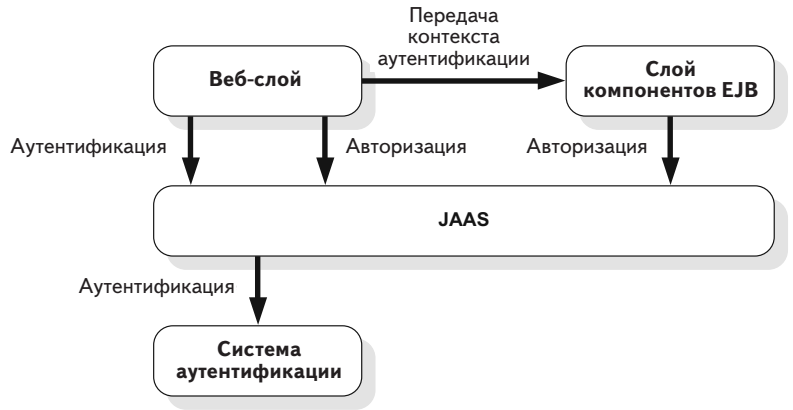

Рис. 6.5. Наиболее типичный сценарий управления безопасностью в Java EE с помощью JAAS

Как показано на рис. 6.5, пользователь входит в приложение через веб-слой. Веб-слой получает от пользователя информацию об аутентификации и проверяет

ее с помощью JAAS. В случае успеха создается действительный объект `Principal`. В этот момент объекту `Principal` присваивается одна или более ролей. После этого, при обращении к каждому ресурсу в веб-слое или в слое компонентов EJB, сервер приложений будет проверять право на доступ к ресурсу. При необходимости объект `Principal` прозрачно передается из веб-слоя в слой компонентов EJB.

Детальное обсуждение особенностей аутентификации и авторизации в веб-слое далеко выходит за рамки данной книги, как и чрезвычайно редкий сценарий отдельной аутентификации на уровне EJB с применением JAAS. Тем не менее, мы познакомим вас с основными моментами, связанными с безопасностью в веб-слое, чтобы вам было от чего отталкиваться в будущем, а затем перейдем к управлению авторизацией в EJB 3.

Аутентификация и авторизация в веб-слое

Спецификация сервлетов веб-слоя (http://java.sun.com/products/servlet) благополучно скрывает множество низкоуровневых тонкостей, связанных с аутентификацией и авторизацией. Как разработчику, вам достаточно сообщить контейнеру сервлета, какие ресурсы должны находиться под защитой системы безопасности и как должна осуществляться их защита – то есть, перечислить роли, обеспечивающие право доступа к ресурсам. Обо всем остальном позаботится контейнер сервлетов.

Поддержка безопасности в веб-слое настраивается с помощью элементов `login-config` и `security-constraint` в файле `web.xml`. В листинге 6.5 показано, как выглядят настройки безопасности административных страниц в приложении ActionBazaar.

Листинг 6.5. Фрагмент файла web.xml с настройками безопасности для функции отмены заказа и других

```
<login-config>
  <!-- ❶ Устанавливается метод аутентификации FORM -->
  <auth-method>FORM</auth-method>
  <!-- ❷ Имя области действия аутентификации -->
  <realm-name>ActionBazaarRealm</realm-name>
  <form-login-config>
    <!-- ❸ Форма, используемая для аутентификации -->
    <form-login-page>/login.faces</form-login-page>
    <!-- ❹ Страница с сообщением об ошибке,
          на случай неудачи аутентификации -->
    <form-error-page>/login_error.faces</form-error-page>
  </form-login-config>
</login-config>

<security-constraint>
  <web-resource-collection>
    <web-resource-name>
      Action Bazaar Administrative Component
    </web-resource-name>
    <!-- ❺ Шаблон URL блокируемых страниц -->
    <url-pattern>/admin/*</url-pattern>
  </web-resource-collection>
```

```
  <auth-constraint>
    <!-- ❻ Роль, дающая право доступа к коллекции ресурсов -->
    <role-name>CSR</role-name>
  </auth-constraint>
</security-constraint>
```

Эти настройки определяют, как веб-контейнер должен получать и проверять информацию об аутентификации. В данном случае ввод информации осуществляется с помощью нестандартной формы, поэтому в качестве метода аутентификации указан метод `FORM` ❶. Когда аутентификация выполняется с помощью веб-формы, необходимо указать форму, а также страницу с сообщением об ошибке. Существует еще два метода аутентификации: `BASIC` и `CLIENT-CERT`. При использовании метода `BASIC` веб-браузер будет отображать стандартный диалог, запрашивающий имя пользователя и пароль. `CLIENT-CERT` – это более сложная форма аутентификации, вообще не связанная с вводом имени/пароля. При ее использовании клиент отправляет веб-серверу публичный ключ сертификата, хранимого веб-браузером по протоколу Secured Socket Layer (SSL), а сервер проверяет содержимое сертификата. После этого провайдер JAAS проверяет права доступа.

Далее указывается имя области (realm), которую должен использовать контейнер для аутентификации ❷. Область – это абстракция контейнера над системой аутентификации, управляемой с помощью JAAS. Области (по сути, механизм извлечения информации для сверки данных, введенных пользователем) настраиваются с применением административных инструментов самого контейнера. Обычно контейнеры предоставляют несколько реализаций областей для извлечения проверочной информации из базы данных или LDAP. Если потребуется организовать аутентификацию с применением некоторой внешней системы, не поддерживаемой контейнером, вам придется обеспечить собственную реализацию области. Реализация областей в разных контейнерах осуществляется по-разному.

После определения области определяется форма для ввода имени пользователя и пароля ❸, а также страница с сообщением об ошибке ❹. В качестве адресов указываются URL, которые будут запрашиваться браузером. В нашем веб-приложении страницы XHTML обрабатываются как страницы JSF с расширением `*.jsf`. В зависимости от настроек расширение может отличаться.

Следующие два элемента определяют коллекцию страниц, доступ к которым следует обезопасить ❺, и роль, необходимую для обращения к этим страницам ❻. Указанный здесь шаблон URL (`/admin/*`), говорит о том, что права доступа должны проверяться для всех файлов в каталоге `admin`. Имя роли должно быть связано с именем группы в области.

В листинге 6.6 приводится разметка формы аутентификации. Когда форма будет передана браузером на сервер, контейнер извлечет параметры запроса с именем пользователя и паролем, и выполнит аутентификацию.

Листинг 6.6. Пример формы аутентификации для приложения ActionBazaar

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

```
<HTML xmlns:f=http://java.sun.com/jsf/core
  xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>ActionBazaar Admin Login</title>
  </h:head>
  <h:body>
    <!-- ❶ Определить действие для формы -->
    <form action="j_security_check" method="POST">
      <h:panelGrid columns="2">
        <f:facet name="header">
          Authentication
        </f:facet>
        <h:outputText value="Username"/>
        <!-- ❷ Поле ввода имени пользователя -->
        <input type="text" name="j_username" size="25"/>
        <h:outputText value="Password"/>
        <!-- ❸ Поле ввода пароля -->
        <input type="password" size="15" name="j_password"/>
        <h:panelGroup>
          <input type="submit" value="Submit"/>
          <input type="reset" value="Reset"/>
        </h:panelGroup>
      </h:panelGrid>
    </form>
  </h:body>
</HTML>
```

Контейнер должен содержать реализацию действия `j_security_check` для обработки формы ❶. В разметке присутствуют оба поля, необходимых для ввода имени пользователя ❷ и пароля ❸. Метод аутентификации `FORM` позволяет определить собственную форму аутентификации и использовать ее взамен универсального модального диалога, который выводит браузер, когда используется метод `BASIC`. А теперь поговорим немного об аутентификации и авторизации на уровне компонентов EJB.

Аутентификация и авторизация в EJB

Существует также возможность организовать аутентификацию в автономных клиентских приложениях, таких как настольные приложения на основе библиотеки Swing. Однако на практике она используется редко, потому что для решения этой задачи требуется кропотливый труд по реализации поддержки всех механизмов системы безопасности, предоставляемых контейнером. Впрочем, многие контейнеры приложений включают в себя JAAS-модуль аутентификации, упрощающий эту задачу. Однако, так как эта книга посвящена EJB, а не специфическим реализациям в разных контейнерах, мы не будем пытаться охватить их здесь – это слишком зыбкая тема, в которой очень многие факторы зависят от конкретного производителя и версии продукта.

Теперь, когда мы познакомились с тем, как осуществляется аутентификация, перейдем к обсуждению авторизации. Для начала займемся исследованием декларативным управлением безопасностью.

6.4.4. Декларативное управление безопасностью в EJB

Декларативное управление безопасностью чем-то напоминает модель CMT управления транзакциями. Вы сообщаете контейнеру о своих желаниях посредством аннотаций или файлов с настройками, а контейнер заботится об их воплощении. Аннотации и/или настройки в файлах могут применяться как к классам целиком, так и к отдельным их методам. Контейнер просматривает результаты преобразования ролей, используемых сервером, в группы приложения и затем сопоставляет их со списком допустимых ролей, чтобы определить возможность обращения к конкретному методу или компоненту. Если обращение невозможно, возбуждается исключение.

Чтобы дискуссия получилась более предметной, давайте посмотрим, какие проблемы безопасности имеются в приложении ActionBazaar. Представителям службы поддержки клиентов (Customer Service Representatives, CSR) при определенных обстоятельствах разрешено удалять ставки, сделанные клиентами – например, если клиент узнает от продавца какие-то факты, не упомянутые в описании. Но в оригинальной реализации операции удаления ставки не выполняется никаких проверок, чтобы дать доступ к ней только представителям службы поддержки.

Проанализировав логику соглашений об именовании в формах и ссылках, умный хакер, после нескольких проб и ошибок, сможет настрочить небольшой сценарий на Perl, который будет конструировать POST-запросы, удаляющие ставки. Затем хакер может инициировать «гонку цен», пытаясь продать свой товар подороже, делая с другой учетной записи все более и более высокие ставки, вынуждая добропорядочных пользователей поднимать свои ставки. Когда потенциальные покупатели прекратят повышать ставки, с помощью утилиты хакер сможет удалить свои ставки, и никто не поймет, что же собственно произошло.

Через какое-то время служба поддержки конечно же обнаружит эту схему, разбираясь с жалобами клиентов, удивленных фактом победы в торгах, когда они явно видели, что была предложена более высокая ставка. Чтобы предотвратить подобное жульничество, необходимо настроить систему безопасности так, чтобы только представители службы поддержки могли удалять ставки. Декларативная реализация такой защиты представлена в листинге 6.7.

Листинг 6.7. Защита функции удаления ставок с использованием декларативного управления безопасностью

```
// ❶ Объявление ролей для компонента
@DeclareRoles({"BIDDER", "CSR", "ADMIN"})
@Stateless(name = "BidManager")
public class BidManagerBean implements BidManager {
  // ❷ Список ролей, разрешающих доступ к методу
  @RolesAllowed({"CSR","ADMIN"})
  public void cancelBid(Bid bid) {
    ...
  }
```

```
  // ❸ Разрешает доступ для всех системных ролей
  @PermitAll
  public List<Bid> getBids(Item item) {
    return item.getBids();
  }
}
```

В этом листинге можно видеть несколько наиболее часто используемых аннотаций поддержки системы безопасности, определенных в JSR-250, включая `javax.annotation.security.DeclarRoles`, `javax.annotation.security.RolesAllowed` и `javax.annotation.security.PermitAll`. Имеется еще две аннотации, отсутствующие здесь, но которые будут обсуждаться далее: `javax.annotation.security.DenyAll` и `javax.annotation.security.RunAs`. Начнем наши изыскания с аннотации `@DeclareRoles`.

Аннотация @DeclareRoles

Аннотация `@DeclareRoles` перечисляет роли, используемые для проверки прав доступа к компоненту EJB. Она может применяться только к классу целиком. В отсутствие этой аннотации, контейнер найдет все аннотации `@RolesAllowed` и сам сконструирует список ролей для класса. Если класс компонента наследует класс другого компонента, списки ролей будут объединены. Согласно листингу 6.7, доступ к компоненту `BidManagerBean` ограничивается ролями `BIDDER`, `CSR` и `ADMIN` ❶.

Аннотация @RolesAllowed

Аннотация `@RolesAllowed` ❷ является решающей при декларативном управлении безопасностью. Она может применяться либо к прикладным методам компонента EJB, либо ко всему классу. При применении ко всему классу, она сообщает контейнеру список ролей, которым позволено обращаться к любым методам компонента. Если эта аннотация применяется к методу, она определяет правила доступа к этому конкретному методу. Огромная гибкость аннотации становится особенно очевидной, после осознания того факта, что с ее помощью можно переопределять настройки, выполненные на уровне класса. Например, доступ ко всему компоненту `BidManagerBean` можно было бы предоставить только администраторам и разрешить обращаться к отдельным методам, таким как `getBids`, другим ролям. Но имейте в виду, что такая чехарда с ролями внутри класса быстро может завести в тупик. В таких ситуациях лучше и проще использовать отдельный компонент.

Аннотации @PermitAll и @DenyAll

Аннотации `@PermitAll` и `@DenyAll` говорят сами за себя – они открывают или закрывают доступ сразу всем. Аннотация `@PermitAll` в листинге 6.7 ❸ сообщает контейнеру, что любой пользователь сможет получить текущие ставки для указанного лота. Используйте эту аннотацию с большой осторожностью, особенно на уровне класса, потому что при небрежном использовании с ее помощью можно проделать брешь в системе безопасности.

Аннотация `@DenyAll` делает недоступным класс или метод для любой роли. Она не имеет большой практической ценности, превращая метод в бесполезный фрагмент кода, пресекая любые попытки обращения к нему. Однако использование ее эквивалента в конфигурационном файле дает возможность запретить доступ к методам не изменяя ни одной строчки кода. Это может пригодиться, когда понадобится закрыть доступ к какой-нибудь функции после развертывания приложения.

@RunAs

Аннотация `@RunAs` напоминает утилиту `sudo` в Unix, которая позволяет выполнить команду от имени другого пользователя. Часто этим другим пользователем оказывается администратор. Данная аннотация дает возможность выполнять операции с разными наборами привилегий, которые могут быть более или менее ограниченными. Например, методу `cancelBid` в листинге 6.7 может потребоваться вызвать компонент, управляющий сохранением записей в журнале и удаляющий запись, созданную в момент добавления ставки. В этой гипотетической ситуации компонент управления записями в журнале может требовать привилегий роли `ADMIN`. С помощью аннотации `@RunAs` можно временно присвоить пользователю с ролью `CSR` привилегии роли `ADMIN`, чтобы вызываемый компонент EJB считал, что к нему обращается администратор:

```
@RunAs("ADMIN")
@RolesAllowed({"CSR"})
public void cancelBid(Bid bid, Item item) {...}
```

Эта аннотация должна использоваться с большой осторожностью. Как и аннотация `@PermitAll`, она способна проделать брешь в системе безопасности. Теперь, когда вы получили представление о декларативном управлении безопасностью, перейдем к знакомству с программным управлением.

6.4.5. Программное управление безопасностью в EJB

Декларативное управление безопасностью обладает широкими возможностями, но иногда бывает необходимо организовать более избирательное управление. Например, может потребоваться менять поведение метода в зависимости от роли пользователя или даже имени пользователя. Кроме того, может оказаться желательно проверить принадлежность пользователя сразу двум группам или убедиться, что пользователь входит в одну группу и не входит в другую. Используя программное управление безопасностью легко можно реализовать подобные сценарии. Прежде чем углубиться в изучение, хочу предупредить, что программная и декларативная модели управления безопасностью не являются взаимоисключающими. В отличие от управления транзакциями, класс не требуется отмечать какой-то специфической аннотацией, устанавливающей декларативную или программную модель, как это было в случае с транзакциями.

Программное управление безопасностью осуществляется посредством объекта SessionContext. Из него можно получить объект Principal и/или проверить с его помощью привилегии вызывающего кода на принадлежность определенной роли. В листинге 6.8 приводится реализация функции отмены ставки, обсуждавшейся в предыдущем разделе, но на этот раз управление безопасностью осуществляется программно.

Листинг 6.8. Защита функции удаления ставок с использованием программного управления безопасностью

```
@Resource
// ❶ Внедрение контекста EJB
private SessionContext context;
public void cancelBid(Bid bid) {
  // ❷ Проверка авторизации
  if(!context.isCallerInRole("CSR") && !context.isCallerInRole("ADMIN"))
  {
    // ❸ Возбудить исключение при нехватке прав
    throw new SecurityException(
      "You do not have permission to cancel an order.");
    ...
  }
}
```

В листинге сначала производится внедрение контекста EJB ❶. Далее с помощью метода isCallerInRole объекта EJBContext проверяется – обладает ли вызывающий код правами роли CSR ❷. Если проверка не увенчалась успехом, возбуждается исключение java.lang.SecurityException, извещающее пользователя о недостаточности прав ❸. В противном случае методу удаления ставки разрешается продолжить работу. В следующем разделе мы поближе познакомимся с двумя методами управления безопасностью, предоставляемыми объектом контекста EJB – isCallerInRole и getCallerPrincipal.

isCallerInRole и getCallerPrincipal

Программное управление безопасностью осуществляется достаточно просто. Основные операции управления выполняются с помощью двух методов, доступных через интерфейс javax.ejb.EJBContext, как показано ниже:

```
public interface EJBContext {
  ...
  public java.securityPrincipal getCallerPrincipal();
  public boolean isCallerInRole(java.lang.String roleName);
  ...
}
```

Вы уже видели метод isCallerInRole в действии. Его имя говорит само за себя. За кулисами объект контекста EJB извлекает объект Principal, связанный с текущим потоком выполнения, и проверяет наличие роли с именем, совпадающим с указанным вами. Метод getCallerPrincipal дает прямой доступ

к объекту `java.security.Principal`, представляющему текущий контекст аутентификации. Единственный метод в интерфейсе `Principal`, представляющий интерес, – это метод `getName`, возвращающий имя пользователя, с привилегиями которого выполняется код. Чаще всего это имя будет совпадать с именем текущего пользователя. Это означает, что в простых случаях с его помощью можно проверять имена отдельных пользователей.

Например, представьте, что вы решили дать клиентам возможность отменять свои заказы в течение пяти минут. Эта функция предоставляется вдобавок к возможности удаления ставок сотрудниками службы поддержки. Реализовать это можно с помощью метода `getCallerPrincipal`, как показано ниже:

```
public void cancelBid(Bid bid) {
  if(!context.isCallerInRole("CSR") &&
      !context.isCallerInRole("ADMIN") &&
      (!bid.getBidder().getUsername().equals(
      context.getCallerPrincipal().getName()) &&
      bid.getBidDate().getTime() >= (new Date().getTime() - 60*1000))) {
    throw new SecurityException("You do not have permission to cancel an
      order.");
    ...
  }
}
```

Важно отметить, что нет никаких гарантий, какое имя вернет объект `Principal`. В некоторых окружениях он может вернуть имя роли, имя группы или любую произвольную строку, имеющую смысл для системы аутентификации. Прежде чем использовать метод `Principal.getName`, вы должны прочитать документацию с описанием вашего конкретного окружения. Как видите, программное управление безопасностью имеет один большой недостаток – смешивание кода поддержки безопасности с прикладной логикой, а также возможность жесткого определения имен ролей и пользователей в коде. В предыдущих версиях EJB не было никакой возможности обойти этот недостаток. Но в EJB 3 проблему можно решить с помощью интерцепторов. В следующем разделе мы покажем, как это делается.

Использование интерцепторов в программном управлении безопасностью

В EJB 3 имеется возможность определять интерцепторы, которые будут вызываться до и после любых прикладных методов компонентов EJB. Этот инструмент идеально подходит для реализации сквозных задач, которые не желательно было бы дублировать в каждом методе. Решение, представленное в листинге 6.8, можно также реализовать с применением интерцепторов, как показано в листинге 6.9.

Листинг 6.9. Применение интерцепторов в программном управлении безопасностью

```
public class SecurityInterceptor {
  @Resource
  private SessionContext sessionContext;
```

```
  // ❶ Пометить метод как интерцептор
  @AroundInvoke
  public Object checkUserRole(InvocationContext context)
      throws Exception {
    // ❷ Обращение к EJBContext из InvocationContext
    if(!sessionContext.isCallerInRole("CSR")) {
      throw new SecurityException("No permission to cancel bid.");
    }
    return context.proceed();
  }
}
@Stateless
public class BidManagerBean implements BidManager {
  // ❸ Назначается интерцептор для метода
  @Interceptors(SecurityInterceptor.class)
  public void cancelBid(Bid bid) {...}
}
```

Метод `checkUserRole` класса `SecurityInterceptor` отмечен как интерцептор типа `AroundInvoke`, который вызывается при каждом обращении к перехватываемому методу ❶. В методе с помощью объекта `Principal` проверяется наличие привилегий роли `CSR` ❷. Если соответствие указанной роли не обнаруживается, возбуждается исключение `SecurityException`. Класс `BidManagerBean`, в свою очередь, указывает, что класс `SecurityInterceptor` играет роль интерцептора для метода `cancelBid` ❸.

Обратите внимание, что хотя интерцептор помогает убрать из прикладного кода жестко зашитые имена, он фактически не избавляет от этого недостатка, а просто переносит определения имен в интерцепторы. Это обстоятельство легко может сделать ситуацию неконтролируемой, если только не используется достаточно простая схема, когда большинство методов EJB имеют сходные требования к безопасности и можно повторно использовать небольшое число интерцепторов по всему приложению. В действительности вам потребуется написать ряд специализированных интерцепторов, учитывающих все возможные комбинации использования аутентификации. Сравните это с относительно простым подходом, основанном на использовании декларативного управления безопасностью с применением аннотаций или дескрипторов развертывания.

6.4.6. Эффективное использование поддержки безопасности в EJB

Как и в случае с транзакциями, для управления безопасностью рекомендуется использовать декларативный подход. Старайтесь не использовать конкретные имена пользователей и избегайте обращений к методу `Principal.getName()`. Как уже упоминалось выше, метод `getName` возвращает строку, в зависимости от используемой системы аутентификации. Возможно вас не волнуют проблемы переносимости кода между серверами приложений, однако в будущем реализация с учетом специфических особенностей той или иной системы аутентификации может

сыграть с вами злую шутку. Декларативная модель намного проще для анализа и меньше подвержена ошибкам, чем модель программного управления. Применяя декларативный подход можно аннотировать классы и при необходимости переопределять настройки безопасности для отдельных методов. Если в реализации программного управления забыть проверить привилегии или допустить ошибку при проверке, можно незаметно для себя создать брешь в системе безопасности. В случае с транзакциями ошибки будут проявляться в базе данных и их можно проанализировать. Ошибки в поддержке безопасности, напротив, не проявляются так явно. Как было показано на примере приложения ActionBazaar, первыми такие ошибки обычно находят пользователи, обнаруживающие внешнее влияние.

Всегда используйте поддержку безопасности, управляемую контейнером. Веб-контейнер автоматически передает контекст безопасности контейнеру EJB. Это гарантирует преемственность между веб-слоем и слоем прикладной логики. Кроме того, проверки безопасности должны осуществляться и на уровне представления (WAR), и на уровне прикладной логики (EJB). Поддержка безопасности должна быть реализована четко и ясно – не питайте себя надеждами, что пользователи не поймут, как сгенерировать POST-запрос, удаляющий записи или получающий сведения, на доступ к которым у них нет прав, потому что пользователи постоянно находятся в поиске чего-то подобного.

6.5. В заключение

В этой главе мы обсудили теоретические основы управления транзакциями с применением моделей CMT и BMT, основные понятия безопасности и приемы управления ею с применением программной и декларативной моделей. И транзакции, и безопасность являются сквозными задачами, которые в идеале не должны пересекаться с прикладной логикой. EJB 3 занимается управлением безопасностью и транзакциями, пытаясь максимально соответствовать этому идеалу. Минимизация присутствия сквозных задач в прикладной логике упрощает код и снижает число ошибок, которые неизбежны при повторении одной и той же логики во многих местах.

Изначально, при использовании встроенного механизма сохранения данных в контейнере, транзакции охватывают вызовы методов компонентов EJB. На фронте безопасности на компоненты вообще не накладывается никаких ограничений на ее поддержку. По умолчанию веб-слой может вызывать любые компоненты, и любой компонент может вызывать любые другие компоненты с привилегиями любого пользователя. В обоих случаях для управления транзакциями и безопасностью лучше использовать декларативный подход. Декларативное управление безопасностью допускается смешивать с программным, но модели CMT и BMT управления транзакциями нельзя смешивать в одном компоненте. Кроме того, использование модели BMT в одном компоненте и CMT в другом может приводить к быстро нарастающим сложностям, если один из этих компонентов использует другой. И для управления транзакциями, и для управления безопасностью может потребоваться выполнить дополнительную настройку контейнера.

Обсуждением проблем управления транзакциями и безопасностью мы завершаем рассмотрение сеансовых компонентов и компонентов, управляемых сообщениями. В следующей главе мы познакомимся с таймерами и рассмотрим вопросы планирования выполнения заданий во времени.

ГЛАВА 7. Планирование и таймеры

Эта глава охватывает следующие темы:

- основы службы EJB Timer Service;
- основы cron;
- разные типы таймеров;
- объявление таймеров cron;
- использование специальных таймеров.

До сих пор мы рассматривали только операции с компонентами EJB, инициируемые извне, такие как вызов метода из веб-слоя или обработка сообщения, доставленного посредством службы JMS. Чтобы произошел вызов метода, пользователь должен был щелкнуть на каком-нибудь элементе веб-страницы или должна была произойти доставка сообщения JMS. И действительно, большая часть операций в приложении выполняется по запросу пользователя, однако иногда бывает необходимо, чтобы приложение само выполнило какие-то действия, через некоторый интервал или в определенный момент времени. Например, может быть желательно, чтобы приложение рассылало напоминания по электронной почте в начале дня, выполняло ночью обработку накопившихся записей или сканировало стороннюю базу данных в поисках определенных записей каждые 15 минут. Реализовать такое поведение можно с помощью службы таймеров EJB Timer Service. Эта служба поможет вам создавать приложения, планирующие свои операции.

В версии 3.1 служба EJB Timer Service претерпела существенные изменения и дополнения, в число которых вошла поддержка cron-подобного планирования. В версиях EJB до 3.1 поддерживалась только возможность задержки (например, на 10 минут) и выполнения через регулярные интервалы (истекающие, например, каждые 20 минут). Прежде, чтобы запланировать выполнение операций на определенное время и/или дату, необходимо было обращаться к сторонним решениям. Но, несмотря на широту возможностей, эти решения привносили дополнительные сложности. Полная поддержка cron-подобного планирования является насущной

необходимостью для многих приложений, и эта поддержка должна быть стандартизована и присутствовать во всех контейнерах.

В этой главе мы посмотрим, как создавать таймеры декларативно или программно, а также исследуем проблемы взаимодействия таймеров с транзакциями и системой безопасности EJB. Но сначала познакомимся с основами планирования.

7.1. Основы планирования

В этой главе рассматриваются вопросы планирования вызовов методов компонентов EJB с использованием службы таймеров EJB Timer Service. По сути служба Timer Service мало чем отличается от обычного таймера, встроенного в кухонную плиту. Когда таймер отсчитывает установленное время, вам остается только вынуть готовый пирог из духовки или добавить в горшок новый ингредиент. То же самое происходит и в контейнере: когда приложение создает таймер – программно или путем объявления в конфигурационном файле – оно взводит таймер. Когда таймер отсчитывает установленный интервал, контейнер извлекает компонент из пула (если это не компонент-одиночка) и вызывает требуемый метод, как показано на рис. 7.1. В случае с компонентом-одиночкой используется существующий экземпляр этого компонента. Метод вызывается точно так же, как если бы его вызов был инициирован пользователем через веб-интерфейс. Контейнер автоматически выполнит внедрение всех необходимых зависимостей, без всякого вмешательства с вашей стороны. Если таймер окажется периодическим (например, срабатывающим каждые 10 минут), автоматически будет создан новый таймер, которые сработает через 10 минут.

Рис. 7.1. Суть службы Timer Service

В последующих разделах мы будем учиться устанавливать таймеры, а также познакомимся с возможностями планировщика cron, интерфейсом и с различными типами таймеров. В этом разделе приводится довольно много сведений, усвоить которые будет непросто, но в действительности все механизмы, представленные здесь, фактически являются всего лишь мощными таймерами для варки яиц. Итак, начнем со знакомства с основными возможностями службы Timer Service.

7.1.1. Возможности Timer Service

Давайте займемся исследованием фундаментальных возможностей службы EJB Timer Service. С ее помощью можно запланировать выполнение некоторой задачи на определенную дату и время, через регулярные интервалы времени или спустя определенный промежуток времени. Несмотря на высокую надежность и способность переживать перезапуск сервера, служба Timer Service все же не поддержи-

вает возможность планирования реального времени. Это означает, что механизмы планирования EJB не подходят для ситуаций, когда требуется реагировать на события в течение измеримого интервала времени. Под измеримым интервалом понимается промежуток, в течение которого приложение должно гарантированно обработать запрос или выдать ответ. Служба Timer Service используется в ситуациях, когда не предъявляется жестких требований к времени реакции (то есть, когда не может произойти ничего плохого, если приложение опоздает на наносекунду). Например, службу Timer Service можно с успехом использовать для рассылки электронной почты в полночь. Однако использовать ее для мониторинга показаний трубок Пито на самолете и рассчитывать тягу двигателей – это плохая идея.

Теперь посмотрим, какие типы компонентов поддерживаются таймерами, исследуем их надежность и способы создания. А после достаточно абстрактного обсуждения перейдем к практике.

Поддерживаемые типы компонентов

Начиная с версии EJB 3.1, службу EJB Timer Service стало возможно использовать с сеансовыми компонентами без сохранения состояния, компонентами-одиночками и компонентами, управляемыми сообщениями. Сеансовые компоненты с сохранением состояния не поддерживаются – возможно в будущем такая поддержка появится, но на сегодняшний день она отсутствует. Это означает, что при работе с компонентами, кроме компонентов-одиночек, вы сами должны организовать сохранение их состояния между срабатываниями таймера. Каждый раз, когда срабатывает таймер, из пула может быть извлечен другой экземпляр сеансового компонента без сохранения состояния, как показано на рис. 7.2. В ситуациях, где необходимо сохранять состояние компонента, можно использовать компоненты-одиночки, но имейте в виду, что при большом количестве таймеров, управляющих единственным компонентом-одиночкой, могут возникать конфликты при вызове метода в одно и то же время. А теперь обсудим надежность таймеров.

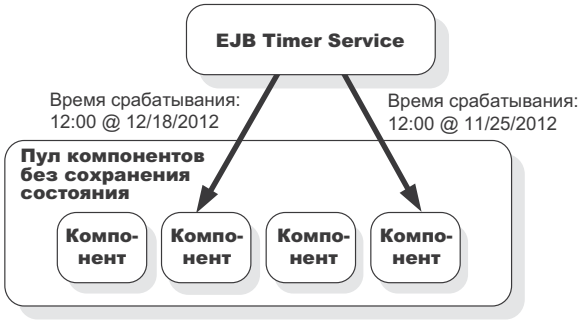

Рис. 7.2. Служба Timer Service и пул сеансовых компонентов без сохранения состояния

Надежность

Для любого таймера надежность играет важную роль. Даже простые будильники снабжаются батарейками, чтобы в случае пропадания питания от электросети часы продолжали показывать точное время и подавали сигнал в нужный момент времени. Благодаря этому вы можете спокойно спать, будучи уверенными, что даже в самой неблагоприятной ситуации будильник разбудит вас вовремя, и вы не проспите на работу или на самолет, улетающий на Гавайи. Если питание будет отсутствовать в момент, когда должен прозвенеть звонок, даже самые совершенные часы ничего не смогут сделать. Таймеры службы EJB Timer Service спокойно переживают перезапуск сервера без какого-либо вмешательства с вашей стороны.

Когда вы планируете выполнение операции, контейнер создает таймер, который вызовет указанный метод в назначенное время. В момент срабатывания, таймер сразу же вызывает метод, если он принадлежит компоненту-одиночке, или предварительно извлечет экземпляр сеансового компонента без сохранения состояния из пула. Этот сценарий развития событий изображен на рис. 7.3. Обратите внимание, что компоненты-одиночки не сериализуются между перезапусками сервера, поэтому в них важно предусмотреть возможность сохранения состояния. Сеансовые компоненты с сохранением состояния не поддерживаются таймерами.

Рис. 7.3. Таймеры и сеансовые компоненты без сохранения состояния или компоненты-одиночки

Создание таймеров

Запланировать выполнение операций можно в конфигурационном файле, программно или с помощью аннотаций. Каждый вариант соответствует определенным потребностям. В конфигурационном файле обычно настраивается выполнение операций по некоторому фиксированному расписанию. Примером может служить ночное удаление временных файлов или сбор информации о выполненной работе. Программный способ обычно используется для создания таймеров по требованию конечного пользователя, а также, когда таймеры являются неотъемлемой частью логики работы приложения. Например, приложение может предоставлять администратору возможность запланировать рассылку по электронной почте

в определенную дату и время – организовать подобное через конфигурационный файл было бы непросто, так как потребовало бы повторно развернуть приложение после изменения настроек. Как будет показано далее в этой главе, планирование осуществляется достаточно просто.

7.1.2. Таймауты

На первый взгляд таймаут может выглядеть как наказание расшалившегося ребенка стоянием в углу[1]. Но в EJB таймауты не имеют никакого отношения к наказаниям в начальной школе. Под термином «таймаут» понимается действие, или метод, выполняемое при срабатывании таймера. Срабатывание таймера действует как звонок, дающий команду на выполнение каких-то операций – вы настраиваете таймер и указываете ему, какое действие должно быть выполнено. Действием может быть реализация метода `ejbTimeout`, определяемого интерфейсом `javax.ejb.TimedObject`, или любой другой метод, отмеченный аннотацией `@Scheduled`. Таймауты могут также настраиваться в конфигурационных файлах XML. Таймаут связывается с таймером. В оставшейся части главы мы подробно рассмотрим приемы создания и таймеров и определения таймаутов.

Примечание. *Таймаут – это действие, которое выполняется в момент срабатывания таймера.*

7.1.3. Cron

Под названием *cron* подразумевается планировщик заданий, имеющийся во многих Unix-системах, включая Mac OS X, Linux и Solaris. Он осуществляет приложения или сценарии в указанные моменты времени для выполнения таких задач, как ротация файлов журналов, перезапуск служб и так далее. Конфигурационный файл планировщика, известный как *crontab*, определяет команды для запуска и расписание. Cron – имеет чрезвычайно гибкие возможности планирования. Например, с помощью cron можно запланировать выполнение некоторого задания в первую субботу каждого месяца. В версии EJB 3.1 служба Timer Service так же обзавелась средствами планирования, сопоставимыми по своим возможностям с cron. Поэтому в Интернете появилось множество статей, расхваливающих новые возможности планирования в EJB. Прежде аналогичные механизмы не отличались широтой возможностей и были представлены сторонними решениями, такими как Quartz.

На рис. 7.4 представлен формат файла *crontab*. В нем имеется пять позиций, имеющих отношение к настройке расписания, за которыми следует команда, которую требуется выполнить. Звездочки соответствуют всем возможным значениям. Например, звездочка в первой позиции (крайняя слева) соответствует каждой минуте. Такая организация настроек открывает широкие возможности планиро-

[1] Слово «time-out» в английском языке имеет множество смыслов, в зависимости от контекста, в том числе и «наказание», обычно – наказание ожиданием, например, стоянием в углу. – *Прим. перев.*

вания выполнения сценариев. Конечно, этот планировщик – не система пакетной обработки (batch-queuing system), но все же его возможностей вполне достаточно для решения повседневных задач.

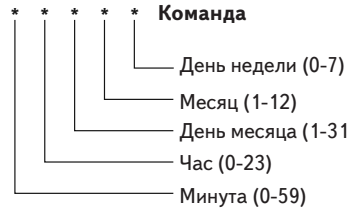

Рис. 7.4. Формат файла crontab

Если вы не знакомы с cron, взгляните на следующий фрагмент. Это – содержимое файла *crontab* из дистрибутива CentOS Linux. В нем демонстрируется настройка выполнения групп заданий раз в час, раз в неделю и раз в месяц. Сценарий `cron.hourly` будет выполняться раз в час, и так далее:

```
20 * * * * root run-parts /etc/cron.hourly
25 1 * * * root run-parts /etc/cron.daily
50 3 * * 0 root run-parts /etc/cron.weekly
43 4 16 * * root run-parts /etc/cron.monthly
```

Таймауты и cron являются ключом к пониманию службы EJB Timer Service, о которой рассказывается далее.

7.1.4. Интерфейс Timer

Чтобы запланировать отложенный вызов метода компонента, необходимо создать таймер. Таймер должен знать, через какое время сработать и какой метод вызвать. Соответственно, при создании таймера необходимо передать контейнеру следующую информацию: время срабатывания и метод для вызова, который в терминологии EJB называют *таймаутом*. На основе этой информации контейнер создаст таймер. Конкретная реализация таймеров зависит от создателей контейнера – таймеры должны быть надежными и, соответственно, их реализация достаточно сложна.

Как будет показано далее, существует три способа настройки таймеров: с помощью аннотаций, программно и посредством конфигурационных файлов. Управлять отдельными таймерами можно с помощью интерфейса `javax.ejb.Timer`, представленного в листинге 7.1. Этот интерфейс предоставляет единственную операцию `cancel` и несколько методов для извлечения информации о таймере. Для получения представления таймера, которое может быть сериализовано, используется метод `getHandle()`. Он возвращает `javax.ejb.TimerHandle`, с помощью которого можно получить экземпляр таймера. Объект `TimerHandle` может существовать только внутри контейнера – его нельзя передать удаленному коду. Информацию, указанную при создании таймера, можно извлечь вызовом

метода `getInfo()`. Это может быть любой сериализованный объект, но в атрибуте `info` аннотации `Schedule` допускается передавать только строки. Метод `getNextTimeout()` возвращает объект `Date` с временем срабатывания таймера. Обратите внимание, что к моменту, когда вы будете выполнять проверку, таймер может уже сработать. Для таймеров на основе cron метод `getSchedule()` возвращает `javax.ejb.ScheduleExpression` с информацией о расписании. Метод `getTimeRemaining()` возвращает число миллисекунд, оставшихся до срабатывания таймера. Метод `isCalendarTimer()` возвращает `true`, если таймер действует на основе механизма cron. Наконец, последний метод интерфейса, `isPersistent()` возвращает `true`, если таймер способен пережить перезапуск сервера.

Листинг 7.1. Интерфейс Timer

```
public interface Timer {
  void cancel();
  TimerHandle getHandle();
  Serializable getInfo();
  Date getNextTimeout();
  ScheduleExpression getSchedule();
  long getTimeRemaining();
  boolean isCalendarTimer();
  boolean isPersistent();
}
```

Экземпляр таймера можно получить обращением к `javax.ejb.TimerService`. Интерфейс `TimerService` содержит ряд методов для программного создания таймеров, а также методы для извлечения всех действующих таймеров. Важно отметить, что список таймеров постоянно изменяется – таймер, извлеченный с помощью этого интерфейса, может оказаться недействительным к моменту, когда вы приступите к его исследованию. В последующих разделах мы детальнее рассмотрим интерфейс `TimerService`, потому что он является основным инструментом создания таймеров.

До сих пор мы не касались темы использования транзакций и поддержки безопасности применительно к таймерам. Проблема транзакций имеет две стороны: как влияют транзакции на создание таймеров, и как методы, вызываемые таймерами, участвуют в транзакциях. Ответ на первый вопрос можно увидеть в следующем фрагменте кода. Если вызов `entityManager` потерпит неудачу, транзакция будет отмечена, как предназначенная для отката, что в свою очередь приведет к остановке таймера, если он еще не сработал. То же относится и к остановке (отмене) таймера: если метод, останавливающий таймер, произведет откат транзакции, это повлечет отмену остановки таймера.

```
public void addBid(Bid bid) {
  timerService.createTimer(15*60*1000,15*60*1000,bid);
  entityManager.persist(bid);
  ...
}
```

Метод, вызываемый таймером, действует в контексте транзакции. Если произошел откат транзакции, контейнер попробует перезапустить таймер. Это происходит в случае, когда компонент использует транзакции, управляемые контейнером, и метод, вызываемый таймером, отмечен значением `REQUIRED` или `REQUIRES_NEW`.

Что касается поддержки безопасности: метод, вызываемый таймером, действует вне контекста безопасности. То есть, если вызвать метод `getCallerPrincipal`, он вернет представление неаутентифицированного пользователя. То есть, если в методе, вызываемом таймером, попытаться обратиться к какому-нибудь защищенному методу, эта попытка потерпит неудачу. Не забывайте, что таймеры могут создаваться посредством конфигурационных файлов и потому не могут быть связаны с каким-либо пользователем. Теперь, когда мы познакомились с основами таймеров, пришло время посмотреть, какие типы таймеров существуют.

7.1.5. Типы таймеров

Существует два типа таймеров, которые поддерживают компоненты EJB: таймеры задержки и календарные таймеры. Таймеры задержки были добавлены в версии EJB 2.1 и могут рассматриваться как усовершенствованные таймеры для варки яиц. Создавая эти таймеры, можно указать, как следует вызывать указанный метод: через регулярные интервалы, в определенный момент, однократно, спустя заданный промежуток времени, и так далее. Например, таймер задержки можно использовать для обновления кэшированного списка ставок в компоненте-одиночке каждые 20 минут. Эти таймеры имеют свою область применения, но они не так гибки, как календарные таймеры.

В календарных таймерах используется та же концепция, что лежит в основе планировщика cron, о которой рассказывалось выше. На основе календарных таймеров можно строить весьма сложные расписания. Cron-подобная схема обладает широкими возможностями и упрощает поддержку планирования прикладных операций. Например, процедуру синхронизации данных можно запланировать на середину ночи, чтобы не оказывать отрицательного влияния на производительность системы днем, когда с ней будут работать люди.

Знакомство с типами таймеров является важным шагом к пониманию особенностей службы Timer Service. Это поможет вам сравнить службы, доступные в версиях ниже EJB 3.1 и выше. Если вы работаете с существующей системой EJB 3, вам придется задействовать внешние библиотеки, такие как Quartz.

Другой подход к изучению функциональности Timer Service заключается в исследовании особенностей создания таймеров. Таймеры могут создаваться декларативно или программно. Декларативные таймеры встраиваются в приложение и могут изменяться только посредством конфигурационных файлов, что влечет за собой необходимость перезапускать приложение. Программные таймеры создаются во время выполнения для организации планирования прикладных операций. Начнем с декларативного способа создания таймеров.

7.2. Декларативные таймеры

Декларативные таймеры можно создавать, помещая аннотации перед методами компонентов или включая объявления в конфигурационный файл приложения. Они лучше подходят для решения обычных задач поддержки или запуска прикладных операций, которые не изменяются с течением времени. Например, декларативные таймеры с успехом можно использовать для составления сводных отчетов о проделанной работе по ночам или, как в случае с приложением ActionBazaar, для загрузки последней информации из службы доставки, чтобы знать, какие заказы были приняты к исполнению. Поскольку настройка таймеров в конфигурационных файлах для разных контейнеров может выполняться по-разному, мы сосредоточимся в этом разделе на аннотации `@Schedule`. Эта аннотация была добавлена в версии EJB 3.1 и с ее помощью создаются календарные таймеры. Для создания таймера задержки следует использовать программный интерфейс, обсуждаемый в разделе 7.3.

7.2.1. Аннотация @Schedule

Добавление аннотации `@Schedule` перед методом приводит к созданию декларативного таймера. Аннотация может использоваться только в компонентах-одиночках, сеансовых компонентах без сохранения состояния и компонентах, управляемых сообщениями. В следующем фрагменте приводится определение аннотации. При использовании аннотации без атрибутов, таймер будет срабатывать каждый день в полночь. В атрибуте `info` можно передать текст с описанием, который может извлекаться во время выполнения – он не оказывает влияния на работу таймера:

```
@Target(value=METHOD)
@Retention(value=RUNTIME)
public @interface Schedule {
  String dayOfMonth() default "*";
  String dayOfWeek() default "*";
  String hour() default "0";
  String info() default "";
  String minute() default "0";
  String month() default "*";
  boolean persistent() default true;
  String second() default "0";
  String timezone() default "";
  String year() default "*";
}
```

Аннотация `@Schedule` предъявляет к целевому методу те же требования, что и аннотация `@Timeout`. Этот метод не должен возвращать значение и может иметь единственный аргумент – объект `javax.ejb.Timer`, обсуждавшийся выше. Ниже приводятся допустимые прототипы целевых методов:

```
void <METHOD>()
void <METHOD>(Timer timer)
```

При декларативном подходе для одного метода может быть создано несколько таймеров. Сделать это можно с помощью аннотации `@Schedules`. Эта аннотация принимает массив аннотаций `@Schedule`. Помните, что таймер не связан с каким-то определенным экземпляром компонента, поэтому неважно, сколько таймеров будет зарегистрировано для одного и того же метода – если все они сработают в одно и то же время, каждый из них будет оперировать отдельным экземпляром. В момент срабатывания таймера из пула извлекается новый экземпляр компонента – между таймерами и компонентами нет отношения «один-к-одному».

7.2.2. Аннотация @Schedules

Аннотация `@Schedules` используется с целью создания сразу нескольких календарных таймеров, для единственного метода и принимает массив аннотаций `@Schedule`. Вот как выглядит ее определение:

```
@Target({METHOD})
@Retention(RUNTIME)
public @interface Schedules {
  javax.ejb.Schedule[] value();
}
```

Прежде чем приступать к практическим примерам, вернемся на шаг назад и посмотрим, какие параметры принимает аннотация `@Schedule`.

7.2.3. Параметры аннотации @Schedule

Календарный таймер определяется с помощью выражения, формат которого близко напоминает формат, используемый утилитой cron. Выражение включает семь атрибутов: `second`, `minute`, `hour`, `dayOfMonth`, `month`, `dayOfWeek` и `year`. Для каждого из этих атрибутов существуют четко очерченный круг значений и синтаксических правил, включая диапазоны, списки и приращения. Все эти атрибуты и их значения по умолчанию перечислены в табл. 7.1.

Таблица 7.1. Атрибуты выражений в аннотации `@Schedules`

Атрибут	Допустимые значения	По умолчанию
`second`	[0, 59]	0
`minute`	[0, 59]	0
`hour`	[0, 23]	0
`dayOfMonth`	[1, 31]	*
`month`	[1, 12] или {“Jan”, “Feb”, “Mar”, “Apr”, “May”, “Jun”, “Jul”, “Aug”, “Oct”, “Nov”, “Dec”}	*
`dayOfWeek`	[0, 6] или {“Sun”, “Mon”, “Tue”, “Wed”, “Thu”, “Fri”, “Sat”}	*
`year`	Четырехзначный год	*

Следует отметить, что по умолчанию таймеры используют настройки часового пояса сервера. Чтобы задать другой часовой пояс, следует указать атрибут `timezone` со значением из IANA Time Zone Database. Допустимые значения хранятся в поле Zone Name. Строковое представление можно также получить с помощью класса `java.util.TimeZone`. А теперь перейдем к практике – попробуем задействовать декларативные таймеры в приложении ActionBazaar.

7.2.4. Пример использования декларативных таймеров

Исследуем возможности декларативного календарного таймера на примере реализации рассылки ежемесячных и праздничных информационных бюллетеней. В ежемесячном бюллетене будут перечисляться наиболее активные торги с описанием лотов и продавцов. Праздничные бюллетени будут рассылаться по большим праздникам и памятным датам, таким как «Черная пятница». Обе задачи будут решены с применением декларативных календарных таймеров.

С этой целью необходимо добавить в приложение ActionBazaar новый сеансовый компонент без сохранения состояния `Newsletter`. Схематическая структура компонента представлена в листинге 7.2. Класс `Newsletter` будет использовать механизм хранения данных контейнера. Так как запросить список пользователей можно, только обладая привилегиями администратора, чтобы не возникали исключения, вызванные нарушением прав доступа, компонент `Newsletter` будет вызываться с правами администратора. Помните, что методы, вызываемые таймером, выполняются с привилегиями неаутентифицированного пользователя.

Важно также учесть вероятность ситуации, когда контейнер может потерпеть аварию в то время, как компонент `Newsletter` занимается рассылкой бюллетеня. Контейнер попытается повторно вызвать метод после перезапуска контейнера. Список рассылки может оказаться очень длинным, а служба поддержки электронной почты не использует транзакции. Чтобы обойти это ограничение, можно воспользоваться механизмом JMS. В процессе обхода списка рассылки метод может добавлять в очередь новое сообщение JMS для каждого пользователя. Если по какой-то причине метод потерпит неудачу, электронные письма не будут отправлены и произойдет откат транзакции. Кроме того, при таком подходе нагрузка на электронную почту может быть сбалансирована созданием нескольких экземпляров компонента, управляемого сообщениями, обрабатывающими сообщения из очереди. Код, реализующий такое поведение, не показан в листинге, но его можно найти в загружаемых примерах к книге. Здесь демонстрируется, как можно использовать различные аспекты EJB для создания надежной службы.

Листинг 7.2. Рассылка бюллетеней и напоминаний

```
@RunAs("Admin") // Метод будет вызываться с привилегиями роли Admin
@Stateless
public class NewsletterBean {
```

```
  @PersistenceContext
  private EntityManager em;

  // Рассылка бюллетеня будет выполняться в полночь
  // первого числа каждого месяца
  @Schedule(second="0", minute="0", hour="0", dayOfMonth="1",
    month="*", year="*")
  public void sendMonthlyNewsletter() {
    ...
  }

  // Аннотация @Schedules может создавать несколько таймеров
  @Schedules({
    // В полдень последнего четверга ноября
    @Schedule(second="0", minute="0", hour="12",
      dayOfMonth="Last Thu", month="Nov", year="*"),
    // В полдень 18 декабря
    @Schedule(second="0", minute="0", hour="12",
      dayOfMonth="18", month="Dec", year="*")
  })
  // Метод отмечен, как требующий запуска новой транзакции
  @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
  public void sendHolidayNewsletter() {
    ...
  }
  ...
}
```

В этом листинге класс `NewsletterBean` объявлен, как действующий с привилегиями администратора. Это позволяет компоненту извлекать список пользователей, доступ к которому ограничен по соображениям безопасности. Метод `sendMonthlyNewsletter` отмечен аннотацией `@Schedule`, указывающей, что метод должен вызываться первого числа каждого месяца. Метод `sendHolidayNewsletter` отмечен аннотацией `@Schedules`, настраивающей его вызов в День благодарения и на Рождество.

Теперь, когда вы получили представление о декларативных таймерах, перейдем к знакомству с синтаксисом правил, чтобы посмотреть, какие возможности у нас имеются. Пример в листинге 7.2 – это лишь вершина айсберга.

7.2.5. Синтаксис правил в стиле cron

Синтаксис определения правил для календарных таймеров не особенно сложен. В табл. 7.1 перечислены атрибуты, допустимые значения и значения по умолчанию. Но эта таблица не дает полного представления об имеющихся возможностях. Вы наверняка обратили внимание, что в атрибутах `second`, `minute` и `hour` аннотации `@Schedule` передаются строковые, а не целочисленные значения. Это обусловлено тем, что в атрибутах можно передавать более сложные значения. Например, можно создать таймер, срабатывающий только с понедельника по пятницу каждую вторую неделю. Если бы в атрибутах допускалось передавать только единственные значения, определить подобное расписание было бы невозможно.

Атрибуты поддерживают следующие синтаксические формы: единственное значение, групповой символ, список, диапазон и приращение.

Единичные значения

Единичные значения являются наиболее простыми. В табл. 7.1 перечисляются допустимые значения, которые можно указывать в атрибутах. Значения нечувствительны к регистру символов. Для некоторых атрибутов существует два представления – числовое и текстовое – как в случае с днями недели, где понедельник может быть представлен как `"Mon"` или `"0"`. Ниже приводится несколько примеров единичных значений в атрибутах:

```
@Schedule(second="0", minute="1", hour="23", dayOfMonth="1", month="Apr",
    dayOfWeek="Mon", year="2015")
```

Это выражение определяет таймер, который сработает в 11:01:00, первого апреля и в первый понедельник апреля, в 2015 году. У кого-то это выражение может вызвать замешательство, из-за того, что в нем используются атрибуты `dayOfMonth` и `dayOfWeek`. Дело в том, что эти два атрибута объединяются по ИЛИ, тогда как все остальные – по И. То есть, таймер, созданный с данным выражением, сработает дважды: 1 апреля 2015 и 6 апреля 2015.

Групповой символ

Групповой символ `*` соответствует всем допустимым значениям для данного атрибута. Если вам доводилось пользоваться командой `ls` в Unix или `dir` в Windows, вы должны понимать, о чем идет речь. Групповой символ используется в ситуациях, когда необходимо указать в атрибуте, что он совпадает со всеми возможными значениями, но не хочется перечислять полный список этих значений. Он поддерживается всеми атрибутами, но обычно его не рекомендуют применять к атрибутам `second` и `minute`. Запуск операций каждую секунду или минуту может отрицательно сказаться на производительности системы. Слишком часто срабатывающие таймеры ухудшают масштабируемость приложения и его отзывчивость. Ниже приводится пример использования группового символа:

```
@Schedule(second="*", minute="*", hour="*", dayOfMonth="*", month="*",
    dayOfWeek="*", year="*")
```

Это выражение определяет таймер, который будет срабатывать каждую секунду, каждой минуты, каждого часа, каждого дня, каждого месяца, каждого года. Этот пример не особенно полезен – он просто демонстрирует, что все атрибуты поддерживают групповой символ.

Список

Список состоит из единичных значений, разделенных запятыми. Список может содержать повторяющиеся значения – они просто игнорируются. Пробельные символы между значениями и запятыми также игнорируются. Список не может содержать групповой символ, вложенные списки или приращения. Но список мо-

жет содержать диапазоны. Например, можно указать диапазоны с понедельника по вторник и с четверга по пятницу. Ниже приводится пример использования списков:

```
@Schedule(second="0,29", minute="0,14,29,59", hour="0,5,11,17",
    dayOfMonth="1,15-31",year="2011,2012,2013")
```

Это выражение определяет таймер, который будет срабатывать в 2011, 2012 и 2013 годах, первого числа, а так же с 15 по 31 число, через каждые 15 минут и 30 секунд в течение каждого шестого часа. То есть, таймер сработает в 0:0:0 1 января 2011, в 0:0:29 1 января 2011, и так далее. Обратите внимание, что атрибут `dayOfWeek` был опущен, чтобы упростить и без того сложный пример.

Диапазон

Диапазоны уже демонстрировались в примере со списками. Они позволяют задать множество значений, включая граничные. Диапазон определяется как пара единичных значений, разделенных дефисом (-), где значение, предшествующее дефису, является начальным значением диапазона, а значение, следующее за дефисом, – конечным. Диапазоны не могут содержать групповые символы, списки или приращения. Ниже приводится пример использования диапазонов:

```
@Schedule(hour="0-11",dayOfMonth="15-31",dayOfWeek="Mon-Fri",month="11-12",
    year="2010-2020")
```

Это выражение определяет таймер, который будет срабатывать в 2011–2020 годах, с ноября по декабрь, с понедельника по пятницу в последнюю половину месяца, с полуночи до полудня. Диапазоны можно было бы указать еще и в атрибутах `minute`, `second` и `dayOfWeek`, но это слишком усложнило бы пример.

Приращения

Приращения позволяют создавать таймеры, срабатывающие в течение фиксированных интервалов. Приращение определяется как пара единичных значений, разделенных символом косой черты, где первое значение задает начало интервала, а второе – протяженность интервала. Приращения могут указываться только в атрибутах `second`, `minute` и `hour`. Они продолжают действовать до достижения границы в единице измерения уровнем выше – если приращение указано в атрибуте `second`, приращение завершится по достижении следующей минуты. То же относится к минутам и часам. Групповые символы могут указываться перед косой чертой, но не после нее. Взгляните на простой пример:

```
@Schedule(second="30/10", minute="*/20", hour="*/6")
```

Это выражение определяет таймер, который будет срабатывать каждые 10 секунд после задержки в 30 секунд, через каждые 20 минут, каждый шестой час. То есть, если создать таймер в полночь, он сработает в 06:20:40, 6:40:30, 12:20:40, 12:40:30, и так далее. Другие атрибуты отсутствуют в примере, потому что они не поддерживают приращения.

7.3. Программные таймеры

В предыдущем разделе мы познакомились с декларативными таймерами и аннотациями `@Schedule` и `@Schedules`, с помощью которых они создаются. Декларативные таймеры удобно использовать, когда планируемое задание и его расписание известны заранее. Их с успехом можно применять, например, для ротации файлов журналов, пакетной обработки записей в нерабочие часы и для решения других прикладных задач. Специализированные таймеры, напротив, используются для выполнения операций, которые неизвестны заранее, и создаются динамически, во время выполнения, по мере необходимости. Если вернуться к примеру с рассылкой в приложении ActionBazaar, можно отметить, что жестко определенное расписание рассылки страдает отсутствием гибкости. Так как приложение ActionBazaar используется в разных странах, необходима возможность определять дополнительные праздничные и памятные дни для рассылки. Очевидно, что пересборка приложения для добавления дополнительных праздничных дней, это уже перебор. Более удачное решение – использовать специализированные таймеры и позволить маркетинговым администраторам планировать рассылки во время работы приложения.

Итак, специальные таймеры создаются программно. Функционально они очень близки к декларативным таймерам. Единственное отличие лишь в том, что специальные таймеры создаются программно, во время выполнения, и каждый компонент может иметь не больше одного метода, вызываемого таймером, потому что только один метод можно отметить аннотацией `@Timeout`. Как будет показано далее, существует несколько методов на выбор, с помощью которых можно создавать программные таймеры.

7.3.1. Знакомство с программными таймерами

Специальные таймеры создаются посредством интерфейса `javax.ejb.TimerService`. Объект `TimerService` может быть внедрен с помощью аннотации `@Resource` или получен обращением к `javax.ejb.SessionContext`. Интерфейс `TimerService` предоставляет множество разновидностей метода `createTimer`, упрощающих создание таймеров в разных ситуациях. С помощью `TimerService` можно создавать таймеры двух типов: интервальные таймеры (или таймеры задержки) и календарные. Календарные таймеры уже рассматривались выше. Интервальные таймеры появились в версии EJB 2.1 и поддерживаются до сих пор.

В листинге 7.3 приводится определение интерфейса `TimerService`. Кроме метода `getTimers()`, все остальные отвечают за создание таймеров. Каждый метод возвращает объект `javax.ejb.Timer`. Объект `Timer` можно использовать для получения информации о таймере, такой как возможность сохранения, оставшееся время до срабатывания, а также для остановки таймера. Метод `getHandle()` возвращает `javax.ejb.TimerHandle`, поддерживающий сериализацию. Объект `javax.ejb.Timer` не может быть сериализован и потому не может храниться в переменной-члене сеансового компонента с сохранением состояния, в сущности

JPA и так далее. Объект `javax.ejb.TimerHandle` имеет метод, позволяющий получить текущий экземпляр `Timer`.

Листинг 7.3. Интерфейс TimerService

```
public interface TimerService {

  // ❶ Методы создания календарных таймеров
  Timer createCalendarTimer(ScheduleExpression schedule)
    throws IllegalArgumentException, IllegalStateException,
    EJBException;
  Timer createCalendarTimer(ScheduleExpression schedule,
    TimerConfig timerConfig) throws IllegalArgumentException,
    IllegalStateException, EJBException;

  // ❷ Методы создания интервальных таймеров EJB 2.1
  Timer createIntervalTimer( Date initialExpiration,
    long intervalDuration, TimerConfig timerConfig ) throws
    IllegalArgumentException, IllegalStateException, EJBException;
  Timer createIntervalTimer( long initialDuration,
    long intervalDuration, TimerConfig timerConfig ) throws
    IllegalArgumentException, IllegalStateException,
    EJBException;
  Timer createTimer( long duration, Serializable info )
    throws IllegalArgumentException, IllegalStateException,
    EJBException;
  Timer createTimer( long initialDuration, long intervalDuration,
    Serializable info ) throws IllegalArgumentException,
    IllegalStateException, EJBException;
  Timer createTimer( Date expiration, Serializable info )
    throws IllegalArgumentException, IllegalStateException,
    EJBException;
  Timer createTimer( Date initialExpiration, long intervalDuration,
    Serializable info ) throws IllegalArgumentException,
    IllegalStateException, EJBException;

  // ❸ Методы создания таймеров однократного действия
  Timer createSingleActionTimer( Date expiration,
    TimerConfig timerConfig ) throws IllegalArgumentException,
    IllegalStateException, EJBException;
  Timer createSingleActionTimer(long duration, TimerConfig timerConfig)
    throws IllegalArgumentException, IllegalStateException,
    EJBException;
  Collection<Timer> getTimers() throws IllegalStateException, EJBException;
}
```

Методы создания календарных таймеров ❶ принимают `javax.ejb.ScheduleExpression` и необязательный объект `javax.ejb.TimerConfig`. Объект выражения `ScheduleExpression` определяет, когда должен сработать таймер, и имеет методы для настройки даты, времени и других параметров. Второй блок ❷ содержит методы создания интервальных таймеров, которые срабатывают через фиксированный интервал времени и могут срабатывать многократно. В третьем

блоке ❸ содержатся методы создания таймеров однократного действия, которые срабатывают единожды, через указанный интервал времени. Многие методы дополнительно принимают ссылку `info` на объект, в которой может также передаваться значение `null`, если использовать этот объект не нужно. Единственное требование, предъявляемое к этому объекту, – он должен реализовать интерфейс `java.io.Serializable`; а касательно всего остального – решать вам. Объект `info`, это контейнер для информации, передаваемой таймеру, – любой информации по вашему выбору.

Не упомянутый выше метод `getTimers()` возвращает коллекцию всех таймеров. Вы можете выполнять итерации по этому списку, останавливать таймеры и определять, когда в следующий раз они должны сработать. Но имейте в виду, что к моменту, когда объект таймера будет извлечен из `TimerService`, он может уже сработать – будьте осторожны, не ставьте выполнение программы в зависимость от возможности найти и остановить таймер до того, как он сработает.

Листинг 7.3 ссылается на два дополнительных класса: `javax.ejb.ScheduleExpression` и `javax.ejb.TimerConfig`. Класс `ScheduleExpression` используется для определения времени срабатывания таймера. Он имеет конструктор без аргументов и методы для установки дня недели, месяца, года, часов, секунд и так далее. Каждый метод возвращает один и тот же экземпляр `ScheduleExpression`, что позволяет составлять цепочки из вызовов методов и выполнять настройки в одной строке кода. Объект `TimerConfig` – это контейнер для хранения дополнительной информации с настройками таймера. Он включает объект `info`, который передается методу, вызываемому таймером, и флаг, указывающий – сможет ли таймер пережить перезапуск сервера.

При организации планирования с помощью `TimerService`, компонент должен иметь специальный метод для вызова таймером, так как таймер создается для текущего компонента, куда внедряется `TimerService` – нельзя установить таймер для другого компонента. Для этого один из методов компонента должен быть отмечен аннотацией `@TimeOut` или сам компонент должен реализовать интерфейс `TimedObject`. Это будет продемонстрировано в примере ниже.

Таймеры, созданные с помощью `TimerService`, участвуют в транзакциях. Когда новый таймер создается в контексте транзакции, таймер останавливается в случае отката транзакции. Метод, вызываемый таймером, может выполняться в контексте транзакции. Если транзакция потерпит неудачу, контейнер позаботится об отмене всех изменений, выполненных методом, и повторно вызовет его. Теперь, когда вы получили первое представление о специальных таймерах, рассмотрим пример из приложения ActionBazaar.

7.3.2. Пример использования программных таймеров

Коммерческому отделу ActionBazaar часто требуется рассылать короткие рекламные сообщения, чтобы привлечь потенциальных покупателей и продавцов зайти на сайт. Выбор времени рассылки играет важную роль в этом случае. Если сообще-

ние придет в конце рабочего дня, получатель наверняка забудет о нем, когда вернется домой. Если сообщение придет перед длинными праздниками или выходными, оно окажется похороненным в папке входящих сообщений. Как результат, создание и утверждение рекламного сообщения не обязательно должно совпадать с временем, когда его следовало бы отправить. При выгрузке рекламного сообщения ActionBazaar запрашивает дату и время рассылки. Это чем-то напоминает предыдущий пример рассылки информационных бюллетеней, за исключением того, что рекламная рассылка не выполняется с такой же регулярностью. Информация с расписанием рассылки подготавливается сотрудником коммерческого отдела и упаковывается в объект `ScheduleExpression`, а затем вызывается метод `scheduleFlyer` компонента `FlyerBean`. Определение этого компонента представлено в листинге 7.4.

Листинг 7.4. Реализация FlyerBean

```
@Stateless
public class FlyerBean {

  private static final Logger logger = Logger.getLogger("FlyerBean");

  // ❶ Внедрить службу таймеров
  @Resource
  private TimerService timerService;

  public List<Timer> getScheduledFlyers() {
    // ❷ Получить список таймеров для этого компонента
    Collection<Timer> timers = timerService.getTimers();
    return new ArrayList<Timer>(timers);
  }

  public void scheduleFlyer(ScheduleExpression se, Email email) {
    // ❸ Создать новый объект TimerConfig
    //   для включения сообщения электронной почты
    TimerConfig tc = new TimerConfig(email,true);
    // ❹ Запланировать рассылку
    Timer timer = timerService.createCalendarTimer(se,tc);
    // ❺ Зафиксировать в журнале время расссылки
    logger.info("Flyer will be sent at: "+ timer.getNextTimeout());
  }

  // ❻ Аннотировать метод для вызова таймером
  @Timeout
  public void send(Timer timer) {
    if(timer.getInfo() instanceof Email) {
      // ❼ Извлечь рекламное сообщение
      Email email = (Email)timer.getInfo();
      // Извлечь списки покупателей/продавцов и выполнить рассылку
    }
  }
}
```

Компонент FlyerBean, представленный в этом листинге, отвечает за планирование рассылки рекламных сообщений и осуществляет рассылку при срабатывании таймера. Объект TimerService внедряется посредством аннотации ❶. Метод getScheduledFlyers предназначен для отображения расписания на веб-странице, чтобы сотрудники коммерческого отдела могли видеть, рассылка каких рекламных сообщений, и на какое время запланирована ❷. Метод scheduleFlyer осуществляет планирование рассылки. Пользовательский интерфейс создает экземпляр ScheduleExpression, который затем передается интерфейсу TimerService. Экземпляр TimerConfig создается с целью сохранения рекламного сообщения, а также, чтобы указать, что таймер должен восстанавливаться при перезапуске сервера ❸. Объект таймера Timer создается обращением к службе TimerService, которой передается расписание ScheduleExpression и настройки TimerConfig ❹. После создания таймера из него можно извлечь информацию о моменте срабатывания ❺. Метод send отмечен аннотацией @TimeOut ❻, которая указывает, что данный метод будет вызываться таймером.

Этот пример достаточно прост. Служба TimerService создает таймеры, которые связываются с текущим компонентом. Метод для вызова таймера либо отмечается аннотацией @TimeOut, либо является реализацией интерфейса javax.ejb.TimedObject.

Однако в листинге отсутствует один важный фрагмент кода, который конструирует объект ScheduleExpression. Объект ScheduleExpression, как упоминалось выше, поддерживает возможность составления цепочек из вызовов методов, способствуя уменьшению числа строк кода. Ниже показано, как сконструировать объект расписания для организации рассылки рекламы в день Святого Валентина непосредственно перед обедом:

```
ScheduleExpression se = new ScheduleExpression();
se.month(2).dayOfMonth(14).year(2012).hour(11).minute(30); // 2/14/2012 @
    11:30
```

Итак, программные таймеры относительно просты в использовании. Чтобы создать такой таймер, необходимо внедрить службу таймеров Timer Service в компонент, отметить метод для вызова таймером аннотацией javax.ejb.TimeOut и запланировать его вызов с помощью одного из методов объекта javax.ejb.TimerService. Что может быть проще!

7.3.3. Эффективное использование программных таймеров EJB

Программные таймеры дают возможность планировать операции прямо во время выполнения программы. Они отлично подходят для реализации расписаний, составляемых пользователем, как в примере с рекламной рассылкой, так как для таких операций не существует заранее известного расписания. Еще одним примером применения программных таймеров в ActionBazaar могло бы служить завершение торгов для лотов. В этом случае с помощью программного таймера можно было

бы завершать торги для каждого лота и отсылать победителю электронное письмо с поздравлениями. В момент создания первой ставки можно создавать таймер, запланированный на срабатывание в момент завершения торгов. Как видите, программные таймеры универсальны и, в отличие от декларативных таймеров, могут создаваться и настраиваться во время выполнения, не требуя изменять конфигурационный файл или перекомпилировать приложение. Декларативные таймеры, в свою очередь, с успехом могут применяться для выполнения операций, известных на стадии разработки, таких как удаление неактивных учетных записей раз в месяц.

При использовании программных таймеров важно подумать о том, как они будут сохраняться контейнером приложения. Повторное развертывание приложения может уничтожить имеющиеся таймеры. Такое поведение может быть нежелательным, если, к примеру, вы просто остановили приложение, чтобы исправить ошибку. Поэтому очень важно понимать, как обеспечить сохранение таймеров в моменты перезапуска сервера. Что должно происходить при переносе приложения на другой сервер? Разработчики серверов приложений уже решили многие из этих проблем, поэтому вам остается только исследовать свой контейнер и точно определить, как должно вести себя приложение в разных ситуациях, помимо простого перезапуска сервера.

Вдобавок к проблемам, связанным с перезапуском сервера, следует так же уделить внимание проблеме злоупотреблений таймерами. Важно внимательно подойти к вопросу дробления задач. При создании новой учетной записи в приложении ActionBazaar пользователь указывает адрес электронной почты, который необходимо проверить. Если адрес не подтверждается в течение 24 часов, учетную запись следует удалить – нельзя допускать переполнения системы большим числом неактивных учетных записей. Не так важно, когда именно произойдет удаление неподтвержденной учетной записи – точно через 24 часа, через 24 часа 30 минут или через 25 часов. Поэтому было бы явным излишеством создавать программный таймер для каждой новой учетной записи. Гораздо практичнее определить декларативный таймер, срабатывающий каждый час и удаляющий учетные записи, которые не были активированы в последние 24 часа. Удаление множества учетных записей можно выполнять в контексте одной транзакции и сервер не будет перегружен слежением за десятками таймеров.

7.4. В заключение

В этой главе мы познакомились с возможностями механизма планирования в EJB. Версия EJB 3.1 сделала большой шаг вперед в области планирования, в сравнении с EJB 3. В версии 3.1 появился cron-подобный планировщик. В версиях, предшествующих EJB 3.1, для более или менее сложного планирования приходилось пользоваться сторонними решениями, такими как Quartz. Однако, «прикручивание» сторонних библиотек чревато усложнением кода и влечет дополнительные юридические проблемы, в зависимости от приемлемости лицензии. Поэтому появление новых возможностей планирования является большим улучшением.

Как было показано в этой главе, средства планирования в EJB можно разделить на декларативные и программные. Декларативное планирование осуществляется путем размещения аннотаций перед методами сеансовых компонентов без сохранения состояния, определяющих когда и как часто они должны вызываться. Программное планирование осуществляется во время выполнения, посредством объекта `javax.ejb.TimerService`. Оба подхода, программный и декларативный, поддерживают не только интервальные таймеры, перекочевавшие из предыдущих версий EJB, но и календарные. Выбор типа таймера зависит от решаемой им задачи и зависит от того, известно ли расписание на момент разработки или определяется во время выполнения. В следующей главе мы займемся исследованием веб-служб.

ГЛАВА 8. Компоненты EJB как веб-службы

Эта глава охватывает следующие темы:

- основы веб-служб;
- основы SOAP (JAX-WS);
- экспортирование компонентов EJB в виде веб-служб SOAP;
- основы REST (JAX-RS)
- экспортирование компонентов EJB в виде веб-служб REST.

В этой главе мы займемся исследованием возможности экспортирования компонентов EJB в виде веб-служб SOAP и REST. Веб-службы давно превратились в промышленный стандарт связи между приложениями (Application-To-Application, A2A) и электронной коммерции (Business-To-Business, B2B). Они составляют основу разработки программного обеспечения в соответствии с принципами сервис-ориентированной архитектуры (Service-oriented Architecture, SOA), согласно которому функциональность приложений экспортируется в виде слабосвязанных служб.

Как веб-службы связаны с EJB 3? Вы можете рассматривать их как способ экспортирования сеансовых компонентов без сохранения состояния, открывающий доступ к ним для клиентов, написанных на Java и других языках программирования, действующих на самых разных платформах. Веб-службы можно считать альтернативным подходом к экспортированию компонентов, не требующим использования механизмов RMI или Java. Клиентом может быть приложение для iPhone на Objective-C, для .NET, выполняющееся на сервере, или веб-приложение на PHP и JavaScript – возможности практически не ограничены. Ваши компоненты EJB вместе с прикладными и веб-службами составляют технологический стек с универсальными методами, которые могут вызываться из любых приложений на любых языках и выполняющихся на любых платформах.

В этой главе предполагается, что вы уже знакомы с веб-службами, поэтому мы не будем углубляться в принципы их разработки. Этой теме посвящены целые книги и было бы невозможно охватить ее целиком в единственной главе. Детальное обсуждение технологий REST и SOAP вы сможете найти в книгах «Restlet in Action» Джерома Лувеля (Jerome Louvel) (Manning, 2012) и «SOA Patterns» Арнона Ротем-Гал-Оза (Arnon Rotem-Gal-Oz) (Manning, 2012). Для начала в этой главе мы познакомимся с основами веб-служб SOAP и REST, а затем попробуем применить вновь полученные знания для экспортирования сеансовых компонентов EJB без сохранения состояния. Попутно мы внедрим поддержку веб-служб в приложение ActionBazaar и обсудим наиболее эффективные приемы использования веб-служб.

8.1. Что такое «веб-служба»?

Очень непросто дать некое универсальное определение веб-служб, с которым бы согласились все без исключения. Тем не менее, говоря простым языком, *веб-служба* – это стандартная платформа, обеспечивающая возможность взаимодействий между сетевыми приложениями. Большинство разработчиков под этим понимают обмен сообщениями в формате XML по протоколу HTTP/HTTPS. И формат XML и протокол HTTP/HTTPS определяются стандартами и обеспечивают возможность обслуживания самых разных клиентов, основанных на разных технологиях. Например, веб-службой, созданной с использованием Java EE, смогут пользоваться клиенты, написанные на C#, Python, C++, Objective-C и любом другом языке, как показано на рис. 8.1.

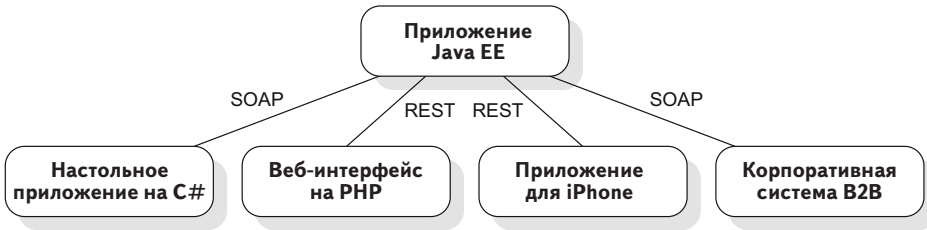

Рис. 8.1. Веб-службы обеспечивают возможность взаимодействий между сетевыми приложениями, такими как приложение на основе Java EE и веб-интерфейс на PHP

8.1.1. Свойства веб-служб

Когда клиент обращается к веб-службе, в конечном счете происходит вызов метода, который обрабатывает запрос и возвращает ответ. Если рассуждать в терминах методов Java, параметрами запроса могут быть примитивы Java или объекты. То же справедливо и для ответа. Как следует из названия, веб-служба – это, в первую очередь, служба, выполняющая определенные операции, такие как размещение заказа, проверка состояния заказа, отмена заказа и так далее. Обычно веб-службы не используются для создания аналогов методов доступа, таких как `setName()`,

`setPhoneNumber()` и других. Это слишком мелкое деление на задачи, к тому же веб-службы не хранят информацию о состоянии. Поясним подробнее последнее утверждение: веб-службы – это службы, а не объекты. Веб-службы не поддерживают понятие сеанса, потому два последовательных обращения к веб-службе рассматриваются как совершенно независимые. Этим они отличаются от технологии Java RMI, где имеется прокси-объект, являющийся представлением удаленного объекта, способный вызывать удаленные методы. Помимо отсутствия поддержки информации о состоянии, вызовы веб-служб выполняются синхронно – клиент должен дождаться, пока веб-служба обработает запрос и вернет ответ.

8.1.2. Транспорты

Обычно для обмена сообщениями веб-службы используют протокол HTTP или HTTPS из-за их вездесущности. Использование широко известных портов и протоколов упрощает развертывание веб-служб, так как маршрутизаторы и брандмауэры уже настроены на обслуживание этого трафика и нет необходимости открывать дополнительные порты, что в некоторых окружениях невозможно. Это также упрощает и разработку, потому что основой для веб-служб могут служить уже имеющиеся веб-серверы. Веб-службы, особенно веб-службы на основе технологии SOAP, могут также использовать другие транспорты, такие как SMTP и FTP. Однако мы не будем рассматривать их, так как их изучение далеко выходит за рамки данной главы.

8.1.3. Типы веб-служб

Двумя самыми заметными стандартами реализации веб-служб являются SOAP и REST. В соответствии со стандартом SOAP, сообщения определяются посредством языка описания веб-служб (Web Services Description Language, WSDL) и кодируются в формат XML. Описание на языке WSDL определяет структуру входящих запросов и исходящих ответов. В соответствии с этим описанием специализированные инструменты могут генерировать код, вызывающий веб-службу и обрабатывающий ответ. Такие инструменты существуют практически для всех языков и платформ. Например, вам потребуется организовать взаимодействие с веб-службой из приложения .NET, можно воспользоваться инструментами и библиотеками, созданными в корпорации Microsoft, которые делают разработку клиентов довольно простым делом. Приложение wsdl.exe генерирует клиентский программный код на языке C# на основе WSDL-файла. Разработчику не нужно писать код, обрабатывающий XML-запрос или генерирующий ответ. Аналогичные инструменты существуют и для других языков, включая C++, Python, PHP и так далее.

В технологии передачи репрезентативного состояния (Representational State Transfer, REST), как и SOAP, используется формат XML и протокол HTTP. Но веб-службы REST используют совершенно иной, значительно более простой подход, чем веб-службы SOAP. Основная идея технологии REST заключается в назначении уникального идентификатора URL каждой веб-службе. Параметры

веб-службе передаются точно так же, как параметры форм. Кроме того, каждая веб-служба отображается в один из методов протокола HTTP: `GET`, `PUT`, `DELETE`, `POST`, `HEAD` и `OPTIONS`. Эти операции документированы в RFC 2616. Веб-служба, возвращающая значение и не вносящая никаких изменений в состояние веб-приложения, должна быть реализована как операция `GET`. Метод `PUT` используется для определения операций, сохраняющих данные на сервере, такие как ставка в приложении ActionBazaar.

В конце главы мы сравним технологии SOAP и REST, и представим рекомендации по выбору между ними.

8.1.4. Java EE API для веб-служб

В этой главе мы погрузимся в изучение тонкостей экспортирования компонентов EJB в виде веб-служб SOAP и RESTful. Просматривая спецификацию Java EE или статьи в Интернете, вы неизбежно столкнетесь с такими аббревиатурами, как JAX-WS (JSR 224) и JAX-RS (JSR 399), которые в Java EE 7 определяют стандартные API для разработки веб-служб на основе технологий SOAP и RESTful, соответственно. Существует множество разных реализаций этих спецификаций. Однако, так как стандарт Java EE 7 явно требует поддержки JAX-WS и JAX-RS, вам не придется беспокоиться по поводу выбора между реализациями, если только вы не решите воспользоваться сторонними решениями. В числе доступных реализаций можно назвать, например, Metro (http://metro.java.net/), включенную в состав GlassFish, и Apache Axis 2 (http://axis.apache.org/).

В документации с описанием Java EE можно также столкнуться с аббревиатурой JAX-RPC. Технология JAX-RPC – это предшественница JAX-WS, поддерживавшаяся в J2EE 1.4. JAX-RPC –устаревшая технология и ее следует избегать в новых приложениях. Технология JAX-WS намного проще в использовании и обладает более богатыми возможностями. Сложность JAX-RPC создала веб-службам на Java плохую репутацию.

В дополнение к только что упомянутым технологиям JAX-WS и JAX-RS, мы также затронем JAXB – средство связывания данных XML для Java (Java Architecture for XML Binding). Как следует из названия, этот набор инструментов применяется для преобразования объектов Java в формат XML и обратно. С помощью инструментария JAXB можно сгенерировать модель данных из схемы XML и затем сохранить ее обратно в XML, или аннотировать программный код. Последний прием мы будем рассматривать в этой главе. Мы не можем дать полный охват всех возможностей в рамках этой главы, но мы расскажем достаточно, чтобы вы получили представление об имеющихся возможностях.

В Java EE 7 впервые появилась встроенная поддержка JSON (JSR 353). JSON – формат записи объектов JavaScript (JavaScript Object Notation), обеспечивающий компактное представление данных. Мы будем использовать формат JSON во время знакомства с веб-службами RESTful, которые часто используют его для формирования ответов. Этот формат особенно удобен, если вы конструируете собственный интерактивный веб-сайт с большим объемом программного кода на

JavaScript. Формат XML слишком многословен, потребляет существенную ширину канала и требует значительных вычислительных ресурсов, как на стороне сервера, так и на стороне клиента. Для интерактивных веб-приложений на основе технологии AJAX формат JSON выглядит намного привлекательнее.

8.1.5. Веб-службы и JSF

Возможно вы приступили к чтению этой главы, надеясь освоить создание веб-служб, чтобы позднее применить полученные знания в конструировании пользовательского интерфейса, использующего приемы AJAX и самих веб-служб. Технология JSF предоставляет встроенную поддержку AJAX и обеспечивает передачу информации между клиентом и сервером. В JSF имеются собственные функции на JavaScript и серверный код поддержки, упрощающие разработку веб-страниц с применением технологии AJAX. Однако мы не будем подробно обсуждать JSF, так как это далеко выходит за рамки данной книги. А теперь, после знакомства с основными понятиями веб-служб, перейдем к более глубокому исследованию вопросов создания традиционных веб-служб на основе SOAP с использованием JAX-WS.

8.2. Экспортирование компонентов EJB с использованием SOAP (JAX-WS)

Начиная с Java EE 7, стандартом веб-служб SOAP на платформе Java EE считается спецификация JAX-WS 2.2. Разработка веб-служб на основе SOAP с использованием JAX-WS не представляет никаких сложностей. Создать и развернуть такую веб-службу с помощью аннотаций можно за минуты. Поддержка веб-служб SOAP на платформе Java прошла длинный путь с первых дней своего существования. Но, прежде чем перейти к коду, мы коротко расскажем об их основах.

8.2.1. Основы SOAP

SOAP – это протокол распределенных взаимодействий, сродни COBRA и Java RMI. Он позволяет приложениям общаться друг с другом, обмениваясь сообщениями посредством сетевых протоколов, из которых чаще всего используются HTTP/HTTPS. Обмен осуществляется сообщениями в формате XML. Структура сообщений определяется с помощью языка WSDL и схемы XML. Использование протокола HTTP в качестве транспорта для передачи сообщений XML привело к широкому распространению SOAP для организации связи между совершенно разными системами, написанными на разных языках и действующими на разных платформах. Например, с помощью SOAP можно обращаться к компонентам EJB из приложений на Python, C#, Objective-C и других языках. Кроме того, так как протокол SOAP имеет четкое определение, то с обеих сторон – на стороне сервера и на стороне клиента – можно использовать инструменты, автоматически генерирующие программный код для обработки сообщений. Одним словом, веб-службы SOAP пользуются достаточно большой популярностью.

Первоначально протокол SOAP было разработан в корпорации Microsoft в 1998 году. В настоящее время развитием этой спецификации занимается консорциум W3C. На момент написания этих строк текущей была спецификация SOAP 1.2. Не менее важными, чем спецификация SOAP, являются профили WS-I. WS-I (www.ws-i.org) – это промышленный консорциум, определяющий стандарты взаимодействий для веб-служб. Даже при том, что протокол SOAP является нейтральным к выбору языка программирования, многие разработчики реализаций SOAP по-разному интерпретируют положения спецификации, из-за чего возникают проблемы. Консорциум WS-I выпустил профили и примеры приложений, обеспечивающие ясные и недвусмысленные рекомендации по обеспечению настоящей совместимости в разработке веб-служб. Ниже перечислены профили, уместные в контексте этой главы:

- *базовый профиль WS-I* – определяет суженный набор действительных служб для SOAP, WSDL и UDDI для поддержания совместимости;
- *базовый профиль безопасности WS-I* – руководство по определению безопасных и совместимых веб-служб;
- *простой профиль связывания SOAP* – дополнительный профиль для базового профиля WS-I; связывает операции с конкретным транспортным протоколом SOAP.

Основные реализации JAX-WS, в том числе Metro, JBossWS (используется сервером приложений GlassFish) и Apache Axis, являются совместимыми с профилями WS-I. Совместимость с профилями WS-I играет важную роль в создании веб-служб, которые могут использоваться приложениями, написанными для самых разных платформ. Теперь, когда вы познакомились с краткой историей SOAP и некоторых сопутствующих спецификаций, посмотрим, как выглядит настоящее сообщение SOAP и исследуем его структуру.

Структура сообщения SOAP

Сообщения SOAP имеют относительно простую структуру, как показано на рис. 8.2. Внешний элемент – это «конверт», содержащий заголовок и тело сообщения. Заголовок может отсутствовать, а его содержимое никак не связано с содержимым тела сообщения. Заголовки обычно обрабатываются самим веб-сервером. Тело содержит собственно информацию, передаваемую в сообщении – данные в формате XML, предназначенные для обработки веб-службой.

Рис. 8.2. Структура сообщения SOAP

В листинге 8.1 приводится пример сообщения SOAP из приложения ActionBazaar. В этом сообщении отсутствует заголовок. В сообщение включена информация для протокола HTTP – как можно видеть, это сообщение должно быть доставлено службе по адресу `/BidService/MakeBid`.

Листинг 8.1. Сообщение SOAP для создания ставки в приложении ActionBazaar

```
<!-- HTTP-заголовок, который обрабатывается веб-сервером и реализацией SOAP -->
POST /BidService/MakeBid HTTP/1.1
Content-type: text/xml;charset="utf-8"
Soapaction: ""
Accept: text/xml,multipart/related, text/html, image/gif, image/jpeg,
  *; q=.2, */*;q=.2
User-Agen: JAX-WS RI 2.1.6 in JDK 6
Host: localhost:8090
Connection: keep-alive
Content-Length: 230

<!-- XML-сообщение SOAP, обрабатываемое веб-службой -->
<S:Envelop xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <bid xmlns="http://com.actionbazaar/bid">
      <bidPrice></bidPrice>
      <itemId></itemId>
      <bidderId></bidderId>
    </bid>
  </S:Body>
</S:Envelop>
```

Разновидности веб-служб

Существует две основные разновидности веб-служб: RPC-ориентированные[1] и документо-ориентированные. Несмотря на то, что название «RPC-ориентированные» предполагает вызов процедур, здесь не имеется в виду модель программирования. В действительности суть заключается в структурировании XML-сообщений. На ранних этапах веб-службы RPC приобрели немалую популярность, но позднее маятник качнулся в сторону документо-ориентированных веб-служб.

Фактически название «документо-ориентированные» означает, что в обмене со службами участвуют документы. Эту разновидность веб-служб следует использовать, если предполагается обмен документами в формате XML, имеющими стандартный формат или формат, разработанный у вас на предприятии. При этом вы можете структурировать сообщения как угодно. Название «RPC», напротив, предполагает вызов метода, соответственно данные в формате XML будут содержать имя метода и набор параметров. Чтобы почувствовать разницу, поэкспериментируйте с аннотацией `SOAPBinding`, о которой рассказывается далее в этой главе, и исследуйте файл WSDL, генерируемый автоматически механизмом JAX-WS.

Независимо от разновидности веб-службы, у вас на выбор имеется два варианта представления данных: литеральное и кодированное представление. Для документо-ориентированной службы предпочтительнее использовать литеральное представление, а для RPC-ориентированной – кодированное. Преимущество литеральных документов становится особенно очевидным, когда предполагается,

[1] Remote Procedure Call – вызов удаленных процедур. – *Прим. перев.*

что веб-служба будет использоваться клиентами, написанными на разных языках и действующими на разных платформах, например .NET. Не используйте кодированную форму представления документов – она практически никем не используется и не поддерживается.

Структура WSDL

Файл на языке WSDL составляет основу любой веб-службы SOAP, потому что содержит полное ее описание. В нем может присутствовать автоматически сгенерированный код для сервера и клиента, осуществляющий трансформацию данных (маршалинг) и фактическую их передачу. Структура файла WSDL показана на рис. 8.3, а пример документа WSDL для службы BidService можно найти в листинге 8.4.

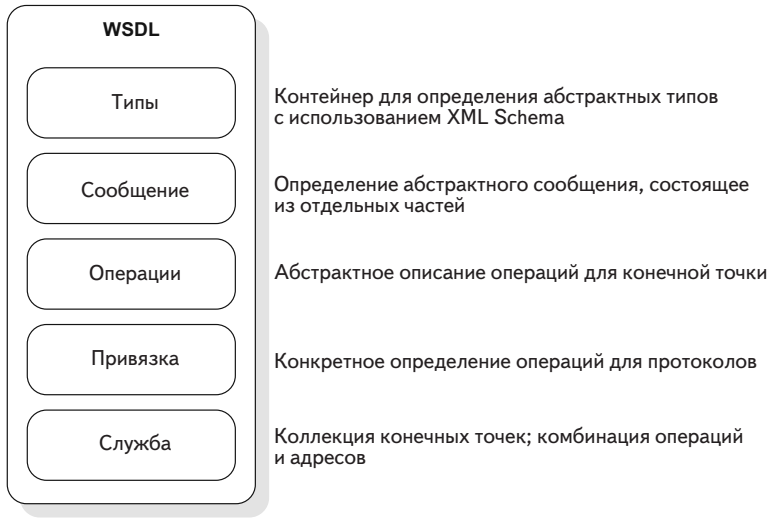

Рис. 8.3. Структура документа WSDL

В разделе «Типы», определяются типы данных, составляющих сообщения. Они определяются с применением языка XML Schema. В разделе «Сообщение» определяются сообщения, поддерживаемые службой. Каждое сообщение состоит из одной или нескольких частей, каждая из которых может описывать входные или входные данные. В разделе «Операции» определяются группы операций (абстрактный интерфейс). Каждая операция должна иметь уникальное имя и включать комбинацию входных и выходных элементов, и ссылки на элементы сообщений. Раздел «Привязка» связывает операции (интерфейсы) с фактическими протоколами, такими как SOAP. Раздел «Служба» описывает привязки как службы и определяет конечные точки.

Не волнуйтесь, если что-то из вышесказанного осталось для вас непонятным – после того, как вы напишете пару-тройку документов WSDL, многое для вас станет проще и понятнее.

Стратегии веб-служб

В процессе создания новой веб-службы необходимо написать два артефакта: файл WSDL с определением службы и класс Java с ее реализацией. Если предполагается обмениваться данными со сложной структурой, также необходимо создать файл схемы с определением данных и сослаться на него из определения WSDL. Имея файл WSDL и файл схемы, с помощью специализированных инструментов можно сгенерировать код для работы с веб-службой. Например, утилита `wsimport` генерирует по заданному файлу WSDL весь программный код на языке Java, необходимый для вызова веб-службы. Аналогичные инструменты существуют также для других языков программирования и платформ – создание клиентов веб-служб SOAP является достаточно стандартной задачей.

Если сосредоточиться на двух основных артефактах – WSDL и реализации на Java – в вашем распоряжении имеется несколько разных подходов. Для каждой службы может быть избран свой подход – они не являются взаимоисключающими в рамках проекта. Имеющиеся подходы изображены на рис. 8.4.

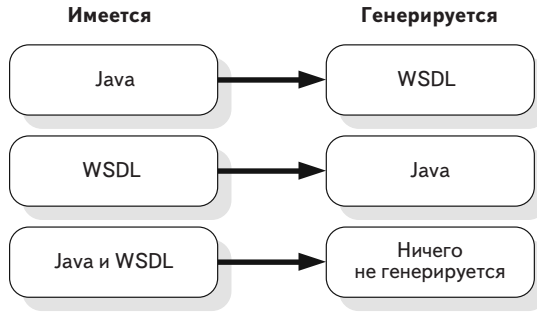

Рис. 8.4. Подходы к созданию веб-служб

Когда имеется реализация на Java, на ее основе с помощью инструментов JAX-WS можно сгенерировать файл WSDL. Этот подход убережет вас от необходимости вникать в тонкости WSDL и схем XML. Просто отметьте класс аннотацией `@WebService`, добавьте аннотации `@WebMethod` перед требуемыми методами и на свет появится новая веб-служба. Однако, несмотря на практическую целесообразность, этот подход имеет несколько недостатков. Во-первых, сгенерированный код может неожиданно оказаться зависимым от некоторых особенностей языка Java, таких как Java Generics. Если параметрами метода являются объекты, то фреймворк привязки данных вынужден будет сгенерировать схему. Сгенерированная схема может оказаться оптимальной, но несовместимой с другими языками.

Если в качестве отправной точки избрать файл с описанием на языке WSDL, тогда с помощью встроенных инструментов Java можно сгенерировать реализацию службы для серверной стороны. Такой подход несколько сложнее, так как требует полного знания языка WSDL. С помощью инструмента `wsimport` можно сгенерировать реализацию службы для серверной стороны (впрочем, он способен

также генерировать код для стороны клиента). В результате вы получите интерфейсы, и вам останется лишь написать классы, реализующие их. После того, как интерфейсы будут сгенерированы, вам потребуется обеспечить синхронизацию этих интерфейсов и реализации. Кроме того, при использовании сложных объектов в качестве входных параметров и возвращаемых значений, сгенерированный код может оказаться неоптимальным.

Третий подход состоит в том, чтобы вручную написать и файл WSDL, и соответствующий код на языке Java. Этот подход позволяет объединить все самое лучшее из двух миров: точное описание WSDL и ясный код Java. При этом устраняются проблемы, которыми страдают инструменты генерации кода – непереносимое описание WSDL и неоптимальный код на Java. Однако это самый сложный путь, так как он требует обеспечить безупречное соответствие между WSDL и кодом на Java. Для реализации служб, которые, как предполагается, должны обслуживать клиентов, написанных на разных языках и выполняющихся на разных платформах, это определенно самый лучший выбор.

Все три решения требуют использования фреймворка связывания данных. В большинстве окружений, по умолчанию роль такого фреймворка играет JAXB (JSR 222). Он отвечает за преобразование данных в формате XML в объекты Java, которые затем будут передаваться службе. Например, служба для работы со ставками имеет метод `placeBid`. Этот метод принимает в качестве параметра объект `Bid`.

Мы еще вернемся к этим подходам, когда будем обсуждать приемы эффективного использования. Но прежде попробуем ответить на вопрос: когда следует использовать службы SOAP?

8.2.2. Когда следует использовать службы SOAP

В этом разделе мы попробуем дать ответ на вопрос: «Когда следует использовать веб-службы SOAP?». Это неоднозначный вопрос и разными людьми может восприниматься по-разному. Например, для кого он может звучать так: «Следует ли отдать предпочтение веб-службам SOAP перед другими технологиями, такими как Java RMI или RESTful?». Кто-то может воспринимать его как вопрос о том, следует ли вообще предоставлять прикладные службы сторонним системам посредством технологий, подобных SOAP. Однако второй вопрос слишком широк для данной главы. Мы не будем рассматривать его и рекомендуем обратиться к другим источникам, посвященным проектированию сервис-ориентированных архитектур (Service-Oriented Architectures, SOA) и более детально обсуждающим веб-службы SOAP, таким как «SOA Governance in Action» Джоса Дирксена (Jos Dirksen) (Manning, 2012). Давайте сосредоточимся на первом вопросе.

В сравнении с подходами к реализации прикладной функциональности в других системах, стек Java EE 7 предоставляет выбор из нескольких совершенно разных технологий, включая SOAP. Экспортировать прикладные службы можно также посредством веб-служб Java RMI, JMS и RESTful. Каждая из этих технологий имеет свои достоинства и компромиссы, в зависимости от ваших требований.

Самой первой технологией экспортирования прикладных служб на языке Java была Java Remote Method Invocation (RMI). Она сделала возможным появление EJB и по умолчанию поддерживается для всех компонентов. Java RMI оптимально подходит для ситуаций, когда клиентами являются приложения на Java. Технология RMI менее требовательна к вычислительным ресурсам в сравнении с SOAP. Но она непригодна, когда необходимо обеспечить доступность служб для приложений, написанных на других языках, так как не является кроссплатформенной. SOAP – это кроссплатформенное решение, которое поддерживает возможность взаимодействий с клиентами, написанными на C#, C++, COBOL, Python и многих других языках.

Веб-службы можно также экспортировать посредством JMS. Многие разработчики едва ли заметят сходство JMS и SOAP, тем не менее, обе технологии основаны на обмене сообщениями. Механизмы обмена сообщениями имеют длинную историю развития, намного более длинную, чем веб-службы. JMS реализует асинхронную передачу сообщений с хранением в промежуточной очереди, тогда как SOAP предполагает синхронные взаимодействия. Технология SOAP имеет четко очерченный интерфейс, благодаря чему есть возможность использовать инструменты, автоматического создания клиентского программного кода. Такое немыслимо при использовании JMS, зато эта технология лучше подходит для интеграции двух систем, когда не требуется, чтобы одна система предоставляла внешний интерфейс другой системе.

Следует также отметить, что веб-службы на основе SOAP, в отличие от служб на основе Java RMI или JMS, легко доступны через корпоративные брандмауэры. Сообщения SOAP обычно передаются с использованием вездесущих протоколов HTTP/HTTPS. А теперь давайте поговорим о том, когда следует открывать доступ к компонентам EJB с помощью технологии SOAP.

8.2.3. Когда следует экспортировать компоненты EJB в виде веб-служб SOAP

Механизм JAX-WS позволяет экспортировать в виде веб-служб обычные классы Java (POJO), а также сеансовые компоненты EJB без сохранения состояния или компоненты-одиночки. Если сравнить исходный код веб-служб на основе POJO и веб-служб на основе компонентов EJB, вы не найдете существенных отличий. Веб-службы на основе компонентов EJB будут включать дополнительную аннотацию `@Stateless/@Singleton` и, возможно, другие аннотации или особенности EJB, с которыми мы уже знакомы, такие как использование службы таймеров или транзакции. Обе разновидности веб-служб – на основе POJO и EJB – поддерживают внедрение зависимостей и события жизненного цикла, такие как `@PostConstruct` и `@PreDestroy`, но, экспортируя компоненты EJB, вы получаете ряд важных преимуществ.

Во-первых, веб-службы на основе компонентов EJB автоматически получают возможности декларативного управления транзакциями и безопасностью, доступные только для компонентов EJB. Эти две крупные особенности дают компонентам EJB огромную фору перед обычными объектами POJO. Кроме того, без каких-

либо дополнительных усилий вы можете использовать интерцепторы и службу таймеров.

Во-вторых, и самое важное, веб-службы на основе компонентов EJB дают возможность без лишних усилий и дублирования кода экспортировать прикладную логику. Экспортирование веб-службы EJB ничем не отличается от экспортирования компонента EJB посредством RMI – достаточно просто заменить аннотацию `@Remote` на аннотацию `@WebService`. Кроме того, компоненты EJB можно одновременно экспортировать с применением обеих технологий. Экспортирование прикладной логики никогда не была таким простым делом.

В табл. 8.1 сравниваются веб-службы на основе EJB и на основе POJO. Здесь вы увидите несколько существенных преимуществ EJB перед POJO.

Таблица 8.1. Сравнение обычных веб-служб Java с веб-службами на основе EJB

Особенность	Веб-служба POJO	Веб-служба EJB
POJO	Да	Да
Внедрение зависимостей, включая ресурсы, единица хранения и так далее	Да	Да
Поддержка событий жизненного цикла	Да	Да
Декларативное управление транзакциями	Нет	Да
Декларативное управление безопасностью	Нет	Да

Теперь, после знакомства с основами веб-служб SOAP и обсуждения вопроса, когда их следует использовать, посмотрим, как используются веб-службы в приложении ActionBazaar.

8.2.4. Веб-служба SOAP для ActionBazaar

Рассмотрим два примера, демонстрирующих порядок экспортирования компонентов EJB в виде веб-служб. С их помощью мы покажем два разных подхода к созданию веб-служб. Первый пример демонстрирует возможность автоматического создания файла WSDL контейнером. А во втором примере мы расскажем, как написать файл WSDL и файл схемы JAXB, как на их основе сгенерировать компоненты и как написать код, реализующий полученные интерфейсы. Вы можете использовать эти примеры как отправные точки для своих исследований. Второй пример определенно представляет больший интерес, так как демонстрирует большее количество деталей.

Служба PlaceBid

В первом примере мы экспортируем в виде веб-службы компонент `PlaceBid`. С ее помощью клиенты, написанные на других языках программирования, смогут отправлять ставки. Данный компонент EJB будет экспортироваться как RPC-кодированная веб-служба SOAP. Файл WSDL будет генерироваться во время выполнения автоматически, с помощью механизма JAX-RS.

В листинге 8.2 приводится код веб-службы PlaceBid. Он состоит из двух частей: интерфейса веб-службы и сеансового компонента без сохранения состояния.

Листинг 8.2. PlaceBid

```
@WebService // ❶ Экспортирует веб-службу
@SOAPBinding(style=SOAPBinding.Style.RPC, // ❷ Определяет параметры
    use=SOAPBinding.Use.ENCODED)          //   связывания
public interface PlaceBidWS {
  public long submitBid(long userId, long itemId, double bidPrice);
}

@Stateless
// ❸ Определяет конечную точку
@WebService(endpointInterface = "com.actionbazaar.ws.PlaceBidWS")
public class PlaceBid implements PlaceBidWS{

  @Override
  // ❹ Этот метод экспортируется
  public long submitBid(long userId, long itemId, double bidPrice) {
    ...
  }

  // ❺ Этот метод не экспортируется
  public List<Bid> getBids() {
    ...
  }
}
```

Аннотация @WebService ❶ в этом листинге определяет интерфейс экспортируемой веб-службы. Вслед за ней определяются параметры связывания конечной точки с протоколом SOAP: с применением кодирования RPC ❷. Реализация класса тоже отмечена аннотацией @WebService, со ссылкой endpointInterface, указывающей на интерфейс веб-службы ❸; компонент также реализует этот интерфейс. Далее следует реализация экспортируемого метода ❹. Обратите внимание, что метод getBids ❺ не экспортируется, потому что он отсутствует в определении интерфейса.

Если предположить, что приложение ActionBazaar развернуто в GlassFish, тогда сгенерированный файл WSDL будет доступен по адресу: http://localhost:8080/PlaceBidService/PlaceBid?wsdl. С помощью утилиты wsimport, входящей в комплект инструментов Java, можно сгенерировать клиентский код для программного доступа к этой службе. В следующем примере мы рассмотрим новые аннотации более подробно.

Служба UserService

В этом примере мы экспортируем новый метод createUser сеансового компонента UserService. Но на этот раз мы начнем не с объектов Java, а с файлов WSDL и XSD.

Существующий метод `createUser` компонента `UserService` принимает объект `com.actionbazaar.account.User`. Это абстрактный объект. Он наследует классы `Bidder`, `Employee` и `SellerI`, а также ссылается на другие объекты. Инструменты автоматического создания кода для такого сложного объекта могут добавить в описание WSDL нежелательные зависимости, которые сделают его несовместимым с другими языками программирования.

Чтобы избежать этих проблем, новый метод `createUser` компонента `UserService` будет использовать объект передачи данных (Data Transfer Object, DTO). Объект передачи данных – это упрощенная версия прикладного объекта, созданная специально для передачи данных. Так как такие объекты передачи данных не имеют сложной иерархии наследования, инструментам автоматизации проще справляться с ними. В данном случае мы начнем с создания схемы XSD с определением объекта `UserDTO`, которая показана в листинге 8.3.

Листинг 8.3. UserDTO.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
  version="1.0"
  targetNamespace="http://com.actionbazaar/user"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://com.actionbazaar/user"
  elementFormDefault="qualified">
  <xs:element name="user" type="tns:UserDTO"/>
  <xs:element name="createUserResponse" type="tns:CreateUserResponse"/>
  <xs:complexType name="UserDTO">
    <xs:sequence>
      <xs:element id="firstName" name="firstName" type="xs:string"
        minOccurs="1" maxOccurs="1" nillable="false"/>
      <xs:element id="lastName" name="lastName" type="xs:string"
        minOccurs="1" maxOccurs="1" nillable="false"/>
      <xs:element id="birthDate" name="birthDate" type="xs:date"
        minOccurs="1" maxOccurs="1" nillable="false"/>
      <xs:element id="username" name="username" type="xs:string"
        minOccurs="1" maxOccurs="1" nillable="false"/>
      <xs:element id="password" name="password" type="xs:string"
        minOccurs="1" maxOccurs="1" nillable="false"/>
      <xs:element id="email" name="email" type="xs:string"
        minOccurs="1" maxOccurs="1" nillable="false"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="CreateUserResponse">
    <xs:sequence>
      <xs:element name="userId" type="xs:integer"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Кроме описания объекта `UserDTO` схема включает также описание объект `CreateUserResponse`. Этот объект используется для передачи идентификатора пользователя обратно, клиентскому коду, чтобы его можно было использовать при последующих обращениях к службе.

Теперь, когда у нас имеется готовая схема, можно воспользоваться генератором JAXB, входящего в библиотеку JDK, и с его помощью превратить схему в настоящие объекты Java:

```
xjc -d <базовый_каталог>/ActionBazaar-ejb/src/main/java/ -p
     com.actionbazaar.ws.dto UserDTO.xsd
```

После создания объектов Java можно перейти к формированию файла WSDL для веб-службы. Наш файл WSDL представлен в листинге 8.4. Его следует сохранить в каталоге META-INF/wsdl. Так как описание WSDL содержит ссылки на файл *UserDTO.xsd*, последний следует сохранить в том же каталоге. Файл *UserService.wsdl* определяет одну службу, `createUser`, которую еще предстоит реализовать.

Листинг 8.4. UserService.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:tns="http://com.actionbazaar/user"
xmlns="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://com.actionbazaar/user"
name="UserService" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <types>
    <xsd:schema>
      <!-- Импортирует схему с определением объектов -->
      <xsd:import namespace="http://com.actionbazaar/user"
        schemaLocation="UserDTO.xsd"/>
    </xsd:schema>
  </types>

  <message name="createUserRequest">
    <!-- UserDTO -->
    <part name="user" element="tns:user"/>
  </message>
  <message name="createUserResponse">
    <!-- ResponseDTO -->
    <part name="response" element="tns:createUserResponse"/>
  </message>
  <portType name="CreateUser">
    <operation name="createUser">
      <input message="tns:createUserRequest"/>
      <output message="tns:createUserResponse"/>
    </operation>
  </portType>
  <binding name="UserPortBinding" type="tns:CreateUser">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
      style="document"/>
    <operation name="createUser">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
```

```
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <!-- Экспортирует UserService -->
  <service name="UserService">
    <port name="UserPort" binding="tns:UserPortBinding">
    <soap:address
      location="http://localhost:8080/UserService/CreateUser"/>
    </port>
  </service>
</definitions>
```

Описание WSDL, показанное в этом листинге, выглядит достаточно простым. Краткое описание отдельных элементов WSDL вы найдете выше в этой главе. На основе этого документа WSDL можно автоматически сгенерировать код для использования на стороне клиента. В листинге 8.5 приводится реализация веб-службы, соответствующая документу WSDL из листинга 8.4.

Листинг 8.5. Интерфейс и реализация UserService

```
@WebService(
  name="CreateUser",
  serviceName="UserService",
  targetNamespace="http://com.actionbazaar/user",
  portName = "UserPort",
  wsdlLocation= "UserService.wsdl")
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT)
public interface UserServiceWS {
  @WebMethod(operationName="createUser")
  @SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
    use=SOAPBinding.Use.LITERAL,
    parameterStyle = SOAPBinding.ParameterStyle.BARE)
  public @WebResult(name="response",
    targetNamespace="http://com.actionbazaar/user")
  CreateUserResponse
  createUser(
    @WebParam(name="user", mode= WebParam.Mode.IN,
      targetNamespace="http://com.actionbazaar/user")
    UserDTO user);
  }

@Stateless(name="UserService")
@WebService(endpointInterface = "com.actionbazaar.ws.UserServiceWS")
public class UserServiceBean implements UserService, UserServiceWS {
  ...
  @Override
  public CreateUserResponse createUser(UserDTO user) {
    ...
  }
}
```

Код в этом листинге реализует веб-службу, соответствующую документу WSDL из листинга 8.4. Теперь реализация получилась намного сложнее, чем в первом примере. Нам потребовалось с помощью аннотаций отобразить параметр и возвращаемое значение в сущности, которые определены в схеме. Кроме того, нам потребовалось отобразить метод в соответствующий элемент WSDL. Данный пример станет более понятным, когда мы поближе познакомимся с аннотациями.

Генерирование клиентского кода

Сгенерировать клиентский код на Java для взаимодействия с этой службой можно с помощью команды `wsimport`. Эта команда генерирует все классы, описанные в файле *UserDTO.xsd*, наряду с заготовками методов, вызывая которые можно обращаться к веб-службе. Благодаря этой особенности, команду `wsimport` удобно использовать для быстрого тестирования создаваемых служб.

8.2.5. Аннотации JAX-WS

Теперь, после знакомства с простым примером реализации веб-службы на основе SOAP в приложении ActionBazaar, мы займемся более детальным изучением аннотаций. Обсуждению SOAP и JAX-WS можно посвятить целую книгу. Этот раздел может претендовать лишь на роль краткого введения. SOAP – это комплексный набор технологий и спецификаций.

Чтобы создать и запустить веб-службу SOAPнеобходимо добавить, как минимум, следующие две аннотации:

- `@javax.jws.WebService` – отмечает интерфейс или класс веб-службы;
- `@javax.jws.soap.SOAPBinding` – определяет разновидность (документ/RPC) и необходимость кодирования (документ, кодированный/литеральный).

С помощью этих двух аннотаций можно создавать простейшие веб-службы SOAP. Рассмотрим каждую аннотацию в отдельности.

Аннотация @WebService

Аннотация `@WebService` применяется к классам или интерфейсам компонентов. Когда она применяется к классу компонента, контейнер EJB автоматически генерирует интерфейс. Если интерфейс уже имеется, аннотацию `@WebService` можно применить к интерфейсу и класс компонента получит следующие методы:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface WebService {
  String name() default "";
  String targetNamespace() default "";
```

```
  String serviceName() default "";
  String portName() default "";
  String wsdlLocation() default "";
  String endpointInterface() default "";
}
```

Аннотация @WebMethod

Аннотация `@javax.jws.WebMethod` применяется к методам, которые должны экспортироваться как веб-службы. По умолчанию экспортируются все методы класса, поэтому данную аннотацию можно использовать, чтобы сделать метод недоступным или изменить свойства экспортируемого метода, такие как имя операции или действие SOAP. Ниже приводится полное определение аннотации:

```
@ Retention(RetentionPolicy.RUNTIME)
@ Target({ElementType.METHOD})
public @interface WebMethod {
  String operationName() default "";
  String action() default "";
  boolean exclude() default false;
}
```

Параметры `operationName` и `action` аннотации `@WebMethod` определяют имя операции и действие SOAP, соответственно. Следующий пример демонстрирует их использование:

```
@WebMethod(operationName = "addNewBid",
           action = http://actionbazaar.com/NewBid)
public Long addBid(…) {
  ...
}
```

Параметр `operationName` отражается на сгенерированном описании WSDL, как показано ниже:

```
<portType name="PlaceBidBean">
    <operation name = "addNewBid">
    ...
    </operation>
</portType>
```

Обратите внимание, что метод с фактическим именем `addBid` экспортируется в веб-службе под именем `addNewBid`. Этот прием можно использовать для отображения контракта службы в фактическую реализацию. Даже если с течением времени реализация изменится, контракт останется неизменным. Если параметр `operationName` не указан, по умолчанию его значением становится фактическое имя метода в реализации. Этот же прием может пригодиться в ситуациях, когда файл WSDL создается отдельно и по каким-то причинам невозможно экспортировать метод под его фактическим именем.

Аналогично параметр `action` используется для создания элемента `SOAPAction` в файле WSDL:

```
<operation name ="addNewBid">
    <soap:operation soapAction = "http://actionbazaar.com/NewBid"/>
</operation>
```

Элемент `SOAPAction` определяет содержимое элемента `header` в HTTP-запросе с сообщением. Клиенты используют его при взаимодействии с веб-службой SOAP по протоколу HTTP. По значению поля `SOAPAction` заголовка конечная точка определяет истинного получателя сообщения, что избавляет ее от необходимости выуживать эту информацию из тела сообщения SOAP. Теперь, после знакомства с аннотацией `@WebMethod`, перейдем к аннотации `@WebParam`, связанной с ней естественными узами.

Аннотация @WebParam

Аннотация `@javax.jws.WebParam` применяется с целью настройки параметра сообщения в генерируемом документе WSDL. Этой аннотацией отмечаются аргументы экспортируемых методов. Ниже приводится полное определение аннотации:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.PARAMETER})
public @interface WebParam {
  String name() default "";
  String partName() default "";
  String targetNamespace() default "";
  Mode mode() default Mode.IN;
  boolean header() default false;
  static final enum Mode {
    public static final IN,
    public static final OUT,
    public static final INOUT;
  }
}
```

Следующий фрагмент демонстрирует порядок использования аннотации, где видно, что параметры можно использовать для передачи данных в веб-службу, возврата результатов из нее или для того и другого одновременно:

```
@WebMethod
public Long addBid(
    @WebParam(name="user", mode= WebParam.Mode.IN) String userId) {
  ...
}
```

Свойство `name` определяет имя параметра для сообщения в WSDL. Если имя не указано, по умолчанию используется фактическое имя аргумента.

Свойство `targetNamespace` определяет пространство имен XML для сообщения. Если свойство `targetNamespace` не указано, сервер будет использовать пространство имен веб-службы.

Свойство `mode` определяет направление параметра и может принимать одно из трех значений: `IN`, `OUT` и `INOUT`. Оно указывает направление передачи данных: в

параметре типа IN передаются входные данные для службы, в параметре типа OUT возвращаются результаты и в параметре типа INOUT данные передаются в обоих направлениях. Параметр типа INOUT действует подобно указателю в C++, передача которого дает методу возможность вернуть более чем один результат. Если в свойстве mode указано значение OUT или INOUT, сам параметр должен иметь тип javax.xml.ws.Holder<T>. Обсуждение этого типа далеко выходит за рамки этой книги, поэтому за дополнительной информацией обращайтесь к спецификации JAX-WS.

Свойство header определяет, откуда должны извлекаться параметры – из тела сообщения или из заголовка. Значение true в этом свойстве означает, что параметры должны извлекаться из заголовка. Эта настройка применяется в ситуациях, когда сообщение SOAP подвергается промежуточной обработке перед передачей, такой как маршрутизация между серверами. Промежуточные обработчики исследуют только содержимое заголовка. Значение true в свойстве header генерирует следующий код WSDL:

```
<operation name = "addNewBid">
  <soap:operation soapAction="urn:NewBid"/>
  <input>
    <soap:header message="tns:PlaceBid_addNewBid" part="user" use="literal"/>
    <soap:body use="literal" parts="parameters"/>
  </input>
</operation>
```

Свойство partName управляет созданием элемента name в wsdl:part или элементом схемы XML параметра, если для веб-службы выбран тип связывания RPC или, если выбран тип связывания document с параметром BARE. Если свойство name не указано для типа RPC и указано свойство partName, в качестве имени элемента будет использоваться значение partName.

Аннотация @WebResult

Аннотация @WebResult очень похожа на аннотацию @WebParam. Она применяется совместно с аннотацией @WebMethod и определяет имя значения, возвращаемого в ответ на сообщение, в WSDL, как показано ниже:

```
@WebMethod
@WebResult(name="bidNumber"
public Long addBid(...) {}
```

Определение аннотации @WebResult практически полностью повторяет определение аннотации @WebParam, кроме свойства mode:

```
@Retention(value = RetentionPolicy.RUNTIME)
@Target(value = {ElementType.METHOD})
public @interface WebResult {
  public String name() default "";
  public String partName() default "";
  public String targetNamespace() default "";
```

```
  public boolean header() default false;
}
```

Как можно догадаться, свойство `name` определяет имя возвращаемого значения в WSDL. Свойство `targetNamespace` определяет пространство имен XML для возвращаемого значения. Эти свойства имеют смысл для документо-ориентированных веб-служб, где возвращаемые значения связываются с определенным пространством имен XML.

Если свойство `header` установлено в значение `true`, возвращаемое значение будет включено в заголовок. Как и в аннотации `@WebParam`, свойство `partName` используется, чтобы определить имя возвращаемого значения.

Аннотации @OneWay и @HandlerChain

В спецификации метаданных веб-служб определены еще две аннотации: `@OneWay` и `@HandlerChain`. Мы дадим здесь лишь краткое описание, так как подробное обсуждение этих аннотаций далеко выходит за рамки книги.

Аннотация `@OneWay` используется для определения веб-служб, не имеющих возвращаемого значения. То есть, когда метод имеет тип `void`. Например:

```
@WebMethod
@OneWay
public void pingServer() {
  ...
}
```

В данном случае метод `pingServer` ничего не возвращает. Аннотация `@OneWay` оптимизирует сообщение, отражая факт отсутствия возвращаемого значения.

Аннотация `@HandlerChain` используется для определения набора обработчиков, которые должны вызываться в ответ на сообщение SOAP. С логической точки зрения, обработчики похожи на интерцепторы EJB. Обработчики могут быть двух типов:

- логические обработчики (`@javax.xml.ws.handler.LogicalHandler`), оперирующие свойствами контекста сообщения и данными в теле сообщения;
- обработчики протокола (`@javax.xml.ws.handler.soap.SOAPHandler`), оперирующие свойствами контекста сообщения и сообщениями, характерными для протокола.

Теперь, после знакомства с основными аннотациями JAX-WS, рассмотрим наиболее эффективные приемы использования SOAP в EJB.

8.2.6. Эффективное использование веб-служб SOAP в EJB

Комбинация JAX-WS и EJB существенно упрощает экспортирование прикладной логики в виде веб-служб. После прочтения этого раздела у многих может появиться желание экспортировать все компоненты в виде веб-служб SOAP. Но не

спешите, на этом пути вас поджидает большое число ловушек, которых следует остерегаться. Так же как разбрасывание `synchronize` по всему коду приложения не сделает его безопасным для выполнения в многопоточной среде, декорирование классов аннотацией `@WebService` не принесет мгновенного успеха. Давайте посмотрим, с какой стороны лучше подходить к EJB и SOAP.

Сначала необходимо определить, имеет ли смысл открывать доступ к компонентам для внешних систем, а затем выяснить, действительно ли взаимодействие с внешними системами может быть организовано только с помощью веб-служб. Если нет, вероятно, лучшим выбором будет JMS или Java RMI. Эти технологии имеют более низкие накладные расходы – и в смысле сетевого трафика, и в смысле вычислительных ресурсов. Формат XML слишком дорог для передачи и обработки. Если предполагается предоставлять услуги третьим сторонам, тогда SOAP является хорошим выбором. Но если услуги предоставляются внутренним Java-системам, RMI и JMS выглядят предпочтительнее.

Если вы решите экспортировать услуги посредством SOAP, экспортируйте только нужные методы. При наличии разветвленной иерархии объектов вам, вероятно, потребуется организовать обмен объектами DTO. Также желательно ограничить объем данных, передаваемых через веб-службы – едва ли кому-то захочется, чтобы механизму JAXB всякий раз приходилось преобразовывать большую часть базы данных в документ XML. Например, если говорить о приложении ActionBazaar, веб-служба, обрабатывающая запрос на получение информации о лоте, не должна вместе со списком ставок извлекать имена пользователей и их пароли.

Проанализируйте, какая разновидность веб-служб лучше подходит для вашей ситуации – RPC-ориентированная или документо-ориентированная. Выбрать ту или иную разновидность можно с помощью аннотации `@SOAPBinding`. RPC-ориентированные службы обычно имеют более высокую производительность. Но документо-ориентированные веб-службы обладают большей гибкостью, благодаря возможности использовать схемы XML для определения сообщений. Аналогично, избегайте использования кодированных сообщений, потому что это ухудшает совместимость служб и переносимость между разными реализациями SOAP. Для универсальных и переносимых служб рекомендуется использовать комбинацию параметров `document/literal`.

Проектируйте веб-службы так, чтобы они генерировали как можно меньший объем сетевого трафика. Избегайте передачи больших объектов. Вместо целого объекта лучше отправить его идентификатор или ссылку. Все объекты, передаваемые по протоколу SOAP, сериализуются в формат XML. Представление данных в формате XML может достигать огромных размеров, делая передачу объектов более дорогостоящей операцией, чем извлечение тех же объектов на месте. Кроме того, избегайте экспортирования компонентов EJB, выполняющих продолжительные операции в контексте транзакций, или отмечайте их, как не имеющие возвращаемых значений, чтобы к ним можно было обращаться асинхронно.

Используйте для параметров методов типы данных JAX-WS, чтобы улучшить совместимость с разнородными веб-службами. Представьте, что получится, если параметрами веб-службы будут объекты с иерархией наследования, включающей

`Collections`, `HashMaps` и `Lists`. Использование таких типов данных в WSDL ухудшит совместимость вашего приложения. Если совместимость играет важную роль, проверьте свое приложение на соответствие базовому профилю WS-I.

Существует множество механизмов, которые помогут вам повысить безопасность ваших веб-служб. Требования к безопасности должны перевешивать требования к производительности, потому что безопасность ценится гораздо выше. Ценность безопасности данных обычно оказывается выше, чем видится изначально. Это утверждение справедливо и для систем/приложений в целом, но для веб-служб, взаимодействующих с заранее неизвестными приложениями, оно приобретает особую значимость. Теперь, после знакомства с JAX-WS, можно переходить к обсуждению создания веб-служб RESTful с использованием JAX-RS.

8.3. Экспортирование компонентов EJB с использованием REST (JAX-RS)

В первой половине главы мы исследовали веб-службы SOAP. Во второй половине мы займемся изучением веб-служб RESTful. В последние несколько лет наблюдался взрывоподобный рост популярности веб-служб RESTful, и тому есть веские причины. Подход RESTful создает ощущение «возврата к основам». Технология SOAP развивается уже более десяти лет и давно переросла рамки простого и стандартного подхода к организации обмена сообщениями в формате XML. Формально аббревиатура SOAP расшифровывается, как Simple Object Access Protocol (простой протокол доступа к объектам), но в настоящее время эта технология официально называется SOAP, а полная расшифровка была убрана из спецификаций, так как ее уже никак нельзя назвать простой.

Концепция веб-служб RESTful впервые была описана в диссертации доктора Роя Филдинга (Roy Fielding) с названием «Architectural Styles and the Design of Network-based Software Architectures» («Архитектурные стили и проектирование архитектур сетевых приложений»). В этой диссертации доктор Филдинг исследовал истоки появления архитектуры Веб и сформулировал ряд архитектурных принципов, которые объединил под общим названием «Передача репрезентативного состояния» (Representational State Transfer, REST). Эти принципы появились как результат работы над спецификациями HTTP 1.0 и 1.1, в которой он принимал участие как один из ведущих авторов. В отличие от SOAP, REST не является стандартом.

Веб-службы RESTful целиком основаны на протоколе HTTP и используют унифицированные идентификаторы ресурсов (Uniform Resource Identifiers, URI), не нашедшие широкого применения в SOAP. В технологии SOAP преобладающее использование протокола HTTP в качестве транспортного механизма обусловлено лишь его распространенностью и простотой проникновения через брандмауэры. Тогда как веб-службы RESTful основаны на использовании базовых операций HTTP – `DELETE`, `GET`, `HEAD`, `OPTIONS`, `POST` и `PUT` – и URI. Таким образом, основу CRUD-функциональности[2] составляют операции HTTP. Телом сообщений для

[2] Create, Read, Update, Delete (создать, прочитать, изменить, удалить). – *Прим. перев.*

этих служб может быть все, что угодно – XML, графика, текст и так далее. В отличие от технологии SOAP, где операция определяется содержимым сообщения, для обращения к веб-службам RESTful используются уникальные адреса URL. Параметры передаются в виде запросов. Все это в технологии SOAP не используется или используется недостаточно широко.

Прикладной интерфейс для создания веб-служб RESTful на платформе Java EE определяет JAX-RS. Механизм JAX-RS никак не зависит от EJB и может использоваться для экспортирования в виде веб-служб RESTful даже простых Java-объектов (POJO). Но, как уже было не раз показано выше, EJB предлагает множество дополнительных возможностей, включая интеграцию с транзакциями. Экспортирование компонентов EJB в виде веб-служб REST концептуально не отличается от экспортирования тех же компонентов в виде веб-служб SOAP. А теперь перейдем к основам REST.

8.3.1. Основы REST

В качестве одного из транспортных механизмов для веб-служб SOAP используется протокол HTTP. Посредством запросов HTTP POST и GET на сервер выгружается блок XML, где выполняется его обработка. Идентификатор URI запроса определяет службу, но фактически выполняемая операция и возвращаемый клиенту результат определяются содержимым в формате XML. Просто глядя на URI запроса, не всегда можно сказать, какая операция запрашивается и будут ли изменены данные на сервере. Для взаимодействий с веб-службами REST используются все возможности протокола HTTP.

Представьте ситуацию, когда браузер отправляет на сервер HTML-форму методом HTTP GET. Вы заполняете поля формы и щелкаете на кнопке **Submit** (Отправить). В адресной строке браузера вы увидите нечто подобное:

```
http://localhost:8080/wse/join.html?firstname=Ryan&lastname=
Cuprak&container=seller
```

Адрес URL имеет следующую структуру:

```
<схема>:<порт>/<идентификатор ресурса>
```

В данном случае элементы «схема», «порт» и «идентификатор ресурса» имеют следующие значения:

- схема: `http`;
- порт: `8080`;
- идентификатор ресурса: `/wse/join.html?firstname=Ryan&lastname=Cuprak&container=seller`.

В запрос включено все, что необходимо для его обработки. Если посмотреть, что в действительности передается между браузером и сервером, можно увидеть дополнительные, скрытые от глаз, данные. Вся эта информация объединяется воедино и отправляется на сервер для обработки. Благодаря ей сервер будет знать, на какой платформе выполняется клиент, является ли он мобильным или полноцен-

ным браузером и сможет сформировать соответствующий ответ. С обсуждаемым запросом в сеть передается примерно следующее содержимое:

```
GET /wse/join.html?firstname=Ryan&lastname=Cuprak&container=seller HTTP/1.1
Host: localhost:8090
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4) AppleWebKit/534.57.2
      (KHTML, like Gecko) Version/5.1.7 Safari/534.57.2
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://localhost:8090/wse/join.html
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Cookie: __utma=111872281.1889907053.1320526418.1320526418.1320526418.1;
  treeForm_tree-hi=treeForm:tree:configurations:default-config:loggerSetting;
  JSESSIONID=ba0e0c6c96fa10944cbf7d69309d
Connection: keep-alive
```

Отладка обращений к веб-службам

При разработке веб-служб часто бывает необходимо отладить сообщения, участвующие в обмене. Перехватывать запросы и ответы можно с помощью дополнительных инструментов, таких как `tcpmon` (http://ws.apache.org/commons/tcpmon/). Использование этого инструмента позволит увидеть точное содержимое запроса и ответа. Вы можете обнаружить, что запросы в корпоративной сети перехватываются прокси-сервером (обычное явление для больших предприятий) или, что запрос изначально формируется неверно. Интегрированные среды разработки, такие как NetBeans имеют встроенную поддержку утилиты `tcpmon`, благодаря чему вы сможете отлаживать запросы, отправляемые на сервер, и реализацию их обработки на сервере.

Мы не будем углубляться в детали строения запросов. Протокол HTTP определяется в документе RFC 2616, который можно найти на сайте www.w3.org. Для успешного создания или использования веб-служб RESTful не требуется полного знания спецификации, но вы определенно почувствуете необходимость в более глубоких знаниях, создав несколько собственных веб-служб или поработав с инструментами на других платформах. Эти знания будут особенно полезны для тех, кому предстоит заниматься проектированием новых служб.

Запрос, описанный выше, использует метод `GET`. Для каждого HTTP-запроса к серверу должен быть выбран один из восьми методов. Метод определяет порядок обработки запроса и операцию, которую сервер должен выполнить. Мы сосредоточимся только на шести методах, потому что другие два – `TRACE` и `CONNECT` – не имеют отношения к веб-службам RESTful. Вот эти методы:

- `GET` – операция чтения, которая не изменяет данных, хранящихся на сервере.
- `DELETE` – удаляет указанный ресурс. Многократное выполнение этой операции не вызывает никакого эффекта. После удаления ресурса все последующие попытки удалить его просто игнорируются.
- `POST` – изменяет данные на сервере. Запрос может включать или не включать данные, которые требуется сохранить, а ответ на него может содержать или не содержать возвращаемые данные.

- PUT – сохраняет на сервере данные, переданные в теле запроса, с идентификатором ресурса, указанном в URL. Многократное выполнение этой операции не вызывает никакого эффекта, потому что при этом обновляется один и тот же ресурс.
- HEAD – возвращает только код ответа и заголовки, связанные с запросом.
- OPTIONS – запрос на получение информации о доступных вариантах взаимодействий в цепочке запрос/ответ для указанного URI.

Все эти методы имеют две характеристики, которые следует знать и помнить: безопасность и идемпотентность. Безопасными считаются методы, которые просто извлекают данные. К таким методам относятся GET и HEAD. Идемпотентными считаются методы, которые производят один и тот же эффект, независимо от того, сколько раз будет выполнен запрос. К идемпотентным методам относятся GET, HEAD, PUT и DELETE. Остальные методы считаются либо небезопасными, либо неидемпотентным. Веб-службы должны учитывать эту семантику. Обращение к веб-службе RESTful в ActionBazaar с использованием метода HTTP DELETE не должно приводить к одновременному удалению существующей и созданию новой ставки. Это не совсем то, что ожидает типичный клиент. Даже если такая возможность описана в документации, она нарушает основополагающие принципы REST.

В дополнение к HTTP-методам необходимо также принимать во внимание HTTP-коды состояния. Они определены в RFC и возвращаются в ответ на каждый HTTP-запрос. Вы наверняка неоднократно сталкивались с этими кодами. Например, вы могли скопировать и вставить адрес сайта в адресную строку, и получить страницу с сообщением об ошибке: «HTTP Status 404 – the requested resource isn't available». Коды состояния, определенные в RFC, также будут возвращаться в ответ на обращения к веб-службам RESTful – их можно рассматривать как аналоги сообщений об ошибках SOAP. Если в процессе обработки запроса веб-службой RESTful будет возбуждено исключение, скорее всего клиент получит код состояния 500, соответствующий внутренней ошибке сервера.

До настоящего момента мы занимались исключительно обсуждением протокола HTTP. У кого-то мог даже возникнуть вопрос: какое отношение это обсуждение имеет к веб-службам RESTful? Ответ: самое непосредственное – веб-службы RESTful используют базовые HTTP-операции в комбинации с адресами URL. При взаимодействии с веб-службой SOAP клиент создает документ XML и отправляет его веб-службе. Веб-служба анализирует содержимое документа XML и определяет, какую операцию следует выполнить. При взаимодействии с веб-службой RESTful производится обращение по адресу URL, подобному тому, что был представлен выше, с использованием одного из HTTP-методов, таких как GET, DELETE или PUT. В этих взаимодействиях не участвуют документы XML – все необходимое содержится в методе и URL. Параметры вызова передаются внутри URL, как это делается при отправке веб-форм. Дополнительное содержимое, такое как документы Microsoft Word, документы XML, видеоролики и так далее, включается в содержимое запроса, то есть, отправляется в виде составных MIME-данных.

Итак, веб-службы RESTful можно рассматривать как «возврат к основам». Вызов веб-службы RESTful так же прост, как переход по адресу URL в браузере.

Параметры встраиваются в URL, а не передаются в виде документа XML. Результат, возвращаемый службой, может быть чем угодно – графикой, простым текстом, XML, JSON и так далее. Если приложение-клиент поддерживает стандарт HTML 5, веб-служба может возвращать ответ в компактном формате JSON. Реализации JavaScript в браузерах обрабатывают формат JSON на порядок эффективнее, чем сообщения SOAP.

Ниже приводятся примеры адресов URL, которые теоретически могут соответствовать вызовам веб-служб RESTful в приложении ActionBazaar:

- (`DELETE`): http://actionbazaar.com/deleteBid?bidId=25
- (`PUT`): http://actionbazzar.com/placeBid?item=245&userId=4243&amount=5.66

Они более компактны и очевидны, чем их аналоги в SOAP. Код для обработки этих запросов легко можно написать с применением поддержки сервлетов еще 10 лет тому назад. То есть, веб-службы RESTful не являются чем-то новым – просто теперь протокол HTTP используется еще и для реализации веб-служб, а не только для транспортировки документов HTML. Все это можно было делать и раньше.

Механизм JAX-RS избавляет от необходимости писать сервлеты и дает возможность вызывать методы непосредственно, с параметрами из URL, преобразуя данные по мере необходимости. А теперь, прежде чем перейти к коду, посмотрим, где и когда имеет смысл использовать службы REST.

8.3.2. Когда следует использовать REST/JAX-RS

Простота веб-служб RESTful подразумевает легкость их реализации и использования. Для этого не требуется применения каких-то специальных инструментов – большинство платформ и языков уже обладают всем необходимым для взаимодействий с веб-службами RESTful. Потребность в сторонних библиотеках и особенности реализации делают использование технологии SOAP намного более сложным делом. При реализации служб SOAP на языке Java, высока вероятность, что такие службы будет сложно использовать из приложений на других языках, таких как C#. Профили WS-I призваны уменьшить эту вероятность, но она все еще имеет место. Обратите внимание, что реализация собственного протокола сокетов могла бы оказаться еще более сложным делом, чем использование технологии SOAP; некоторые проблемы совместимости не могут служить поводом для полного отказа от ее применения – никакое решение не является идеальным.

Так как веб-службы RESTful способны возвращать все, что угодно, ответ может быть адаптирован для конкретного клиента. Сообщения SOAP излишне многословны и требуют значительных вычислительных ресурсов для обработки запросов и создания ответов в формате XML. Веб-службы RESTful резко отличаются в этом отношении; они используют протокол HTTP и адреса URL для представления запросов, а в качестве ответов может возвращаться все, что угодно. Запросы компактны и требуют минимальных усилий для их обработки. Ответы могут быть компактными и адаптированными для конкретного клиента. Например, приложениям Web 2.0 веб-служба RESTful может возвращать ответ в формате JSON, благодаря чему он может использоваться клиентами непосредственно, и не требуется

генерировать автоматически или писать вручную сложный программный код для обработки документов XML.

Компактность запросов и ответов особенно важны там, где задержки и вычислительные ресурсы являются ограничивающими факторами. Чем больше сообщение SOAP, тем больше времени требуется для его отправки и тем больше вычислительных ресурсов потребляется для его обработки. Веб-службы RESTful предлагают намного более широкие возможности по оптимизации запросов и ответов в ситуациях, когда производительность является руководящим фактором. Например, последние версии iOS (iOS 5) включают API для работы с веб-службами RESTful, но не имеют встроенной поддержки SOAP. Несмотря на встроенные механизмы экономии заряда аккумулятора в iPhone и iPad, даже небольшая оптимизация позволяет экономить его еще больше.

Теперь, когда мы выяснили некоторые ключевые показания к использованию веб-служб RESTful, необходимо также понять, когда следует экспортировать компоненты EJB посредством REST.

8.3.3. Когда следует экспортировать компоненты EJB в виде веб-служб REST

В этом разделе мы попытаемся ответить на вопрос: когда следует экспортировать компоненты EJB в виде веб-служб RESTful. Как вы уже знаете, компоненты EJB предоставляют множество полезных услуг посредством простых объектов POJO. Если веб-службе необходимы такие особенности, как транзакции, тогда компоненты EJB следует экспортировать с применением REST. Однако есть ряд архитектурных моментов, которые следует учитывать. Во-первых, роль провайдеров служб RESTful могут играть только сеансовые компоненты без сохранения состояния. Во-вторых, веб-службы RESTful намного ближе к сетевым форматам, чем их SOAP-собратья; SOAP – это более высокоуровневая абстракция. Необходимо особенно позаботиться о том, чтобы не осложнить интерфейс поддержкой REST. А теперь рассмотрим каждый из пунктов более внимательно.

Так как веб-службы RESTful не имеют хранимого состояния, комбинация REST и EJB имеет смысл только в контексте сеансовых компонентов EJB без сохранения состояния. Компоненты с сохранением состояния и компоненты, управляемые сообщениями, несовместимы с природой REST и такая комбинация теряет всякий смысл. Компоненты-одиночки можно экспортировать в виде веб-служб REST, но такие службы не будут иметь высокой производительности из-за того, что только один компонент сможет обрабатывать запросы. В итоге, роль веб-служб REST могут играть только сеансовые компоненты без сохранения состояния.

Слой компонентов EJB должен быть нейтральным по отношению к типам клиентов. Компонент `BidManager` в приложении ActionBazaar должен возвращать список ставок пользователя в виде объекта `java.util.List`, а не в виде строки в формате JSON. JSON – это уже конкретное представление данных, адаптированное для удобства веб-клиентов. Опасность веб-служб RESTful состоит в том, что они более тесно связаны с транспортным протоколом.

Другая проблема, связанная с веб-службами RESTful, – типы данных, обрабатываемых службой. Экспортирование компонента EJB в виде веб-службы REST оправданно при достаточно мелком дроблении задач, решаемых методами, и если эти методы принимают и возвращают значения простых типов или строки. Компоненты EJB, обменивающиеся сложными объектами, хуже подходят на эту роль. Даже при том, что для сериализации данных JAX-RS использует JAXB, поддержка SOAP (с ее возможностью определять схемы) значительно упрощает создание клиентов, обменивающихся сложными документами XML.

Наконец, необходимо подумать о проблеме упаковки. Если приложение развертывается из файла EAR с отдельными модулями, реализующими веб-слой и слой компонентов EJB, это может вызывать проблемы при развертывании служб. Контейнеры могут не сканировать модуль с компонентами EJB в поисках веб-служб RESTful. Обязательно проверьте поведение своего контейнера, чтобы избежать неожиданностей.

Существует множество проблем, которые следует рассмотреть, прежде чем принять решение об экспортировании компонента EJB в виде веб-службы RESTful. Несмотря на простоту и легковесность веб-служб RESTful, необходимо все же выделить дополнительное время на планирование и обдумывание в графике разработки.

8.3.4. Веб-служба REST для ActionBazaar

Приложение ActionBazaar экспортирует несколько важнейших своих услуг в виде веб-служб RESTful. Одним из ключевых компонентов, экспортируемых посредством JAX-RS, является `BidService`, с которым мы уже встречались неоднократно на протяжении книги. Он включает операции для добавления, удаления и изменения ставок. Учитывая быстрое распространение смартфонов, таких как iPhone, есть смысл экспортировать ряд основных услуг приложения ActionBazaar в виде веб-служб RESTful, чтобы упростить создание клиентов для этих устройств. Применение технологии SOAP могло бы потребовать значительных вычислительных ресурсов и большого объема трафика. Хотя смартфоны вполне пригодны для обработки запросов SOAP, реализация RESTful все же больше подходит для них.

Чтобы начать процесс разработки, необходимо сформулировать идентификаторы URI для основных операций в `BidService`:

- вернуть список ставок, сделанных указанным пользователем в указанной категории за указанный диапазон дат:
  `/bidService/list/{userId}/{category}?startDate=x&endDate=x`
- добавить новую ставку и принять XML в качестве входных данных:
  `/bidService/addBid`
- вернуть экземпляр ставки в формате XML или JSON:
  `/bidService/getBid/{bidId}`
- отменить ставку с указанным идентификатором:
  `/bidService/cancel/{bidId}`

Вот как может выглядеть вызов одной из этих операций:

```
http://actionbazaar.com:8080/bidService/getBid/83749
```

Если ввести эту строку в веб-браузере, он отобразит текст в формате XML, сгенерированный фреймворком JAXB для объекта `Bid`.

Чтобы реализовать веб-службу RESTful, в локальный интерфейс следует добавить аннотации JAX-RS. Это освободит реализацию от тонкостей, связанных с JAX-RS и сделает ее более читаемой. Помимо следования общепринятой практике, применение интерфейсов устраняет перегрузку кода аннотациями. Структура получившегося в результате класса представлена на рис. 8.5.

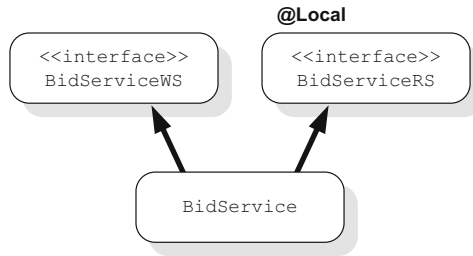

Рис. 8.5. Структура класса BidService

Исходный код класса `com.actionbazaar.buslogic.BidService` показан в листинге 8.6. В нем присутствует четыре экспортируемых метода. Этот пример иллюстрирует применение аннотаций, которые будут описаны далее в этой главе. После демонстрации адресов URL выше, должно быть очевидно, каким образом эти аннотации экспортируют целевые методы. Как можно убедиться, экспортирование компонента EJB в виде веб-службы RESTful осуществляется просто. Кроме того, эту службу легко протестировать с помощью веб-браузера – для этого не требуется создавать отдельного клиента.

Листинг 8.6. RESTful-интерфейс BidServiceRS

```
// Отмечает класс, как содержащий веб-службу RESTful;
// параметр аннотации - путь к корню службы
@Local @Path("/bidService")
public interface BidServiceRS {

    @POST                          // Вызывается HTTP-методом POST
    @Path("/addBid")               // Полный путь: /bidService/addBid
    @Consumes("application/xml") // Объект Bid поступает в формате XML
    public void addBid(Bid bid);

    @GET                           // Вызывается HTTP-методом GET
    @Path("/getBid/{bidId}")
    @Produces({"application/json",application/xml"}) // Возвращает JSON или XML
    public Bid getBid(@PathParam("bidId") long bidId);// bidId - параметр метода

    @DELETE                        // Вызывается HTTP-методом DELETE
```

```
    @Path("delete/{bidId}")
    public void cancelBid(@PathParam("bidId") long bidId);

    @GET
    @Produces("text/plain")
    @Path("/list/{userId}/{category}")
    public String listBids(
            @PathParam("category")String category,
            @PathParam("userId")long userId,
            @QueryParam("startDate") String startDate, // Извлекаются из
            @QueryParam("endDate") String endDate);    // параметров запроса
}
```

Один метод в этом листинге получает объект `Bid` в формате XML и один возвращает его в этом формате. Фактически экземпляр `Bid` – это объект DTO, было бы крайне нежелательно, чтобы преобразование применялось также к ссылкам на `Item` и `Bidder`. За преобразование отвечает фреймворк JAXB. Так как класс `Bid` является чрезвычайно простым DTO, мы вручную добавили аннотации JAXB – объект не генерируется на основе схемы XML, как это обычно делается. Определение класса этого объекта DTO показано в листинге 8.7.

Листинг 8.7. Класс Bid объекта DTO, используемый в аннотациях JAXB

```
@XmlRootElement(name="Bid") // Корневой элемент XML-документа
// Сериализовать все нестатические, нетранзитивные поля
@XmlAccessorType(XmlAccessType.FIELD)
public class Bid {

    @XmlElement             // Сериализовать в отдельный элемент
    // Использовать кроссплатформенный объект Date, совместимый с .NET
    private XMLGregorianCalendar bidDate;

    @XmlAttribute           // Сериализовать в атрибут корневого элемента
    private Long bidId;

    @XmlElement
    private double bidPrice;

    @XmlElement
    private long itemId;

    @XmlElement
    private long bidderId;

    public Bid() {
    }

    // Методы доступа к полям объекта не показаны
}
```

При вызове метода `getBid`, объект DTO из листинга 8.7 будет преобразовываться в документ XML, представленный в листинге 8.8, который поможет вам лучше понять, как аннотации JAXB управляют преобразованием данных в формат XML.

Листинг 8.8. Пример вывода Bid DTO в формате XML

```
<Bid bidId="10">
  <bidDate>2012-05-15T20:01:35.088-04:00</bidDate>
  <bidPrice>10.0</bidPrice>
  <itemId>45</itemId>
  <bidderId>77</bidderId>
</Bid>
```

Если при обращении к методу `getBid` клиент сообщит, что желает получить результат в формате JSON, веб-служба вернет данные в виде строки JSON, как показано в листинге 8.9. Результаты в формате JSON идеально подходят для приложений Web 2.0, написанных на JavaScript или для смартфонов. После получения этих данных на стороне клиента, они становятся доступны клиентскому приложению в виде ассоциативного массива, работать с которым намного проще, чем с разметкой XML.

Листинг 8.9. Пример вывода Bid DTO в формате JSON

```
{
  "@bidId":"10",
  "bidDate":"2012-05-15T21:52:58.771-04:00",
  "bidPrice":"10.0",
  "itemId":"45",
  "bidderId":"77"
}
```

Если попытаться выполнить этот код, реализация JAX-RS почти наверняка сообщит, что не знает, как создать экземпляр интерфейса `BidServiceRS`. Чтобы связать сеансовый компонент без сохранения состояния со службой, необходимо реализовать `javax.ws.rs.core.Application` и вернуть экземпляры всех интерфейсов, отмеченных аннотациями JAX-RS. Реализация экземпляра `Application`, представленная в листинге 8.10, должна быть зарегистрирована в сервлете JAX-RS, в файле `web.xml` (здесь не показан). Описанный прием будет работать с реализацией Metro, но другие реализации JAX-RS могут действовать иначе.

Листинг 8.10. Настройка экземпляров интерфейса JAX-RS

```
public class SystemInit extends Application {
    public Set<Class<?>> getClasses() {
        return Collections.emptySet();
    }

    public Set<Object> getSingletons() {
        Set<Object> classes = new HashSet<Object>();
        try {
            InitialContext ctx = new InitialContext();
            Object bidServiceRS =
                ctx.lookup("java:global/WebServiceExperiment/BidService");
            classes.add(bidServiceRS);
```

```
        } catch (Throwable t) {
            t.printStackTrace();
        }
        return classes;
    }
}
```

Теперь, после знакомства с примером реализации простой службы JAX-RS, познакомимся поближе с аннотациями.

8.3.5. Аннотации JAX-RS

В этом разделе мы исследуем наиболее важные аннотации JAX-RS, покроем основные случаи их применения и подробнее опишем примеры кода из приложения ActionBazaar, продемонстрированные выше. Тем использования механизма JAX-RS можно было бы посвятить целую книгу, поэтому, если вы решите экспортировать свои компоненты EJB в виде веб-служб RESTful, обязательно продолжите изучение этой темы с помощью других источников.

Веб-службы RESTful намного проще в реализации, чем веб-службы SOAP. Чтобы создать веб-службу RESTful, достаточно применить следующие аннотации:

- `@javax.ws.rs.Path` – эта аннотация должна присутствовать перед классом, реализующим веб-службу RESTful;
- `@javax.ws.rs.GET/POST/PUT/DELETE` – эти аннотации должны помещаться перед методами. Они определяют реализации обработчиков запросов для разных HTTP-методов и не имеют параметров.

Чтобы определить простейшую действующую веб-службу RESTful, достаточно применить аннотацию `@Path` к классу и аннотации, определяющие HTTP-методы, к методам класса. Давайте исследуем эти аннотации, начав с `@GET`. Имейте в виду, что мы не будем рассматривать все аннотации JAX-RS – это далеко выходит за рамки данной главы.

Аннотация @GET

Аннотация `@javax.ws.rs.GET` является аннотацией-меткой, определяющей метод веб-службы, который должен вызываться в ответ на запрос HTTP GET. Ниже приводится определение аннотации `GET`:

```
@Target(value = {ElementType.METHOD})
@Retention(value = RetentionPolicy.RUNTIME)
@HttpMethod(value = "GET")
public @interface GET {}
```

Если в составе запроса `GET` предполагается передавать дополнительные параметры, необходимо учитывать верхнее ограничение на размер URI запроса. Это ограничение определяется не спецификацией HTTP – оно зависит от браузера. Веб-браузеры традиционно ограничивают длину URI от 4 Кбайт до 8 Кбайт.

Аннотация @POST

Аннотация `@javax.ws.rs.POST` является аннотацией-меткой, определяющей метод веб-службы, который должен вызываться в ответ на запрос HTTP POST. Ниже приводится определение аннотации `POST`:

```
@Target(value = {ElementType.METHOD})
@Retention(value = RetentionPolicy.RUNTIME)
@HttpMethod(value = "POST")
public @interface POST {}
```

Метод HTTP POST используется для передачи данных на сервер, например, для отправки фотографии товара в приложение ActionBazaar. В сообщении передается ссылка на новую или имеющуюся запись в базе данных. Обратите внимание, что в результате операции не всегда создается новая запись. В одном запросе POST можно передавать несколько вложений, если объявить его как включающий составные MIME-данные; смотрите для примера аннотацию `@Consumes`.

Аннотация @DELETE

Аннотация `@javax.ws.rs.DELETE` является аннотацией-меткой, определяющей метод веб-службы, который должен вызываться в ответ на запрос HTTP DELETE. Ресурс, идентифицируемый строкой URI, должен быть удален, при этом все последующие операции с тем же идентификатором ресурса не должны производить никакого эффекта, так как ресурс уже удален. Ниже приводится определение аннотации `DELETE`:

```
@Target(value = {ElementType.METHOD})
@Retention(value = RetentionPolicy.RUNTIME)
@HttpMethod(value = "DELETE")
public @interface DELETE {}
```

Аннотация @PATH

Аннотация `@javax.ws.rs.Path` помещается перед классом и перед методами. Когда она помещается перед классом, она объявляет класс службой JAX-RS. Класс, экспортируемый как веб-служба RESTful, обязательно должен быть отмечен этой аннотацией. Когда аннотация помещается перед методом, этот метод экспортируется как веб-служба RESTful.

Параметр аннотации `@Path` определяет относительный URI службы. Идентификатор URI, указанный в аннотации перед методом, объединяется с URI в аннотации перед классом и определяет полный путь к веб-службе RESTful. Аннотация определена, как показано ниже:

```
@Target(value = {ElementType.TYPE, ElementType.METHOD})
@Retention(value = RetentionPolicy.RUNTIME)
public @interface Path {
  public String value();
}
```

Параметры могут включаться в URI с помощью фигурных скобок ({}). Параметры запроса связываются с параметрами метода с помощью аннотации @java.ws.rs.PathParam, которая описывается далее. Фигурных скобок с аннотациями может быть столько, сколько потребуется.

Аннотация @PathParam

Аннотация @javax.ws.rs.PathParam отображает параметры, указанные в @javax.ws.rs.Path, в фактические параметры Java-метода. Механизм JAX-RS автоматически будет выполнять преобразование типов при вызове методов. Ниже приводится определение аннотации:

```
@Target(value = {ElementType.PARAMETER, ElementType.METHOD,
    ElementType.FIELD})
@Retention(value = RetentionPolicy.RUNTIME)
public @interface PathParam {
  public String value();
}
```

Взгляните на следующий пример, где показано, как объявить метод веб-службы, принимающий два параметра, userId и category:

```
@GET
@Path("/list/{userId}/{category}")
public List<Item> listBids(
        @PathParam("category")String category,
        @PathParam("userId")long userId
    ) {
  ...
}
```

В этом примере параметры URI запроса, category и userID, отображаются в фактические параметры метода. Если предположить, что класс отмечен аннотацией @Path("/bidService"), тогда строка вызова метода будет выглядеть примерно так:

```
http://localhost:8080/actionbazaar/bidService/list/14235/cars
```

Здесь также предполагается, что приложение развернуто в контексте ActionBazaar. Работу этой службы легко можно протестировать с помощью веб-браузера.

Аннотация @QueryParam

Аннотация @javax.ws.rs.QueryParam позволяет извлекать параметры запроса и отображать их в фактические параметры метода. Эта аннотация определена, как показано ниже:

```
@Target(value = {ElementType.PARAMETER, ElementType.METHOD,
    ElementType.FIELD})
@Retention(value = RetentionPolicy.RUNTIME)
public @interface QueryParam {
  public String value();
}
```

Следующий фрагмент демонстрирует применение аннотации:

```
@Path("/bidService")
@Stateless(name="BidService")
public class BidServiceBean {
  @GET @Produces("text/plain")
  @Path("/list/{userId}/{category}")
  public String listBids(
    @PathParam("category")String category,
    @PathParam("userId")long userId,
    @QueryParam("startDate") String startDate,
    @QueryParam("endDate") String endDate
  ) {
      ...
  }
}
```

Эту службу можно вызвать из веб-браузера, введя следующую строку URL: `http://localhost:8080/wse/bidService/list/3232/cars?startDate=10252012&endDate=10302012`. Параметры запроса, например `startDate` и `endDate`, будут отображены в одноименные параметры метода.

> **Преобразование типов**
>
> Возможно, вы обратили внимание, что в этом примере передаются начальная и конечная даты в виде значений типа `java.lang.Strings`. Объект `java.util.Date` не имеет конструктора, который принимал бы строковый параметр. Некоторые реализации, такие как Apache CFX, включают расширения для регистрации обработчиков, позволяющих решать подобные проблемы и гарантирующих, что интерфейсы компонентов не будут подвержены ограничениям JAX-RS.

Аннотация @Produces

Аннотация `@javax.ws.rs.Produces` определяет типы данных многоцелевых расширений электронной почты Интернета (Multipurpose Internet Mail Extensions, MIME), которые службы могут возвращать клиентам. Клиенту может возвращаться один или более типов. Типы, поддерживаемые клиентом, служба должна определить по заголовкам запроса. Ниже приводится определение аннотации:

```
@Inherited
@Target(value = {ElementType.TYPE, ElementType.METHOD})
@Retention(value = RetentionPolicy.RUNTIME)
public @interface Produces {
  public String[] value() default {"*/*"};
}
```

Следующий пример демонстрирует применение этой аннотации к методу, который может возвращать данные в формате XML или JSON, в зависимости от требований клиента:

```
@GET
@Path("/getBid/{bidId}")
@Produces({"application/json","application/xml"})
public Bid getBid(@PathParam("bidId") long bidId) {
  ...
}
```

Если вызвать этот метод из веб-браузера, можно увидеть данные в формате XML или JSON. Реализация JAX-RS может сгенерировать документ XML (с помощью JAXB) или строку в формате JSON.

В число часто используемых MIME-типов, которые могут возвращаться методами веб-служб, входят:

- `application/xml` – XML;
- `application/json` – текст в формате JSON;
- `text/plain` – простой текст;
- `text/html` – разметка HTML;
- `application/octet-stream` – произвольные двоичные данные;

Это не полный список, в него включены лишь наиболее часто используемые типы.

Аннотация @Consumes

Аннотация `@javax.ws.rs.Consumes` похожа на аннотацию `@Produces`. Единственное различие в том, что данная аннотация определяет MIME-тип, принимаемый службой. Эта аннотация используется, когда метод может принимать данные разных типов. Она определена, как показано ниже:

```
@Inherited
@Target(value = {ElementType.TYPE, ElementType.METHOD})
@Retention(value = RetentionPolicy.RUNTIME)
public @interface Consumes {
  public String[] value() default {"*/*"};
}
```

Следующий пример демонстрирует применение аннотации:

```
@POST @Path("/addBid")
@Consumes("application/xml")
public void addBid(Bid bid) {
  ...
}
```

Этот метод принимает объект `Bid` в формате XML. Преобразование документа XML в экземпляр `com.actionbazaar.dto.Bid` реализация JAX-RS выполняет с помощью фреймворка JAXB. Обратите внимание, что объект DTO используется для создания экземпляра `com.actionbazaar.persistence.Bid`, имеющего ссылки на `Item` и `Bidder` – было бы нежелательно выполнять сериализацию всего дерева объектов.

Как упоминалось прежде, мы рассмотрели лишь с часть аннотаций, предоставляемых механизмом JAX-RS. Для обсуждения всех деталей JAX-RS потребовалось бы написать отдельную книгу. Кстати, мы можем порекомендовать вам отличную книгу, посвященную теме REST и Java: «Restlet in Action» Джерома Лувеля (Jerome Louvel) (Manning, 2012).

В следующем разделе мы посмотрим, как эффективнее использовать EJB и веб-службы RESTful. Но имейте в виду, что это лишь краткое введение, так как тема эффективного использования REST весьма обширна.

8.3.6. Эффективное использование веб-служб REST в EJB

Экспортирование сеансовых компонентов в виде веб-служб REST требует предварительного планирования и тщательной проработки внешнего интерфейса. В отличие от веб-служб SOAP, веб-службы RESTful намного ближе к формату данных, передаваемых по сети. Вам придется позаботиться об устранении конфликтов с требованиями, предъявляемыми клиентами, такими как требование возвращать данные в формате JSON, определяя соответствующим образом интерфейсы сеансовых компонентов. Чтобы упростить задачу, экспортируемые компоненты должны иметь достаточно простые интерфейсы. Как отмечалось выше, желательно, чтобы такие компоненты принимали и возвращали данные простых типов или строки.

Аннотации JAX-RS следует применять к локальным интерфейсам. Применение их непосредственно к классам компонентов сделает реализацию трудночитаемой и установит тесную связь компонента с механизмом JAX-RS. Код с нагромождением аннотаций будет сложно читать и сопровождать. В будущем может потребоваться использовать шаблон проектирования «Адаптер» (Adapter), чтобы отделить компонент, и в этом случае наличие отдельного интерфейса упростит подобное архитектурное изменение.

Чтобы адаптировать интерфейс существующего компонента EJB для поддержки веб-служб RESTful, следует применить шаблон проектирования «Адаптер». При применении шаблона «Адаптер» веб служба RESTful реализуется как простой объект POJO. Для внедрения ссылок на компоненты EJB в такой объект можно использовать механизм CDI (внедрения контекстов и зависимостей). Подробнее о механизме CDI рассказывается в следующей главе. Согласно этому шаблону проектирования, адаптер веб-службы RESTful должен транслировать сообщения в форму, которую способен принимать компонент EJB, и тем самым исключить «просачивание» требований интерфейса RESTful в прикладную логику.

Экспортируя компоненты EJB в виде веб-служб REST, необходимо точно следовать семантике REST. Операции, выполняемые компонентом EJB, должны отображаться в соответствующие HTTP-методы. Метод компонента, удаляющий запись, должен вызываться в ответ на запрос HTTP DELETE. Метод, извлекающий данные, должен вызываться в ответ на запрос HTTP GET. Соблюдение принципов безопасности и идемпотентности методов является важным условием создания

надежных решений. Если прежде вы работали в основном с веб-службами SOAP, вам может потребоваться несколько изменить свой образ мышления.

Прежде всего следует стремиться сохранить веб-службы RESTful максимально простыми. Под веб-службами RESTful в первую очередь подразумевается легковесность. Признаком простоты может служить возможность вызова веб-службы из браузера. Если для вызова веб-службы необходим дополнительный код, вероятно, такую службу лучше реализовать с применением технологии SOAP. Веб-службы RESTful не имеют тесной связи с форматом XML – если вдруг выяснится, что служба интенсивно использует фреймворк JAXB для преобразования объектов в формат XML и обратно, это может служить поводом задуматься о пересмотре архитектуры.

8.4. Выбор между SOAP и REST

В этой главе обсуждались два разных подхода к организации веб-служб: SOAP и REST. Первое решение, которое вам придется принять при создании новой веб-службы – какой из двух подходов выбрать. На этот вопрос нет простого ответа. Вам потребуется взвесить множество факторов, характерных для приложения. Первым из них является наличие в приложении функциональности, уже экспортируемой посредством SOAP или REST. Если такой функциональности нет, тогда нужно выяснить, какая модель лучше отражает потребности в службах, REST или SOAP. Модель REST предъявляет более строгие требования к организации API. Следует также учитывать типы обслуживаемых клиентов и необходимость предоставления файла WSDL, чтобы обеспечить возможность автоматизированного создания кода.

Часто решающим фактором в выборе того или иного подхода является популярность. В настоящее время REST является более популярным подходом к созданию новых веб-служб. SOAP – слишком сложная и тяжеловесная технология. Сложность и тяжеловесность часто делают экспортирование простых служб неоправданно сложным делом. Как результат, REST рассматривается многими как «возврат к основам». Рост популярности REST во многом обусловлен популярностью служб RESTful, предоставляемых такими интернет-гигантами, как Amazon, Flickr и Google. Если вы не ограничены в выборе каким-то одним подходом, то мы покажем вам выгоды и компромиссы каждого из них.

Начнем с SOAP. Основными преимуществами SOAP является самодокументируемость и превосходная поддержка дополнительными инструментам. Кроме того, существует несколько спецификаций WS, расширяющих SOAP поддержкой надежных сообщений, транзакций и системы безопасности. Самодокументируемая природа SOAP способствовала появлению множества инструментов, действующих по принципу «укажи и щелкни», которые способны автоматически генерировать код, вызывающий веб-службы. Благодаря этому бизнес-аналитики или системные интеграторы могут использовать веб-службы, совершенно не понимая механики работы SOAP и происходящего за кулисами. С другой стороны, не имея

хороших инструментов или не зная, как эффективно их использовать, правка документов WSDL вручную может оказаться очень нелегким делом. Поскольку основной упор делается на применение инструментов автоматизации, необходимо знать их ограничения, касающиеся поддержки разнородных платформ, и как они интерпретируют схемы XML при создании программного кода.

Веб-службы SOAP также поддерживают возможность маршрутизации через промежуточные узлы. Каждый узел может исследовать заголовки сообщений, выполнять некоторые операции и передавать сообщения дальше. Это очень полезная особенность в сложных окружениях.

Технология REST, напротив, очень проста: каждая операция REST отображается на URL, а метод HTTP определяет тип выполняемой операции. Веб-службы RESTful должны отображаться в операции CRUD. При создании веб-служб необходимо принимать во внимание методы HTTP и тщательно прорабатывать структуру URL. Результатом работы веб-службы RESTful необязательно должен быть документ XML – это может быть строка JSON, простой текст, изображение и так далее. Параметры вызова службы передаются в параметрах запроса. Благодаря использованию параметров запроса и возможности возвращать результаты в компактных форматах, таких как JSON, веб-службы RESTful предъявляют весьма низкие требования к пропускной способности сети и вычислительным ресурсам. Как следствие, веб-службы RESTful оказываются более масштабируемыми.

Хотя концептуально технология REST существенно проще, она не имеет такой обширной поддержки дополнительными инструментами, как SOAP. Несмотря на то, что WSDL теперь поддерживает REST, эта особенность не получила широкого распространения. Вы не сможете автоматизировать создание клиентов, однако разработка кода, вызывающего службу, не представляет сложностей, к тому же на многих платформах (например, Java и iOS) имеются библиотеки, упрощающих реализацию взаимодействий со службами RESTful.

В табл. 8.2 представлена матрица решений, которая поможет вам сделать выбор между SOAP и REST. Немаловажным фактором при принятии решения является также наличие действующих веб-служб в вашем приложении. Выпишите плюсы обеих технологий и затем подсчитайте и сравните их количества. Исходя из конкретных потребностей, некоторые факторы могут оказаться важнее других.

Таблица 8.2. Матрица решений для выбора между SOAP и REST

	SOAP	REST
Требуется поддержка дополнительными инструментами (автоматизация создания клиентов)	X	
Отсутствуют инструменты создания/редактирования WSDL		X
Ограниченность вычислительных ресурсов		X
Ограниченность пропускной способности сети		X
Операции отображаются в схему CRUD		X
Обработка документов XML	X	

Таблица 8.2. (окончание)

	SOAP	REST
Транзакции, координация и так далее	X	
Маршрутизация/обработка сообщений	X	X
Требуется применение протокола SMTP	X	
Поддержка других форматов вывода, кроме XML (JSON, текст, графика и так далее)		X
Сообщения типа «точка–точка»		X
Преобразование входящих/исходящих сообщений в соответствии со схемой	X	

Теперь, когда у вас есть простая матрица решений, которая поможет сделать правильный выбор, давайте подведем итоги и перейдем к JPA.

8.5. В заключение

В этой главе вы узнали, как экспортировать сеансовые компоненты без сохранения состояния в виде веб-служб SOAP или RESTful. В первой половине главы мы занимались исследованием веб-служб SOAP. Веб-службы SOAP являются документо-ориентированными. Мы рассмотрели приемы создания простых веб-служб SOAP, а также структуру WSDL и сообщений SOAP. Провели сравнение RPC-ориентированных и документо-ориентированных, а также кодированных и литеральных разновидностей веб-служб. Познакомились с основными аннотациями JAX-WS и фреймворком JAXB. Кроме того, мы обсудили различные подходы к разработке веб-служб, например, автоматическое создание серверной реализации на основе WSDL или создание WSDL на основе имеющейся реализации, а также рассмотрели наиболее эффективные приемы использования веб-служб.

Во второй половине главы мы погрузились в изучение веб-служб RESTful. В кратком введении было дано сравнение веб-служб RESTful и SOAP. Вы имели возможность увидеть, что веб-службы RESTful существенно проще и основаны на уже имеющейся инфраструктуре протокола HTTP. Веб-службы RESTful используют HTTP-методы `GET`, `PUT`, `DELETE` и другие для реализации услуг – услуги тесно связаны с конкретными HTTP-методами и потому должны придерживаться их семантики. Кроме того, для определения конкретных услуг веб-службы RESTful используют URI и параметры запросов, встроенные в URL. Веб-службы RESTful могут возвращать текст, JSON, XML и так далее, давая возможность адаптировать формат результата для конкретного клиента. Ближе к концу были представлены некоторые аннотации JAX-RS и приемы эффективного использования веб-служб. В следующей главе мы приступим к изучению JPA.

Часть III

ИСПОЛЬЗОВАНИЕ EJB СОВМЕСТНО С JPA И CDI

В этой части вы получите полное представление о взаимоотношениях EJB 3 с JPA и CDI. В главе 9 дается введение в предметное моделирование и особенности отображения сущностей JPA в предметную область. В ней также будут описываться особенности настройки отношений между объектами предметной области средствами JPA и использование механизма наследования объектов. Глава 10 охватывает управление сущностями JPA через CRUD-операции. В ней вы познакомитесь также с диспетчером сущностей `EntityManager`, жизненным циклом сущностей, сохранением и извлечением сущностей и понятием области видимости сущностей. Глава 11 рассказывает о языке JPQL и более подробно – о приемах извлечения данных. В ней вы научитесь создавать и выполнять стандартные запросы JPQL, а также использовать запросы SQL, когда это действительно необходимо. Глава 12 служит введением в технологию CDI и рассказывает, как она дополняет EJB. Здесь вы познакомитесь также с веб-расширениями CDI и с дополнительными средствами взаимодействий, жизненно важными для современных браузеров, обладающих интерфейсом с вкладками.

ГЛАВА 9. Сущности JPA

Эта глава охватывает следующие темы:

- JPA и несоответствие интерфейсов;
- моделирование предметной области;
- реализация объектов предметной области с помощью JPA;
- определение отношений между объектами предметной области с помощью JPA;
- использование наследования в объектах предметной области.

Java Persistence API (JPA) – это стандартный прикладной интерфейс Java, позволяющий создавать, извлекать, изменять и удалять данные в реляционных базах данных. Возможность взаимодействий с базами данных в языке Java имелась практически с самых первых дней его существования. JDBC API дает разработчикам прямой доступ к базе данных и возможность выполнять запросы SQL. Но, несмотря на все свои преимущества, интерфейс JDBC оказался слишком прост – в нем отсутствуют многие необходимые возможности. Вследствие этого поверх JDBC было разработано множество сторонних инструментов объектно-реляционного отображения (Object-Relational Mapping, ORM). Интерфейс JPA является стандартом объектно-реляционного отображения для платформ Java SE и Java EE. С точки зрения разработчика это означает возможность создания приложений баз данных без использования сторонних классов.

Цель этой главы – дать краткий обзор наиболее часто используемых возможностей JPA. Сначала посмотрим, как превратить любые объекты POJO предметной модели в сущности, управляемые механизмом JPA. Затем узнаем, как отображать сущности в таблицы базы данных. Потом исследуем разные стратегии, поддерживаемые механизмом JPA для определения и создания первичных ключей в базе данных. Наконец, займемся отношениями между таблицами базы данных (такими, как, например, связь множества ставок `Bid` с одним лотом `Item`) и узнаем, как JPA управляет отображением этих отношений. Покончив со всем этим, в

следующей главе мы займемся исследованием `EntityManager` и посмотрим, как в действительности реализуются операции CRUD.

9.1. Введение в JPA

JPA – это стандартное решение задачи, известной как объектно-реляционное отображение. Выражаясь простым языком, объектно-реляционное отображение – это транслятор (переводчик). Представьте двух человек, ведущих беседу. Один говорит на английском, а другой – на французском языке. Им нужен третий человек, переводчик, который знает оба языка. Объектно-реляционное отображение как раз и есть такой переводчик, знающий язык базы данных и язык предметной модели вашего приложения на Java. В данном случае роль переводчика будет играть JPA. Он отвечает за передачу данных в обе стороны между базой данных и предметной моделью на Java.

Объектно-реляционное отображение выполняет сложную работу. Для каждой структуры в базе данных должна поддерживаться возможность отображения в произвольную предметную модель на Java. Эту проблему отображения часто называют проблемой несоответствия интерфейсов (impedance mismatch). Обзор JPA мы начнем с описания проблемы несоответствия интерфейсов в следующем разделе, а затем до конца главы будем знакомиться с ключевыми особенностями JPA, помогающими решить ее.

9.1.1. Несоответствие интерфейсов

Под термином *несоответствие интерфейсов* (impedance mismatch) подразумеваются различия между объектно-ориентированной и реляционной парадигмами, и сложности, которые несут эти различия. Чаще всего проблема несоответствия интерфейсов возникает на уровне хранения, где находится предметная модель. Корень проблемы заключается в отличии фундаментальных целей двух технологий.

Когда полю объекта Java присваивается другой объект, обычно копируется только ссылка (указатель), а не сам объект. Иными словами, доступ к объектам в языке Java осуществляется по ссылке, а не по значению. Если бы это было невозможно, вам, вероятно, потребовалось бы сохранять идентичность присваиваемого объекта (например, в переменной типа `int`) и разыменовывать ее при необходимости. С другой стороны, в реляционных базах данных отсутствует понятие доступа к объектам по ссылке. Реляционные базы данных практически всегда используют идентичности. Эти идентичности указывают, где находится фактический объект, и запросы к базе данных используют эти идентичности для поиска требуемых данных.

Кроме того, язык Java предлагает такую роскошь, как наследование и полиморфизм, отсутствующие в реляционном мире. Наконец, объекты Java включают и данные (переменные экземпляра), и поведение (методы). Таблицы баз данных, напротив, хранят только данные в строках и столбцах, и не обладают поведенческими характеристиками.

Эти различия выдвигаю проблему несоответствия интерфейсов на передний план. Реляционные базы данных и предметные модели на языке Java используются для решения одной и той же концептуальной проблемы, но они используют совершенно разные языки. Опытный администратор баз данных в первую очередь сосредоточится на эффективности хранения данных в нормализованных таблицах, применяя первичные ключи, внешние ключи и ограничения для поддержания целостности данных. Опытный разработчик на Java сконцентрируется на создании исчерпывающей предметной модели, используя наследование, инкапсуляцию и полиморфизм, чтобы реализовать сложную схему поведения и бизнес-правил. В табл. 9.1 перечисленные некоторые из наиболее ярких несоответствий между объектно-ориентированным и реляционным мирами.

Таблица 9.1. Несоответствие интерфейсов: различия между объектно-ориентированным и реляционным мирами

Предметная модель (Java)	Реляционная модель (база данных)
Объекты, классы	Таблицы, строки
Атрибуты, свойства	Столбцы
Идентичность	Первичный ключ
Отношение/ссылка на другую сущность	Внешний ключ
Наследование/полиморфизм	Не поддерживается
Методы	Косвенно: SQL-логика, хранимые процедуры, триггеры
Код переносим	Переносимость зависит от конкретного типа базы данных

В оставшейся части главы мы посмотрим, как преодолеть проблему несоответствия интерфейсов, в частности с использованием JPA в качестве переводчика между реляционной и предметной моделями. Мы увидим, как JPA отображает строки и столбцы таблиц в объекты Java.

9.1.2. Взаимосвязь между EJB 3 и JPA

Прежде чем погрузиться в исследование JPA, познакомимся с историей этого механизма и его взаимоотношениями с компонентами EJB. В версии Java EE 5 контейнер EJB был полностью перестроен и переориентирован на использование легковесных POJO, аннотаций и парадигмы преобладания умолчаний перед настройками. Вместе с EE 5 вышла и первая версия прикладного интерфейса Java Persistence API. На тот момент определение JPA являлось частью спецификации EJB. В этом был определенный смысл, потому что тогда механизмы JPA были тесно связаны с компонентами EJB, которые в первую очередь отвечали за определение бизнес-правил и поддержание целостности прикладных данных. Но с развитием спецификации JPA и наполнением ее новыми возможностями, она превратилась в самостоятельную спецификацию Java Specification Request (JSR).

Версия JPA 2.1 определена в документе JSR-338 и доступна на веб-сайте Java Community Process: http://jcp.org/en/jsr/detail?id=338. Самым интересным следствием выделения интерфейса JPA в отдельную спецификацию является ликвидация тесной связи с Java Enterprise Edition (Java EE) и появление возможности использовать его в приложениях Java Standard Edition (Java SE). Иными словами, чтобы воспользоваться всеми богатствами JPA, больше не нужно иметь сервер Java EE или контейнер EJB. Благодаря этому открывается возможность использовать JPA там, где прежде это было невозможно. Но, перед тем как начать использовать JPA в своих приложениях, вам придется определить объекты для хранения прикладных данных и отношения между ними – то есть, выполнить предметное моделирование, о чем мы и поговорим далее.

9.2. Предметное моделирование

Зачастую первым этапом разработки промышленных приложений является создание предметной модели (domain model). То есть, составление списка сущностей предметной области и определение отношений между ними.

В следующем разделе вашему вниманию предлагается введение в предметное моделирование. Затем мы исследуем предметную область приложения ActionBazaar и идентифицируем основные компоненты модели, такие как объекты, отношения и кардинальность (cardinality). Мы представим краткий обзор, как предметное моделирование поддерживается в JPA и затем создадим простой объект предметной области в виде класса Java.

9.2.1. Введение в предметное моделирование

Предметная модель – это концептуальное представление задачи, решаемой приложением. Предметная модель состоит из объектов Java, представляющих данные приложения и отношения, или связи, между данными. Данные могут представлять нечто фактически существующее, например заказчика, или что-то более эфемерное, как, например, предпочтения заказчика. Отношение – это связь между объектами. Важно помнить, что предметная модель описывает объекты и отношения между ними. Предметная модель не описывает, как приложение будет манипулировать объектами – это относится уже к прикладной логике компонентов EJB и MDB.

9.2.2. Предметная модель приложения ActionBazaar

В этой книге мы занимаемся разработкой базовой функциональности приложения ActionBazaar, непосредственно связанной с аукционными торгами и продажей. Для начала посмотрим, какие действия составляют основу приложения ActionBazaar. Итак. как показано на рис. 9.1, основными в приложении ActionBazaar являются следующие действия:

- продавец размещает объявление на сервере ActionBazaar;
- объявления организуются в категории, удобные для поиска и просмотра;
- покупатели делают ставки;
- высшая ставка выигрывает.

Рис. 9.1. Основная функциональность ActionBazaar. Продавцы размещают объявления в соответствующих категориях. Покупатели делают ставки, и высшая ставка выигрывает

Чтобы определить объекты предметной области, достаточно просмотреть этот список и выделить имена существительные: продавец, объявление, категория, покупатель, ставка и заказ. Наша цель – выявить объекты предметной области, или сущности, которые необходимо хранить в базе данных. В реальном мире выявление объектов предметной области может занимать часы и требовать многократного повторения анализа прикладной задачи. Но для начала достаточно нарисовать простенькую диаграмму, случайно разбросав объекты, как показано на рис. 9.2.

Рис. 9.2. Сущности предметной модели ActionBazaar

Теперь, когда имеется модель объектов, необходимо обозначить, как они взаимодействуют друг с другом. Для этого достаточно нарисовать стрелки между объектами, показывающие связи между ними (эти стрелки отражают отношения между объектами) и модель закончена. Отношения в случае с приложением ActionBazaar показаны на рис. 9.3.

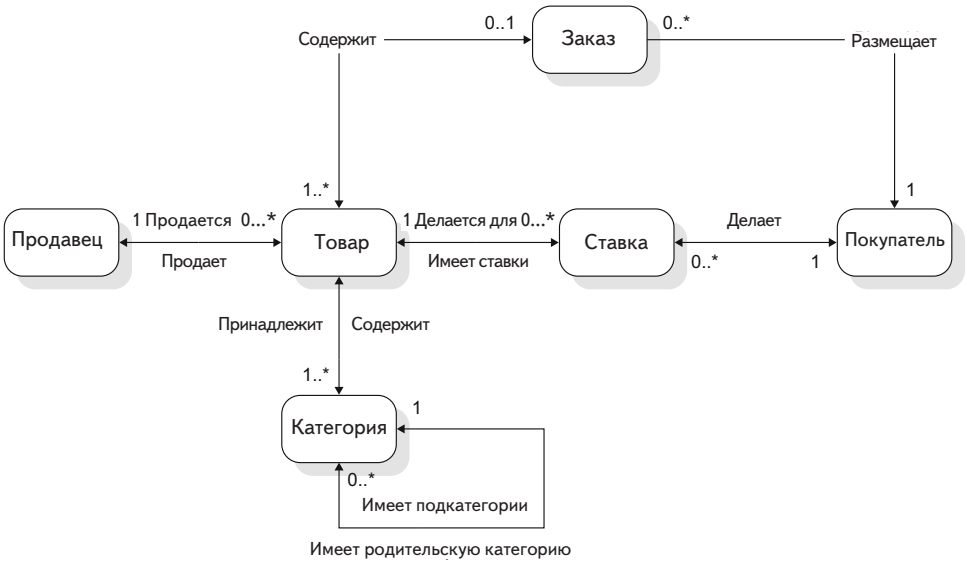

Рис. 9.3. Предметная модель ActionBazaar состоит из сущностей и отношениями между ними. Сущности связаны отношениями, а отношения могут иметь вид «один к одному», «один ко многим», «многие к одному» и «многие ко многим». Отношения могут быть одно- и двунаправленными

Рисунок 9.3 понятен без лишних пояснений. Например, товар продается продавцом, продавец может продавать более одного товара, товар можно отнести к одной или более категориям, каждая категория может быть подкатегорией другой категории, покупатель предлагает цену на товар, и так далее. Обратите также внимание, что хотя предметная модель описывает возможные связи между объектами, она не описывает способ обработки объектов. Например, даже при том, что на схеме видно, что заказ включает один или более товаров, она ничего не сообщает о том, когда формируется связь между заказом и товаром. Рассуждая логически, нетрудно сделать вывод, что товар в заказе был выигран покупателем, предложившим самую высокую цену. Соответственно связь, скорее всего, устанавливается прикладной логикой после окончания торгов и когда победитель подтвердит желание совершить покупку. Далее мы посмотрим, как эту предметную модель превратить в классы Java.

Объекты предметной области в виде классов Java

Теперь перейдем к исследованию кода, использующего JPA. Но перед этим посмотрим на самые обычный объект Java, представляющий относительно сложный объект предметной области, `Category`. Определение класса `Category` приводится в листинге 9.1. Как можно заметить – это самый обычный POJO, без каких-либо аннотаций JPA. Обычно так начинается реализация предметной модели. POJO – это только *кандидат* на превращение в сущность, хранимую в базе данных.

Листинг 9.1. Category – объект предметной области на языке Java

```
package com.actionbazaar.listing01;
import java.sql.Date;

// Класс Category - обычный POJO
public class Category {

    // Атрибут id уникально идентифицирует экземпляр Category
    protected Long id;

    // Следующие два атрибуты хранят сведения о категории
    protected String name;
    protected Date modificationDate;

    // Следующие три атрибуты определяют
    // отношения категории с другими объектами
    protected Set<Item> items;
    protected Category parentCategory;
    protected Set<Category> subCategories;

    // Конструктор без аргументов
    public Category() {
    }

    public Long getId() {
        return this.id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return this.name;
    }

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

    public Date getModificationDate() {
        return this.modificationDate;
    }

    public void setModificationDate(Date modificationDate) {
```

```
        this.modificationDate = modificationDate;
    }

    public Set<Item> getItems() {
        return this.items;
    }

    public void setItems(Set<Item> items) {
        this.items = items;
    }

    public Set<Category> getSubCategories() {
        return this.subCategories;
    }

    public void setSubCategories(Set<Category> subCategories) {
        this.subCategories = subCategories;
    }

    public Category getParentCategory() {
        return this.parentCategory;
    }

    public void setParentCategory(Category parentCategory) {
        this.parentCategory = parentCategory;
    }
}
```

Класс `Category` имеет несколько защищенных (`protected`) переменных экземпляра, для каждой из которых имеются методы чтения и записи, соответствующие соглашениям об именовании, принятым в JavaBeans. Свойства, кроме `name` и `modificationDate`, играют особую роль в предметном моделировании. Свойство `id` используется для хранения уникального номера, идентифицирующего категорию. Свойство `items` хранит все товары, относящиеся к данной категории, и представляет отношение «многие ко многим» между элементами и категориями. Свойство `parentCategory` представляет самоссылочное (self-referential) отношение «многие к одному» между родительской и дочерней категориями. Свойство `subCategories` представляет отношение «один ко многим» между категорией и подкатегориями.

> **JavaBeans.** *Согласно правилам, все классы JavaBeans имеют конструктор без аргументов и защищенные или приватные переменные экземпляров, доступные только посредством методов с именами, следующими шаблону* `getXX` *и* `setXX`*, как показано в листинге 9.1, где* `XX` *– имя свойства (переменной экземпляра).*

Класс `Category`, как видно из листинга 9.1, является вполне приемлемой реализацией объекта предметной области на языке Java. Проблема только в том, что он все еще остается простым POJO. Пока в классе `Category` нет ничего, указывающего на то, что он управляется JPA. В нем также нет ничего, что сообщало бы интерфейсу JPA, как следует управлять классом `Category`. Например, в свойстве `id` нет

ничего такого, что говорило бы о том, что оно хранит уникальный идентификатор экземпляра `Category`. Кроме того, свойства отношений (`items`, `subCategories`) не определяют ни направлений, ни множественности этих отношений. Далее мы решим некоторых из этих проблем, добавив в POJO аннотации JPA, и начнем с идентификации класса `Category`, как объекта предметной области.

9.3. Реализация объектов предметной области с помощью JPA

В предыдущих разделах вы познакомились с основными понятиями предметного моделирования и идентифицировали части предметной модели приложения ActionBazaar. В этом разделе вы увидите некоторые аннотации JPA в действии, по мере реализации модели. В первую очередь мы познакомимся с аннотацией `@Entity`, превращающей любой POJO в сущность, управляемую механизмом JPA. Затем мы расскажем вам об отличиях полей и свойств с точки зрения хранения данных и об идентичности сущностей. В заключение мы обсудим встроенные объекты.

9.3.1. Аннотация @Entity

Аннотация `@Entity` отмечает простой объект POJO предметной области как сущность, управляемую механизмом JPA. Ее можно считать своеобразным аналогом аннотаций `@Stateless`, `@Stateful` и `@MessageDriven`. Ниже показано, как превратить класс `Category` в сущность:

```
@Entity
public class Category {
    //...
    public Category() { /**/ }
    public Category(String name) { /**/ }
    //...
}
```

Вот и все! Благодаря аннотации `@Entity`, JPA теперь будет знать, что это – сущность, которая будет участвовать во взаимодействиях с базой данных. Как видно из этого фрагмента, все неабстрактные сущности должны иметь либо общедоступный (`public`), либо защищенный (`protected`) конструктор без аргументов. Этот конструктор будет использоваться механизмом JPA для создания новых экземпляров сущностей – вам никогда не придется создавать эти экземпляры вручную при получении данных средствами JPA.

Сильной стороной JPA является сохранение поддержки всех особенностей объектно-ориентированного программирования для сущностей, таких как наследование и полиморфизм, которые по сути продолжают оставаться простыми Java-объектами. Сущность может наследовать другую сущность или даже класс, не являющийся сущностью. Например, на рис. 9.4 показано отличное решение – классы `Seller` и `Bidder` объектов предметной области наследуют общий

класс `User` (при этом класс `User` необязательно должен быть отмечен аннотацией `@Entity`). В листинге 9.2 приводится объявление родительского класса `User`, отмеченного как сущность.

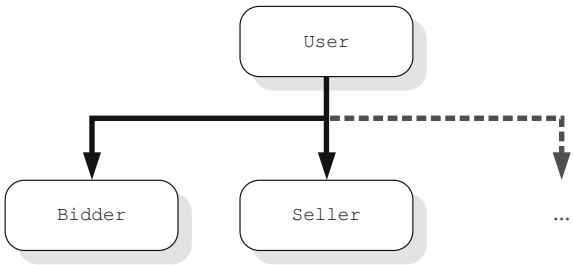

Рис. 9.4. Сущности поддерживают наследование. Сущности Bidder и Seller наследуют класс User

Листинг 9.2. Сущность User

```
@Entity
public abstract class User {
    // ...
    String userId;
    String username;
    String email;
    byte[] picture;
    // ...
}

@Entity
public class Seller extends User { /**/ }

@Entity
public class Bidder extends User { /**/ }
```

Все поля класса `User` (`userId`, `username`, `email`) будут сохраняться при сохранении сущности `Seller` или `Bidder`. Все было бы иначе, если бы класс `User` не был отмечен как сущность, – в этом случае значения унаследованных свойств просто отбрасывались бы при сохранении экземпляров `Seller` и `Bidder`. Этот листинг демонстрирует также один любопытный недостаток – экземпляры класса `User` также могут сохраняться в базе данных, что не всегда желательно. Решить эту проблему можно, объявив класс `User` абстрактным – абстрактные сущности допустимы, но они не могут использоваться для создания экземпляров.

Конечной целью механизма сохранения является сохранение свойств сущности в базе данных. Аннотация `@Entity` – это только начало. Согласно принципу предпочтения соглашений перед настройками, действующему в Java EE, такого объявления может быть вполне достаточно. Однако в JPA имеется еще множество аннотаций, помогающих определить, как данные должны сохраняться в базу данных. Далее мы посмотрим, как JPA определяет, какая таблица в базе данных должна хранить данные из сущности.

9.3.2. Определение таблиц

Данные предметной модели хранятся в множестве таблиц в базе данных. Некоторые сущности легко отображаются в единственную таблицу. Более сложные сущности могут храниться в двух и более таблицах.

Важно помнить, что ведя речь о работе с несколькими таблицами, *в данном случае* мы подразумеваем данные для единственной сущности, хранящиеся в нескольких таблицах. Возьмем в качестве примера таблицу пользователей. В одной таблице может храниться основная информация о пользователях, а картинки, связанные с пользователями, – в другой. Это пример того, как данные о пользователях могут храниться в нескольких таблицах. Именно об этом мы и будем говорить в данном разделе.

О чем не будет рассказываться в этом разделе, так это о работе с несколькими таблицами в контексте отношений между сущностями. Примером может служить информация о пользователях и их адреса. С одним пользователем может быть связано множество адресов. Обсуждение отношений между сущностями мы начнем рассматривать в разделе 9.4.

Отображение сущности в единственную таблицу

Аннотация `@Table` определяет таблицу, в столбцы которой отображается сущность. Параметр `name` является наиболее важным. По умолчанию все сохраняемые данные из сущности отображаются в таблицу с именем, указанным в параметре `name` аннотации. Как видно из определения аннотации, она имеет еще ряд параметров:

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface Table {
    String name() default "";
    String catalog() default "";
    String schema() default "";
    Index[] indexes() default {};
    UniqueConstraint[] uniqueConstraints() default {};
}
```

Аннотация `@Table` является необязательной. По умолчанию JPA будет предполагать, что имя таблицы совпадает с именем класса сущности, отмеченного аннотацией `@Entity`. В случае с классом `Category` из листинга 9.1, JPA будет предполагать, что соответствующая таблица в базе данных так же называется `Category`. Аннотация `@Table` позволяет определить другое имя таблицы. Например, если в базе данных таблица называется `ITEM_CATEGORY`, с помощью аннотации `@Table` можно настроить JPA на использование этого имени:

```
@Table(name="ITEM_CATEGORY")
public class Category
```

Параметры `catalog` и `schema` предусмотрены на тот случай, если потребуется определить местоположение таблицы в базе данных. Схемы и каталоги – вполн-

не обычные особенности баз данных и мы не будем обсуждать их в дальнейшем. Но примера ради предположим, что таблица `ITEM_CATEGORY` находится в схеме `ACTIONBAZAAR`. В этом случае схему можно было бы указать, как показано ниже:

```
@Table(name="ITEM_CATEGORY", schema="ACTIONBAZAAR"))
public class Category
```

Параметры `catalog` и `schema` редко используются на практике, потому что эти тонкости обычно указываются в настройках источников данных сервера Java EE. Более того, считается плохой практикой, когда в коде определяются подобные детали, касающиеся баз данных, потому что любые изменения в них потребуют выполнить пересборку кода вместо перенастройки сервера EE.

Параметр `uniqueConstraints` также используется достаточно редко. Кроме того, нет никаких гарантий, что он будет использоваться вашей реализацией JPA. Большинство реализаций поддерживают весьма дружественную для разработчика особенность, известную как автоматическое создание схемы, когда реализация автоматически создает для сущностей отсутствующие объекты базы данных. Это поведение не обозначено в спецификации JPA и в разных реализациях настраивается по-разному. Параметр `uniqueConstraints` сообщает реализации, какие столбцы в автоматически созданной таблице должны иметь ограничение уникальности. Но напомним еще раз, что нет никаких гарантий наличия поддержки автоматического создания таблиц, а даже если она и имеется, нет никаких гарантий, что параметр `uniqueConstraints` будет оказывать на нее влияние.

За исключением этапа разработки, для быстрого создания прототипов и модульного тестирования, использование поддержки автоматического создания таблиц практически всегда считается неудачным решением. Но, если вы работаете над открытым проектом, такая поддержка может позволить членам сообщества быстро установить и запустить ваше приложение, чтобы оценить его возможности. Ваше приложение может настраивать JPA (с применением файла `persistence.xml`, о котором рассказывается ниже) на использование источника данных по умолчанию в сервере EE. Это позволит другим развернуть ваше приложение на своих серверах EE практически без вмешательства в настройки. Если вы предпочтете использовать источник данных по умолчанию, ожидайте получения просьб написать инструкцию по настройке вашего приложения в разных серверах EE.

Источник данных по умолчанию

В спецификации Java EE 7 наконец-то был стандартизован адрес источника данных для сервера EE в каталоге JNDI. Он определен, как:

java:comp/DefaultDataSource

Этот источник данных легко можно получить с помощью аннотации `@Resource`:

```
@Resource(lookup="java:comp/DefaultDataSource")
DataSource defaultDs;
```

С помощью аннотации `@Table` можно определить только одну таблицу. Но в реляционных базах данных информация часто хранится в нескольких таблицах. Далее мы посмотрим, как с помощью аннотаций `@SecondaryTable` определить две или более таблиц.

Отображение сущности в множество таблиц

Аннотации `@SecondaryTables` (во множественном числе) и `@SecondaryTable` (в единственном числе) позволяют определять сущности, которые должны храниться в двух или более таблицах. В некоторых редких ситуациях это вполне оправданно. Вернемся к сущности `User` из листинга 9.2. Она содержит свойство `byte[] picture`. Администратор баз данных в компании ActionBazaar решил хранить данные в двух таблицах, `USERS` и `USER_PICTURES`, как показано на рис. 9.5.

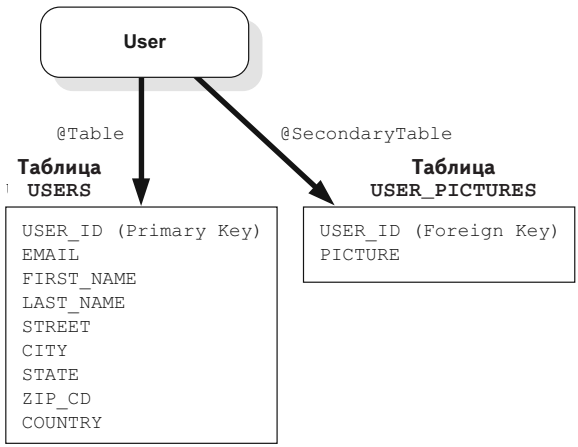

Рис. 9.5. Хранение информации о пользователях в двух таблицах

Это вполне оправданное решение, потому что хранение больших двоичных изображений может существенно замедлять скорость выполнения запросов к таблице. Аннотация `@SecondaryTable`, определение которой приводится ниже, позволяет получать данные для сущности более чем из одной таблицы:

```
@Target({ TYPE })
@Retention(RUNTIME)
public @interface SecondaryTable {
    String name();
    String catalog() default "";
    String schema() default "";
    PrimaryKeyJoinColumn[] pkJoinColumns() default {};
    UniqueConstraint[] uniqueConstraints() default {};
}
```

Обратите внимание, что за исключением элемента `pkJoinColumns`, определение этой аннотации идентично определению аннотации `@Table`. Элемент `pkJoinColumns` является ключевой особенностью данной аннотации. Для примера исследуем следующее объявление сущности `User`, отображаемой в две таблицы:

```
@Entity
@Table(name="USERS")
@SecondaryTable(name="USER_PICTURES",
    pkJoinColumns=@PrimaryKeyJoinColumn(name="USER_ID"))
public class User
```

Параметр `pkJoinColumns=@PrimaryKeyJoinColumn(name="USER_ID")` сообщает JPA, что столбец `USER_PICTURES.USER_ID` является внешним ключом, связанным с одноименным столбцом, являющимся первичным ключом в таблице `USERS` (о настройке первичного ключа в таблице `USERS` рассказывается в следующем разделе, в описании аннотации `@Id`). Для извлечения сущности `User` из двух таблиц JPA будет использовать операцию соединения. В этом примере участвуют только две таблицы. Если потребуется организовать хранение сущностей более чем в двух таблицах, можно воспользоваться аннотацией `@SecondaryTables` (во множественном числе).

У многих из вас может возникнуть законный вопрос, касающийся аннотации `@SecondaryTable`: как JPA узнает, какие свойства объекта Java должны хранятся в других таблицах? В конце концов, в определении сущности `User` (листинг 9.2) нет ничего, что говорило бы о том, что данные для свойства `picture` хранятся в отдельной таблице. Чтобы ответить на этот вопрос, нужно разобраться с тем, как JPA отображает свойства в столбцы таблиц.

9.3.3. Отображение свойств в столбцы

В предыдущих разделах мы узнали, как выполняется отображение сущностей в таблицы базы данных. Теперь пришла пора посмотреть, как выполняется отображение свойств сущностей в столбцы таблиц. Отображение свойства хранимого объекта в столбец таблицы осуществляется с помощью аннотации `@Column`. Сначала мы познакомимся с основами использования аннотации `@Column`. Затем рассмотрим некоторые дополнительные особенности, появившиеся в версии JPA 2.0, упрощающие отображение свойств в столбцы. И в заключение обсудим подробнее обсудим переходные поля (transient fields).

Аннотация @Column

Для начала взгляните на листинг 9.3, где приводится пример отображения некоторых свойств сущности `User`.

Листинг 9.3. Отображение свойств сущности User

```
@Entity
@Table(name="USERS")
public class User {
    // ❶ Свойство userId отображается в столбец USER_ID
    @Column(name="USER_ID")
    String userId;

    // ❷ Свойство username отображается в столбец USER_NAME
    @Column(name="USER_NAME")
```

```
    String username;

    // ❸ Свойство email отображается в столбец EMAIL,
    //   поэтому нет необходимости использовать аннотацию @Column
    String email;

    //...Другие методы доступа опущены для краткости
    // ❹ Методы доступа к свойству email
    //   с именами, соответствующими соглашениям для JavaBean
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
}
```

В этом листинге демонстрируется использование аннотации `@Column` для определения правил отображения свойств в столбцы таблицы `USERS`. Например, для свойства `userId` предполагается, что в таблице `USERS` имеется столбец `USER_ID`, в котором будет храниться значение свойства `userId` ❶. Аналогично определяется отображение свойства `username` – JPA будет сохранять значение этого свойства в столбце `USER_NAME` таблицы `USERS` ❷.

Свойство `email` выбивается из общего ряда. Оно не имеет соответствующей аннотации `@Column` ❸. Как же тогда JPA узнает, в какой столбец отображается это свойство? Помните о преимуществе соглашений перед настройками в Java EE? В соответствии с соглашениями, JPA отображает все свойства объектов Java, имеющие общедоступные или защищенные методы доступа в стиле JavaBeans, в одноименные столбцы. Свойство `email` имеет стандартные методы доступа, поэтому по умолчанию JPA автоматически будет пытаться сохранять его значение в одноименном столбце ❹. Так как свойство `email` имеет имя «email», будет предполагаться, что в таблице `USERS` имеется столбец `EMAIL`. Благодаря принципу предпочтения соглашений перед настройками, если имена всех свойств объекта совпадают с именами соответствующих им столбцов, аннотацию `@Column` можно вообще не использовать.

В листинге 9.3 демонстрируется самый простой случай применения аннотации `@Column`. Остальные ее атрибуты можно увидеть в листинге 9.4.

Листинг 9.4. Атрибуты 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;
    int scale() default 0;
}
```

Как видите, аннотация `@Column` имеет множество других атрибутов, но все они редко используются на практике. Наиболее полезными из них являются атрибуты `insertable` и `updatable`. Если атрибуту `insertable` присвоить значение `false`, JPA не будет включать свойство объекта Java в SQL-инструкцию `INSERT` при сохранении новых данных. Аналогично, если атрибуту `updatable` присвоить значение `false`, JPA не будет включать свойство в SQL-инструкцию `UPDATE` при обновлении хранящихся данных. Почему может возникнуть желание отказаться от сохранения некоторых данных? На то может быть множество причин, но чаще всего это обусловлено тем, что такие данные генерируются самой базой данных, обычно каким-либо триггером. Хорошим примером может служить автоматически генерируемый первичный ключ.

Остальные параметры (`unique`, `nullable`, `length`, `precision`, `scale`, `table` и `columnDefinition`) используются реализациями JPA для поддержки автоматического определения схемы и создания таблиц в базе данных, если они отсутствуют. Как уже отмечалось в разделе 9.3.2, использование этой возможности практически всегда не лучшее решение, поэтому мы оставляем эти атрибуты вам для самостоятельного исследования.

После изучения листинга 9.3 у кого-то из вас может возникнуть вопрос: почему аннотации `@Column` находятся непосредственно перед свойствами объекта `User`? Они должны располагаться именно перед свойствами? Можно ли размещать аннотации перед методами чтения или записи? Мы обсудим эти вопросы после того, как разберемся в различиях между доступом к данным на основе полей и свойств.

Доступ к данным на основе полей и свойств

Интерфейс JPA дает определенную свободу размещения аннотации `@Column`. Наиболее очевидный выбор – перед переменными экземпляров. Такой способ называется доступом к данным на основе полей, например:

```
@Column(name="USER_ID")
private Long id;
```

Когда аннотация размещается перед полем, механизм JPA напрямую обращается к приватной переменной. Это может быть допустимо, если объект предметной области является простым компонентом с методами доступа, не выполняющими никаких промежуточных операций с данными. Другой вариант – размещение аннотаций перед методами чтения (аннотации перед методами записи игнорируются). Такой способ называется доступом к данным на основе свойств, например:

```
@Column(name="USER_ID")
public Long getId() { return id; }
public Long setId(Long id) {
```

```
    this.id = (id <= 0) ? 0 : id;
}
```

Обратите внимание, что в этом примере аннотация находится перед методом чтения и метод записи включает некоторую прикладную логику. Когда используется доступ к данным на основе свойств, JPA получает значение вызовом метода чтения и устанавливает новое значение вызовом метода записи. Если метод чтения и/или метод записи содержит некоторую прикладную логику, следует использовать способ доступа к данным на основе свойств, чтобы механизм JPA использовал методы чтения и записи вместо прямого доступа к приватной переменной.

Версия JPA 1.0 позволяла использовать доступ к данным либо только на основе полей, либо только на основе свойств, не давая возможности смешивать их в пределах иерархии объектов. Например, в приложении ActionBazaar имеется суперкласс `User` с дочерними классами `Seller` и `Bidder`. В версии JPA 1.0 во всех трех объектах (`User`, `Seller`, `Bidder`) можно было использовать доступ к данным либо только на основе полей, либо только на основе свойств. Положение дел изменилось в версии JPA 2.0, с появлением аннотации `@Access`.

Сначала с помощью аннотации `@Access` на уровне класса необходимо определить тип доступа по умолчанию для всей сущности. Тип доступа может быть определен как `AccessType.FIELD` или `AccessType.PROPERTY`. Ниже приводится пример настройки типа доступа `FIELD` для сущности `User`:

```
@Entity
@Access(FIELD)
public class User { ... }
```

Теперь можно определить сущность `Seller` с некоторыми особенностями:

```
@Entity
@Access(FIELD)
public class Seller extends User {
    //...остальная часть определения класса опущена для краткости

    @Transient
    private double creditWorth;

    @Column(name="CREDIT_WORTH")
    @Access(AccessType.PROPERTY)
    public double getCreditWorth() { return creditWorth; }
    public void setCreditWorth(double cw) {
        creditWorth = (cw <= 0) ? 50.0 : cw;
    }
}
```

Обратите внимание, что с помощью аннотации `@Access` для всего класса `Seller` установлен способ доступа к данным `FIELD`. Но в нем дополнительно используются аннотации `@Transient` и `@Access`, определяющие доступ к `creditWorth`, как к свойству. То есть, доступ ко всем данным в `Seller` будет осуществляться прямым обращением к полям, кроме `creditWorth`. Когда требуется выполнить подобное переопределение, всегда следует использовать пару аннотаций, `@Transient` и

@Access. Это предотвратит попытку механизма JPA отобразить creditWorth дважды.

В предыдущем примере для класса Seller установлен способ доступа к данным на основе полей и для поля creditWorth отдельно установлен способ доступа как к свойству. Однако легко можно определить и противоположное поведение. Для всего класса Seller можно установить способ доступа PROPERTY, а для creditWorth установить способ доступа как к полю. Например:

```
@Entity
@Access(PROPERTY)
public class Seller extends User {
    //...остальная часть определения класса опущена для краткости

    @Column(name="CREDIT_WORTH")
    @Access(Accesss.TypeFIELD)
    private double creditWorth;

    @Transient
    public double getCreditWorth() { return creditWorth; }
    public void setCreditWorth(double cw) {
        creditWorth = (cw <= 0) ? 50.0 : cw;
    }
}
```

Аннотация @Access стала отличным дополнением к остальным особенностям JPA, и сняла ограничение, требующее, чтобы для всех объектов в иерархии предметной модели использовался какой-то один способ доступа, на основе полей или на основе свойств. Она позволяет смешивать два разных способа доступа. Это особенно удобно, когда объекты в иерархии предметной модели являются частью нескольких проектов, которые не так легко изменить.

В примерах выше использована новая для нас аннотация @Transient. Это еще одна часто используемая аннотация JPA, препятствующая механизму JPA управлять элементами, которые помечены ею. Описание аннотации @Transient приводится в следующем разделе.

Определение переходных полей

Коль скоро JPA автоматически пытается сохранять свойства объектов Java с использованием стандартных методов доступа JavaBeans, возможно ли определить свойство, которое не будет сохраняться механизмом JPA? Да, возможно. В JPA имеется аннотация @Transient, которая сообщает, что отмеченное ею свойство должно игнорироваться. Если свойство отмечено аннотацией @Transient, JPA не будет пытаться извлекать, вставлять или обновлять соответствующее значение в базе данных. Представьте, что в объект User были добавлены свойства birthday (дата рождения) и age (возраст), как показано ниже:

```
public class User {
    @Column(name="DATE_OF_BIRTH")
    Date birthday;
```

```
    @Transient
    int age;
    //...остальная часть определения класса опущена для краткости
    public void setBirthday(Date birthday) {
        this.birthday = birthday;
        // вычислить возраст...
    }
}
```

Метод `setBirthday()` автоматически вычисляет возраст (`age`). В приложении это свойство играет вспомогательную роль, и нет никакого смысла хранить его значение в базе данных, поэтому оно отмечено аннотацией `@Transient`.

Аннотация `@Column` в самом начале указывает, в какой столбец таблицы отображается свойство объекта. Однако в JPA имеется еще несколько аннотаций, позволяющих определять особые данные, такие как первичные ключи, даты, отметки времени (timestamps), коды и двоичные данные. Примеры этих данных мы рассмотрим далее.

9.3.4. Типы представления времени

Типы представления времени используются для представления дат и времени суток. Большинство баз данных поддерживает несколько разных типов представления времени с разной степенью точности, таких как `DATE` (хранит день, месяц и год), `TIME` (хранит только время суток, но не хранит день, месяц и год) и `TIMESTAMP` (хранит день, месяц, год и время суток). Аннотация `@Temporal` помогает указать, какой из этих типов следует использовать.

JPA может отображать значения из базы данных в свойства типа `java.util.Date` или `java.util.Calendar`. При записи значений в базу данных JPA будет использовать только требуемую часть значения свойства. То есть, если в базе данных определено, что столбец хранит только время суток, JPA извлечет из свойства типа `java.util.Date` или `java.util.Calendar` только время суток и отбросит дату.

Кроме того, JPA может также отображать Java-типы `java.sql.Date`, `java.sql.Time` и `java.sql.Timestamp`. Использование их с аннотацией `@Temporal` выглядит несколько избыточным, потому что JPA автоматически выводит типы столбцов в базе данных, опираясь на типы Java. Однако избыточность в данном случае не повредит и применение аннотации `@Temporal` поможет внести дополнительную ясность в программный код.

Для примера представьте, что требуется сохранить дату записи данных в базу без времени суток. Реализовать это можно любым из следующих способов:

```
@Temporal(TemporalType.DATE)
protected java.util.Date creationDate;

@Temporal(TemporalType.DATE)
protected java.util.Calendar creationDate;

// Без аннотации
protected javax.sql.Date creationDate;
```

9.3.5. Перечисления

Взгляните еще раз на рис. 9.4 и представьте, что модель данных включает перечисление с типами пользователей приложения. Определение этого перечисления может выглядеть так:

```
public enum UserType { SELLER, BIDDER, CSR, ADMIN }
```

Так как в реляционной базе данных отсутствует понятие объекта или иерархии типов, используя это перечисление можно организовать хранение в таблице информации о типе пользователя. В реляционных базах данных также отсутствует понятие перечислений (`enum`), поэтому `UserType` нужно преобразовать в что-то более понятное базе данных. В этом вам поможет аннотация `@Enumerated`. Ниже показаны примеры использования аннотации `@Enumerated`:

```
@Enumerated(EnumType.STRING)
protected UserType userType1;

@Enumerated(EnumType.ORDINAL)
protected UserType userType2;
```

Различие между этими двумя примерами заключается в том, что в первом из них используется параметр `EnumType.STRING`, а во втором – параметр `EnumType.ORDINAL`. Тип `EnumType` управляет представлением перечисления в базе данных.

Каждый элемент перечисления имеет строковое представление. Так, значение `UserType.SELLER` будет представлено строкой `SELLER`, `UserType.BIDDER` – строкой `BIDDER`, и так далее. Соответственно, значение свойства, отмеченного аннотацией `@Enumerated(EnumType.STRING)`, будет сохраняться в базе данных в строковом представлении.

Кроме того, каждый элемент перечисления имеет числовой индекс. Так, для перечисления, объявленного выше, значению `UserType.SELLER` соответствует индекс 0, значение которого можно получить обращением к перечислению по индексу: `UserType.values()[0];`. Аналогично, значению `UserType.BIDDER` соответствует индекс 1, значению `UserType.CSR` – индекс 2, и так далее. Соответственно, значение свойства, отмеченного аннотацией `@Enumerated(EnumType.ORDINAL)`, будет сохраняться в базе данных в числовом (порядковом) представлении.

По умолчанию JPA использует порядковые значения. То есть, по умолчанию значение `UserType.ADMIN` будет сохранено как число 3, а не как строка `"ADMIN"`. Важно так же помнить, что после сохранения данных в базе они никак не будут связаны с данными в коде. Это означает, что если изменить определение перечисления в коде, это никак не отразится на значениях, хранящихся в базе. Изменяя определение перечисления в коде, вы рискуете столкнуться с ситуацией, когда код будет неправильно интерпретировать данные, хранящиеся в базе, или вообще не сможет с ними работать. Если сохранить значения перечисления в порядковом виде и затем изменить порядок следования элементов в перечислении, типы сразу всех учетных записей пользователей будут интерпретироваться неправильно. Если сохранить значения перечисления в строковом виде и затем

изменить имя какого-нибудь элемента перечисления (например, BIDDER заменить на BUYER), при попытке прочитать данные возникнет исключение времени выполнения, потому что значение BIDDER, имеющееся в базе данных, больше не определено в коде.

9.3.6. Коллекции

Итак, к настоящему моменту вы узнали, как отображаются основные типы единичных значений в столбцы таблиц базы данных. Но в предметной модели имеются также коллекции. Например, пользователи приложения ActionBazaar могут иметь несколько телефонных номеров. В листинге 9.5 приводится дополненная версия сущности User, содержащая коллекцию номеров телефонов.

Листинг 9.5. Определение сущности User со списком телефонных номеров

```
@Entity
@Table(name="USERS")
public class User {
    private Collection<String> telephoneNumbers;
    //...
}
```

В этом листинге имеется коллекция объектов String, предназначенных для хранения номеров телефонов. Теперь нужно сообщить механизму JPA, как он должен извлекать эти данные из базы. В этом нам поможет аннотация @ElementCollection. Для отображения коллекций простых (таких как java.lang.String, java.util.Integer, и так далее) или встраиваемых типов (подробнее о встраиваемых типах рассказывается в описании аннотации @EmbeddedId, в разделе 9.3.7) в JPA 2.0 была реализована аннотация @ElementCollection. В версиях, предшествовавших JPA 2.0, также имелась возможность отобразить коллекцию объектов, как будет показано в разделе 9.4, но аннотация @ElementCollection делает решение этой задачи намного проще. В листинге 9.6 приводится дополненная версия сущности User, осуществляющая отображение коллекции номеров.

Листинг 9.6. Отображение коллекции телефонных номеров

```
@Entity
@Table(name="USERS")
public class User {
    // ❶ Используется для определения коллекций
    @ElementCollection
    // ❷ Имя таблицы для хранения коллекции можно переопределить,
    //    если имя по умолчанию не подходит
    @CollectionTable(name="PHONE_NUMBERS",
        // ❸ Имя столбца в таблице для хранения коллекции так же
        //    можно переопределить, если имя по умолчанию не подходит
        joinColumns=@JoinColumn(name="USER_ID"))
    @Column(name="NUMBER")
    private Collection<String> telephoneNumbers;
```

```
    //...
}
```

Аннотация `@ElementCollection` определяет таблицу для хранения коллекции в базе данных и используется в сущностях для обозначения свойств-коллекций ❶. Таблицы, предназначенные для хранения коллекций, это самые обычные таблицы в базе данных. Имена таких таблиц по умолчанию конструируются из имен сущностей и свойств. В соответствии с определением в листинге 9.6, таблица получила бы имя по умолчанию `USER_TELEPHONENUMBER`, но в аннотации `@CollectionTable` явно указано, что таблица должна иметь имя `"PHONE_NUMBERS"` ❷. В качестве имени столбца в таблице коллекции по умолчанию выбирается имя свойства. В данном случае столбец получил бы имя по умолчанию `TELEPHONENUMBER`. Но в аннотации `@Column` явно указано, что столбец должен иметь имя `"NUMBER"` ❸. И, наконец, атрибут `joinColumns` сообщает, что столбец `PHONE_NUMBERS.USER_ID` играет роль внешнего ключа, ссылающегося на первичный ключ таблицы `USERS`.

В листинге 9.6 демонстрируется пример использования коллекции строк. Но имейте в виду, что аннотация `@ElementCollection` может также работать со встраиваемыми объектами. Подробнее о встраиваемых объектах рассказывается в разделе 9.3.7, и все же, давайте посмотрим, как выглядят встраиваемые объекты. Ниже приводится пример встраиваемого объекта `Address`:

```
@Embeddable
public class Address {
    @Column(name="HOME_STREET")
    private String street;
    @Column(name="HOME_CITY")
    private String city;
    @Column(name="HOME_STATE")
    private String state;
    @Column(name="HOME_ZIP")
    private String zipCode;
}
```

Сущность `User` может включать свойство, представляющее коллекцию объектов `Address`:

```
@Entity
@Table(name="USERS")
public class User {
    @ElementCollection
    @CollectionTable(name="HOMES")
    private Set<Address> shippingAddresses;
}
```

При работе с коллекциями особую важность приобретают первичные ключи в базе данных. Именно с помощью первичных ключей формируются отношения между таблицами. Используя эти отношения, механизм JPA автоматически извлекает необходимые данные из других таблиц в свойства-коллекции. Далее мы посмотрим, как настроить идентификацию первичных ключей в JPA.

9.3.7. Определение идентичности сущностей

Под словами «определение идентичности сущностей» в действительности подразумевается настройка идентификации первичных ключей. Все таблицы в базе данных должны поддерживать возможность уникальной идентификации строк, чтобы при необходимости изменить данные, изменениям подверглась только одна строка, а остальные оставались нетронутыми. Аналогично, когда механизм JPA извлекает данные из базы данных и преобразует их в объекты предметной модели, ему необходим некоторой способ уникальной идентификации объектов, находящихся в памяти. Для этого требуется определить, какие столбцы таблиц играют роль первичных ключей. Решить эту задачу можно тремя способами:

- с помощью аннотации `@Id`;
- с помощью аннотации `@IdClass`;
- с помощью аннотации `@EmbeddedId`.

Аннотация @Id

Когда первичный ключ таблицы состоит из единственного столбца, можно воспользоваться аннотацией `@Id`. Вернемся к определению объекта `Category` из листинга 9.1 и предположим, что свойство `id` хранит значение столбца первичного ключа таблицы `CATEGORY`. В этом случае достаточно просто отметить это свойство аннотацией `@Id`, чтобы сообщить JPA, что это значение можно использовать для уникальной идентификации объектов `Category`. Как показано в следующем примере, аннотацию можно применить к свойству:

```
@Id
private long id;
```

или к методу чтения:

```
@Id
public long getId() { return id; }
```

Значение свойства, отмеченного аннотацией `@Id`, будет использоваться механизмом JPA для уникальной идентификации объектов `Category`. То есть, если механизму JPA потребуется выяснить, являются ли два объекта `Category` представлением одной и той же категории, ему достаточно будет просто сравнить значения свойств, отмеченных аннотацией `@Id`. Аннотация `@Id` может применяться к свойствам простых типов (`int`, `long`, `double`, и так далее), в этом случае значения будут сравниваться непосредственно, и к свойствам-объектам, реализующим интерфейс `Serializable` (`String`, `Date`, и так далее), в этом случае JPA будет вызывать метод `equals()`.

Важно помнить, что аннотация `@Id` может использоваться, только если первичный ключ таблицы состоит из единственного столбца. В большинстве современных проектов с этой целью в таблицу добавляется отдельный столбец. Но в унаследованных проектах первичный ключ может состоять из нескольких столбцов. В JPA поддерживается два способа определения первичных ключей, состоящих из нескольких столбцов, которые описываются далее.

Аннотация @IdClass

Первый способ определения первичных ключей из нескольких столбцов реализован в виде аннотации `@IdClass`. Эту аннотацию можно применить к нескольким свойствам объекта, в паре с аннотацией `@Id` (столбцам в таблице, составляющим первичный ключ) и затем определить класс, следующий шаблону компаратора и определяющий порядок сравнения нескольких значений. Давайте посмотрим, как это выглядит на практике. Допустим, что таблица `CATEGORY` является частью унаследованного проекта и в качестве первичного ключа в ней используются название категории и дата создания. Как показано в листинге 9.7, первое, что следует сделать, – определить в классе `Category` два свойства и отметить их аннотацией `@Id`, чтобы механизм JPA знал, что они уникально идентифицируют сущность.

Листинг 9.7. Категория с первичным ключом, состоящим из двух столбцов

```
@Entity
public class Category {
    // ❶ Свойство name является одной частью первичного ключа
    @Id @Column(name="CAT_NAME") private String name;
    // ❷ Свойство createDate является другой частью первичного ключа
    @Id @Column(name="CAT_DATE") private java.util.Date createdDate;
    // ...
}
```

Теперь у нас имеются свойства `name` ❶ и `createdDate` ❷, отмеченные аннотацией `@Id`, сообщающей, что эта пара свойств уникально идентифицирует категорию. Но JPA пока не знает, как с помощью этих свойств проверить равенство двух объектов `Category`. Для этого необходимо определить новый класс, `CategoryKey`, как показано в листинге 9.8.

Листинг 9.8. Класс CategoryKey

```
import java.io.Serializable;
import java.sql.Date;
// ❶ Спецификация JPA требует, чтобы класс реализовал интерфейс Serializable
public class CategoryKey implements Serializable {
    private static final long serialVersionUID = 1775396841L;
    // ❷ Два поля, составляющих первичный ключ
    String name;
    Date createDate;

    // ❸ Обязательный конструктор без аргументов
    public CategoryKey(){}

    // ❹ Обязательный для переопределения метод equals, выполняющий сравнение
    public boolean equals(Object other) {
        if (other instanceof CategoryKey) {
            final CategoryKey otherCategoryKey = (CategoryKey) other;
            return (otherCategoryKey.name.equals(name)
                    &&
                    otherCategoryKey.createDate.equals(createDate));
```

```
        }
        return false;
    }

    // ❺ Обязательный для переопределения метод hashCode, возвращающий хэш-код
    public int hashCode() {
        return super.hashCode();
    }
}
```

Давайте чуть подробнее остановимся на классе `CategoryKey`. Спецификация JPA требует, чтобы подобные классы реализовали интерфейс `Serializable` ❶, что и сделано в данном случае, а также определяется `serialVersionUID`. Поля `name` и `createDate` ❷, составляющие первичный ключ, повторяются в классе `CategoryKey`. Требуется также определить конструктор без аргументов ❸, чтобы дать механизму JPA возможность создать экземпляр `CategoryKey`. Фактическое сравнение свойств `name` и `createDate` выполняет переопределенный метод `equals()` ❹. Наконец, также требуется переопределить метод `hashCode()` ❺.

Мы почти закончили. Последнее, что осталось сделать, указать, что сравнение значений первичного ключа в `Category` должно выполняться с помощью `CategoryKey`. Вернемся к листингу 9.7 и добавим в реализацию `Category` аннотацию `@IdClass`:

```
@Entity
@IdClass(CategoryKey.class)
public class Category {
    // ...
}
```

Итак, как же все это работает? JPA извлекает данные из таблицы `CATEGORY` и создает объекты `Category`, присваивая свойствам `name` и `createdDate` значения из соответствующих столбцов таблицы. Когда наступает момент сравнить два объекта `Category`, создается новый экземпляр класса `CategoryKey` для каждого объекта `Category`, участвующего в сравнении. Затем значения свойств `name` и `createdDate` копируются из объектов `Category` в объекты `CategoryKey`, и вызывается метод `equals()` объекта `CategoryKey`, чтобы проверить равенство объектов `CategoryKey`. Если они оказываются равны, JPA делает вывод, что и соответствующие им объекты `Category` тоже равны.

Это лишь один из способов отображения первичных ключей, состоящих из нескольких столбцов, в предметной модели. Далее описывается другой способ – с применением аннотации `@EmbeddedId`.

Аннотация @EmbeddedId

Второй способ отображения первичного ключа, состоящего из нескольких столбцов, заключается в использовании аннотации `@EmbeddedId`. Применение этой аннотации очень близко напоминает применение аннотации `@IdClass`, но в отличие от нее `@EmbeddedId` используется в паре с аннотацией `@Embeddable` и превращает объект предметной модели в составной объект. Рассмотрим практический пример. Допустим, что таблица `CATEGORY` является частью унаследованного проекта

и в качестве первичного ключа в ней используются название категории и дата создания. Чтобы воспользоваться аннотацией `@EmbeddedId` для отображения первичного ключа, сначала необходимо определить класс `@Embeddable CategoryId`, представляющий первичный ключ.

Листинг 9.9. Класс CategoryId

```
import java.sql.Date;
import javax.persistence.Embeddable;
// ❶ Аннотация @Embeddable позволяет встраивать этот объект в другие объекты
@Embeddable
public class CategoryId {
    // ❷ Два поля, составляющих первичный ключ
    String name;
    Date createDate;

    // ❸ Обязательный конструктор без аргументов
    public CategoryId() {}

    // ❹ Обязательный для переопределения метод equals, выполняющий сравнение
    public boolean equals(Object other) {
        if (other instanceof CategoryId) {
            final CategoryId otherCategoryKey = (CategoryId) other;
            return (otherCategoryKey.name.equals(name)
                    &&
                    otherCategoryKey.createDate.equals(createDate));
        }
        return false;
    }

    // ❺ Обязательный для переопределения метод hashCode, возвращающий хэш-код
    public int hashCode() {
        return super.hashCode();
    }

    public String getName() {
        return name;
    }

    public Date getCreateDate() {
        return createDate;
    }
    // ...
}
```

Как видите, этот листинг практически идентичен листингу 9.8. Наиболее существенным отличием является применение аннотации `@Embeddable` ❶ к классу `CategoryId`. С помощью этой аннотации вы сообщаете механизму JPA, что класс `CategoryId` можно вставлять, или встраивать, в другие классы. Свойства `name` и `createDate` ❷ все так же составляют первичный ключ. Класс `CategoryId` имеет конструктор без аргументов ❸, переопределяет `equals()` ❹, реализующий проверку на равенство, и включает обязательную собственную версию метода `hashCode()` ❺.

Класс CategoryId из листинга 9.9 не особенно отличается от класса CategoryKey из листинга 9.8, но вот класс Category будет совершенно иным. При использовании аннотации @ClassId класс Category имеет отдельные свойства, составляющие первичный ключ в таблице CATEGORY. Однако при использовании аннотации @EmbeddedId дело обстоит совершенно иначе: отдельные свойства name и createDate замещаются классом CategoryId и класс Category превращается в составной класс. Необходимые изменения представлены в листинге 9.10.

Листинг 9.10. Использование аннотации @EmbeddedId в классе Category

```
@Entity
public class Category {
    // ❶ Аннотация @EmbeddedId определяет встроенный
    //    объект как первичный ключ
    @EmbeddedId
    // ❷ Объект CategoryId хранит первичный ключ
    private CategoryId categoryId;
    // ...

    // ❸ Следующие методы чтения, обеспечивают доступ
    //    к данным в первичном ключе
    public String getName() {
        return categoryId.getName();
    }

    public Date getCreateDate() {
        return categoryId.getCreateDate();
    }
}
```

Рассмотрим этот пример внимательнее. Прежде всего, обратите внимание на отсутствие свойств name и createDate. Их заменило свойство categoryId ❶, отмеченное аннотацией @EmbeddedId ❷. В результате получился составной объект Category, который должен создавать экземпляр класса CategoryId для хранения первичного ключа таблицы CATEGORY. Методы чтения в классе Category теперь возвращают значения свойств из встроенного объекта ❸.

Аннотация @Embeddable

Мы познакомились с аннотацией @Embeddable на примере первичных ключей, состоящих из нескольких столбцов, а также узнали, как использовать комбинацию аннотаций @EmbeddedId и @Embeddable для их отображения. Однако важно понимать, что поддержка первичных ключей не является единственным предназначением аннотации @Embeddable. Любой объект можно отметить аннотацией @Embeddable и использовать его в рамках предметной модели. Отличным примером может служить представление адреса. Информацию об адресе могут использовать самые разные объекты предметной модели, и вместо того, чтобы дублировать эти свойства, можно создать объект @Embeddable Address и встраивать его с помощью @Embedded в объекты предметной модели. Мы рекомендуем вам не останавливаться на достигнутом и продолжить самостоятельное исследование аннотаций @Embedded, @Embeddable и @AttributeOverride.

Теперь, когда вы знаете, как отображаются первичные ключи в объектах предметной модели, можно посмотреть, как JPA генерирует значения первичных ключей при вставке новых данных в базу.

9.3.8. Генерирование значений первичных ключей

В предыдущем разделе мы узнали, как реализовать отображение первичных ключей в предметной модели на языке Java. В простейшем случае с этой целью можно использовать аннотацию `@Id`. В более сложных ситуациях для отображения первичных ключей, состоящих из нескольких столбцов, можно использовать аннотацию `@IdClass` или `@EmbeddedId`. Но мы пока не знаем, как генерируются значения первичных ключей.

Для генерации значений первичных ключей обычно используются два способа. Первый: значения первичных ключей образуются естественным образом из фактических данных. Например, для уникальной идентификации строк в таблице с информацией о пользователях вполне достаточно может оказаться имени, фамилии и телефонного номера. То есть, первичный ключ должен состоять из трех столбцов. Второй способ: введение искусственных данных, никак не связанных с хранимой информацией и предназначенных исключительно для уникальной идентификации строк в таблице. С этой целью обычно используются числовые первичные ключи, состоящие из единственного столбца.

Хотя оба подхода имеют свои достоинства и недостатки, тем не менее, подход, основанный на введении искусственных данных, намного проще и удобнее. Существует пять популярных методов генерации значений первичных ключей, поддерживаемых в JPA:

- `AUTO`;
- `IDENTITY`;
- `SEQUENCE`;
- `TABLE`;
- `CODE`.

Далее мы познакомимся с аннотацией `@GeneratedValue` и посмотрим, как с ее помощью организовать создание значений первичных ключей, применяя перечисленные методы.

AUTO

Метод `AUTO` используется по умолчанию и является простейшим случаем использования `@GeneratedValue`. Он освобождает разработчика от необходимости выполнения каких-то особых операций с базой данных и возлагает создание значений первичного ключа на механизмы по умолчанию, настроенные в базе данных. В листинге 9.11 показано, как могло бы выглядеть объявление объекта `User` при использовании метода `AUTO`.

Листинг 9.11. Метод AUTO

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO) // Метод AUTO
    @Column(name="USER_ID")
    protected Long userId;
}
```

IDENTITY

Метод `IDENTITY` основан на использовании столбца специального типа, поддерживающего возможность автоматического увеличения собственного числового значения для уникальной идентификации строк в таблице. Такой тип столбца поддерживается в большинстве баз данных. В Microsoft SQL Server этот тип называется `IDENTITY`; в MySQL – `AUTO_INCREMENT`; в PostgreSQL – `SERIAL`; в Oracle этот тип столбцов отсутствует. Когда данные вставляются в таблицу, значение специального поля автоматически увеличивается и его значение сохраняется как первичный ключ строки. В листинге 9.12 показано, как могло бы выглядеть объявление объекта `User` при использовании метода `IDENTITY`.

Листинг 9.12. Метод IDENTITY

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY) // Метод IDENTITY
    @Column(name="USER_ID")
    protected Long userId;
}
```

SEQUENCE

При использовании метода `SEQUENCE`, для создания уникальных числовых значений используются *последовательности* (sequence) базы данных. Этот метод наиболее популярен в базах данных Oracle, потому что в Oracle отсутствует тип столбцов с автоматически увеличивающимися значениями, как описывалось в предыдущем разделе. Чтобы настроить JPA на использование последовательности в базе данных в качестве генератора значений первичного ключа, сначала необходимо определить последовательность. Следующая инструкция SQL создает последовательность в Oracle:

```
CREATE SEQUENCE USER_SEQUENCE START WITH 1 INCREMENT BY 10;
```

Затем указать используемую последовательность с помощью аннотации `@SequenceGenerator` и задействовать эту последовательность в аннотации `@GeneratedValue`. В листинге 9.13 показано, как могло бы выглядеть объявление объекта `User` при использовании метода `SEQUENCE`.

Листинг 9.13. Метод SEQUENCE

```
@Entity
// ❶ Внутреннее имя последовательности для JPA
@SequenceGenerator(name="USER_SEQUENCE_GENERATOR",
        // ❷ Имя последовательности в базе данных
        sequenceName="USER_SEQUENCE", initialValue=1, allocationSize=10)
public class User {
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, // ❸ Метод SEQUENCE
            // ❹ Внутреннее имя используемой последовательности
            generator="USER_SEQUENCE_GENERATOR")
    @Column(name="USER_ID")
    protected Long userId;
}
```

Первой в этом листинге следует аннотация `@SequenceGenerator`, определяющая имя последовательности `USER_SEQUENCE_GENERATOR` ❶. Это внутреннее имя для JPA, которое будет использоваться в других аннотациях для ссылки на последовательность. В атрибуте `sequenceName` указывается фактическое имя последовательности в базе данных, в данном случае: `USER_SEQUENCE` ❷. В аннотации `@GeneratedValue` указывается метод `Generation_Type.SEQUENCE` ❸, а атрибуту `generator` присваивается внутреннее имя последовательности ❹.

Действие аннотации `@SequenceGenerator` распространяется на весь модуль. То есть, ее необязательно использовать внутри класса. Любая аннотация `@SequenceGenerator` определяет настройки для всех сущностей, поэтому внутренние имена последовательностей должны быть уникальными.

TABLE

Метод `TABLE` основан на использовании небольшой таблицы в базе данных для получения автоматически увеличивающихся числовых значений. Обычно такая таблица состоит из двух столбцов. В первом столбце определяется уникальное имя последовательности, а во втором – текущее значение. Чтобы настроить JPA на использование метода `TABLE`, сначала необходимо создать таблицу:

```
CREATE TABLE SEQUENCE_GENERATOR_TABLE (
  SEQUENCE_NAME VARCHAR2(80) NOT NULL,
  SEQUENCE_VALUE NUMBER(15) NOT NULL,
  PRIMARY KEY (SEQUENCE_NAME));
```

Следующим шагом после создания таблицы, который показан в листинге 9.14, является вставка имен и начальных значений последовательностей вручную.

Листинг 9.14. Вставка имени и начального значения последовательности

```
-- Вставить последовательность с именем USER_SEQUENCE и начальным значением 1
INSERT INTO SEQUENCE_GENERATOR_TABLE
  (SEQUENCE_NAME, SEQUENCE_VALUE) VALUES ('USER_SEQUENCE', 1);
```

Затем, с помощью аннотации `@TableGenerator` настраивается подключение к этой таблице и в аннотации `@GeneratedValue` – использование данного генерато-

ра последовательности. В листинге 9.15 показано, как могло бы выглядеть объявление объекта `User` при использовании метода `TABLE`.

Листинг 9.15. Метод TABLE

```
@Entity
// ❶ Внутреннее имя последовательности для JPA
@TableGenerator (name="USER_TABLE_GENERATOR",
    // ❷ Имя таблицы в базе данных
    table="SEQUENCE_GENERATOR_TABLE",
    // ❸ Столбец в таблице с именем последовательноти
    pkColumnName="SEQUENCE_NAME",
    // ❹ Столбец в таблице со значением последовательности
    valueColumnName="SEQUENCE_VALUE",
    // ❺ Имя используемой последовательности (в таблице)
    pkColumnValue="USER_SEQUENCE")
public class User {
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE, // ❻ Метод TABLE
        // ❼ Внутреннее имя используемой последовательности
        generator="USER_TABLE_GENERATOR")
    @Column(name="USER_ID")
    protected Long userId;
}
```

Первой в этом листинге следует аннотация `@TableGenerator` с именем `USER_TABLE_GENERATOR` ❶. Это внутреннее имя для JPA, которое будет использоваться в других аннотациях для ссылки на последовательность. Атрибут `table` определяет имя таблицы ❷. Атрибут `pkColumnName` определяет имя столбца в таблице, хранящего имена последовательностей ❸. Атрибут `valueColumnName` определяет имя столбца, хранящего значения последовательностей ❹. Наконец, атрибут `pkColumnValue` ❺ определяет имя используемой последовательности (см. инструкцию `INSERT` в листинге 9.14). В аннотации `@GeneratedValue` указывается метод `GenerationType.TABLE` ❻ и в атрибуте `generator` – внутреннее имя последовательности ❼.

Имейте в виду, что действие аннотации `@TableGenerator` распространяется на весь модуль. То есть, ее необязательно использовать внутри класса. Любая аннотация `@TableGenerator` определяет настройки для всех сущностей, поэтому внутренние имена последовательностей должны быть уникальными.

Первичные ключи, equals() и hashCode()

Важно отметить, что особое значение для первичных ключей имеют методы `equals()` и `hashCode()`. Обычно для сравнения объектов предметной модели по первичному ключу требуется предоставить собственные реализации этих методов. Но при использовании любого из методов автоматической генерации первичных ключей, их значения остаются неизвестными, пока механизм сохранения не вставит данные в таблицу. То есть, до момента вставки, значение первичного ключа в объекте предметной модели будет равно `NULL`. Соответственно, методы `equals()` и `hashCode()` должны корректно обрабатывать пустые значения.

CODE

Метод `CODE` используется в ситуациях, когда значение первичного ключа желательно было бы вычислять вручную, в прикладном коде, без применения автоматических механизмов базы данных. Для этого JPA поддерживает множество разных способов, но наиболее популярным является способ, основанный на использовании аннотации `@PrePersist`, с помощью которой можно организовать присваивание значения свойству первичного ключа перед вставкой данных в базу. В листинге 9.16 показано, как могло бы выглядеть объявление объекта `User` при использовании метода `CODE`.

Листинг 9.16. Метод CODE

```
@Entity
public class User {
    @Id
    @Column(name = "USER_ID")
    protected String userId;

    // ❶ Этот метод автоматически будет вызван перед вставкой данных
    @PrePersist
    public void generatePrimaryKey() {
        // ❷ Генерирует значение первичного ключа
        userId = UUID.randomUUID().toString();
    }
}
```

В этом листинге можно заметить несколько отличий в классе `User`. Во-первых, отсутствует аннотация `@GeneratedValue`. Это обусловлено тем, что теперь значение первичного ключа вычисляется вручную, а не автоматически. Во-вторых, появилась аннотация `@PrePersist` ❶, сообщающая механизму JPA, что он должен вызывать метод `generatePrimaryKey()` перед вставкой данных в базу. И, в-третьих, используется объект `UUID` из JDK, как простой пример одного из способов вычисления значения первичного ключа ❷.

Итак, мы завершили знакомство с основами чтения/записи данных из таблиц в базе данных с использованием JPA. Далее мы займемся рассмотрением отношений между таблицами и узнаем, как JPA строит деревья объектов, опираясь на эти отношения.

9.4. Отношения между сущностями

Как показано на рис. 9.3, модель предметной области в приложении ActionBazaar включает множество разных сущностей, связанных отношениями друг с другом. До сих пор основное внимание уделялось сохранению и извлечению данных для единственной сущности. Теперь пришла пора посмотреть, как JPA поддерживает отношения между сущностями. Под отношением в действительности понимается хранение ссылки на одну сущность в другой. Например, объект предложения цены (ставки) `Bid` хранит ссылку на объект объявления `Item`, для которого это предло-

жение было сделано. Соответственно, между объектами `Bid` и `Item` существует отношение. Напомним, что отношения могут быть одно- и двунаправленными. Отношение между объектом покупателя `Bidder` и объектом ставки `Bid`, как показано на рис. 9.3, является однонаправленным, потому что объект `Bidder` хранит ссылку на объект `Bid`, а объект `Bid` не имеет ссылки на объект `Bidder`. Отношение `Bid-Item`, напротив, является двунаправленным, в том смысле, что объекты `Bid` и `Item` хранят ссылки друг на друга. Отношения могут относиться к одному из четырех типов: «один к одному», «один ко многим», «многие к одному» и «многие ко многим». Все типы отношений выражаются в JPA с помощью аннотаций. В следующих разделах мы по очереди познакомимся с каждым типом отношений и соответствующими им аннотациями.

9.4.1. Отношение «один к одному»

Для выражения одно- и двунаправленных отношений «один к одному» используется аннотация `@OneToOne`. В структуре модели предметной области приложения ActionBazaar (рис. 9.3) нет отношений «один к одному». Но такое отношение легко представить, если вообразить отношение между объектами `User` и `BillingInfo`. Объект `BillingInfo` может содержать данные, касающиеся кредитной карты пользователя, счета в банке, и так далее. Давайте посмотрим, как может выглядеть однонаправленное отношение «один к одному».

Однонаправленное отношение «один к одному»

Допустим, что в объекте `User` имеется ссылка на объект `BillingInfo`, а обратная ссылка отсутствует. Иными словами, отношение является однонаправленным, как показано на рис. 9.6.

Рис. 9.6. Однонаправленное отношение «один к одному» между объектами User и BillingInfo

Реализация этого отношения приводится в листинге 9.17.

Листинг 9.17. Однонаправленное отношение «один к одному»

```
@Entity
public class User {
    @Id
    protected String userId;
    protected String email;

    // ❶ Отношение "один к одному" между User и BillingInfo
    @OneToOne
    protected BillingInfo billingInfo;
```

```
}

// ❷ Сущность, связанная отношением с User
@Entity
public class BillingInfo {
    @Id
    protected Long billingId;
}
```

В этом листинге класс `User` хранит ссылку на `BillingInfo` в свойстве `billingInfo`. Так как свойство `billingInfo` хранит ссылку на единственный экземпляр `BillingInfo` ❶, отношение относится к типу «один к одному». Аннотация `@OneToOne` определяет это отношение в базе данных ❷.

Аннотацию `@OneToOne`, как и большинство других аннотаций JPA, можно применять к свойствам класса или к методам чтения. Она имеет несколько атрибутов, коротко описываемых в табл. 9.2, но все они используются довольно редко.

Таблица 9.2. Атрибуты аннотации @OneToOne

Атрибут	Описание
`targetEntity`	Класс объекта, участвующего в отношении. Может пригодиться, когда отношения определяются интерфейсами и имеется несколько классов реализаций.
`cascade`	Определяет, как должны распространяться изменения по данным связанным отношением.
`fetch`	Определяет момент, когда должны извлекаться связанные данные.
`optional`	Определяет, должно ли отношение интерпретироваться как необязательное (см. листинг 9.18).
`mappedBy`	Определяет сущность, владеющую отношением. Этот атрибут употребляется только на стороне, не владеющей отношением (см. листинг 9.18).

Двунаправленное отношение «один к одному»

При использовании однонаправленных отношений навигация в пределах предметной модели может осуществляться только в одном направлении. В реализации, представленной в листинге 9.17, через объект `User` можно добраться до объекта `BillingInfo`, но обратного пути не существует – нельзя получить объект `User`, имея объект `BillingInfo`. В двунаправленных отношениях объекты предметной модели хранят ссылки друг на друга, что позволяет двигаться в любом из двух направлений. Двунаправленные отношения «один к одному» реализуются путем применения аннотаций `@OneToOne` с обеих сторон отношения. Как это делается, показано в листинге 9.18, который является доработанной версией из листинга 9.17.

Листинг 9.18. Двунаправленное отношение «один к одному»

```
@Entity
public class User {
    @Id
```

```
    protected String userId;
    protected String email;
    // ❶ Отношение "один к одному"
    @OneToOne
    protected BillingInfo billingInfo;
}

@Entity
public class BillingInfo {
    @Id
    protected Long billingId;
    protected String creditCardType;
    // ...

    // ❷ Владельцем этого двунаправленного отношения "один к одному"
    //    является User.billingInfo
    @OneToOne(mappedBy="billingInfo", optional=false)
    protected User user;
}
```

В этом листинге класс `User` связан отношением с классом `BillingInfo` посредством свойства `billingInfo` ❶. Пока нет никаких отличий от однонаправленного отношения «один к одному» в листинге 9.17. Но в этом примере отношение является двунаправленным, потому что класс `BillingInfo` также имеет ссылку на класс `User` в виде свойства `user` ❷. Аннотация `@OneToOne` перед свойством `user` имеет атрибуты `mappedBy` и `optional`. Атрибут `mappedBy="billingInfo"` сообщает механизму JPA, что «владельцем» отношения является свойство `billingInfo` класса `User`, а атрибут `optional="false"` – что экземпляр `BillingInfo` не может существовать без экземпляра `User`. Любопытно отметить, что аннотация `@OneToOne` в классе `User` не имеет этого атрибута. Из этого следует, что экземпляр `User` может существовать без экземпляра `BillingInfo`, а экземпляр `BillingInfo` без экземпляра `User` – нет.

9.4.2. Отношения «один ко многим» и «многие к одному»

Отношения «один ко многим» и «многие к одному» являются наиболее распространенными типами отношений в корпоративных системах. Состоя в отношениях этого типа, одна сущность может хранить две ссылки и более на другие сущности. Обычно это означает, что сущность имеет свойство-коллекцию типа `java.util.Set` или `java.util.List`, хранящее множество экземпляров другой сущности. Кроме того, если отношение между двумя сущностями является двунаправленным, с одной стороны это отношение будет относиться к типу «один ко многим», а с другой «многие к одному».

На рис. 9.7 изображено отношение между сущностями `Item` и `Bid` в приложении ActionBazaar. С точки зрения `Item`, для одного объявления `Item` может быть сделано несколько предложений цены (ставок) `Bid`; соответственно отношение `Item-Bid` относится к типу «один ко многим». С точки зрения `Bid`, множество

предложений цены может быть сделано для одного объявления `Item`; соответственно отношение `Bid-Item` относится к типу «многие к одному».

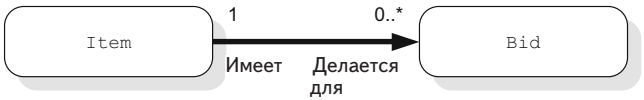

Рис. 9.7. Для одного объявления может поступить множество предложений цены («один ко многим»)

Тип отношения определяется точкой зрения. Этим отношения «один ко многим» и «многие к одному» отличаются от отношений «один к одному», в которых с любой стороны имеется единственная ссылка на сущность. Отношения «один ко многим» и «многие к одному», как и отношения «один к одному», имеют сторону-«владельца» – любые отношения имеют сторону-«владельца», которая определяется с помощью атрибута `mappedBy`. В листинге 9.19 показано, как реализуется отношение между `Item` и `Bid`.

Листинг 9.19. Двунаправленные отношения «один ко многим» и «многие к одному»

```
import java.sql.Date;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
@Entity
public class Bid {
    @Id
    protected Long bidId;
    protected Double amount;
    protected Date timestamp;
    ...
    // ❶ Отношение "многие к одному"
    @ManyToOne
    protected Item item;
    ...
}

import java.sql.Date;
import java.util.Set;

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

@Entity
public class Item {
    @Id
    protected Long itemId;
    protected String title;
    protected String description;
```

```
    protected Date postdate;
    ...
    // ❷ Отношение "один ко многим", владельцем которого является Bid
    @OneToMany(mappedBy="item")
    protected Set<Bid> bids;
    ...
}
```

Как вы наверняка помните, тип отношения определяется точкой зрения, поэтому сначала рассмотрим этот листинг с точки зрения класса `Bid`. Итак, с точки зрения `Bid` множество предложений цены может быть сделано для одного объявления `Item`. Соответственно отношение `Bid-Item` принадлежит к типу «многие к одному». Об этом свидетельствует аннотация `@ManyToOne` ❶ перед свойством `item` в объекте `Bid`. Свойство `item` позволяет по имеющемуся объекту ставки `Bid` получить соответствующий объект объявления `Item`. В терминах отношения `Bid-Item`, можно утверждать, что объект `Item` может существовать без объектов `Bid`, а объекты `Bid` без объекта `Item` – нет. Поэтому объект `Bid` является «владельцем» отношения между `Bid` и `Item`. Напомним, что владелец определяется с помощью атрибута `mappedBy` аннотации в подчиненной сущности, а не в сущности-владельце. Так как владельцем является сущность `Bid`, атрибут `mappedBy` отсутствует в аннотации `@ManyToOne` ❶. Вообще говоря, в двунаправленных отношениях типа «один ко многим» владельцем отношения всегда является сторона `@ManyToOne`.

Теперь посмотрим на этот листинг с точки зрения сущности `Item`. Итак, с точки зрения `Item` для одного объявления может поступить множество предложений цены `Bid`. Соответственно отношение `Item-Bid` принадлежит к типу «один ко многим». Об этом свидетельствует аннотация `@OneToMany` ❷ перед свойством `bids` в объекте `Item`. Свойство `bids` позволяет получить все предложения, сделанные для данного объявления `Item`. Так как `Item` может существовать без объектов `Bid`, а объекты `Bid` без объекта `Item` – нет, «владельцем» отношения является объект `Bid`. Аннотация `@OneToMany` перед свойством `Item.bids` имеет атрибут `mappedBy="item"`, указывающий, что владельцем является свойство `Bid.item`.

Кто я – @ManyToOne или @OneToMany?

При использовании аннотаций `@ManyToOne` и `@OneToMany` иногда возникает путаница, какую аннотацию использовать с той или иной стороны. Поэтому часто полезно задавать себе следующие вопросы.

«Я» один, а «вас» много? Если ответ на этот вопрос утвердительный, значит класс должен содержать список или множество сущностей, и такое отношение относится к типу «один ко многим» (аннотация `@OneToMany`). (См. класс `Item` в листинге 9.19.)

«Нас» много, а «вы» один? Если ответ на этот вопрос утвердительный, значит класс должен содержать единственный экземпляр сущности, и такое отношение относится к типу «многие к одному» (аннотация `@ManyToOne`). (См. класс `Bid` в листинге 9.19.)

Самое интересное наступает, когда утвердительный ответ можно дать на оба вопроса. Чтобы понять, как действовать в этой ситуации, переходите к обсуждению отношения «многие ко многим» и аннотации `@ManyToMany` в следующем разделе.

9.4.3. Отношение «многие ко многим»

Отношения «многие ко многим» достаточно часто встречаются в корпоративных приложениях, хотя и не так часто, как отношения «один ко многим». В отношениях этого типа обе стороны могут хранить коллекции ссылок на связанные сущности. В приложении ActionBazaar в отношении «многие ко многим» находятся объекты `Category` и `Item`, как показано на рис. 9.8.

В отношении `Item-Category` объект объявления `Item` может быть помещено в несколько категорий, в то же время каждая категория `Category` может содержать несколько объявлений. Для определения отношений этого типа используется аннотация `@ManyToMany`. Обычно отношения «многие ко многим» являются двунаправленными. В листинге 9.20 приводится пример определения двунаправленного отношения «многие ко многим» между объектами `Item` и `Category`.

Рис. 9.8. Отношение «многие ко многим» между объектами Item и Category

Листинг 9.20. Двунаправленное отношение «многие ко многим»

```
import java.util.Set;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
@Entity
public class Category {
    @Id
    protected Long categoryId;
    protected String name;
    ...
    // Отношение "многие ко многим"
    @ManyToMany
    protected Set<Item> items;
    ...
}

import java.util.Set;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
@Entity
public class Item {
    @Id
    protected Long itemId;
    protected String title;
    ...
```

```
    // Отношение "многие ко многим", владельцем которого является Category
    @ManyToMany(mappedBy="items")
    protected Set<Category> categories;
    ...
}
```

В этом листинге свойство `items` объекта `Category` отмечено аннотацией `@ManyToMany` и оно же является владельцем отношения. Свойство `categories` объекта `Item`, напротив, является подчиненной стороной в двунаправленном отношении «многие ко многим». Как и в примере определения отношения «один ко многим», в аннотации `@ManyToMany` опущен атрибут `optional`, потому что пустое множество или список неявно подразумевают необязательность отношения, в том смысле, что сущность может существовать даже в отсутствие связей с другими сущностями.

Итак, мы охватили все аннотации определения отношений, имеющиеся в JPA. Используя эти аннотации можно настраивать любые отношения между объектами предметной модели. В табл. 9.3 приводится краткая сводка по аннотациям и их атрибутам.

Таблица 9.3. Аннотации JPA определения отношений между сущностями

Атрибут	@OneToOne	@OneToMany	@ManyToOne	@ManyToMany
targetEntity	Да	Да	Да	Да
cascade	Да	Да	Да	Да
fetch	Да	Да	Да	Да
optional	Да	Нет	Да	Да
mappedBy	Да	Да	Нет	Да

На этом мы завершаем знакомство со способами отображения отношений между сущностями. Автоматическое отображение отношений во время извлечения данных из базы является одной и из самых мощных особенностей JPA. Но, как разработчики на Java, при конструировании предметной модели мы можем применять и другие объектно-ориентированные приемы, которые также поддерживаются в JPA. Одной из таких особенностей является возможность создания иерархий объектов через наследование, поэтому далее мы посмотрим, как средствами JPA осуществляется отображение наследования.

9.5. Отображение наследования

До сих пор мы учились извлекать данные из таблиц и записывать их в таблицы с помощь JPA, и определять отношения между сущностями. Далее мы посмотрим, как наследование объектов Java отражается на строении таблиц. Чтобы было проще понять, о чем пойдет речь, рассмотрим простой пример. Как вы помните, в приложении ActionBazaar имеется два разных типа пользователей: покупатели (bidders) и продавцы (sellers). С точки зрения предметной модели, здесь наследо-

вание используется с целью определения свойств, общих для обеих разновидностей объекта `User`, и уникальных для каждого из двух объектов, `Bidder` и `Seller`. На рис. 9.9 показано, как выглядит это дерево наследования.

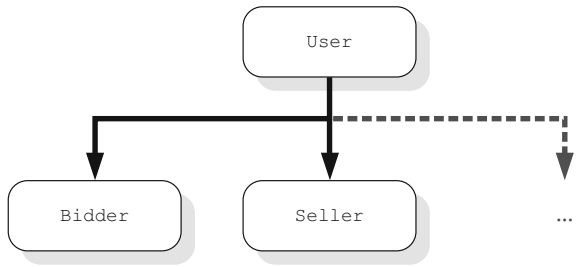

Рис. 9.9. Сущности Bidder и Seller наследуют общие свойства из сущности User

Это настолько привычно в объектно-ориентированном программировании, что большинство разработчиков на Java реализуют подобное наследование, не задумываясь ни на секунду. Это так очевидно, что `Bidder` и `Seller` должны наследовать `User`. Но для администраторов баз данных наследование объектов представляет большую проблему. Почему? Потому что большинство баз данных не поддерживают наследование между таблицами (наследование таблиц имеется в PostgreSQL, но это редкое явление). Соответственно перед администратором встает вопрос: как отобразить иерархию наследования объектов, подобную той, что изображена на рис. 9.9, в таблицы реляционной базы данных. Существует три стратегии, помогающие решить эту задачу:

- единой таблицы;
- соединения таблиц;
- отдельных таблиц для каждого класса.

9.5.1. Стратегия единой таблицы

В соответствии со стратегией единой таблицы, которая является для JPA стратегией по умолчанию, все классы в иерархии модели предметной области отображаются в одну общую таблицу. То есть, все данные из всех объектов в предметной модели хранятся в общей таблице. Разные объекты в предметной модели идентифицируются с помощью специального столбца, называемого *дискриминатором* (discriminator). Столбец-дискриминатор в каждой строке содержит значение, уникальное для данного типа объектов (см. раздел 9.3.5, где рассказывается о типах-перечислениях). Чтобы проще было понять эту схему, рассмотрим ее реализацию. В настоящий момент в приложении ActionBazaar предполагается, что объекты всех типов, представляющие пользователей, включая `Bidder` и `Seller`, отображаются в таблицу `USERS`. На рис. 9.10 показано, как могла бы выглядеть такая таблица.

	USER_ID	USERNAME		USER_TYPE	CREDIT_WORTH		BID_FREQUENCY
Покупатели	1	eccentric-collector	...	B	NULL	...	5.70
	2	packrat	...	B	NULL	...	0.01
Продавцы	3	snake-oil-salesman	...	S	$10,000.00	...	NULL

Рис. 9.10. Сохранение данных о пользователях в единой таблице

Как показано на рис. 9.10, таблица `USERS` содержит данные, общие для всех учетных записей (`USER_IDD`, `USERNAME`). Она также содержит данные, уникальные для `Bidder` (`BID_FREQUENCY`) и для `Seller` (`CREDIT_WORTH`). Строки 1 и 2 содержат записи, соответствующие классу `Bidder`, а строка 3 – запись, соответствующую классу `Seller`. Это обозначено значениями `B` и `S` в столбце дискриминатора `USER_TYPE`. Дискриминатор `USER_TYPE` будет содержать значения, соответствующие всем типам пользователей в предметной модели ActionBazaar. Механизм JPA будет отображать каждый тип, сохраняя данные в соответствующих столбцах, устанавливая нужное значение в столбце `USER_TYPE` и записывая в остальные столбцы значение `NULL`. В листинге 9.21 показано, как реализовать стратегию единой таблицы.

Листинг 9.21. Стратегия единой таблицы

```
@Entity
// ❶ Сущность User отображается в таблицу USERS
@Table(name="USERS")
// ❷ Для предметной модели будет использоваться стратегия единой таблицы
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
// ❸ Имя и тип столбца-дискриминатора
@DiscriminatorColumn(name="USER_TYPE",
    discriminatorType=DiscriminatorType.STRING, length=1)
public abstract class User { ... }

@Entity
// ❹ Значение столбца-дискриминатора для объектов Bidder
@DiscriminatorValue(value="B")
public class Bidder extends User {...}

// ❺ Значение столбца-дискриминатора для объектов Seller
@Entity
@DiscriminatorValue(value="S")
public class Seller extends User {...}
```

С помощью аннотаций `@Inheritance` ❷ и `@DiscriminatorColumn` ❸ определяется, что для данной предметной модели механизм JPA должен использовать стратегию единой таблицы. Аннотации `@Inheritance` и `@DiscriminatorColumn` всегда должны помещаться перед корневым объектом модели. В данном примере таковым является объект `User`. Обратите внимание, что объект `User` отмечен аннотацией `@Table(name="USERS")` ❶ с определением имени таблицы, а объекты

Bidder и Seller нет. Это обусловлено тем, что в стратегии единой таблицы существует только одна таблица, поэтому все данные для разных типов учетных записей будут храниться вместе, в таблице USERS. Чтобы можно было отличить данные, соответствующие классу Bidder, с помощью аннотации @DiscriminatorValue настраивается сохранение значения "B" для этого класса в столбце USER_TYPE таблицы USERS ❹. Аналогично настраивается сохранение значения "S" для класса Seller ❺. Применение аннотации @DiscriminatorValue не является обязательным, в ее отсутствие значениями столбца дискриминатора будут служить имена классов, например, "Seller" для объекта Seller.

9.5.2. Стратегия соединения таблиц

Стратегия соединения таблиц основана на использовании отношения «один к одному» между предметной моделью и таблицами в базе данных. При ее использовании для каждой сущности в предметной модели создается отдельная таблица. Взгляните на рис. 9.11, чтобы понять суть стратегии соединения таблиц.

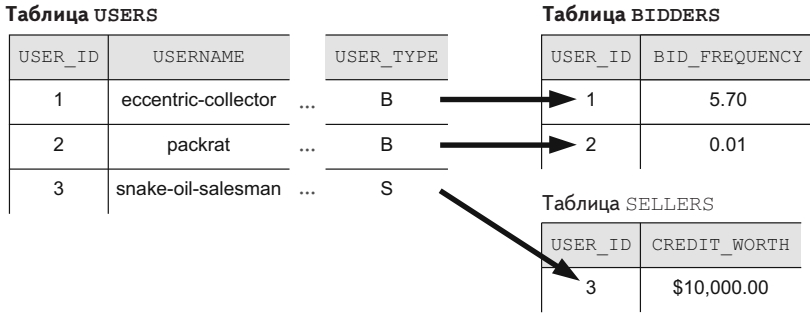

Рис. 9.11. Хранение всех учетных записей с использованием отображения «один к одному» Java-объектов в таблицы

Как показано на рис. 9.11, таблица USERS является родительской и хранит данные, общие для всех типов учетных записей (USERNAME). Таблицы BIDDERS и SELLERS являются дочерними, о чем свидетельствует тот факт, что столбец USER_ID в таблицах BIDDERS и SELLERS является внешним ключом, ссылающимся на столбец USER_ID в таблице USERS. Таблица BIDDERS хранит данные, уникальные для учетных записей покупателей (BID_FREQUENCY), а таблица SELLERS – данные, уникальные для учетных записей продавцов (CREDIT_WORTH). Эти таблицы являются прямым отображением предметной модели. Так, сущность User отображается в таблицу USERS, Seller – в таблицу SELLERS и Bidder – в таблицу BIDDERS. В листинге 9.22 показано, как реализовать стратегию соединения таблиц.

Листинг 9.22. Стратегия соединения таблиц

```
@Entity
// Сущность User отображается в таблицу USERS
@Table(name="USERS")
```

```
// ❶ Для предметной модели будет использоваться стратегия соединения таблиц
@Inheritance(strategy=InheritanceType.JOINED)
// ❷ Имя и тип столбца-дискриминатора
@DiscriminatorColumn(name="USER_TYPE",
    discriminatorType=DiscriminatorType.STRING, length=1)
public abstract class User { ... }

@Entity
// ❸ Сущность Bidder отображается в таблицу BIDDERS
@Table(name="BIDDERS")
// ❹ Значение столбца-дискриминатора для объектов Bidder
@DiscriminatorValue(value="B")
// Использовать BIDDERS.USER_ID для соединения с USERS.USER_ID
@PrimaryKeyJoinColumn(name="USER_ID")
public class Bidder extends User {...}

@Entity
// ❺ Сущность Seller отображается в таблицу SELLERS
@Table(name="SELLERS")
// Значение столбца-дискриминатора для объектов Seller
@DiscriminatorValue(value="S")
// Использовать SELLERS.USER_ID для соединения с USERS.USER_ID
@PrimaryKeyJoinColumn(name="USER_ID")
public class Seller extends User {...}
```

Аннотации `@DiscriminatorColumn` ❷ и `@DiscriminatorValue` ❶ в этом листинге используются точно так же, как в реализации стратегии единой таблицы. В атрибуте `strategy` аннотации `@Inheritance` указано значение `JOINED`. Кроме того, посредством применения аннотаций `@PrimaryKeyJoinColumn` в обеих сущностях, `Bidder` ❹ и `Seller` ❻, между родительской и дочерними таблицами реализованы отношения «один к одному». В обоих случаях атрибут `name` определяет, что столбцы `BIDDER.USER_ID` и `SELLER.USER_ID` являются внешними ключами, ссылающимися на `USER.USER_ID`. Сущности `Bidder` и `Seller` также отмечены аннотациями `@Table`, определяющими имена таблиц для отображения (❸ и ❺).

Стратегия соединения таблиц является, пожалуй, лучшим выбором с точки зрения архитектуры базы данных. Однако, с точки зрения производительности она проигрывает стратегии единой таблицы, потому что требует соединения нескольких таблиц для обеспечения полиморфизма запросов.

9.5.3. Стратегия отдельных таблиц для каждого класса

При использовании стратегии отдельных таблиц, для каждой сущности в предметной модели создается отдельная таблица, почти как в стратегии соединения таблиц, но между ними есть важное отличие: в стратегии отдельных таблиц для каждого класса отсутствуют какие-либо отношения между таблицами. Все данные, которые являются разделяемыми в стратегии соединения таблиц, не являются разделяемыми в стратегии отдельных таблиц и дублируются в каждой таблице. Взгляните на рис. 9.12, где показано, как действует эта стратегия.

Таблица USERS

USER_ID	USERNAME
1	super-user

Таблица SELLERS

USER_ID	USERNAME	CREDIT_WORTH
1	snake-oil-salesman	

Таблица BIDDERS

USER_ID	USERNAME	BID_FREQUENCY
1	eccentric-collector	5.70
2	packrat	0.01

Рис. 9.12. При использовании стратегии отдельных таблиц, таблицы в базе данных имеют одинаковые столбцы

Взглянув на рис. 9.12, легко увидеть отличие стратегии отдельных таблиц. Как и прежде, сущность `User` отображается в таблицу `USERS`, `Seller` – в таблицу `SELLERS` и `Bidder` – в таблицу `BIDDERS`. Но в этой стратегии не используются реляционные возможности базы данных. Таблицы не связаны между собой отношениями и потому не имеют общих данных. Как следствие, общие данные, такие как `USER_ID` и `USERNAME`, повторяются в каждой таблице.

Из-за дублирования данных, при использовании стратегии отдельных таблиц необходимо проявлять осторожность, сохраняя данные, особенно если эти данные, такие как `USER_ID` и `USERNAME`, должны быть уникальны. Данные должны быть уникальными не только в «дочерней» таблице, но и в «родительской» (слова «родительская» и «дочерняя» взяты в кавычки, потому что в действительности таблицы не связаны отношениями). Например, допустим, что потребовалось сохранить новую учетную запись покупателя со следующими данными: `USER_ID = 10`, `USERNAME = "ActionBazaarUser123"`. Эти значения могут быть уникальными для таблицы `BIDDERS`, но при использовании стратегии отдельных таблиц эти же данные необходимо сохранить в таблице `USERS`. Однако для таблицы `USERS` эти значения могут оказаться неуникальными, из-за того, что могут уже использоваться в учетной записи продавца.

Стратегия отдельных таблиц оказывается самой простой в реализации, в сравнении с остальными стратегиями. В листинге 9.23 показано, как реализовать стратегию отдельных таблиц.

Листинг 9.23. Стратегия отдельных таблиц для каждого класса

```
@Entity
// ❶ Сущность User отображается в таблицу USERS
@Table(name="USERS")
// ❷ Для предметной модели будет использоваться стратегия отдельных таблиц
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
```

```
public class User { ... }

@Entity
// ❸ Сущность Bidder отображается в таблицу BIDDERS
@Table(name="BIDDERS")
public class Bidder extends User {...}

@Entity
// ❹ Сущность Seller отображается в таблицу SELLERS
@Table(name="SELLERS")
public class Seller extends User {...}
```

В этом листинге видно, что атрибут `strategy` аннотации `@Inheritance` определяет стратегию `TABLE_PER_CLASS` ❷. Разумеется, в этом листинге отсутствуют аннотации `@DiscriminatorColumn` и `@DiscriminatorValue` – в данном случае они не нужны, потому что все данные для каждой сущности сохраняются в отдельной таблице, соответственно нет необходимости различать записи по типу учетной записи. Наконец, с помощью аннотаций `@Table` реализовано отображение каждой сущности в собственную таблицу (❶, ❸ и ❹).

Эта стратегия является самой сложной с точки зрения обеспечения надежности. В результате поддержка этой стратегии в спецификации определена как необязательная. Мы рекомендуем стараться избегать применения этой стратегии.

Этим примером мы заканчиваем анализ трех стратегий отображения объектно-ориентированного наследования. Выбор стратегии не так прост, как может показаться. Правильный выбор вам поможет сделать сравнение, которое приводится в табл. 9.4.

Таблица 9.4. Сравнение стратегий отображения объектно-ориентированного наследования

Особенность	Единая таблица	Соединение таблиц	Отдельные таблицы
Поддержка таблиц	Одна таблица для всех классов в иерархии сущностей: • обязательные столбцы могут хранить пустые значения; • число столбцов в таблице растет с появлением новых подклассов.	Одна для родительского класса и отдельные таблицы для каждого подкласса, хранящие полиморфные свойства. Отображаемые таблицы нормализованы.	Одна таблица для каждого конкретного класса в иерархии сущностей.
Используется столбец-дискриминатор?	Да	Да	Нет
SQL-инструкция, генерируемая для извлечения иерархии сущностей	Простая инструкция `SELECT`.	Инструкция `SELECT` с операцией соединения нескольких таблиц.	По одной инструкции `SELECT` для подкласса или объединение `UNION` инструкций `SELECT`.

Таблица 9.4. (окончание)

Особенность	Единая таблица	Соединение таблиц	Отдельные таблицы
SQL-инструкция, генерируемая для вставки и изменения	Единственная инструкция `INSERT` или `UPDATE` для каждой сущности в иерархии.	Множество инструкций `INSERT`, `UPDATE`: по одной для родительского класса и по одной для каждого подкласса.	Одна инструкция `INSERT` или `UPDATE` для каждого подкласса.
Полиморфные отношения	Хорошо	Хорошо	Плохо
Полиморфные запросы	Хорошо	Хорошо	Плохо

9.6. В заключение

На этом мы завершаем введение в JPA. Не забывайте, что интерфейс JPA обладает массой возможностей и в настоящее время охватывается собственной спецификацией JSR (версия JPA 2.1 определяется спецификацией JSR-338, доступной на сайте Java Community Process: http://jcp.org/en/jsr/detail?id=338). Мы рекомендуем считать эту главу лишь кратким введением и продолжить самостоятельное исследование возможностей JPA.

В начале главы мы познакомились со сложностями отображения предметной модели на языке Java в таблицы базы данных. Как мы узнали, из-за разного представления данных в разных технологиях существует проблема несовместимости интерфейсов, что требует применения переводчика, JPA, для преобразования данных. Затем мы познакомились с простой предметной моделью Java для приложения ActionBazaar. Мы дали имена сущностям и определили отношения между ними. После этого мы приступили к исследованию богатого множества аннотаций JPA для чтения/записи данных в базе и определения отношений между данными. Мы познакомились с аннотацией `@Entity`, превращающей простые объекты в сущности JPA; с аннотациями `@Table` и `@SecondaryTable`, отображающими сущности в таблицы базы данных; с аннотациями `@Column`, `@Transient`, `@Temporal` и `@Enumerated`, отображающими свойства сущностей в столбцы таблиц; с аннотациями `@Id`, `@IdClass` и `@EmbeddedId`, идентифицирующими первичные ключи; и с аннотацией `@GeneratedValue`, генерирующей первичные ключи. Далее мы увидели, как определяются отношения между сущностями с помощью аннотаций `@OneToOne`, `@ManyToOne`, `@OneToMany` и `@ManyToMany`. В заключение мы исследовали стратегии отображения наследования, такие как стратегия единой таблицы, стратегия соединения таблиц и стратегия отдельных таблиц, и особенности их определения с помощью аннотаций `@Inheritance`, `@DiscriminatorColumn`, `@DiscriminatorValue` и `@PrimaryKeyJoinColumn`. В главе 10 мы приступим к знакомству с объектом `EntityManager` и узнаем, как пользоваться сущностями из этой главы для фактического сохранения данных в базе.

ГЛАВА 10. Управление сущностями

Эта глава охватывает следующие темы:

- объект `EntityManager`;
- жизненный цикл сущностей;
- сохранение и извлечение сущностей;
- область видимости контекста сохранения.

В главе 9 мы познакомились с основами сущностей JPA. Вы видели, как реализовать объекты предметной области, как определять отношения между объектами и отображать наследование. В этой главе мы познакомимся с объектами `EntityManager` и посмотрим, как управлять сущностями. Обсудим назначение объекта `EntityManager` и как внедрять его в классы EJB. Вы познакомитесь с жизненным циклом сущностей и как с помощью `EntityManager` сохранять, находить, объединять и удалять сущности, то есть, выполнять обычные операции создания, чтения, изменения и удаления в базе данных. Здесь вы также познакомитесь с разными областями видимости контекста сохранения.

Давайте начнем эту главу с исследования объекта `EntityManager`. Объект `EntityManager` – это основа JPA, и практически все операции выполняются с его применением.

10.1. Введение в использование EntityManager

Объект `EntityManager` является, пожалуй, самым важным и интересным элементом Java Persistence API. Он управляет жизненным циклом сущностей. В этом разделе вы познакомитесь с интерфейсом `EntityManager` и его методами. Мы исследуем разные этапы жизненного цикла сущности, а также разные контексты сохранения (persistence contexts).

10.1.1. Интерфейс EntityManager

В некотором смысле `EntityManager` играет роль моста между объектно-ориентированным и реляционным мирами, как показано на рис. 10.1. когда прикладной код создает сущность, объект `EntityManager` преобразует ее в новую запись в базе данных. Когда прикладной код изменяет сущность, объект `EntityManager` находит соответствующие данные в базе и приводит их в соответствие с изменениями в сущности. Аналогично `EntityManager` удаляет данные из базы, когда запрашивается удаление сущности. С другой стороны моста, когда выполняется попытка «найти» сущность в базе данных, `EntityManager` создает объект сущности, заполняет его данными из базы и «возвращает» в объектно-ориентированный мир.

Рис. 10.1. Объект EntityManager играет роль моста между объектно-ориентированным и реляционным мирами. Он интерпретирует параметры объектно-реляционного отображения, указанные для сущности, и сохраняет сущность в базе данных

Помимо выполнения этих явных CRUD-операций создания, чтения изменения и удаления, `EntityManager` также старается автоматически синхронизировать сущности с базой данных, пока они находятся *в его области досягаемости* (именно такую синхронизацию мы будем подразумевать под «управлением» сущностями в следующем разделе). Можно смело утверждать, что `EntityManager` является важнейшим интерфейсом в JPA и реализует большую часть волшебства объектно-реляционного отображения.

Несмотря на широту возможностей, объект `EntityManager` имеет достаточно простой и понятный интерфейс, особенно если сравнить его с шагами по подготовке к отображению, обсуждавшимися в предыдущей главе, и с прикладным интерфейсом запросов, который будет обсуждаться в следующей главе. После знакомства с некоторыми основными понятиями в следующих нескольких разделах, интерфейс `EntityManager` будет выглядеть для вас тривиально простым. Вы наверняка согласитесь с этим, взглянув на табл. 10.1, где перечислены некоторые из наиболее часто используемых методов из интерфейса `EntityManager`.

Таблица 10.1. Объект EntityManager используется для выполнения CRUD-операций. Здесь перечислены наиболее часто используемые его методы

Сигнатура метода	Описание
`public void persist(Object entity);`	Сохраняет сущность в базе данных и делает ее управляемой сущностью.
`public <T> T merge(T entity);`	Подключает сущность к контексту сохранения `EntityManager` и возвращает подключенную сущность.

Таблица 10.1. (продолжение)

Сигнатура метода	Описание
`public void remove(Object entity);`	Удаляет сущность из базы данных
`public <T> T find(Class<T> entityClass, Object primaryKey);`	Находит экземпляр сущности по первичному ключу. Этот метод имеет перегруженные версии и доступен в разных формах.
`public void flush();`	Синхронизирует данные в базе с состоянием сущностей в контексте сохранения `EntityManager`.
`void setFlushMode(FlushModeType flushMode);`	Изменяет режим синхронизации для контекста сохранения объекта `EntityManager`. Поддерживается два режима сохранения: `AUTO` и `COMMIT`. По умолчанию используется режим `AUTO`, в соответствии с которым `EntityManager` пытается автоматически синхронизировать сущности с базой данных.
`public FlushModeType getFlushMode();`	Возвращает текущий режим синхронизации.
`public void refresh(Object entity);`	Обновляет сущности, читая данные из базы. Этот метод имеет перегруженные версии и доступен в разных формах.
`public Query createQuery(String jpqlString);`	Создает динамический запрос с использованием инструкций JPQL. Этот метод имеет перегруженные версии и доступен в разных формах.
`public Query createNamedQuery(String name);`	Создает экземпляр запроса на основе именованного запроса в экземпляре сущности. Этот метод имеет перегруженные версии и доступен в разных формах.
`public Query createNativeQuery(String sqlString);`	Создает динамический запрос с использованием инструкций SQL. Этот метод имеет перегруженные версии и доступен в разных формах.
`public StoredProcedureQuery createStoredProcedureQuery(String procedureName);`	Создает `StoredProcedureQuery` для выполнения хранимой процедуры. Этот метод имеет перегруженные версии и доступен в разных формах.
`public void close();`	Закрывает управляемый приложением объект `EntityManager`.
`public void clear();`	Отсоединяет все управляемые сущности от контекста сохранения. Изменения в сущностях при этом не сохраняются и утрачиваются.
`public boolean isOpen();`	Проверяет, открыт ли объект `EntityManager`.
`public EntityTransaction getTransaction();`	Возвращает объект транзакции, который можно использовать для запуска и завершения транзакции вручную.

Таблица 10.1. (окончание)

Сигнатура метода	Описание
`public void joinTransaction();`	Предлагает объекту `EntityManager` присоединиться к текущей транзакции JTA.

Не волнуйтесь, если назначение каких-то методов вам непонятно. Все они, кроме методов, имеющих отношение к API запросов (`createQuery`, `createNamedQuery` и `createNativeQuery`), будут подробно рассматриваться в следующих разделах. Здесь не указаны некоторые методы объекта `EntityManager` – они используются довольно редко, поэтому мы не будем тратить время на их обсуждение. Однако, после прочтения этой главы мы рекомендуем не останавливаться на достигнутом и продолжить исследование методов объекта `EntityManager` самостоятельно. Последнюю версию спецификации EJB 3 Java Persistence API 2.1 можно найти по адресу: http://jcp.org/en/jsr/detail?id=338.

Даже при том, что механизм JPA не связан тесными узами с контейнером, как сеансовые компоненты или компоненты MDB, сущности, тем не менее, имеют свой жизненный цикл. Это обусловлено тем, что они находятся «под управлением» механизма JPA, который следит за ними и даже автоматически синхронизирует их с базой данных, когда это возможно. Подробнее об этом рассказывается в следующем разделе.

10.1.2. Жизненный цикл сущностей

Жизненный цикл сущностей довольно прост. Вы легко поймете его суть, как только усвоите, что `EntityManager` ничего не знает о POJO, какие бы аннотации вы к нему не применяли, пока вы не сообщите диспетчеру, что он должен интерпретировать POJO как сущность JPA. Этим они полностью отличаются от объектов POJO, отмеченных аннотациями как сеансовые компоненты или компоненты MDB, которые загружаются и управляются контейнерами сразу после запуска приложения. Кроме того, диспетчер `EntityManager` стремится управлять сущностями минимально короткое время. И снова здесь наблюдается полная противоположность компонентам, управляемым контейнером, которые часто продолжают существование до завершения приложения.

Сущность, находящаяся под управлением `EntityManager`, называется *подключенной* (attached), или *управляемой* (managed).С другой стороны, когда `EntityManager` прекращает управление сущностью, такая сущность называется *отключенной* (detached). Сущность, которая никогда не находилась под управлением, называется *переходной* (transient), или *новой*. На рис. 10.2 изображена диаграмма жизненного цикла сущности.

Давайте рассмотрим поближе сущности в подключенном и отключенном состояниях.

Подключенные сущности

Когда речь идет о сущности в подключенном состоянии, подразумевается, что ее синхронизацию с базой данных обеспечивает `EntityManager`. То есть, объект

EntityManager гарантирует, во-первых, загрузку данных из базы в сущность, как только она будет передана под его управление, и, во-вторых, запись данных в базу при изменении свойств сущности (например, в результате вызова ее методов). Достигается это за счет хранения ссылки на сущность объектом EntityManager и периодической проверки необходимости синхронизации. Если EntityManager обнаружит изменение данных в сущности, он автоматически синхронизирует изменения с базой данных. Управление сущностью прекращается, когда она удаляется или выходит из области досягаемости диспетчера.

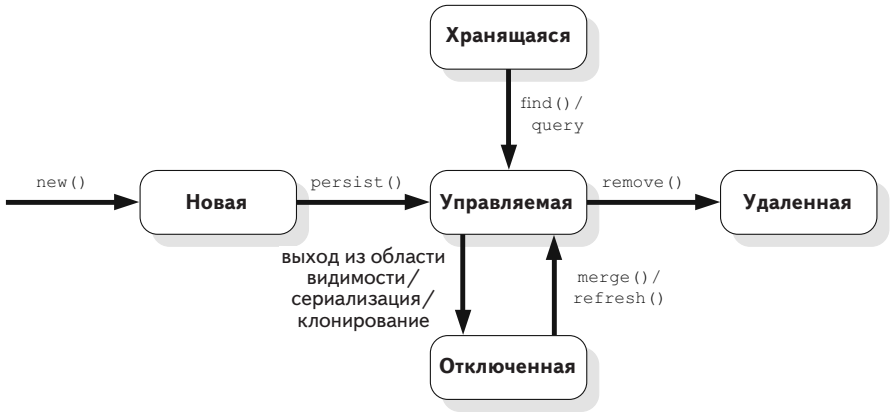

Рис. 10.2. Сущность становится управляемой в момент ее сохранения, слияния или извлечения. Управляемая сущность перестает быть управляемой (отключается) в момент выхода из области видимости, удаления или клонирования

Подключить сущность к контексту EntityManager можно вызовом метода persist или merge. Также подключение выполняется при извлечении сущности из базы данных вызовом метода find или query внутри транзакции. Состояние сущности определяется вызываемым методом. Управляемую сущность в любой момент можно обновить данными из базы вызовом метода refresh.

Сразу после создания экземпляра, как показано в следующем фрагменте, сущность оказывается в новом, или переходном состоянии, потому что EntityManager пока не знает о ее существовании:

```
Bid bid = new Bid();
```

В таком переходном состоянии сущность будет пребывать, пока вызовом метода persist объекта EntityManager не будет создана новая запись в базе данных, соответствующая сущности. Это самый естественный способ подключить сущность Bid из предыдущего фрагмента к контексту EntityManager:

```
manager.persist(bid);
```

Управляемая сущность отключается от контекста, когда выходит из его области видимости, удаляется, сериализуется или клонируется. Например, экземпляр сущности Bid будет отключен в результате подтверждения транзакции.

Сущность, извлеченная из базы данных вызовом метода `find` объекта `EntityManager` или одного из методов запроса в контексте транзакции, автоматически становится управляемой. Извлеченная сущность тут же становится неподключенной, если извлечение произошло вне транзакции.

Подключить отключенную сущность можно одним из двух способов. Первый – вызовом метода `merge`. Метод `merge` принимает отключенную сущность и возвращает подключенную сущность. С помощью переданной ему отключенной сущности метод `merge` находит данные в базе, создает новую подключенную сущность, копирует в нее изменения из исходной неподключенной сущности и возвращает новую, подключенную сущность. Прежде чем транзакция завершится, любые изменения в новой подключенной сущности будут сохранены в базе – таким способом можно записать изменения из неподключенной сущности в базу данных.

Второй способ – вызовом метода `find`. Чтобы получить новую подключенную сущность, достаточно передать первичный ключ из отключенной сущности в запрос. Раз уж мы заговорили об отключенных сущностях, рассмотрим их подробнее.

Отключенные сущности

Отключенная сущность – это сущность, не управляемая объектом `EntityManager` и для нее не гарантируется синхронизация с базой данных. Операции отключения и подключения могут пригодиться при необходимости передачи сущностей между различными слоями приложения. Например, можно отключить сущность, передать ее в веб-слой, обновить, вернуть обратно в слой компонентов EJB и вновь подключить к контексту сохранения данных.

Механика отключения сущностей имеет некоторые особенности. По сути, подключенная сущность отключается, как только выходит из области видимости контекста `EntityManager`. Это можно представить, как разрыв связи между сущностью и объектом `EntityManager` в конце логической единицы обработки, или сеанса. Сеанс взаимодействия с `EntityManager` можно ограничить единственным вызовом метода или распространить во времени до бесконечности. (Чем-то напоминает сеансовые компоненты, не так ли? Как будет показано чуть ниже, это сходство не случайно.) Если сеанс взаимодействия с `EntityManager` ограничен единственным вызовом метода, все подключенные сущности отключатся, как только метод вернет управление, даже если они продолжат использоваться за пределами метода. Если пока вам не все понятно, многое прояснится, когда мы обсудим контекст сохранения объекта `EntityManager` в следующем разделе.

Экземпляры сущностей также отключаются при клонировании или сериализации. Это объясняется тем, что `EntityManager` следит за сущностями, используя ссылки на объекты Java. Так как клонированные или сериализованные экземпляры не могут адресоваться ссылками на оригинальные сущности, у `EntityManager` нет никакой возможности узнать об их существовании. Данная ситуация возникает чаще всего, когда сущности передаются по сети, при обращении к методам удаленных компонентов.

Кроме того, вызов метода `clear` объекта `EntityManager` принудительно отключит все сущности, управляемые этим объектом. Вызов метода `remove` также отключит сущность, но это вполне логично: он удалит все данные из базы, связанные с сущностью, а так как данные будут удалены, бессмысленно продолжать управлять ими. Например, удалить сущность `Bid` можно, как показано ниже:

```
manager.remove(bid);
```

Чтобы проще было запомнить смену этапов жизненного цикла сущности, очень удобно использовать следующую аналогию. Представьте, что сущность – это самолет, а `EntityManager` – авиадиспетчер. Когда сущность находится за пределами зоны ответственности диспетчера, она считается «отключенной» или «новой». Но когда она входит в зону (то есть, становится управляемой), диспетчер начинает руководить движением самолета (синхронизацией ее состояния с базой данных). В конце концов, самолет пересекает зону ответственности диспетчера и покидает ее (отключается), после чего пилот волен продолжать движение по своему плану без команд с земли (то есть, изменять состояние отключенной сущности без синхронизации с базой данных).

Область видимости контекста сохранения – это эквивалент зоны действия радаров авиадиспетчера. Для эффективного использования сущностей очень важно понимать, как действует контекст сохранения. Исследованием отношений между контекстом сохранения, его областью видимости и объектом `EntityManager` мы займемся в следующем разделе.

10.1.3. Контекст сохранения, области видимости и EntityManager

Контекст сохранения (persistence context) играет важную роль в работе внутренних механизмов `EntityManager`. Вы можете выполнять операции сохранения, вызывая методы `EntityManager`, но сам объект `EntityManager` не будет следить за жизненным циклом отдельных сущностей. В действительности, `EntityManager` делегирует задачу управления состоянием сущностей текущему доступному контексту сохранения.

В самом простом представлении, контекст сохранения – это автономная коллекция сущностей, управляемых диспетчером `EntityManager` в данной области видимости. Область видимости – это промежуток времени, в течение которого сущности в коллекции остаются управляемыми.

Чтобы вам проще было разобраться со всей этой кухней, мы познакомимся с различными областями видимости, посмотрим, что они делают, а затем вернемся к обсуждению терминологии. Мы расскажем, как связаны между собой контекст сохранения и область видимости с объектом `EntityManager`, начав с исследования областей видимости.

Существует два типа областей видимости контекста сохранения: *область видимости транзакции* и *расширенная область видимости*.

Транзакционный EntityManager

Объект `EntityManager`, связанный с контекстом, область видимости которого ограничена рамками транзакции, называется *транзакционным*. Если контекст сохранения действует только в пределах транзакции, сущности, подключенные во время действия транзакции, автоматически отключатся после ее завершения. (Все операции, связанные с изменением данных, должны выполняться внутри транзакции, независимо от типа области видимости.) Проще говоря, контекст сохранения продолжает управлять сущностями, пока охватывающая транзакция остается активной. Как только контекст сохранения обнаружит, что транзакция была подтверждена или отменена, он отключит все управляемые им сущности, предварительно синхронизировав их с базой данных.

На рис. 10.3 изображена связь между сущностями, областью видимости транзакции и контекстами сохранения.

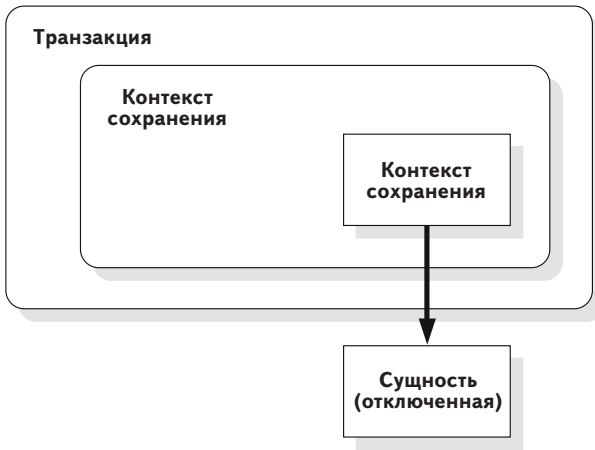

Рис. 10.3. Контекст сохранения с областью видимости в пределах транзакции удерживает сущности подключенными, только пока действует охватывающая транзакция

EntityManager с расширенной областью видимости

Диспетчер `EntityManager` с расширенной областью видимости распространяет свое действие на несколько транзакций. Однако он может работать только с сеансовыми компонентами с сохранением состояния и продолжает действовать, пока компонент не будет уничтожен. Поэтому в контекстах сохранения с расширенной областью видимости продолжительность периода, когда сущности остаются управляемыми, никак не зависит от продолжительности отдельных транзакций. После подключения сущности остаются управляемыми, пока продолжает существовать экземпляр `EntityManager`. Например, `EntityManager` с расширенной областью видимости, созданный в сеансовом компоненте с сохранением состояния, продолжит управлять всеми подключенными сущностями, пока не будет закрыт при уничтожении компонента. Как показано на рис. 10.4, если сущность не отключить явно вызовом метода `remove`, она останется под управлением такого контекста сохранения до самого конца существования компонента.

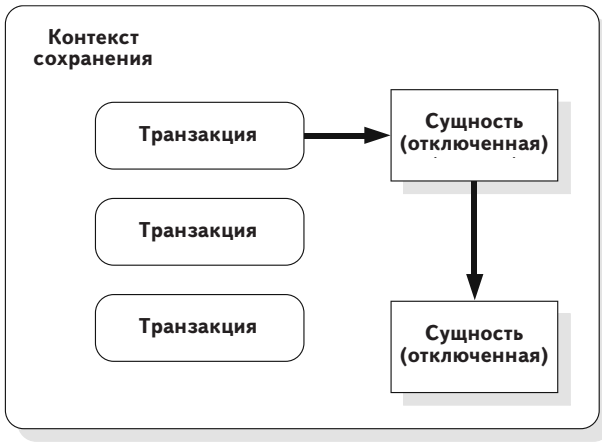

Рис. 10.4. В контексте сохранения с расширенной областью видимости, после подключения любой транзакцией, сущность остается управляемой на протяжении всех последующих транзакций

Термин *область видимости* (*scope*), применительно к контекстам сохранения, имеет тот же смысл, что и применительно к переменным Java. Он описывает, как долго остается активным тот или иной контекст. Транзакционные контексты можно сравнить с локальными переменными в методах – они действуют только в рамках транзакций. Контексты с расширенной областью видимости, напротив, больше похожи на переменные экземпляров, которые остаются активными в течение всего времени существования объекта – они действуют, пока существует сам объект `EntityManager`.

К настоящему моменту мы охватили все теоретические понятия, необходимые для понимания особенностей работы `EntityManager`. Теперь посмотрим на объект `EntityManager` в действии.

10.1.4. Использование EntityManager в ActionBazaar

Займемся далее исследованием интерфейса `EntityManager` на примере реализации сеансового компонента без сохранения состояния `ItemManagerBean` для приложения ActionBazaar. Данный компонент реализует операции для работы с объявлениями. Как видно из листинга 10.1, сеансовый компонент предоставляет методы, добавляющие, изменяющие и удаляющие сущности `Item` с помощью `EntityManager`. Это очень важный компонент, потому что объявления составляют основу приложения ActionBazaar. Пользователи просматривают объявления, делают ставки, а по окончании торгов для победителей генерируются заказы.

Листинг 10.1. `ItemManager` реализует основные операции с объявлениями

```
@Stateless
public class ItemManager {
    @PersistenceContext
    private EntityManager entityManager;
    public void addItem(String name, String description, byte[] picture,
        BigDecimal initialPrice, long sellerId) {}
```

```
    public void saveItem(Item item) {}
    public void deleteItem(Item item) {}
    public Item updateItem(Item item) {}
    public List<Item> findItemByName(String name) {}
    public List<Item> findByDate(Date startDate , Date endDate) {}
    public List<String> getAllItemsNames() {}
    public List<Object[]> getAllItemNamesWithIds() {}
    public List<WinningBidWrapper> getWinningBid(Long itemId) {}
    public List<Tuple> getWinningBidTuple(Long itemId) {}
}
```

Давайте для начала посмотрим, как приобрести ссылку на `EntityManager`.

10.1.5. Внедрение EntityManager

Первое, что необходимо сделать перед выполнением каких-либо операций с хранилищем данных, – получить экземпляр `EntityManager`. Сделать это совсем несложно; достаточно отметить переменную-член типа `EntityManager` аннотацией `@PersistenceContext`, как показано в листинге 10.1, а все остальное – поиск, открытие и закрытие – автоматически сделает контейнер. Внедренный экземпляр `EntityManager` получит область видимости транзакции, если явно не указать иное. Чтобы лучше понять, как выполняется внедрение, рассмотрим определение аннотации:

```
@Target(value = {ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
@Retention(value = RetentionPolicy.RUNTIME)
public @interface PersistenceContext {
    public String name() default "";
    public String unitName() default "";
    public PersistenceContextType type() default
        PersistenceContextType.TRANSACTION;
    public PersistenceProperty[] properties() default {};
}
```

Первый атрибут аннотации, `name`, определяет имя контекста сохранения в каталоге JNDI. Этот атрибут используется в тех редких случаях, когда данная реализация контейнера требует указать имя экземпляра `EntityManager` в каталоге JNDI, чтобы найти его. В большинстве случаев этот атрибут можно не указывать, за исключением ситуаций, когда `@PersistenceContext` применяется к классу для установки ссылки на контекст сохранения.

Атрибут `unitName` определяет имя единицы хранения (persistence unit). Единицы хранения – это группы сущностей, используемых в приложении. Возможность деления на группы может пригодиться в больших приложениях Java EE для отделения друг от друга разных логических областей (таких как пакеты Java). Например, сущности в приложении ActionBazaar можно было бы разделить на две группы: `general` и `admin`.

Единицы хранения можно определить только в дескрипторе развертывания `persistence.xml`. Подробнее об определении единиц хранения будет рассказываться в главе 13. А пока вам достаточно знать, что для получения экземпляра

EntityManager, управляющего определенной единицей хранения – например, admin – необходимо указать ее имя, как показано ниже:

```
@PersistenceContext(unitName="admin")
EntityManager entityManager
```

Если модуль Java EE имеет единственную единицу хранения, атрибут unitName можно опустить. Большинство механизмов хранения правильно определяют имя единицы хранения в этом случае. Но, вообще говоря, если в приложении существуют единицы хранения, их принято указывать с помощью атрибута unitName, потому что спецификация весьма туманно описывает поведение аннотации, если имя единицы хранения не указано явно.

Настройка области видимости для EntityManager

Атрибут type определяет область видимости для EntityManager. Как уже говорилось выше, экземпляры EntityManager, управляемые контейнером, могут иметь либо расширенную область видимости, либо область видимости, ограниченную рамками транзакции. Если атрибут type отсутствует, он получает значение по умолчанию TRANSACTION. Поэтому неудивительно, что обычно этот атрибут используется, только чтобы присвоить ему значение EXTENDED. В коде это выглядит так:

```
@PersistenceContext(type=PersistenceContexType.EXTENDED)
EntityManager entityManager;
```

Для компонентов без сохранения состояния или MDB нельзя использовать расширенную область видимости. Причина достаточно очевидна: задача расширенной области видимости заключается в том, чтобы обеспечить синхронизацию сущностей между вызовами методов компонента, даже если все вызовы методов будут осуществляться в разных транзакциях. А так как ни сеансовые компоненты без сохранения состояния, ни компоненты MDB не предполагают такой возможности, бессмысленно поддерживать расширенную область видимости для компонентов этих типов. С другой стороны, расширенная область видимости идеально подходит для сеансовых компонентов с сохранением состояния. Объект EntityManager с расширенной областью видимости можно использовать, например, для кэширования и поддержания прикладных данных в актуальном состоянии между вызовами методов со стороны клиента, причем, не отказываясь от возможности выполнять каждый метод в отдельной транзакции.

> **Примечание.** *Главное достоинство диспетчеров* EntityManager*, управляемых контейнером, в том, что они дают довольно высокий уровень абстракции. Контейнер сам создает экземпляры* EntityManager*, сохраняет их в каталоге JNDI, внедряет в компоненты и закрывает по окончании их использования (обычно при уничтожении компонента).*

Трудно оценить объем кода, который помогает экономить контейнер. Вспомните об этом, когда будете читать следующий раздел, где описываются приемы

прямого доступа к EntityManagerFactory. Отметьте, что EntityManager не поддерживает работу в многопоточной среде, что требует особой осторожности при внедрении экземпляров EntityManager в компоненты CDI, сервлеты, компоненты JSF и им подобные.

EntityManager и многопоточность

Объекты EntityManager не поддерживают работу в многопоточной среде и потому не должны использоваться в ситуациях, когда обращения к ним могут выполняться более чем из одного потока выполнения. Это означает, что их небезопасно использовать в сервлетах или страницах JSP. Сервлет создается один раз и используется для одновременной обработки множества запросов. Хотя на этапе разработки или во время простого тестирования может показаться, что приложение работает безупречно, при увеличении нагрузки оно может проявлять непредсказуемое поведение. Лучше всего использовать экземпляры EntityManager внутри компонентов EJB. Если по каким-то причинам понадобится обращаться к объектам EntityManager непосредственно, можно использовать прием, представленный ниже, или задействовать EntityManagerFactory:

```
@PersistenceContext(name="pu/actionBazaar" unitName="ActionBazaar")
public class ItemServlet extends HttpServlet {
    @Resource
    private UserTransaction ut;
    public void service(HttpSerlvetRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        Context ctx = new InitialContext();
        EntityManager em = (EntityManager)ctx.lookup(
            "java:comp/env/pu/actionBazaar");
        ...
        ut.begin();
        em.persist(item);
        ut.commit();
        ...
    }
}
```

Другой способ – использовать управляемые приложением экземпляры EntityManager и транзакции JTA. Стоит также отметить, что EntityManagerFactory можно безопасно использовать в многопоточном окружении.

10.1.6. Внедрение EntityManagerFactory

В предыдущем разделе было показано, как внедрить экземпляр EntityManager с помощью контейнера. Когда EntityManager внедряется контейнером, у вас почти не остается рычагов управления жизненным циклом диспетчера. В некоторых ситуациях бывает желательно иметь более полный контроль как над самим экземпляром EntityManager, так и над транзакциями. По этой причине, а также для большей гибкости, JPA предоставляет поддержку внедрения EntityManagerFactory. С помощью EntityManagerFactory можно создавать экземпляры EntityManager,

полностью находящиеся в вашей власти. Например, в листинге 10.2 приводится реализация компонента `ItemManager`, управляющего своим экземпляром `EntityManager` вручную.

Листинг 10.2. Компонент ItemManager, управляющий экземпляром EntityManager непосредственно

```
@Stateless
public class ItemManager {
    @PersistenceUnit
    // ❶ Переменная для внедрения экземпляра EntityManagerFactory
    private EntityManagerFactory entityManagerFactory;
    private EntityManager entityManager;

    @PostConstruct
    private void init() {
        // ❷ Создает EntityManager
        entityManager = entityManagerFactory.createEntityManager();
    }

    public Item updateItem(Item item) {
        // Явно подключает транзакцию JTA
        entityManager.joinTransaction();
        entityManager.merge(item);
        return item;
    }

    @PreDestroy
    private void cleanup() {
        if(entityManager.isOpen()) {
            // ❸ Закрывает EntityManager
            entityManager.close();
        }
    }
```

Код листинга достаточно прост. Он обеспечивает внедрение экземпляра `EntityManagerFactory` ❶ и с его помощью создает экземпляр `EntityManager` после создания компонента ❷. Здесь также предусмотрено закрытие `EntityManager` ❸ перед уничтожением компонента. Данная реализация помогает понять, что делает контейнер для создания `EntityManager`.

Ниже приводится определение аннотации `PersistenceUnit`:

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface PersistenceUnit {
    String name() default "";
    String unitName() default "";
}
```

Атрибуты `name` и `unitName` служат в точности тем же целям, что и одноименные атрибуты аннотации `@PersistenceContext`. Элемент `name` определяет имя экземпляра `EntityManagerFactory` в каталоге JNDI, а атрибут `unitName` – имя единицы хранения.

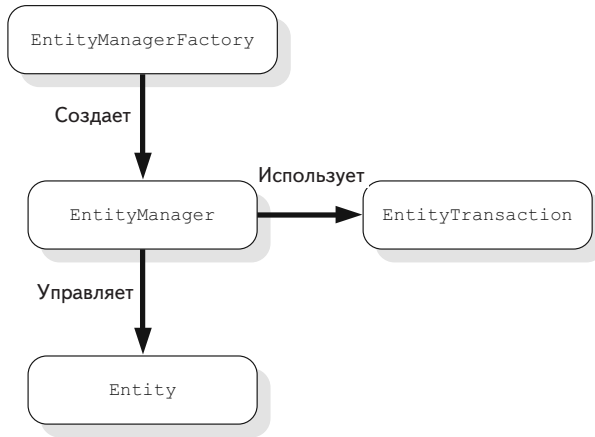

Рис. 10.5. Важнейшие классы/интерфейсы JPA, необходимые при использовании экземпляров EntityManager, управляемых приложением

На рис. 10.5 показаны связи между `EntityManagerFactory`, `EntityManager`, `EntityTransaction` и `Entity` в контексте. Мы пока не знакомы с `EntityTransaction` – этот объект предоставляет прикладной интерфейс для управления транзакцией, когда нет возможности делегировать эту работу контейнеру.

Как видно в листинге 10.2, объект `EntityManagerFactory` имеет метод `createEntityManager`, который создает и возвращает новый экземпляр `EntityManager`. После вызова этого метода на вас возлагается вся ответственность за своевременное уничтожение `EntityManager` – если этого не сделать, приложение могут ожидать разные неприятности.

Из этого примера следует один важный вывод: экземпляр `EntityManager`, управляемый приложением, не включается автоматически в управление транзакциями, поэтому необходимо использовать `EntityTransaction`, чтобы присоединиться к имеющейся транзакции. На практике все же лучше использовать `EntityManager`, управляемый контейнером, потому что чем меньше операций приходится выполнять вручную, тем меньше вероятность допустить ошибку.

Теперь, когда вы получили представление об объектах `EntityManager` и `EntityManagerFactory`, перейдем к знакомству с некоторыми функциями `EntityManager`.

10.2. Операции с хранилищем

Основу JPA API составляют операции `EntityManager`, о которых рассказывается в следующих разделах. Как можно было заметить в листинге 10.1, несмотря на простоту интерфейса `EntityManager`, он обеспечивает полный комплект методов для работы с хранилищем. В дополнение к CRUD-операциям, представленным в листинге 10.1, мы обсудим также редко используемые операции, такие как принудительная синхронизация и обновление. Итак, начнем наш обзор с самого логичного места: операции сохранения новых сущностей в базе данных.

10.2.1. Сохранение сущностей

Метод `addItem` из листинга 10.1 сохраняет сущность `Item` в базе данных. Реализация этого метода приводится в листинге 10.3. Хотя это и не очевидно, данный код демонстрирует, как сохраняются отношения между сущностями, на чем мы остановимся подробнее чуть ниже. А пока сосредоточимся на самом методе сохранения.

Листинг 10.3. Сохранение сущностей

```
public void addItem(String name, String description, byte[] picture,
        BigDecimal initialPrice, long sellerId) {
    Item item = new Item();
    item.setItemName(name);
    item.setPicture(picture);
    item.setInitialPrice(initialPrice);
    BazaarAccount seller = entityManager.find(BazaarAccount.class,sellerId);
    item.setSeller(seller);
    // Сохранение сущности
    entityManager.persist(item);
}
```

Новая сущность `Item`, соответствующая записи в базе данных, впервые создается в методе `addItem`. Все данные для сущности `Item`, подлежащие сохранению в базе, такие как название и описание продаваемого товара, передаются пользователем. Как вы наверняка помните из главы 9, сущность `Item` связана отношением «многие к одному» с сущностью `Seller`. Соответствующая сущность с информацией о продавце извлекается с помощью метода `find` объекта `EntityManager` и записывается в поле сущности `Item`. Затем вызывается метод `persist`, сохраняющий сущность в базе данных, как показано на рис. 10.6. Обратите внимание, что метод `persist` предназначен для создания новых записей в базе данных, а не для обновления существующих. Это означает, что вы должны обеспечить уникальность идентичности (первичного ключа) сущности. Если попытаться сохранить сущность со значением первичного ключа, уже имеющимся в базе данных, будет возбуждено исключение `EntityExistsException`, либо в момент вызова метода `persist`, либо в момент подтверждения транзакции.

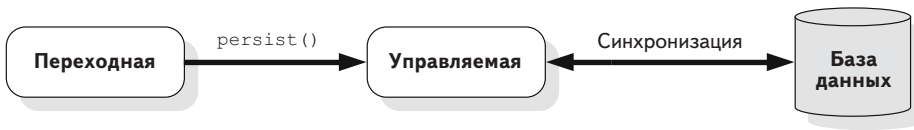

Рис. 10.6. Вызов метода `persist` экземпляра `EntityManager` делает сущность управляемой. Состояние сущности синхронизируется с базой данных при подтверждении транзакции

Если попытаться сохранить сущность, нарушающую какие-либо другие ограничения (например, ограничение уникальности), механизм хранения возбудит

соответствующий подкласс javax.persistence.PersistenceException, обертывающий исключение базы данных.

Как отмечалось выше, метод persist также делает сущность управляемой. Инструкция (или инструкции) INSERT, создающая запись, которая соответствует сущности, необязательно будет выполнена немедленно. Для экземпляров EntityManager с областью видимости, ограниченной рамками транзакции, инструкция обычно выполняется непосредственно перед подтверждением охватывающей транзакции. В данном случае это означает, что инструкции SQL будут выполнены после возврата из метода addItem. Для экземпляров EntityManager с расширенной областью видимости (или управляемых приложением), инструкция INSERT обычно выполняется перед тем, как EntityManager будет закрыт. Также инструкция INSERT может быть выполнена в любой другой момент, вызовом метода принудительной синхронизации.

Как отмечалось выше, все операции, требующие обновления базы данных, должны выполняться в рамках транзакций. Если вызвать метод persist за пределами транзакции, диспетчер сущностей возбудит исключение TransactionRequiredException. То же относится к метода flush, merge, refresh и remove.

10.2.2. Извлечение сущностей по ключу

Интерфейс JPA поддерживает несколько способов извлечения экземпляров сущностей из базы данных. Самый простой – извлечение сущности по первичному ключу с использованием метода find. Все остальные способы основаны на использовании прикладного интерфейса запросов и языка JPQL, о котором рассказывается в главе 11. Метод find уже использовался нами в методе addItem, представленном в листинге 10.3, для извлечения экземпляра Seller, соответствующего добавляемой сущности Item:

```
Seller seller = entityManager.find(Seller.class, sellerId);
```

В первом параметре методу find передается Java-тип извлекаемой сущности. Во втором параметре – значение идентичности сущности, которую надлежит извлечь. Как уже говорилось в главе 9, идентичностью сущности может быть значение простого типа, определяемое с помощью аннотации @Id, или класс составного первичного ключа, определяемый с помощью аннотации @EmbeddedId или @IdClass. В листинге 10.1 первый параметр метода find объявлен как значение простого типа java.lang.Long, соответствующее полю sellerId в сущности Seller, отмеченному аннотацией @Id.

Хотя это не видно в листинге 10.1, метод find полностью поддерживает составные первичные ключи. Чтобы увидеть, как мог бы выглядеть код, использующий такой ключ, на минуту предположим, что идентичность сущности Seller определяется не простым числовым значением, а именем и фамилией продавца. Данная идентичность заключена в класс составного первичного ключа, отмеченного аннотацией @IdClass. В листинге 10.4 показано, как можно заполнить и передать методу find экземпляр такого класса.

Листинг 10.4. Поиск с использованием составного первичного ключа

```
SellerPK sellerKey = new SellerPK();

sellerKey.setFirstName(firstName);
sellerKey.setLastName(lastName);

Seller seller = entityManager.find(Seller.class, sellerKey);
```

Метод `find` исследует определение класса сущности, указанного в первом параметре, и генерирует инструкцию `SELECT` для извлечения данных, подставляя в нее значения первичного ключа из второго параметра. Например, вызов метода `find` из листинга 10.3 мог бы сгенерировать такую инструкцию `SELECT`:

```
SELECT * FROM SELLERS WHERE seller_id = 1
```

Обратите внимание, что если экземпляр сущности, соответствующий указанному ключу, отсутствует в базе данных, метод `find` не будет возбуждать какое-либо исключение. Вместо этого `EntityManager` вернет `null`, то есть пустую сущность, и ваше приложение должно быть готово обработать эту ситуацию. Метод `find` допускается вызывать вне контекста транзакции, но в этом случае он вернет неподключенную сущность, поэтому в общем случае желательно вызывать `find` внутри транзакции. Одной из важнейших особенностей метода `find` является использование им внутреннего кэша диспетчера EntityManager. Если реализация механизма хранения поддерживает кэширование и сущность уже присутствует в кэше, `EntityManager` вернет экземпляр из кэша, минуя обращение к базе данных. Большинство реализаций механизмов хранения, таких как Hibernate и Oracle TopLink, поддерживает кэширование, поэтому вы можете рассчитывать на эту чрезвычайно полезную оптимизацию.

Интерфейс JPA обладает еще более важной особенностью, которую можно рассматривать как оптимизацию, – отложенная и немедленная загрузка. Инструкция `SELECT`, сгенерированная методом `find`, попытается извлечь все поля сущности, потому что такое поведение реализовано по умолчанию. Но иногда это поведение оказывается нежелательным. Возможность изменения режимов извлечения позволит вам оптимизировать скорость работы приложения.

Режимы извлечения сущностей

В предыдущих главах мы упоминали вскользь режимы извлечения, но не рассматривали их сколько-нибудь подробно. Обсуждение темы извлечения сущностей идеально подходит для более полного исследования режимов извлечения.

Как уже говорилось, при извлечении сущности `EntityManager` обычно загружает все ее данные из базы. На языке ORM это называется «жадной» (или немедленной) загрузкой. Если вам приходилось решать проблемы производительности приложений, связанные с досрочным или избыточным кэшированием, вы наверняка уже знаете, что немедленная загрузка не всегда является удачным решением. Классическим примером может служить загрузка больших двоичных объектов (BLOB), таких как изображения. Если только вы не разрабатываете программу

предназначенную для обработки графики, такую как электронный фотоальбом, весьма вероятно, что загрузка изображений, являющихся частью сущностей, во множестве мест в приложении, отрицательно скажется на его производительности. Так как загрузка блоков двоичных данных связана с выполнением продолжительных операций ввода/вывода, извлекать их следует, только когда они действительно необходимы. Вообще эта стратегия оптимизации называется «ленивым» (или отложенным) извлечением.

В JPA имеется несколько средств поддержки отложенного извлечения. Самое простое – отметить поле, подлежащее отложенному извлечению, аннотацией `@Basic`. Например, организовать отложенное извлечение свойства `picture` для сущности `Item` можно следующим образом:

```
@Column(name="PICTURE")
@Lob
@Basic(fetch=FetchType.LAZY)
public byte[] getPicture() {
    return picture;
}
```

В этом случае инструкция `SELECT`, сгенерированная методом `find`, не будет загружать данные из столбца `ITEMS.PICTURE` в поле `picture`. Изображение будет автоматически загружаться при первом обращении к свойству вызовом метода `getPicture`.

Однако, будьте внимательны. Отложенная загрузка – обоюдоострое оружие. Если для столбца указано, что для него используется режим отложенной загрузки, это означает, что `EntityManager` будет выполнять дополнительную инструкцию `SELECT` при первом обращении к этому свойству. А теперь представьте, что получится, если определить отложенный режим загрузки для всех свойств сущности. Базу данных затопит большое число инструкций `SELECT` при обращении к свойствам сущностей в программе. Кроме того, спецификация EJB 3 определяет поддержку отложенной загрузки как необязательную, а это означает, что не все механизмы хранения гарантируют ее реализацию. Обязательно проверьте документацию для механизма хранения, прежде чем тратить время, определяя, какие столбцы следует загружать в отложенном режиме.

Загрузка связанных сущностей

Одной из самых запутанных сторон использования режимов извлечения является управление извлечением связанных сущностей. Как известно, метод `find` объекта `EntityManager` должен извлечь все сущности, связанные отношениями с возвращаемой. Возьмем в качестве примера сущность `Item` из приложения ActionBazaar, которая отлично подходит на роль испытуемого, так как связана отношениями «многие к одному», «один ко многим» и «многие ко многим» с другими сущностями. Единственный тип отношений, не представленный в этой сущности, – «один к одному». Отношением «многие к одному» сущность `Item` связана с сущностью `Seller` (продавец (seller) может разместить несколько объявлений (items), но каждое объявление может быть размещено только одним продавцом);

отношением «один ко многим» – с сущностью `Bid` (для одного объявления может быть сделано несколько предложений цены); и отношением «многие ко многим» – с сущностью `Category` (объявление может быть включено в несколько категорий, а каждая категория может содержать несколько объявлений). Эти отношения изображены на рис. 10.7.

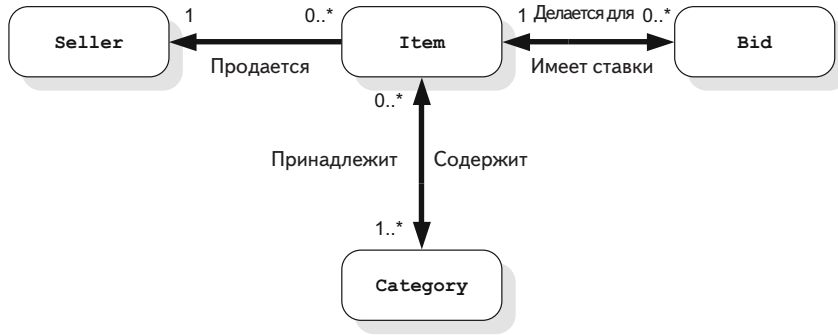

Рис. 10.7. Сущность Item связана отношениями «многие к одному», «один ко многим» и «многие ко многим» с тремя другими сущностями: Seller, Bid и Category, соответственно

Перед тем, как вернуть экземпляр `Item`, метод `find` автоматически извлекает также сущности `Seller`, `Bid` и `Category`, связанные с ним, и заполняет ими соответствующие свойства. Как показано в листинге 10.5, с `Item` связаны: единственная сущность `Seller`, экземпляр которой сохраняется в свойстве `seller`, множество сущностей `Bid`, которые сохраняются в списке `bids`, и множество сущностей `Category`, которые сохраняются в списке `categories`. Возможно, кто-то из вас удивится, узнав, что некоторые из этих отношений извлекаются в отложенном режиме.

Все аннотации отношений, представленные в главе 9, включая `@ManyToOne`, `@OneToMany` и `@ManyToMany`, имеют атрибут `fetch`, управляющий режимами извлечения, точно так же, как одноименный атрибут аннотации `@Basic`, описанной в предыдущем разделе. Ни одна из аннотаций отношений в следующем листинге не определяет атрибут `fetch`, поэтому для всех отношений используется режим извлечения по умолчанию.

Листинг 10.5. Отношения в сущности Item

```
public class Item {
    // "Один ко многим" с сущностями Bid
    @OneToMany(mappedBy = "item", cascade = CascadeType.ALL)
    private List<Bid> bids;

    // "Многие к одному" с сущностями Seller
    @ManyToOne
    @JoinColumn(name = "SELLER_ID", referencedColumnName = "USER_ID")
```

```
    private BazaarAccount seller;

    // "Многие ко многим" с сущностями Category
    @ManyToMany(mappedBy = "items")
    private Set<Category> category;

    ...
}
```

По умолчанию для некоторых отношений используется отложенный режим извлечения, а некоторые загружаются немедленно. Причины таких различий мы обсудим далее на примере сущности `Item`. Сущность `Seller`, связанная с `Item`, извлекается немедленно, потому что в аннотации `@ManyToOne` по умолчанию используется режим `EAGER`. Чтобы понять, почему выбрано такое значение по умолчанию, необходимо разобраться с тем, как реализован режим немедленной загрузки в `EntityManager`. В действительности, каждое немедленно извлекаемое отношение преобразуется в дополнительную инструкцию `JOIN`, приложенную к основной инструкции `SELECT`. Чтобы понять суть сказанного выше, посмотрите, как выглядит инструкция `SELECT`, немедленно извлекающая сущность `Seller`, связанную с сущностью `Item`.

Листинг 10.6. Инструкция SELECT, немедленно извлекающая сущность SELLER, связанную с сущностью Item

```
SELECT
      *
FROM
      ITEMS
-- Внутреннее соединение для отношения "многие к одному"
INNER JOIN
      SELLERS
ON
      ITEMS.SELLER_ID = SELLERS.SELLER_ID
WHERE ITEMS.ITEM_ID = 100
```

Как показано в листинге 10.6, наиболее естественным способом немедленного извлечения сущности `Item` является единственная инструкция `SELECT`, включающая инструкцию `JOIN` соединения таблиц `ITEMS` и `SELLERS`. Важно отметить тот факт, что такое соединение возвратит единственную строку, содержащую столбцы из двух таблиц, `SELLERS` и `ITEMS`. С точки зрения производительности, инструкция `JOIN` выполняется гораздо эффективнее, чем две инструкции `SELECT`, одна для извлечения `Item` и одна для извлечения `Seller`. Последний сценарий в точности описывает то, что происходит в режиме отложенного извлечения, только вторая инструкция `SELECT`, извлекающая `Seller`, выполняется при первом обращении к свойству `seller` сущности `Item`. То же самое относится и к аннотации `@OneToOne`, поэтому для нее по умолчанию так же используется режим немедленной загрузки. Точнее говоря, операция соединения таблиц позволяет добиться большей эффективности там, где она возвращает единственную строку.

Отложенная и немедленная загрузка связанных сущностей

Для аннотаций `@OneToMany` и `@ManyToMany`, напротив, используется режим отложенной загрузки. Обусловлено это тем, что оба отношения возвращают множество сущностей, соответствующих заданной. Например, представьте множество сущностей `Category`, связанных с извлекаемой сущностью `Item`. Операция соединения таблиц `JOIN` для отношений «один ко многим» и «многие ко многим» обычно возвращает более одной строки – по одной для каждой соответствующей сущности.

Проблема становится особенно очевидной, если представить извлечение сразу множества сущностей `Item` (например, в результате выполнения запроса JPQL; подробнее о запросах JPQL рассказывается в следующей главе). Базе данных придется вернуть ($N_1 + N_2 + ... + N_x$) строк, где N_i – число сущностей `Category`, связанных с i-й сущностью `Item`. Для больших чисел N и i число строк в результате может оказаться просто огромным и вызвать значительные задержки в работе базы данных. Именно поэтому в JPA применяется более консервативный подход и для аннотаций `@OneToMany` и `@ManyToMany` по умолчанию используется режим отложенной загрузки. В табл. 10.2 перечислены режимы извлечения по умолчанию для всех аннотаций отношений.

Таблица 10.2. Режимы по умолчанию загрузки связанных сущностей отличаются для разных типов отношений. Режим можно изменить, определив атрибут fetch

Тип отношения	Режим извлечения по умолчанию	Число извлекаемых сущностей
Один к одному	`EAGER`	Извлекается единственная сущность.
Один ко многим	`LAZY`	Извлекается коллекция сущностей.
Многие к одному	`EAGER`	Извлекается единственная сущность.
Многие ко многим	`LAZY`	Извлекается коллекция сущностей.

Однако, как бы то ни было, режим по умолчанию не всегда является оптимальным. Даже при том, что в большинстве случаев стратегия немедленного извлечения имеет смысл только для отношений «один к одному» и «многие к одному», иногда ее применение не выглядит удачным решением. Например, если сущность содержит большое число отношений «один к одному» и «многие к одному», немедленная загрузка всех связанных сущностей может привести к включению большого числа инструкций `JOIN` в запрос, большое число которых в одном запросе ничем не лучше попытки загрузить ($N_1 + N_2 + ... + N_x$) строк. Если это становится проблемой с точки зрения производительности, для некоторых из отношений следует определить режим отложенного извлечения. Ниже показано, как определить режим извлечения для отношения (так получилось, что это уже знакомое нам свойство `seller` в сущности `Item`):

```
@ManyToOne(fetch=FetchType.LAZY)
public Seller getSeller() {
    return seller;
}
```

Также иногда бывает желательно отказаться от режима отложенной загрузки, используемого по умолчанию аннотациями `@OneToMany` и `@ManyToMany`. В частности, при работе с большими наборами данных этот режим может привести к созданию значительного числа запросов `SELECT`. Эта проблема известна под названием «*N* + 1 инструкций SELECT», где «1» обозначает инструкцию `SELECT` для извлечения целевой сущности, а «*N*» – инструкции `SELECT` для извлечения связанных сущностей. Иногда можно обнаружить, что даже для аннотаций `@OneToMany` и `@ManyToMany` лучше использовать немедленный режим загрузки.

К сожалению, нет универсального правила, которым можно было бы руководствоваться при выборе режима извлечения, потому что на него влияет слишком большое число факторов, включая стратегии оптимизации, реализованные разработчиками баз данных, архитектура базы данных, объем хранимых данных и используемые в приложении шаблоны программирования. На практике выбор часто делается методом проб и ошибок. К счастью в JPA подобные настройки производительности могут выполняться с помощью нескольких конфигурационных параметров, без необходимости вносить изменения в программный код.

Обсудив извлечение сущностей, можно перейти к исследованию третьей операции в аббревиатуре CRUD: изменение (updating) сущностей.

10.2.3. Изменение сущностей

Как уже говорилось выше, `EntityManager` гарантирует автоматическое сохранение в базе данных изменений, произведенных в подключенных сущностях. Это означает, что по большей части вам не придется беспокоиться о вызове каких-либо методов вручную, чтобы сохранить изменения. Это, пожалуй, наиболее привлекательная особенность механизмов хранения данных на основе ORM, потому что она скрывает синхронизацию данных и позволяет воспринимать сущности как обычные POJO. Взгляните на код, представленный в листинге 10.7, который изменяет объявление о продаже. Часто после добавления объявления бывает необходимо дополнить описание или изменить фотографию, чтобы привлечь дополнительное внимание потенциальных покупателей.

Листинг 10.7. Прозрачное управление подключенной сущностью

```
@TransactionAttribute(REQUIRED)
public void updateItem(Long itemId, byte picture[],
        String description, StarRating starRating) {
    // Следующие инструкции прозрачно
    // сохраняют изменения в базе данных
    Item item = this.findItem(itemId);
    item.setPicture(picture);
    item.setDescription(description);
    item.setStarRating(starRating);
}
```

Поиск сущности `Item` – единственное, для чего используется `EntityManager` в методе `updateItem`. Однако, когда метод завершит свою работу, `EntityManager`

автоматически сохранит внесенные изменения в базе данных. Обычно фактическая запись производится в момент завершения транзакции, когда `EntityManager` пытается подтвердить все изменения. Но у вас есть возможность произвести сохранение принудительно в любой момент вызовом метода `flush`. Метод `flush` применяется ко всем сущностям, которые были подключены на этот момент. Момент синхронизации определяется параметром типа `FlushModeType`. Если передать в нем значение `FlushTypeMode.AUTO`, изменения будут сохранены в момент выполнения запроса. Если передать значение `FlushTypeMode.COMMIT`, изменения будут сохранены по завершении транзакции, в момент ее подтверждения.

Операции подключения и отключения сущностей

Управляемые сущности – это чрезвычайно удобно, но проблема в том, что довольно сложно обеспечить постоянное подключение сущностей. Иногда сущности приходится отключать и сериализовать для передачи в веб-слой, где они изменяются, находясь за пределами досягаемости `EntityManager`. Кроме того, как вы наверняка помните, сеансовые компоненты без сохранения состояния не могут гарантировать обслуживание запросов от одного и того же клиента одним и тем же экземпляром компонента. Это означает отсутствие гарантий обслуживания сущности одним и тем же экземпляром `EntityManager` в разных обращениях к компоненту и невозможность автоматизации выполнения операций с хранилищем.

Именно эту модель реализует сеансовый компонент `ItemManager`, как следует из листинга 10.1. Экземпляр `EntityManager`, используемый компонентом, имеет область видимости `TRANSACTION`. Так как компонент использует управляемые контейнером транзакции (Container-Managed Transactions, CMT), по завершении транзакции сущность автоматически отключается. Это означает, что сущности, возвращаемые сеансовым компонентом своим клиентам, всегда находятся в отключенном состоянии, также как новые сущности `Item`, созданные методом `addItem`:

```
public Item addItem(String title, String description,
    byte[] picture, double initialPrice, long sellerId) {
    Item item = new Item();
    item.setTitle(title);
    ...
    entityManager.persist(item);
    return item;
}
```

Рано или поздно может понадобиться повторно подключить сущность к контексту сохранения для синхронизации с базой данных. Для этой цели объект `EntityManager` предоставляет метод `merge` (см. рис. 10.8).

Не забывайте, что сущность, переданная методу `merge`, необязательно будет синхронизирована с базой данных немедленно – для этого может потребоваться некоторое время. Метод `merge` можно задействовать в компоненте `ItemManager`, например, для сохранения измененной сущности `Item` в базе данных:

```
public Item updateItem(Item item) {
    entityManager.merge(item);
    return item;
}
```

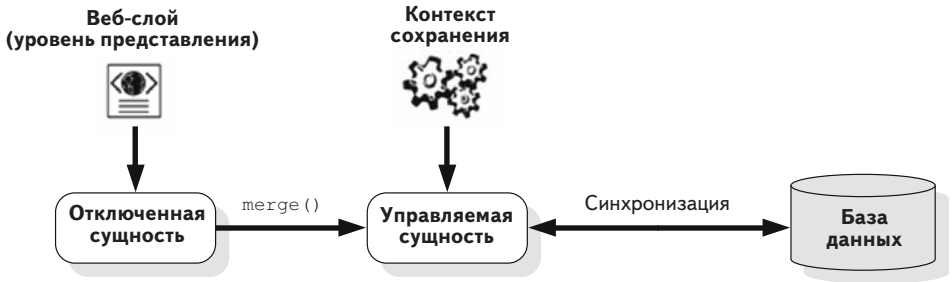

Рис. 10.8. Экземпляр сущности можно отключить, сериализовать и передать в веб-слой, где клиент внесет в него изменения и отправит обратно на сервер. После этого на сервере можно выполнить операцию merge, чтобы подключить сущность к контексту сохранения

Как только метод `updateItem` вернет управление, изменения в сущности `Item` будут сохранены в базе данных. Метод `merge` должен применяться, только к сущностям, существующим в базе данных. Попытка подключения несуществующей сущности приведет к исключению `IllegalArgumentException`. То же исключение будет возбуждено, если `EntityManager` обнаружит попытку подключения сущности, удаленной из базы данных вызовом метода `remove`, даже если инструкция `DELETE` еще не была выполнена.

Подключение отношений

По умолчанию сущности, связанные с подключаемой сущностью, не подключаются автоматически. Например, сущности `Seller`, `Bid` и `Category`, связанные с сущностью `Item`, не будут подключены предыдущим фрагментом кода вместе с сущностью `Item`. Но такое поведение можно изменить с помощью атрибута `cascade` аннотаций `@OneToOne`, `@OneToMany`, `@ManyToOne` и `@ManyToMany`. Атрибут `cascade` должен добавляться в аннотацию на стороне владельца отношения. Если атрибуту `cascade` присвоить значение `ALL` или `MERGE`, связанные сущности будут подключаться автоматически. Например, следующий фрагмент подключит сущность `Seller`, связанную с сущностью `Item`, потому что атрибуту `cascade` присвоено значение `MERGE`:

```
public class Item {
    @ManyToOne(cascade=CascadeType.MERGE)
    public Seller getSeller() {
```

Обратите внимание, что как и большинство методов `EntityManager`, метод `merge` должен вызываться в контексте транзакции, иначе будет возбуждено исключение `TransactionRequiredException`. А теперь перейдем к последней операции в аббревиатуре CRUD: удаление (deleting) сущности.

Отключенные сущности и антишаблон DTO

Имеющие сколько-нибудь серьезный опыт использования EJB 2 наверняка знакомы с антишаблоном использования объектов передачи данных (Data Transfer Object, DTO). Нужно признать, что антишаблон DTO в свое время был необходимым злом из-за сущностей-компонентов. Однако с выходом EJB 3 отключенные сущности превратились в POJO и надобность в антишаблоне DTO исчезла. Вместо того, чтобы создавать отдельные DTO для передачи предметных данных между уровнями прикладной логики и представления, для этой цели достаточно использовать отключенные сущности. Именно эта модель обсуждалась в данной главе.

Но, если ваши сущности дополнительно реализуют некоторую логику, возможно, вам вновь придется прибегнуть к шаблону DTO, чтобы предотвратить непреднамеренное ее использование вне контекста транзакции. В любом случае, если вы решите использовать отключенные сущности взамен DTO, в них следует реализовать поддержку интерфейса `java.io.Serializable`.

10.2.4. Удаление сущностей

Метод `deleteItem` компонента `ItemManagerBean`, определение которого приводится ниже, удаляет сущность `Item` из базы данных. Обратите внимание на одну важную особенность: перед удалением сущность `Item` подключается к контексту сохранения вызовом метода `merge` объекта `EntityManager`:

```
public void deleteItem(Item item) {
    entityManager.remove(entityManager.merge(item));
}
```

Объясняется это тем, что метод `remove` может удалять только подключенные сущности, а методу `deleteItem` передается отключенная сущность `Item`. Если методу `remove` передать отключенную сущность, он возбудит исключение `IllegalArgumentException`. Перед тем, как метод `deleteItem` вернет управление, запись, соответствующая указанной сущности, будет удалена из базы данных с помощью инструкции `DELETE`:

```
DELETE FROM ITEMS WHERE item_id = 1
```

Как и в случае с методами `persist` и `merge`, инструкция `DELETE` необязательно будет выполнена немедленно – гарантируется лишь, что она будет выполнена в некоторый момент. Однако `EntityManager` отметит сущность как удаленную и последующие попытки изменить ее потерпят неудачу (как описывается в предыдущем разделе).

Каскадирование операций удаления

Так же как при подключении и сохранении сущностей, чтобы обеспечить каскадное удаление сущностей, связанных отношениями с указанной, атрибуту `cascade` аннотаций отношений следует присвоить значение `ALL` или `REMOVE`. Например, ниже показано, как указать, что сущность `BillingInfo`, связанная с сущностью `Bidder`, должна удаляться вместе с родительской сущностью `Bidder`:

```
@Entity
public class Bidder {
    @OneToOne(cascade=CascadeType.REMOVE)
    public BillingInfo setBillingInfo() {
```

Если рассуждать логически, в этом есть определенный смысл. Нет никаких причин хранить сущность `BillingInfo` после удаления родительской сущности `Bidder`. Вообще говоря, необходимость каскадного удаления определяется предметной областью. Часто такой способ удаления имеет смысл только для отношений «один к одному» и «один ко многим». Но даже для этих отношений следует проявлять осторожность, применяя каскадное удаление, потому что связанные сущности также могут быть связаны отношениями с другими сущностями. Например, пусть у нас имеется отношение «один ко многим» между сущностями `Seller` и `Item`, и мы определили каскадный режим удаления сущности `Seller` и всех связанных с ней сущностей `Item`. А теперь вспомните, что другие сущности, такие как `Category`, также могут хранить ссылки на сущности `Item`, которые будут удалены, и тогда эти отношения окажутся бессмысленными![1]

Обслуживание отношений

Если вам действительно необходимо обеспечить каскадное удаление сущностей `Item`, связанных с удаляемой сущностью `Seller`, необходимо сначала обойти в цикле все экземпляры `Category`, ссылающиеся на удаляемые сущности `Item`, и разорвать связь, как показано ниже:

```
List<Category> categories = getAllCategories();
List<Item> items = seller.getItems();
for (Item item: items) {
    for (Category category: categories) {
        category.getItems().remove(item);
    }
}
entityManager.remove(seller);
```

Этот код извлечет все экземпляры `Category` в системе[2] и удалит все ссылки на сущности `Item`, связанные с удаляемой сущностью `Seller` из списков ссылок. Затем он удалит указанную сущность `Seller` и связанные с ней сущности `Item`.

Разумеется, метод `remove` должен вызываться в контексте транзакции, иначе он возбудит исключение `TransactionRequiredException`. Кроме того, попытка удалить уже удаленную сущность возбудит исключение `IllegalArgumentException`.

Мы закончили знакомство с особенностями простых CRUD-операций в `EntityManager`. Далее мы займемся исследованием еще двух важных операций с

---

[1] Не самый удачный пример: исчезновение объявления из категории никак не скажется на самой категории. Гораздо больше неприятностей в этом случае может доставить отношение между сущностями `Bid` и `Item`, так как приведет к появлению по-настоящему бессмысленных сущностей `Bid`. – *Прим. перев.*

[2] Обратите внимание: здесь имеются в виду не категории в базе данных, а уже извлеченные сущности `Category`, хранящиеся в памяти приложения. – *Прим. перев.*

хранилищем: принудительная синхронизация изменений с базой данных и принудительное обновление сущностей из базы данных.

10.3. Запросы сущностей

Запросы к базе данных играют не менее важную роль, чем сами данные. После сохранения данных в базе тут же возникает потребность в механизме извлечения их оттуда. В JPA имеется несколько разных способов выполнения запросов, от JQL до Criteria API, низкоуровневые запросы SQL и запросы к хранимым процедурам. Каждый имеет свои плюсы и минусы, в зависимости от ваших потребностей.

До настоящего момента мы извлекали сущности с помощью метода `find` экземпляра `EntityManager`. Метод `find` позволяет выполнять запросы, извлекающие сущности по первичному ключу. Хотя это и удобно, но часто бывает желательно выполнить поиск по каким-то другим критериям. Для этого можно воспользоваться интерфейсом `javax.persistence.Query`. Этот интерфейс позволяет определять запросы, передавать им параметры, выполнять и извлекать данные постранично.

Существует несколько типов запросов, исследованием которых мы займемся в этой и в следующей главе:

- `javax.persistence.Query` – представляет запрос на языке JPQL или SQL. Для типизированных запросов возвращается экземпляр `javax.persistence.TypeQuery<T>`, избавляющий от необходимости выполнять приведение типов результатов.
- `javax.persistence.StoredProcedureQuery` – представляет запрос, вызывающий хранимую процедуру.
- `javax.persistence.criteria.CriteriaQuery` – представляет запрос, сконструированный с помощью метамодели.

В дополнение к запросам этих типов имеются также динамические и именованные запросы. Динамические запросы – это запросы, создаваемые программно, в процессе выполнения приложения, и передаваемые экземпляру `EntityManager` для выполнения. Именованные запросы – это запросы, определяемые в аннотациях или в конфигурационных файлах ORM XML. Именованные запросы упрощают многократное их использование и помогают избежать захламления программного кода. Давайте исследуем эти два типа запросов подробнее.

Запросы и абстрактные запросы

В этой главе мы познакомимся с несколькими разными типами запросов, включая JPQL, SQL и Criteria. Несмотря на сходство имен, интерфейсы `javax.persistence.criteria.CriteriaQuery` и `javax.persistence.Query` не имеют общих предков. Эти два API существенно отличаются друг от друга. Как будет показано далее, Criteria Query API позволяет конструировать запросы с использованием объектов Java, без привлечения строк в свободной форме, которые интерпретируются во время выполнения.

10.3.1. Динамические запросы

Динамические запросы конструируются «на лету», во время выполнения программы. Обычно динамические запросы используются однократно – то есть один запрос не используется совместно множеством компонентов. Если у вас есть опыт использования JDBC (`Statements` или `PreparedStatements`), этот подход покажется вам хорошо знакомым. Динамические запросы могут включать код на языке JPQL или SQL. Например:

```
@PersistenceContext
private EntityManager entityManager;
public List<Category> findAllCategories() {
    TypedQuery<Category> query = entityManager.createQuery(
        "SELECT c FROM Category c",Category.class);
    return query.getResultList();
}
```

В этом примере используется экземпляр `EntityManager`, полученный путем внедрения зависимостей. Метод `findAllCategories` создает экземпляр `TypedQuery`, используя поддержку обобщенных типов (Generics) в Java, чтобы избежать необходимости выполнять приведение типов результатов. После создания запроса вызывается его метод `getResultList()`, который выполняет запрос и возвращает результаты. Динамические запросы можно также использовать для выполнения инструкций на языке SQL и вызова хранимых процедур, создавая их с помощью методов `createNativeQuery` и `createStoredProcedureQuery`, соответственно.

10.3.2. Именованные запросы

В отличие от динамических запросов, именованные запросы должны создаваться заранее. Их можно определять в аннотациях или в XML-файле с настройками объектно-реляционного отображения. Обращение к именованному запросу осуществляется по его имени, что позволяет многократно использовать его в разных программных компонентах. Именованный запрос можно определить с помощью аннотации `@javax.persistence.NamedQuery`:

```
@Entity
@NamedQuery(
    name = "findAllCategories",
    query = "SELECT c FROM Category c WHERE c.categoryName
        LIKE :categoryName ")
    public class Category implements Serializable {
        public List<Category> findAllCategories() {
            TypedQuery<Category> query =
                entityManager.createNamedQuery("findAllCategories",Category.class);
        }
    }
}
```

В больших приложениях может потребоваться определить множество именованных запросов. В таких ситуациях можно воспользоваться аннотацией `@javax.persistence.NamedQueries`, как показано ниже:

```
@Entity
@NamedQueries({
    @NamedQuery(
        name = "findCategoryByName",
        query = "SELECT c FROM Category c WHERE c.categoryName
            LIKE :categoryName order by c.categoryId"
    ),
    @NamedQuery(
        name = "findCategoryByUser",
        query = "SELECT c FROM Category c JOIN c.user u
            WHERE u.userId = ?1"
)})
@Table(name = "CATEGORIES")
public class Category implements Serializable {
    ...
}
```

Примечание. *Имейте в виду, что область видимости именованного запроса распространяется на всю единицу хранения и потому его имя должно быть уникальным для этой единицы. Мы рекомендуем выработать свое соглашение об именовании запросов, чтобы уменьшить риск конфликтов имен в приложении.*

Это было лишь краткое введение в поддержку запросов. С помощью методов `find`, `createQuery`, `createNativeQuery` и `createStoredProcedureQuery` экземпляра `EntityManager` можно извлекать из базы данных любые сущности, соответствующие необходимым критериям. Исследование достоинств и недостатков JPQL – языка запросов для JPA – является темой следующей главы, где вы узнаете, что нужно сделать, чтобы извлечь только те данные, которые вам необходимы.

10.4. В заключение

В этой главе мы исследовали возможности объекта `EntityManager`. Вы узнали, как с его помощью сохранять, находить, подключать и удалять сущности. Мы обсудили контексты сохранения и области видимости, новые/отключенные и управляемые сущности, а также возможность подключения сущностей. Дополнительно была упомянута возможность создания с помощью `EntityManagerFactory` экземпляра `EntityManager`, управляемого приложением, с целью получения более полного контроля над механизмом хранения и транзакциями. В заключение мы коротко обсудили возможность выполнения запросов – сначала мы сравнили динамические и именованные запросы, а затем покрыли основы поддержки запросов в JPA с простыми примерами применения JPQL, Criteria API и SQL.

В следующей главе мы подробно остановимся на языке запросов JPQL. Вы познакомитесь с синтаксисом этого языка и научитесь извлекать любые необходимые вам данные. Там же мы исследуем возможности Criteria API – типизированного механизма запросов для извлечения сущностей.

ГЛАВА 11. JPQL

Эта глава охватывает следующие темы:

- создание и выполнение запросов;
- язык запросов Java Persistence Query Language;
- использование Criteria API;
- использование запросов SQL;
- вызов хранимых процедур.

В главе 10 мы познакомились с прикладным интерфейсом объекта `EntityManager`. Вы узнали, как получить экземпляр `EntityManager` из контейнера и как использовать его для выполнения простых CRUD-операций (создание, чтение, изменение, удаление) с сущностями. Создание, изменение и удаление – это довольно простые операции и они были охвачены в главе 10 достаточно полно. Однако операция чтения намного сложнее, потому что существует множество способов извлечения данных из базы.

В этой главе мы подробнее исследуем операцию чтения данных с применением языка запросов Java Persistence Query Language (JPQL), Criteria API и языка запросов SQL. Каждый из этих способов по-своему подходит к извлечению объектов из реляционной базы данных и к решению разных проблем. В конце этой главы мы рассмотрим поддержку вызова хранимых процедур – новой особенности, появившейся в Java EE 7. Итак, начнем эту главу с погружения в JPQL.

11.1. Введение в JPQL

Основную часть этой главы занимает обсуждение достоинств и недостатков JPQL. Сначала мы познакомимся с определением языка, рассмотрим несколько примеров, иллюстрирующих почти все его аспекты, и попутно рассмотрим некоторые малоизвестные рекомендации.

В группе EJB 3 Expert Group недолго длились дебаты, касающиеся выбора стандартного языка запросов для JPA. Участники быстро пришли к единому мнению, что лучшим выбором будет язык запросов JPQL. JPQL – это расширение для EJB QL, языка запросов в EJB 2. Бессмысленно изобретать новый язык в такой хорошо изученной области, поэтому группа единодушно выбрала язык запросов EJB QL в качестве основы. Благодаря этому, миграция приложений с EJB 2 на EJB 3 осуществляется удивительно просто.

Чем отличается JPQL от SQL?

JPQL оперирует классами и объектами (сущностями) в пространстве Java. SQL оперирует таблицами, столбцами и строками в пространстве баз данных. Хотя JPQL и SQL очень похожи, они оперируют разными мирами.

Парсер, или механизм обработки запросов JPQL, как показано на рис. 11.1, преобразует запрос на языке JPQL в запрос на языке SQL для используемой базы данных.

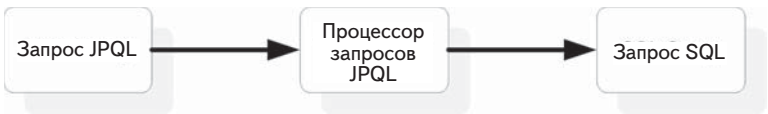

Рис. 11.1. Все запросы JPQL преобразуются в запросы SQL процессором JPQL и затем выполняются базой данных. Процессор запросов входит в состав JPA и обычно реализуется производителем сервера приложений

Язык JPQL настолько похож на язык SQL, что при просмотре исходного кода их легко спутать. Однако, не следует забывать, что несмотря на такое сходство, JPQL и SQL используются совершенно по-разному, и эти различия, обсуждаемые в данной главе, следует знать, чтобы эффективно использовать JPQL в программах.

Надеемся, что этими пространными рассуждениями мы пробудили в вас интерес к запросам на языке JPQL. Что вы скажете, если мы предложим вам продолжить эту тему и заняться исследованием разных типов инструкций JPQL? Далее мы обсудим разные конструкции языка JPQL, такие как инструкции `FROM` и `SELECT`, условные инструкции, подзапросы и разнообразные функции. В заключение этого обсуждения мы рассмотрим инструкции изменения и удаления данных.

11.1.1. Типы инструкций

Язык JPQL поддерживает три типа инструкций, перечисленных в табл. 11.1. На языке JPQL можно писать запросы, выполняющие выборку данных, их изменение и удаление.

Для начала давайте посмотрим, как осуществляется извлечение сущностей с помощью инструкции `SELECT`.

Таблица 11.1. Типы инструкций, поддерживаемые JPQL

Тип	Описание
Выборка	Извлекают сущности и связанные с ними данные.
Изменение	Изменяют одну или более сущностей.
Удаление	Удаляют одну или более сущностей.

Инструкция SELECT

Начнем наши исследования с простого запроса JPQL:

```
SELECT c
FROM Category c
WHERE c.categoryName LIKE :categoryName
ORDER BY c.categoryId
```

Этот запрос включает (или может включать) следующие элементы:

- обязательную инструкцию `SELECT`, определяющую тип объекта, сущность или значения, подлежащие извлечению;
- обязательное предложение `FROM`, определяющее объявление сущности, используемой другими инструкциями;
- необязательное предложение `WHERE`, выполняющее фильтрование результатов, возвращаемых запросом;
- необязательное предложение `ORDER BY`, выполняющее сортировку результатов;
- необязательное предложение `GROUP BY`, выполняющее агрегирование;
- необязательное предложение `HAVING`, выполняющее фильтрование в сочетании с агрегированием.

Инструкции UPDATE и DELETE

В предыдущей главе мы обсуждали изменение и удаление сущностей с использованием методов `EntityManager`. Но их возможности ограничены единственным экземпляром сущности. А как быть, если потребуется удалить более одной сущности сразу? Подобно языку SQL, JPQL имеет инструкции `UPDATE` и `DELETE`, выполняющие изменение и удаление сущностей, и также допускают использование предложения `WHERE`. Эти инструкции очень похожи на подобные им инструкции в языке SQL. Их часто называют инструкциями массового изменения или удаления, потому что в первую очередь используются для изменения и удаления множества сущностей, соответствующих определенным критериям. В этом разделе мы ограничимся обсуждением синтаксиса JPQL, применяемого для реализации операций изменения и удаления.

Использование инструкции UPDATE

В инструкции `UPDATE` можно указать только один тип сущностей и определить предложение `WHERE`, чтобы сузить круг сущностей, затрагиваемых инструкцией. Вот как выглядит синтаксис инструкции `UPDATE`:

```
UPDATE entityName indentifierVariable
SET single_value_path_expression1 = value1, ...
single_value_path_expressionN = valueN
WHERE where_clause
```

В предложении SET можно использовать любые хранимые поля и поля-отношения, представляющие одиночные значения. Допустим, что мы захотели присвоить «золотой» статус и комиссионную скидку 10% всем продавцам, фамилия которых начинается с «Packrat». Сделать это можно с помощью следующего запроса JPQL:

```
UPDATE Seller s
SET s.status = 'Gold', s.commissionRate = 10
WHERE s.lastName like 'Packrat%'
```

Совершенно очевидно, что предложение WHERE в инструкции UPDATE действует точно так же, как в инструкции SELECT. Однако более подробно предложение WHERE будет обсуждаться далее в этой главе.

Использование инструкции DELETE

Подобно инструкции UPDATE, инструкция DELETE в языке JPQL напоминает родственную ей инструкцию в языке SQL. В ней можно указать только один тип сущностей и определить предложение WHERE, чтобы сузить круг сущностей, на которые будет оказано влияние. Вот как выглядит синтаксис инструкции DELETE:

```
DELETE entityName indentifierVariable
WHERE where_clause
```

Например, удалить все экземпляры сущностей, представляющие «серебряных» продавцов, можно с помощью следующего запроса:

```
DELETE Seller s
WHERE s.status = 'Silver'
```

11.1.2. Предложение FROM

Предложение FROM является едва ли не самым важным в языке JPQL. Оно определяет область действия запроса – имена сущностей, на которые будет распространено действие запроса. Чтобы создать запрос, извлекающий сущности Category, предложение FROM следует определить так:

```
FROM Category c
```

Category – это область действия запроса, а c – это идентификатор типа Category.

Идентификация области действия запроса: именование сущностей

Имена сущностей, которые будут играть роль области действия запроса, можно определять в аннотациях @Entity, с помощью атрибута name. Если атрибут name

не определен, по умолчанию ему присваивается имя класса сущности. Имя сущности должно быть уникальным в пределах единицы хранения. Иными словами, нельзя определить две сущности с одинаковыми именами, иначе при развертывании приложения будет сгенерирована ошибка. В этом есть определенный смысл, потому что иначе механизм хранения не сможет определить, на какие сущности должен воздействовать запрос.

В предыдущем примере предполагалось, что аннотация `@Entity` перед классом `Category` не имеет атрибута `name`. Однако ничто не мешает определить другое имя сущностей класса `Category` для использования в запросах:

```
@Entity(name = "CategoryEntity")
public class Category
```

и тогда предложение `FROM` в запросе должно быть изменено, как показано ниже:

```
FROM CategoryEntity c
```

Это изменение необходимо, чтобы парсер JPQL смог правильно отобразить тип сущности.

Переменные

В примере выше определена переменная `c`, которую можно использовать в других инструкциях и предложениях, таких как `SELECT` и `WHERE`. В общем случае синтаксис определения простой переменной имеет следующий вид:

```
FROM entityName [AS] identificationVariable
```

Квадратные скобки (`[]`) не являются частью синтаксиса, а лишь показывают, что оператор `AS` является необязательным. Идентификатор переменной (который, кстати, не чувствителен к регистру) должен быть допустимым идентификатором с точки зрения синтаксиса Java и не должен совпадать с зарезервированными идентификаторами языка JPQL, которые перечислены в табл. 11.2. Имейте также в виду, что идентификатор не может совпадать с именем любой другой сущности в той же единице хранения.

Таблица 11.2. Ключевые слова JPQL, которые нельзя использовать в качестве переменных идентификаторов

Типы	Зарезервированные слова
Инструкции и предложения	SELECT, UPDATE, DELETE, FROM, WHERE, GROUP, HAVING, ORDER, BY, ASC, DESC
Инструкции соединения	JOIN, OUTER, INNER, LEFT, FETCH
Условные инструкции и операторы	DISTINCT, OBJECT, NULL, TRUE, FALSE, NOT, AND, OR, BETWEEN, LIKE, IN, AS, UNKNOWN, EMPTY, MEMBER, OF, IS, NEW, EXISTS, ALL, ANY, SOME
Функции	AVG, MAX, MIN, SUM, COUNT, MOD, UPPER, LOWER, TRIM, POSITION, CHARACTER_LENGTH, CHAR_LENGTH, BIT_LENGTH, CURRENT_TIME, CURRENT_DATE, CURRENT_TIMESTAMP

То есть, нельзя определить следующее предложение FROM:

```
FROM Category User
```

или:

```
FROM Category Max
```

потому что в приложении ActionBazaar уже есть сущность с именем User, а MAX – это зарезервированный идентификатор.

В предложении FROM можно определить несколько переменных, но подробнее о том, в каких случаях эта возможность используется, будет рассказываться далее в этой главе, при обсуждении операции соединения нескольких сущностей.

Выражение маршрута

В примерах запросов JPQL выше использовались выражения, такие как c.categoryName и c.categoryId. Их называют выражениями маршрутов (path expressions). Выражение маршрута – это переменная, за которой следует оператор навигации (.) и хранимое поле или поле связи. Обычно выражения маршрутов используются для сужения области действия запроса в предложениях WHERE или для сортировки – в предложениях ORDER BY.

Поле связи может содержать либо единичное значение-объект, либо коллекцию. Поля связи, представляющие отношения «один ко многим» и «многие ко многим» имеют тип коллекции, соответственно и выражение маршрута имеет тип коллекции. Например, в запросе можно использовать отношение «многие ко многим» между Category и Item для поиска всех сущностей Category, имеющих связанные с ними объявления:

```
SELECT distinct c
FROM Category c
WHERE c.items isn't EMPTY
```

Здесь c.items имеет тип коллекции. Такие выражения известны как выражения-коллекции. Поля связи, представляющие отношения «один к одному» и «многие к одному», имеют тип одиночного объекта, соответственно и выражение маршрута имеет тип одиночного объекта.

С помощью выражения маршрута, имеющего тип одиночного объекта, можно обращаться к другим хранимым полям и полям связи. Например, если допустить, что между Category и User имеется отношение «многие к одному», используя поле связи user в выражении маршрута можно обратиться к хранимому полю, такому как firstName:

```
c.user.firstName
```

или, используя поле связи contactDetails, обратиться к полю email с адресом электронной почты:

```
c.user.contactDetails.email
```

Используя выражения маршрутов, имейте в виду, что они не позволяют обращаться к значениям-коллекциям хранимых полей или полей связи, как показано в следующем примере:

```
c.items.itemName
```

или

```
c.items.seller
```

Это обусловлено отсутствием поддержки доступа к отдельным элементам коллекций. выражение `c.items.itemName` на языке `JPQL` сродни выражению `category.getItems().getItemName()` на языке Java, которое является недопустимым. Далее мы посмотрим, как можно использовать выражения маршрутов в предложении `WHERE`.

Фильтрация с помощью предложения WHERE

Предложение `WHERE` в языке JPQL позволяет фильтровать результаты запроса и возвращать только сущности, соответствующие заданному критерию. Допустим, что требуется извлечь все экземпляры сущности `Category`. Сделать это можно с помощью инструкции `SELECT` без предложения `WHERE`:

```
SELECT c
FROM Category c
```

Этот запрос может вернуть тысячи экземпляров `Category`. А теперь допустим, что в действительности требуется извлечь только экземпляры `Category`, соответствующие некоторому условию. Например, чтобы извлечь экземпляры `Category` со значением поля `categoryId` больше 500, этот запрос необходимо переписать, как показано ниже:

```
SELECT c
FROM Category c
WHERE c.categoryId > 500
```

В предложении `WHERE` поддерживаются литералы практически всех типов языка Java, таких как `boolean`, `float`, `enum`, `String`, `int` и так далее. Однако вы не сможете использовать восьмеричные и шестнадцатеричные числа, а также типы массивов, такие как `byte[]` или `char[]`. Не забывайте, что инструкции `JPQL` преобразуются в инструкции языка SQL – именно SQL ограничивает возможность использования типов BLOB и CLOB в предложении `WHERE`.

Передача параметров: позиционных и именованных

Как уже говорилось выше, запросы JPQL поддерживают два типа параметров: позиционные и именованные. Далее в этой главе будет показано, как устанавливать значения в этих параметрах.

В параметрах можно передавать не только числовые или строковые значения – тип значения зависит от типа выражения маршрута в предложении `WHERE`. Пара-

метры могут иметь более сложные типы, такие как типы других сущностей, но в условных выражениях допускается использовать только выражения маршрутов, имеющие тип одиночного значения.

Условные операторы и выражения

Условия в предложении WHERE, осуществляющем фильтрацию результатов запроса, называют условными выражениями. Условные выражения можно конструировать из выражений маршрутов и операторов, поддерживаемых языком. JPQL может вычислять выражения маршрутов и сопоставлять их с числовыми, строковыми или логическими значениями с помощью операторов отношений. Ниже приводится пример условного выражения:

```
c.categoryName = 'Dumped Cars'
```

В табл. 11.3 перечислены типы операторов, поддерживаемых JPQL, в порядке предшествования

Таблица 11.3. Типы операторов, поддерживаемые JPQL

Тип	Оператор
Навигационные	.
Унарные	+, -
Арифметические	*, /, +, -
Операторы отношений	=, >, >=, <, <=, <>, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF]
Логические	NOT, AND, OR

Сложные условные выражения могут включать другие условные выражения, объединенные логическими операторами, такими как AND или OR. Например, вот как можно реализовать извлечение категорий, соответствующих любому из двух условий:

```
WHERE c.categoryName = 'Dumped Cars'
      OR c.categoryName = 'Furniture from Garbage'
```

С выражениями маршрутов числовых типов можно использовать все типы операторов отношений. А с операндами типов String и Boolean – операторы равенства (=) и неравенства (<>).

Проверка на вхождение в диапазон с помощью BETWEEN

Оператор BETWEEN используется для сравнения переменной с диапазоном значений. Его можно использовать для сопоставления выражений маршрутов с нижней и верхней границами, представленными арифметическими или строковыми выражениями, а также датами и временем, как показано ниже:

```
path_expression [NOT] BETWEEN lowerRange and upperRange
```

Допустим, что необходимо отфильтровать результаты так, чтобы значение `categoryId` находилось внутри указанного диапазона. С этой целью можно использовать предложение `WHERE` и именованные параметры, определяющие границы диапазона:

```
WHERE c.categoryId BETWEEN :lowRange AND :highRange
```

Примечание. *Нижняя и верхняя границы в операторе* `BETWEEN` *должны быть значениями одного типа.*

Оператор IN

Оператор `IN` позволяет создавать условные выражения, проверяющие вхождение значения выражения маршрута в список значений. Ниже приводится синтаксис оператора `IN`:

```
path_expression [NOT] IN (List_of_values)
```

Список может быть статическим списком значений, разделенных запятой, или динамическим, создаваемым с помощью подзапроса. Допустим, что требуется извлечь только сущности `User`, значение поля `userId` в которых входит в статический список. Эту задачу решает следующее предложение `WHERE`:

```
WHERE u.userId IN ('viper', 'drdba', 'dumpster')
```

Если потребуется определить обратное условие, то есть извлечь сущности `User`, значение поля `userId` в которых не входит в статический список, можно воспользоваться следующим предложением `WHERE`:

```
WHERE u.userId not IN ('viper', 'drdba', 'dumpster')
```

Подзапрос – это запрос внутри запроса. Подзапросы могут возвращать единственные значения или множество значений. Подробнее о подзапросах рассказывается в разделе 11.1.4. Давайте рассмотрим пример использования подзапроса в операторе `IN`:

```
WHERE c.user IN (SELECT u
                 FROM User u
                 WHERE u.userType = 'A')
```

В этом выражении выполняется попытка сопоставить значение поля типа `User` со списком пользователей, извлекаемых подзапросом. Когда запрос содержит подзапрос, сначала выполняется подзапрос, а затем выполняется родительский запрос, который использует результаты, извлеченные подзапросом.

Оператор LIKE

Оператор `LIKE` позволяет определять соответствие выражение маршрута типа одиночного значения строковому шаблону и имеет следующий синтаксис:

```
string_value_path_expression [NOT] LIKE pattern_value_
```

Здесь pattern_value – это строковый литерал или входной параметр, который может содержать символы подчеркивания (_) и знаки процента (%). Символ подчеркивания соответствует любому одиночному символу. Взгляните на следующее предложение WHERE:

```
WHERE c.itemName LIKE '_ike'
```

Это выражение вернет TRUE для любого поля c.itemName, имеющего значение mike, bike и так далее. Следуя этой аналогии можно сконструировать сколь угодно сложную строку шаблона, замещая изменяющиеся символы символом подчеркивания. Если строка шаблона состоит из единственного символа подчеркивания, соответствовать ей будут только односимвольные строки.

Знак процента (%) соответствует любому числу символов. Если потребуется найти все категории, названия которых начинаются с «Recycle», это можно сделать с помощью следующего предложения WHERE:

```
WHERE c.categoryName LIKE 'Recycle%'
```

Выражение будет возвращать TRUE для полей c.categoryName, имеющих, например, такие значения: «Recycle from Garbage», «Recycle from Mr. Dumpster» и «RecycleMania – the Hulkster strikes again!».

Допустим, что нужно отфильтровать результаты так, чтобы остались только те данные, для которых строковое выражение не соответствует шаблону. Для этого в сочетании с оператором LIKE можно использовать оператор NOT, как показано в следующем примере:

```
WHERE c.categoryName NOT LIKE '%Recycle%'
```

Выражение будет возвращать FALSE для всех полей c.categoryName, значение которых включает слово «Recycle» в любом месте, потому что в этом шаблоне присутствует два символа % – до и после слова «Recycle».

В большинстве приложений, для большей гибкости, предпочтительнее применять параметры вместо строковых литералов. С этой целью можно использовать позиционные параметры, как показано ниже:

```
WHERE c.categoryName NOT LIKE ?1
```

Запрос с этим предложением WHERE вернет все категории, в которых значение c.categoryName не соответствует шаблону, переданному в позиционном параметре ?1.

Значения NULL и пустые коллекции

До сих пор нам удавалось избегать обсуждения пустых значений (null) и их обработки в выражениях. Но теперь пришло время заняться этой маленькой тайной. Вы должны помнить, что значение null отличается от пустой строки и в запросах на языке JPQL они интерпретируются по-разному. Но не все базы данных интерпретируют значение null и пустые строки по-разному. Вы уже знаете, что запросы на языке JPQL транслируются в запросы на языке SQL. Если база данных возвращает TRUE при сравнении пустой строки со значением null, это значит,

что вы не сможете рассчитывать на непротиворечивость результатов между двумя разными базами данных. Мы рекомендуем проверить, как обстоит дело в вашей базе данных.

Когда условное выражение встречает пустое значение, оно возвращает значение `null` или `UNKNOWN`. Сложное предложение `WHERE`, состоящее из нескольких условных выражений, объединенных логическими операторами, такими как `AND`, может производить пустые или неизвестные результаты. В табл. 11.4 приводится таблица истинности, где перечислены результаты условных выражений, в которых участвует пустое значение.

Таблица 11.4. Таблица истинности операций с логическим операндом и пустым значением

Значение выражения 1	Логический оператор	Значение выражения 2	Результат
TRUE	AND	null	UNKNOWN
FALSE	AND	null	FALSE
NULL	AND	null	UNKNOWN
TRUE	OR	null	TRUE
NULL	OR	null	UNKNOWN
FALSE	OR	null	UNKNOWN
	NOT	null	UNKNOWN

Для проверки результата выражения, возвращающего единственное значение, можно использовать оператор `IS NULL` или `IS NOT NULL`. Для выражения, имеющего пустое значение, оператор `IS NULL` вернет `TRUE`, а оператор `IS NOT NULL` вернет `FALSE`. Например, если потребуется определить, возвращает ли выражение маршрута непустое значение, используйте следующее предложение `WHERE`:

```
WHERE c.parentCategory IS NOT NULL
```

С помощью сравнения `IS NULL` нельзя проверить выражение, возвращающее коллекцию. Проще говоря, `IS NULL` не позволяет проверить, является ли коллекция пустой. Для этой цели в языке JPQL предусмотрена конструкция `IS [NOT] EMPTY`. Ниже приводится предложение `WHERE`, которое оставит в возвращаемом наборе только те категории, которые не содержат ни одного объявления:

```
WHERE c.items IS EMPTY
```

Как уже объяснялось выше, инструкции JPQL преобразуются в инструкции SQL. Однако в языке SQL отсутствует конструкция, эквивалентная конструкции `IS EMPTY`. Поэтому многим из вас будет любопытно узнать, какие инструкции SQL генерируются оператором `IS EMPTY`. Оператор `IS EMPTY` применяется к выражениям, возвращающим коллекции, обычно – к полям, описывающим отношения, поэтому сгенерированная инструкция SQL определит, возвращает ли инструкция `JOIN` для указанного отношения хотя бы одну запись в подзапросе. Чтобы было более понятно, о чем идет речь, рассмотрим следующий запрос JPQL:

```
SELECT c
FROM Category c
WHERE c.items IS EMPTY
```

Как вы наверняка помните, в главе 9 говорилось, что сущности `Category` и `Item` связаны отношением «многие ко многим», посредством ассоциативной таблицы `CATEGORIES_ITEMS`. Соответственно парсер JPQL сгенерирует следующую инструкцию SQL:

```
SELECT
  c.CATEGORY_ID, c.CATEGORY_NAME, c.CREATE_DATE,
  c.CREATED_BY, c.PARENT_ID
FROM CATEGORIES c
WHERE (
  (SELECT COUNT(*)
  FROM CATEGORIES_ITEMS ci, ITEMS i
  WHERE (
    (ci.CATEGORY_ID = c.CATEGORY_ID) AND
    (i.ITEM_ID = ci.ITEM_ID))) = 0)
```

Как видите, здесь используется подзапрос, извлекающий число связанных объявлений для категории с помощью функции `COUNT`, и затем возвращаемое им число сравнивается с 0. Это означает, что если ни одного элемента не будет найдено, выражение `IS EMPTY` вернет `TRUE`.

Приходилось ли вам когда-нибудь проверять присутствие некоторого значения в коллекции? Уверен, что да! В JPQL для этой цели можно использовать оператор `MEMBER OF`. Давайте посмотрим, как он работает.

Проверка присутствия сущности в коллекции

Проверить присутствие переменной, результата выражения, возвращающего единственное значение, или входного параметра в коллекции можно с помощью оператора `MEMBER OF`, синтаксис которого показан ниже:

```
entity_expression [NOT] MEMBER [OF] collection_value_path_expression
```

Ключевые слова `OF` и `NOT` являются необязательными и их можно опустить. Ниже приводится пример использования входного параметра с оператором `MEMBER OF`:

```
WHERE :item MEMBER OF c.items
```

Это условное выражение вернет `TRUE`, если экземпляр сущности (`:item`), переданный в параметре, присутствует в коллекции `c.items` конкретной категории `c`.

Использование функций JPQL

В языке JPQL имеется несколько встроенных функций для выполнения разного рода операций. Эти функции можно использовать в предложениях `WHERE` или `HAVING`. Подробнее о предложении `HAVING` вы узнаете далее в этой главе, когда мы будем знакомиться с агрегатными функциями. А сейчас займемся исследованием трех разновидностей функций JPQL:

- строковые функции;
- арифметические функции;
- функции для работы с датами и временем.

Строковые функции

Строковые функции обычно используются в инструкции SELECT. В табл. 11.5 перечислены все строковые функции, поддерживаемые в языке JPQL. Эти функции применяются для фильтрации результатов запросов. Если вам потребуется выполнять какие-либо промежуточные операции с полученными данными, используйте для этого функции, имеющиеся в языке Java. Главная причина такой рекомендации в том, операции со строками в памяти приложения, выполняются намного быстрее, чем операции в базе данных.

Таблица 11.5. Строковые функции JPQL

Строковые функции	Описание
`CONCAT(string1, string2)`	Возвращает строку, составленную из двух исходных строк.
`SUBSTRING(string, position, length)`	Возвращает подстроку в строке, начиная с позиции `position` и длиной `length`.
`TRIM([LEADING | TRAILING | BOTH] [trim_character] FROM] string_to_trimmed)`	Отсекает указанный символ `trim_character` на концах исходной строки (`LEADING` – в начале, `TRAILING` – в конце или `BOTH` – с обоих концов). Если символ `trim_character` не указан, предполагается, что удалению подлежат пробельные символы.
`LOWER(string)`	Возвращает строку после преобразования символов в нижний регистр.
`UPPER(string)`	Возвращает строку после преобразования символов в верхний регистр.
`LENGTH(string)`	Возвращает длину строки.
`LOCATE(searchString, stringToBeSearched[initialPosition])`	Возвращает позицию строки `searchString` в строке `stringToBeSearched`, поиск начинается с позиции `initialPosition`.

Давайте рассмотрим поближе пару наиболее часто используемых строковых функций. Допустим, что необходимо сравнить результат конкатенации двух строковых выражений со строковым литералом. Поставленную задачу решает следующее предложение WHERE:

```
WHERE CONCAT(u.firstName, u.lastName) = 'ViperAdmin'
```

Если результат конкатенации `u.firstName` и `u.lastName` не совпадет со строкой «ViperAdmin», условное выражение вернет FALSE.

Далее приводится пример использования функции SUBSTRING для проверки совпадения первых трех символов в u.lastName со строкой «VIP»:

```
WHERE SUBSTRING(u.lastName, 1, 3) = 'VIP'
```

Имя каждой строковой функции достаточно отчетливо сообщает о ее назначении. Помимо строковых функций язык JPQL поддерживает также ряд арифметических функций, о которых рассказывается в следующем разделе.

Арифметические функции

Арифметические функции редко применяются в простых CRUD-операциях, зато они часто используются при формировании отчетов. В языке JPQL поддерживается только самый минимум таких функций, но некоторые производители могут добавлять собственные инструменты, специально для создания отчетов. Однако, применение любых инструментов, предусматриваемых отдельными производителями, ухудшает переносимость кода и осложняет переход на платформы других производителей. Арифметические функции, поддерживаемые в JPQL и перечисленные в табл. 11.6, можно применять в предложениях WHERE или HAVING.

Таблица 11.6. Арифметические функции JPQL

Арифметические функции	Описание
ABS(simple_arithmetic_expression)	Возвращает абсолютное значение simple_arithmetic_expression.
SQRT(simple_arithmetic_expression)	Возвращает корень квадратный от simple_arithmetic_expression в виде значения типа double.
MOD(num, div)	Возвращает остаток от целочисленного деления num на div.
SIZE(collection_value_path_expression)	Возвращает число элементов в коллекции.

Большинство арифметических функций имеют имена, достаточно отчетливо сообщающие об их назначении, как, например, SIZE:

```
WHERE SIZE(c.items) = 5
```

Это выражение вернет TRUE, если коллекция c.items содержит 5 элементов, и FALSE – в противном случае.

Функции для работы со временем

Большинство языков включает функции, возвращающие текущие дату, время или временную метку (timestamp). Язык JPQL также предлагает ряд функций для работы со временем, которые перечислены в табл. 11.7. Эти функции транслируются в соответствующие SQL-функции и запрашивают текущие дату, время и временную метку из базы данных.

Имейте в виду: так как функции JPQL извлекают значения времени из базы данных, они могут отличаться от времени, возвращаемого виртуальной машиной

JVM, если они база данных и JVM действуют на разных серверах. Эта проблема проявляется только в приложениях, чувствительных к подобным различиям. Ее можно разрешить, запустив службу времени на всех ваших серверах. Далее мы поближе познакомимся с инструкцией SELECT.

Таблица 11.7. Функции JPQL для работы со временем

Функции	Описание
CURRENT_DATE	Возвращает текущую дату.
CURRENT_TIME	Возвращает текущее время.
CURRENT_TIMESTAMP	Возвращает текущую временную метку.

11.1.3. Инструкция SELECT

В начале этой главы мы уже видели несколько примеров инструкции SELECT, однако до сих пор не углублялись в ее обсуждение. Из предыдущих примеров видно, что SELECT описывает желаемый результат запроса. Ниже приводится синтаксис инструкции SELECT в языке JPQL:

```
SELECT [DISTINCT] expression1, expression2, .... expressionN
```

Инструкция SELECT может включать идентификаторы переменных, выражения маршрутов, возвращающие единственное значение, или агрегатные функции, разделенные запятыми. В предыдущих примерах мы использовали переменные в инструкции SELECT, как показано ниже:

```
SELECT c
FROM Category AS c
```

Здесь также можно использовать выражения маршрутов:

```
SELECT c.categoryName, c.createdBy
FROM Category c
```

Выражения в инструкции SELECT должны возвращать единственное значение. То есть, выражения маршрутов, возвращающие коллекции, в этой инструкции недопустимы. Выражения маршрутов могут быть полями связи, как в предыдущем примере, где `c.createdBy` – это поле связи в сущности `Category`.

Предыдущий запрос может возвращать повторяющиеся сущности. Чтобы предотвратить появление повторяющихся данных в результате, используйте ключевое слово DISTINCT:

```
SELECT DISTINCT c.categoryName, c.createdBy
FROM Category c
```

Следующая инструкция SELECT недопустима:

```
SELECT c.categoryName, c.items
FROM Category
```

потому что выражение `c.items` возвращает коллекцию, а, как говорилось выше, выражения, возвращающие коллекции, нельзя использовать в инструкции `SELECT`.

В следующем разделе мы поговорим об использовании агрегатных функций.

Использование выражения-конструктора в инструкции SELECT

Чтобы получить один или более экземпляров Java-объектов в инструкции `SELECT` можно использовать конструктор. Эта возможность может пригодиться, когда в запросе требуется создать экземпляры, инициализированные данными, полученными из подзапроса:

```
SELECT NEW actionbazaar.persistence.ItemReport (c.categoryID, c.createdBy)
FROM Category
WHERE categoryId.createdBy = :userName
```

Указанный класс не обязательно должен быть сущностью или отображаться в базу данных.

Агрегатные функции

В языке JPQL поддерживаются все обычные агрегатные функции: `AVG`, `COUNT`, `MAX`, `MIN` и `SUM`, которые перечислены в табл. 11.8. Имя каждой функции достаточно отчетливо сообщает о ее назначении. Агрегатные функции часто применяются в запросах, используемых при формировании отчетов. В функциях `AVG`, `MAX`, `MIN` и `SUM` допускается использовать только хранимые поля, но в функции `COUNT` можно использовать любые выражения маршрутов или переменные.

Таблица 11.8. Агрегатные функции, поддерживаемые в языке JPQL

Агрегатные функции	Описание	Тип возвращаемого значения
`AVG`	Возвращает среднее для всех значений указанного поля.	`Double`
`COUNT`	Возвращает число результатов, возвращаемых запросом.	`Long`
`MAX`	Возвращает максимальное значение для указанного поля.	Зависит от типа хранимого поля.
`MIN`	Возвращает минимальное значение для указанного поля.	Зависит от типа хранимого поля.
`SUM`	Возвращает сумму всех значений указанного поля.	Может возвращать значение типа `Long` или `Double`.

Найти максимальное значение поля `i.itemPrice` среди всех объявлений можно с помощью следующего запроса:

```
SELECT MAX(i.itemPrice)
FROM Item i
```

Найти количество сущностей Category можно с помощью агрегатной функции COUNT:

```
SELECT COUNT(c)
FROM Category c
```

Только что вы увидели несколько простых примеров применения агрегатных функций. В следующем разделе мы покажем, как агрегировать результаты на основе выражений маршрутов.

Группировка с помощью GROUP BY и HAVING

В промышленных приложениях нередко требуется объединить данные по значению некоторого хранимого поля. Если принять, что сущности User и Category связаны отношением «один ко многим», тогда следующий запрос вернет список с числом сущностей Category, созданных каждым пользователем c.user:

```
SELECT c.user, COUNT(c.categoryId)
FROM Category c
GROUP BY c.user
```

Этот запрос осуществляет группировку по связанной сущности. Группировку допускается выполнять с использованием выражения, возвращающего единственное значение, которое может быть хранимым полем или полем связи. При выполнении агрегирования с использованием GROUP BY, разрешается использовать только агрегатные функции. Результаты агрегированного запроса можно фильтровать с помощью предложения HAVING. Допустим, что необходимо извлечь информацию только о тех пользователях, которые создали более пяти сущностей Category. Для этого достаточно лишь немного изменить предыдущий запрос:

```
SELECT c.user, COUNT(c.categoryId)
FROM Category c
GROUP BY c.user
HAVING COUNT(c.categoryId) > 5
```

Кроме того, вместе с предложением GROUP BY допускается использовать предложение WHERE:

```
SELECT c.user, COUNT(c.categoryId)
FROM Category c
WHERE c.createDate is BETWEEN :date1 and :date2
GROUP BY c.user
HAVING COUNT(c.categoryId) > 5
```

При наличии предложения WHERE в запросе, содержащем оба предложения, GROUP BY и HAVING, этот запрос обрабатывается в несколько этапов. Сначала выполняется фильтрация результатов с применением предложения WHERE. Затем к отфильтрованным результатам применяется агрегирование, описываемое предложением GROUP BY. И, наконец, применяется предложение HAVING, выполняющее фильтрацию агрегированных результатов.

11.1.4. Управление результатами

Подзапрос – это запрос внутри запроса. Подзапросы можно использовать в предложениях WHERE, HAVING и GROUP BY для фильтрации результатов. В отличие от SQL, JPQL в EJB 3 не поддерживает подзапросы в предложении FROM. Подзапросы всегда выполняются первыми и только потом выполняется главный запрос, использующий результаты, возвращаемые подзапросом.

Вот как выглядит синтаксис подзапроса:

```
[NOT] IN / [NOT] EXISTS / ALL / ANY / SOME (subquery)
```

Как видите, с подзапросами можно использовать операторы IN, EXISTS, ALL, ANY или SOME.

Рассмотрим поближе несколько примеров подзапросов.

Использование оператора IN с подзапросами

Мы уже обсуждали использование оператора IN, когда определяли присутствие результата выражения в списке значений. Этот список значений можно создавать с помощью подвыражения:

```
SELECT i
FROM Item i
WHERE i.user IN (SELECT c.user
                 FROM Category c
                 WHERE c.categoryName LIKE :name)
```

В этом запросе сначала выполняется подзапрос в скобках, возвращающий список пользователей, а затем выясняется, входит ли значение выражения i.item в список.

Оператор EXISTS

Оператор EXISTS (или NOT EXISTS) помогает узнать, возвращает ли подзапрос какие-нибудь результаты. Если подзапрос возвращает хотя бы один результат, оператор EXISTS вернет TRUE, в противном случае будет возвращено значение FALSE. Ниже приводится пример, иллюстрирующий применение оператора EXISTS:

```
SELECT i
FROM Item i
WHERE EXISTS (SELECT c
              FROM Category c
              WHERE c.user = i.user)
```

Если внимательно изучить результаты этого запроса, можно обнаружить, что они идентичны результатам запроса из примера в предыдущем разделе, где обсуждался оператор IN. Вообще оператор EXISTS выглядит предпочтительнее, чем IN, особенно когда таблицы содержат большое число записей. Это объясняется тем, что обычно оператор EXISTS имеет в базах данных более оптимальную реализацию. И снова причина кроется в особенностях работы процессора запросов, выполняющего перевод запросов на языке JPQL в запросы на языке SQL.

Операторы ANY, ALL и SOME

Операторы ANY, ALL и SOME своим действием напоминают оператор IN. Их можно использовать в комбинации с любыми операторами сравнения чисел, такими как: =, >, >=, <, <= и <>.

Ниже приводится пример, демонстрирующий применение оператора ALL к подзапросу:

```
SELECT c
FROM Category c
WHERE c.createDate >= ALL
                (SELECT i.createDate
                 FROM Item i
                 WHERE i.user = c.user)
```

Оператор ALL возвращает TRUE, если все результаты, возвращаемые подзапросом, соответствуют условию; в противном случае он возвращает FALSE. В данном примере оператор ALL вернет FALSE, если хотя бы одно объявление (Item), возвращаемое подзапросом, будет иметь дату создания (createDate) более позднюю, чем дата создания (createDate) категории, извлекаемой основным запросом.

Как следует из названий операторов ANY и SOME, они возвращают TRUE, если хотя бы один из результатов подзапроса соответствует условию. Ниже показан пример использования оператора ANY:

```
SELECT c
FROM Category c
WHERE c.createDate >= ANY
                (SELECT i.createDate
                 FROM Item i
                 WHERE i.seller = c.user)
```

Оператор SOME фактически является псевдонимом (синонимом) оператора ANY; соответственно его можно использовать вместо оператора ANY.

11.1.5. Соединение сущностей

Если у вас есть опыт использования реляционных баз данных и SQL, вы должны быть знакомы с оператором соединения JOIN. Его можно использовать, чтобы получить декартово произведение двух сущностей. Обычно оператор JOIN сопровождается предложением WHERE, определяющим условие соединения сущностей, если простого декартова произведения оказывается недостаточно.

Чтобы создать соединение двух или более сущностей, их необходимо указать в предложении FROM. Соединение сущностей выполняется либо на основе отношений между ними, либо на основе произвольных хранимых полей. При выполнении соединения есть возможность указать, как будет определяться соответствие сущностей условию в операторе JOIN. Например, можно выполнить соединение Category и Item на основе отношений между ними и извлекать только сущности, соответствующие условию соединения. Такие соединения называют внутренними соединениями. Однако, можно извлекать результаты, не только удовлетворяющие

условиям соединения, но и включать сущности с одной из сторон предметной области, не имеющие соответствующих сущностей с другой стороны предметной области. Например, можно извлечь все экземпляры `Category`, даже если для них отсутствуют соответствующие экземпляры `Item`. Такие соединения называются внешними. Обратите внимание, что внешние соединения могут быть левыми, правыми или полными.

Давайте сначала рассмотрим несколько примеров наиболее распространенных типов соединений: внутренних и внешних. После этого мы познакомимся с более редкими (но иногда весьма полезными) выборкой (fetch) и тета-соединениями (theta joins).

Внутренние соединения

Часто в приложениях возникает необходимость выполнить соединение двух или более сущностей, опираясь на некоторое общее отношение между ними. Такое соединение называется внутренним соединением и выполняется с помощью оператора `INNER JOIN`:

```
[INNER] JOIN join_association_path_expression [AS]
  identification_variable
```

Сущности `Category` и `User` в приложении ActionBazaar связаны отношением «многие к одному». Чтобы получить список всех пользователей, соответствующих определенному критерию, можно воспользоваться следующим запросом:

```
SELECT u
FROM User u INNER JOIN u.Category c
WHERE u.userId LIKE ?1
```

Предложение `INNER` является необязательным. Запомните, что когда в запросе присутствует оператор `JOIN`, по умолчанию подразумевается `INNER JOIN`, если явно не указано предложение `OUTER`. А теперь посмотрим на другой конец спектра – внешние соединения.

Внешние соединения

Внешние соединения позволяют извлекать дополнительные сущности, не соответствующие условиям, указанным в операторе `JOIN`, если связи между сущностями являются необязательными. Внешние соединения особенно полезны при формировании отчетов. Допустим, что сущности `User` и `Category` связаны необязательным отношением и необходимо сгенерировать отчет для вывода всех названий категорий, связанных с пользователями. Если пользователь не связан ни с одной категорией, в отчете должно выводиться «NULL». Если указать пользователя слева от оператора `JOIN`, чтобы получить соединение можно использовать ключевую фразу `LEFT JOIN` или `LEFT OUTER JOIN`:

```
SELECT u
FROM User u LEFT OUTER JOIN u.Category c
WHERE u.userId like ?1
```

Этот запрос извлечет также все сущности `User`, не имеющие соответствующих сущностей `Category`. Следует заметить, что если бы использовалось не внешнее соединение, запрос вернул бы только пользователей, имеющих соответствующие категории.

А поддерживаются ли в языке JPQL другие типы соединений, кроме распространенных `INNER JOIN` и `OUTER JOIN`? Хороший вопрос! Да, поддерживаются – выборка и тета-соединение – с которыми мы познакомимся далее.

Оператор FETCH JOIN

В типичном приложении для коммерции может потребоваться запросить определенную сущность и одновременно извлечь связанные сущности. Например, одновременно с сущностью `Bid` в приложении ActionBazaar может понадобиться извлечь и инициализировать связанный экземпляр покупателя. Для этой цели в JPQL имеется оператор `FETCH JOIN`:

```
SELECT b
FROM Bid b FETCH JOIN b.bidder
WHERE b.bidDate >= :bidDate
```

Такой способ выборки часто бывает удобно использовать, когда включен режим отложенной загрузки отношений, но в данном конкретном запросе требуется обеспечить немедленную загрузку. Оператор `FETCH JOIN` можно использовать и во внутренних, и во внешних соединениях.

Тета-соединения

Тета-соединения редко используются на практике, и основаны на соединении сущностей по произвольным хранимым полям или полям связи, а не на отношениях между ними. Например, сущности `Category` в системе ActionBazaar имеют хранимое поле `rating`, определяющее рейтинг категорий. Это поле может принимать значения `DELUXE`, `GOLD`, `STANDARD` и `PREMIUM`. Кроме того, сущность `Item` имеет хранимое поле `star`, определяющее рейтинг объявления; это поле также может принимать значения `DELUXE`, `GOLD`, `STANDARD` и `PREMIUM`. Допустим, что имеются сущности обоих типов, хранящие в этих полях общие значения, такое как `GOLD`, и требуется реализовать их соединение по полям `rating` и `star`. Эту задачу решает следующий запрос:

```
SELECT i
FROM Item i, Category c
WHERE i.star = c.rating
```

Хотя эта разновидность соединений редко используется на практике, ее нельзя полностью исключить.

Могли ли вы представить, что JPQL обладает такими богатыми возможностями? Кто не знал ничего лучше, мог бы подумать, что это совершенно другой язык.... Впрочем, так и есть! И он только и ждет, когда вы попробуете пользоваться им. Мы надеемся, что смогли заложить основы, опираясь на которые вы сможете начать использовать JPQL в своих приложениях.

Итак, мы вышли на финишную прямую в этой главе, и нам осталось рассмотреть всего пару тем. Нам нужно обсудить возможность создания запросов на языке SQL, но перед этим нужно поговорить об операциях массового удаления и изменения.

11.1.6. Операции массового удаления и изменения

В приложении ActionBazaar поддерживается возможность классификации пользователей с использованием таких терминов, как «золотой», «платиновый» и других, исходя из числа успешных торгов в году. В конце года вызывается модуль приложения, который присваивает пользователям соответствующие статусы. Можно было бы с помощью запроса извлечь коллекцию сущностей `User` и затем в цикле изменить их статусы. Однако гораздо проще воспользоваться инструкцией `UPDATE` и с ее помощью изменить коллекцию сущностей, в соответствии с условиями, как показано в следующем примере:

```
UPDATE User u
SET u.status = 'G'
WHERE u.numTrades >=?1
```

В предыдущих примерах уже было показано несколько примеров использования инструкций `DELETE` и `UPDATE`, но до настоящего момента мы не углублялись в их обсуждение. Давайте предположим, что администраторам системы ActionBazaar потребовалась возможность удалять сущности, такие как `User`, исходя из определенных условий. Начнем обсуждение со следующего кода:

```
@PersistenceContext em;
...
// начало транзакции
Query query = em.createQuery("DELETE USER u WHERE u.status = :status ");
query.setParameter("status", 'GOLD');
int results = query.executeUpdate();
// конец транзакции
```

Порядок использования инструкций `UPDATE` и `DELETE`, как показано в этом фрагменте, близко напоминает использование любых других инструкций JPQL, кроме двух важных отличий. Во-первых, для выполнения массового изменения или удаления вместо метода `getResultList` или `getSingleResult` используется метод `executeUpdate` интерфейса `Query`. Во-вторых, метод `executeUpdate` должен вызываться внутри активной транзакции.

Мы рекомендуем изолировать массовые операции и выполнять их в отдельных транзакциях, так как они таят в себе множество скрытых ловушек, а также потому, что они непосредственно транслируются в операции базы данных и могут вызвать несоответствия между данными в базе и в управляемых сущностях. От производителей требуется только обеспечить поддержку операций массового удаления и изменения – согласно спецификации, они не обязаны согласовывать изменения

в базе данных с активными сущностями. Проще говоря, механизм хранения не будет удалять связанные сущности при удалении основной сущности в ходе операции массового удаления.

На данный момент мы охватили почти все основы: запросы, аннотации и JPQL. Но у нас осталась нераскрытой еще одна тема: использование в EJB 3 обычных запросов SQL.

11.2. Запросы Criteria

Запросы JPQL являются чрезвычайно мощным инструментом, но этот инструмент все еще опирается на простые текстовые строки, встроенные в программный код на Java, которые интерпретируются только во время выполнения программы. Это означает, что успешные компиляция и развертывание приложения не гарантируют отсутствие в запросах синтаксических ошибок. Единственный способ убедиться в корректности запросов – выполнить каждый из них в модульных или интеграционных тестах. Если игнорировать регрессионное тестирование, синтаксические ошибки в запросах будут дорого обходиться на этапе разработки, особенно если приложение большое и требуется значительное время на его компиляцию, упаковку и развертывание, для опробования запросов вручную. В версии Java EE 6 появилась поддержка Criteria API – механизма создания типизированных запросов.

Под словами «типизированный API» подразумевается подход к созданию запросов SQL с использованием настоящих объектов Java. Этим он полностью отличается от традиционного способа, основанного на конструировании SQL-инструкций в виде строк и чреватого синтаксическими ошибками. С помощью типизированного интерфейса можно строить запросы, используя исключительно программный код, без применения текстовых строк. При этом гарантируется синтаксическая корректность запросов, хотя они все еще могут содержать логические ошибки. Под логическими ошибками здесь понимаются ошибки, допускаемые человеком, такие как отсутствие предиката в инструкции `DELETE`, из-за того, что программист отвлекся на телефонный звонок. Недостаток такого подхода – более громоздкий код реализации сложных запросов: с использованием Criteria API однострочные инструкции на языке JPQL или SQL легко могут превратиться в многострочный программный код. Решить эту проблему можно созданием предметно-ориентированного языка (Domain-Specific Language, DSL), однако обсуждение этой темы далеко выходит за рамки данной книги и главы.

На данный момент наибольший интерес представляет сама механика статической типизации свойств. Очевидно, что для этого нужно определить модель данных и в этом нам поможет *метамоделирование*. Метамодель – это статическое представление модели данных. Она может быть автоматически сгенерирована компилятором на основе аннотаций JPA. Метамодель используется в сочетании с классом `CriteriaBuilder` для конструирования и выполнения запросов. Чтобы обсуждение получилось более предметным, рассмотрим простой запрос для поиска объявления по названию товара, представленный в листинге 11.1.

Листинг 11.1. Поиск объявления по названию товара с использованием Criteria API

```
public List<Item> findItemByName(String name) {
    // Получить экземпляр CriteriaBuilder от EntityManager
    CriteriaBuilder builder = entityManager.getCriteriaBuilder();

    // Создать новый экземпляр запроса CriteriaQuery для поиска объявления
    CriteriaQuery<Item> query = builder.createQuery(Item.class);

    // Сконструировать запрос на основе метамодели
    Root<Item> root = query.from(Item.class);
    Predicate condition = builder.like(root.get(Item_.itemName),name);
    query.where(condition);
    TypedQuery<Item> q = entityManager.createQuery(query);
    return q.getResultList();
}
```

Этот метод класса `ItemManager` создает Criteria-запрос на основе метамодели и затем выполняет его. Несмотря на громоздкость, этот запрос будет выполняться довольно быстро, потому что в нем отсутствуют строки, требующие динамической интерпретации. Здесь вы не сможете по ошибке использовать `name` вместо `itemName`. Теперь, когда вы получили общее представление о Criteria-запросах, перейдем к обсуждению метамоделей.

11.2.1. Метамодели

Метамодель является представлением сущностей в базе данных. Она имеет некоторое сходство с Reflection API в языке Java и метаданными JDBC. Как и при использовании Reflection API, с помощью метамодели можно выполнять обход атрибутов сущностей и получать базовую информацию о них, включая имена и типы. Поддерживается также возможность определять наличие отношений между сущностями, таких как «один ко многим» или «многие ко многим», и так далее. По аналогии с DatabaseMetaData API в JDBC, с помощью которого можно получить низкоуровневую информацию о строении базы данных, метамодель позволяет получать информацию о классах, управляемых механизмом JPA. Метамодели поддерживают также возможность интроспекции объектно-реляционных отображений.

Однако метамодель – это лишь часть решения. Чтобы обеспечить статическую типизацию, необходимы еще статические объекты, представляющие данные. Эти объекты создаются с помощью *процессора аннотаций*. Процессор аннотаций – это модуль расширения компилятора `javac`, который обрабатывает аннотации на этапе компиляции. В случае с JPA, он обрабатывает аннотации, а также конфигурационный файл `persistence.xml`. Для всех классов, управляемых механизмом JPA, генерируются файлы с исходным кодом на языке Java, которые затем компилируются вместе с проектом и образуют статическую метамодель, используемую во время выполнения. Интегрированные среды разработки выполняют интроспекцию классов метамодели для поддержки функции автодополнения кода.

Чтобы задействовать метамодель, нужно выполнить следующие действия:

1. Снабдить POJO аннотациями JPA.
2. Скомпилировать код, задействовав в процедуре компиляции процессор метамоделей.
3. Реализовать Criteria-запросы с использованием сгенерированных классов.
4. При изменении модели повторить предыдущие этапы.

Процессор аннотаций

Вам потребуется настроить процедуру сборки и, возможно, среду разработки на использование процессора метамодели. О том, как это сделать, можно узнать в документации к вашей реализации JPA. Если вы не планируете использовать Criteria-запросы, тогда вам не потребуется ничего настраивать. После выполнения необходимых настроек модель будет генерироваться и развертываться автоматически.

Интроспекция

Основу любой метамодели составляет интерфейс `Metamodel`. Получить ссылку на реализацию `Metamodel` можно вызовом метода `getMetamodel()` объекта `EntityManager`. С помощью экземпляра `Metamodel` можно получить какую-то определенную сущность или список всех управляемых сущностей. Информация о любой сущности возвращается в виде экземпляра `EntityType`. Объект `EntityType` имеет методы для получения дополнительной информации о сущности, включая информацию об атрибутах. Если прежде вам доводилось использовать механизм рефлексии в Java, то вы не будете испытывать никаких затруднений.

В листинге 11.2 приводится определение интерфейса `Metamodel`. Для устранения необходимости приведения типов в нем используется поддержка обобщенных типов – Java Generics.

Листинг 11.2. Интерфейс Metamodel

```
public interface Metamodel {
    // Извлекает информацию об управляемой сущности в виде EntityType
    public <X extends Object> EntityType<X> entity(Class<X> type);
    public <X extends Object> ManagedType<X> managedType(Class<X> type);
    public <X extends Object> EmbeddableType<X> embeddable(Class<X> type);
    public Set<ManagedType<?>> getManagedTypes();

    // Извлекает список экземпляров EntityType для всех управляемых сущностей
    public Set<EntityType<?>> getEntities();
    public Set<EmbeddableType<?>> getEmbeddables();
}
```

Используя интерфейс `Metamodel`, можно выполнить итерации по всем управляемым классам, как показано в листинге 11.3.

Листинг 11.3. Интроспекция с помощью интерфейса Metamodel во время выполнения

```
// Получить метамодель с помощью EntityManager
Metamodel metaModel = entityManager.getMetamodel();
// Получить множество всех управляемых классов в текущей единице хранения
Set<EntityType<? extends Object>> types = metaModel.getEntities();
for(EntityType<? extends Object> type : types) {
    // Вывести имя сущности
    logger.log(Level.INFO, "--> Type: {0}", type);
    // Получить список типов атрибутов сущности
    Set attributes = type.getAttributes();
    for(Object obj : attributes) {
        // Вывести имя атрибута
        logger.log(Level.INFO, "Name: {0}", ((Attribute)obj).getName());
        // Вывести TRUE, если атрибут является коллекцией
        logger.log(Level.INFO, "isCollection: {0}",
            ((Attribute)obj).isCollection());
    }
}
```

Метамодели имеют большую практическую ценность. С их помощью можно получать списки всех управляемых сущностей и исследовать их атрибуты и отношения. Но, как говорилось выше, метамодель – это лишь часть проблемы. Другой ее частью является сгенерированный код, который обеспечит поддержку статической типизации в запросах.

Сгенерированный код

Процессор аннотаций сгенерирует класс метамодели для каждой управляемой сущности. Сгенерированные классы получат те же имена, с дополнительным символом подчеркивания _ в конце. Сгенерированные классы будут отмечены аннотацией `@Static-Metamodel`. В табл. 11.9 перечислены атрибуты экземпляра класса метамодели, генерируемые в зависимости от типов соответствующих атрибутов сущности. Интерфейс `Metamodel` предоставляет также механизм поиска классов метамодели во время выполнения.

Таблица 11.9. Определения атрибутов экземпляра класса метамодели

Объявление атрибута	Тип атрибута
`public static volatile SingularAttribute<X, Y> y;`	Для атрибута, не являющегося коллекцией.
`public static volatile CollectionAttribute<X, Z> z;`	`java.util.Collection`
`public static volatile SetAttribute<X, Z> z;`	`java.util.Set`
`public static volatile ListAttribute<X, Z> z;`	`java.utilList`
`public static volatile MapAttribute<X, K, Z> z;`	`java.util.Map`

В листинге 11.4 представлен класс метамодели, сгенерированный процессором аннотаций для класса сущности `Item` в приложении ActionBazaar.

Листинг 11.4. Класс метамодели, сгенерированный процессором аннотаций

```
package com.actionbazaar.model;
// Тег, вставляемый автоматически;
// данный пример сгенерирован с помощью EclipseLink
@Generated(value="EclipseLink-2.5.0.v20130321-rNA",
    date="2013-04-12T18:02:56")
// Отмечает класс, являющийся частью статической метамодели
// и отображаемый в управляемую сущность
@StaticMetamodel(Item.class)
// Имя управляемой сущности с символом подчеркивания в конце
public class Item_ {
    // Простой атрибут, представляющий одиночное значение
    public static volatile SingularAttribute<Item, byte[]> picture;
    public static volatile SingularAttribute<Item, String> itemName;
    // Отношение "многие ко многим"
    public static volatile SetAttribute<Item, Category> category;
    public static volatile SingularAttribute<Item, BigDecimal> initialPrice;
    public static volatile SingularAttribute<Item, Date> bidEndDate;
    public static volatile SingularAttribute<Item, String> description;
    // Отношение "один ко многим"
    public static volatile ListAttribute<Item, Bid> bids;
    public static volatile SingularAttribute<Item, Long> itemId;
    public static volatile SingularAttribute<Item, Date> createdDate;
    // Отношение "многие к одному"
    public static volatile SingularAttribute<Item, BazaarAccount> seller;
    public static volatile SingularAttribute<Item, Date> bidStartDate;
}
```

Класс в листинге 11.4 имеет статические свойства, которые обеспечивают статическую типизацию при построении запроса. В Criteria API широко используются обобщенные типы, которые в сочетании со статическими классами гарантируют синтаксическую корректность запросов. Теперь пришло время погрузиться в изучение `CriteriaBuilder`.

11.2.2. CriteriaBuilder

Класс `CriteriaBuilder` является одной из основ Criteria API. Он отвечает за конструирование запросов: стыковку переменных, выражений, предикатов и предложений сортировки. Это фабричный класс, используемый для создания запросов на основе статической метамодели. Получить ссылку на экземпляр `CriteriaBuilder` можно вызовом метода `getCriteriaBuilder()` объекта `EntityManager`, как было показано в листинге 11.1. В табл. 11.10 перечислены пять разных типов запросов, которые можно создавать с помощью `CriteriaBuilder`.

Тип запроса определяет тип ожидаемого возвращаемого значения. Этот тип необязательно должен быть типом управляемой сущности. Например, при конструировании запроса, вычисляющего среднее значение от всех ставок для определенного объявления, можно передать `Long.class`. Точно так же, как было показано в примере создания инструкции `SELECT`, можно указать объект-обертку, не являющийся управляемой сущностью, а используемый только для промежуточного хранения результатов.

Таблица 11.10. Методы создания запросов в CriteriaBuilder

Метод создания запросов	Описание
`createQuery()`	Создает новый объект Criteria-запроса.
`createQuery(java.lang.Class<T> resultClass)`	Создает новый объект Criteria-запроса с определенным типом возвращаемого значения.
`createTupleQuery()`	Создает новый объект Criteria-запроса, возвращающего кортеж результатов.
`createCriteriaDelete(Class<T> targetEntity)`	Создает новый объект Criteria-запроса, выполняющего массовое удаление.
`createCriteriaUpdate(Class<T> targetEntity)`	Создает новый объект Criteria-запроса, выполняющего массовое изменение.

Экземпляр `CriteriaBuilder` предоставляет также фабричные методы для создания выражений, предложений сортировки, инструкций выборки и предикатов. Эти методы перечислены в табл. 11.11. Мы подробнее рассмотрим их в следующих разделах.

Таблица 11.11. Методы CriteriaBuilder, сгруппированные по типам

Тип	Методы
Выражение	`abs`, `all`, `any`, `avg`, `coalesce`, `concat`, `construct`, `count`, `countDistinct`, `currentDate`, `currentTime`, `currentTimestamp`, `diff`, `function`, `greatest`, `keys`, `least`, `length`, `literal`, `locate`, `lower`, `max`, `min`, `mod`, `neg`, `nullif`, `nullLiteral`, `parameter`, `prod`, `quot`, `selectCase`, `size`, `some`, `sqrt`, `substring`, `sum`, `sumAsDouble`, `sumAsLong`, `toBigDecimal`, `toBigInteger`, `toDouble`, `toFloat`, `toInteger`, `toLong`, `toString`, `trim`, `upper`, `values`
Выборка	`array`, `construct`, `tuple`
Сортировка	`asc`, `desc`
Предикат	`and`, `between`, `conjunction`, `disjunction`, `equal`, `exists`, `ge`, `greaterThan`, `greaterThanOrEqualTo`, `gt`, `in`, `isEmpty`, `isFalse`, `isMember`, `isNotEmpty`, `isNotMember`, `isNotNull`, `isNull`, `isTrue`, `le`, `lessThan`, `lessThanOrEqualTo`, `like`, `lt`, `not`, `notEqual`, `notLike`, `or`

11.2.3. CriteriaQuery

Объект `CriteriaQuery` является еще одной из основ Criteria API. Он стыкует между собой отдельные элементы – `SELECT`, `FROM` и необязательное предложение `WHERE` – конструируя объектное представление SQL-запроса. Механизм JPA использует это объектное представление для создания инструкций SQL. Затем JPA выполняет полученную инструкцию, передавая ее базе данных, и упаковывает результат в форму объекта, указанного при создании запроса. Сама схема мало чем отличается от JPQL, кроме того, что в данном случае объектное представление конструируется вручную.

Создание объектов CriteriaQuery выполняется с помощью интерфейса CriteriaBuilder, который мы рассмотрели в предыдущем разделе. Интерфейс CriteriaBuilder используется также для конструирования отдельных элементов запросов, которые затем передаются методам экземпляра CriteriaQuery. Вы можете использовать CriteriaBuilder как фабрику для создания любых необходимых элементов запросов. Основные методы интерфейса CriteriaQuery и типы их параметров перечислены в табл. 11.12.

Таблица 11.12. Основные методы интерфейса CriteriaQuery

Метод	Тип параметра	Описание
groupBy	Выражение	Создает SQL-предложение GROUP BY.
having	Выражение	Создает SQL-предложение HAVING.
multiselect	Выбираемые значения	Создает запрос с множеством выбираемых элементов.
orderBy	Сортировка	Создает SQL-предложение ORDER BY.
select	Выбираемые значения	Создает запрос с единственным выбираемым элементом.
where	Предикат	Создает SQL-предложение WHERE.

11.2.4. Корень запроса

Корень запроса (query root) – это одна из самых сложных для описания абстракций. Корень запроса определяет начальную точку для навигации – сущности, содержащие свойства, которые будут использоваться при конструировании SQL-предложений WHERE, SELECT и JOIN. Запрос может иметь ноль или более корней. Если вы не собираетесь выполнять соединение нескольких таблиц, определять конкретные извлекаемые свойства или применять ограничения к извлекаемым значениям, корень запроса можно не создавать. Говоря иными словами, корень запроса – это объект, используемый при конструировании выражений.

Создать корень запроса можно вызовом метода from экземпляра CriteriaQuery. Несмотря свое имя, метод from не используется для формирования SQL-предложения FROM; сущности автоматически добавляются в предложение FROM, если они используются в выражениях. Возвращаемый объект имеет тип javax.persistence.criteria.Root. Далее этот объект можно использовать при конструировании выражений.

Чтобы лучше понять, как применять корень запроса, рассмотрим простой пример в листинге 11.5.

Листинг 11.5. Извлечение названий всех товаров с помощью корня запроса

```
public List<String> getAllItemsNames() {
    CriteriaBuilder builder = entityManager.getCriteriaBuilder();
    // Создать CriteriaQuery; результатом запроса будет строка
    CriteriaQuery<String> query = builder.createQuery(String.class);
```

```
    // Создать корень запроса типа Item;
    // он будет использоваться для навигации
    Root<Item> root = query.from(Item.class);
    // Использовать объект Root, выбрать свойство itemName
    query.select(root.get(Item_.itemName));
    TypedQuery<String> tq = entityManager.createQuery(query);
    return tq.getResultList();
}
```

Код в этом листинге произведет следующий запрос SQL:

```
SELECT ITEM_NAME FROM ITEMS
```

Теперь, когда вы знаете, как создать корень запроса, обратимся к выражениям предикатов и соединений.

Выражения

Выражения используются в предложениях `SELECT`, `WHERE` и `HAVING`. С помощью выражений определяется, что должен возвращать запрос и какие ограничения должны быть учтены при выполнении запроса. Все выражения реализуют интерфейс `javax.persistence.criteria.Expression<T>`. Существует несколько особенно примечательных дочерних интерфейсов, включая `Predicate`, `Join` и `Path`. Получить экземпляр выражения можно вызовом одного из вспомогательных методов экземпляра `CriteriaBuilder`. Обратите внимание, что во многих случаях для создания выражений вам придется использовать другие выражения. Например, взгляните на сигнатуру метода `lessThanOrEqualTo` экземпляра `CriteriaBuilder`:

```
Predicate lessThanOrEqualTo(Expression<? extends Y> x,
                            Expression<? extends Y> y)
```

Как видите, этот метод принимает два выражения и возвращает предикат. Оба выражения используют обобщенные типы, которые в конкретной реализации должны совпадать – если сравнивается значение типа `Double`, второе значение так же должно иметь тип `Double`, и так далее. Чтобы сослаться на столбец (свойство сущности), нужно создать выражение типа `Path`. Как упоминалось прежде, `Path` – это тип выражений, для конструирования которых используется корень запроса, путем передачи ему атрибута метамодели. Так как подобное пояснение может показаться непонятным, давайте рассмотрим метод `findByDate`, представленный в листинге 11.6.

Листинг 11.6. Извлечение объявлений, размещенных в определенный промежуток времени

```
public List<Item> findByDate(Date startDate , Date endDate) {
    CriteriaBuilder builder = entityManager.getCriteriaBuilder();
    CriteriaQuery<Item> query = builder.createQuery(Item.class);
    // Создать корень запроса, ссылающийся на объекты Item
    Root<Item> itemRoot = query.from(Item.class);
```

```
    // Сконструировать выражение Path,
    // использующее метамодель для ссылки на созданные данные
    Path<Date> datePath = itemRoot.get(Item_.createdDate);
    // Сконструировать предикат, отбирающий объявления
    // с датой создания в указанном диапазоне
    Predicate dateRangePred = builder.between(datePath,
    startDate, endDate);
    query.where(dateRangePred);
    TypedQuery<Item> q = entityManager.createQuery(query);
    return q.getResultList();
}
```

Выражение, сконструированное в листинге 11.6, позволяет извлекать сущности `Item`, свойство `createdDate` которых имеет значение, попадающее в указанный диапазон. Для этого используются корень запроса и метамодель. Выражение `Item_.createDate` гарантирует обращение к существующему атрибуту, а посредством обобщенных типов гарантируется соответствие типов аргументов в вызове метода `between`. В результате выполнения кода в листинге, базе данных будет направлен следующий запрос:

```
SELECT ITEM_ID, BID_END_DATE, BID_START_DATE, CREATEDDATE, DESCRIPTION,
       INITIAL_PRICE, ITEM_NAME, PICTURE, STARRATING, SELLER_ID
FROM ITEMS
WHERE (CREATEDDATE BETWEEN ? AND ?)
```

Это очень простой пример, но он наглядно демонстрирует, как можно использовать Criteria API для создания типизированных запросов в программах, корректность которых проверяется компилятором Java.

А теперь перейдем к соединениям.

Соединения

Интерфейс Criteria API поддерживает возможность соединения классов, связанных отношениями. При этом по умолчанию используется внутреннее соединение. Соединение выполняется либо по объекту `Root`, либо по объекту `Join`. Оба объекта обладают методом `join`, способным принимать атрибут из метамодели, имеющий единственное значение или хранящий коллекцию: `SingularAttribute`, `CollectionAttribute`, `SetAttribute`, `ListAttribute` или `MapAttribute`. Сначала мы рассмотрим, как выполняется соединение по объекту `Root`, а затем перейдем к объекту `Join`.

Для демонстрации возьмем в качестве примера страницу из приложения ActionBazaar, где выводится сводка по выигравшей ставке. Информация для этой страницы извлекается из сущностей `Order`, `Item`, `Bid` и `BazaarAccount`. Ее можно было бы получить путем обхода объектов, извлекая сначала `Order`, а затем последовательно `Bid`, `Item` и `BazaarAccount`, но такой подход слишком неэффективен, так как придется выполнить множество обращений к базе данных. Гораздо эффективнее воспользоваться поддержкой соединений в Criteria API. Пример в листинге 11.7 выполняет четыре соединения.

Листинг 11.7. Получение сводной информации по выигравшей ставке с помощью соединений

```
public List<WinningBidWrapper> getWinningBid(Long itemId) {
    CriteriaBuilder builder = entityManager.getCriteriaBuilder();
    CriteriaQuery<WinningBidWrapper> query =
        builder.createQuery(WinningBidWrapper.class);
    Root<Item> itemRoot = query.from(Item.class);
    Root<Order> orderRoot = query.from(Order.class);
    Root<Bid> bidRoot = query.from(Bid.class);
    Root<BazaarAccount> userRoot = query.from(BazaarAccount.class);
    // Выполнить внутреннее соединение по сущностям Order и Bid
    Join<Order,Bid> j1 = orderRoot.join(Order_.bid);
    // Выполнить внутреннее соединение по сущностям Order и Item
    Join<Order,Item> j2 = orderRoot.join(Order_.item);
    // Выполнить внутреннее соединение по сущностям Order и BazaarAccount
    Join<Order,BazaarAccount> j3 = orderRoot.join(Order_.bidder);
    Path<Long> itemIdPath = itemRoot.get(Item_.itemId);
    // Создать объект Path для использования в предикате и
    // конструирования предложения WHERE, чтобы обеспечить
    // извлечение только интересующего объявления
    Predicate itemPredicate = builder.equal(itemIdPath,itemId);
    query.where(itemPredicate);
    // Определить список полей для создания объекта-обертки,
    // включив в него только необходимые поля
    query.select(
        builder.construct(
            WinningBidWrapper.class,
            userRoot.get( BazaarAccount_.username ),
            bidRoot.get( Bid_.bidPrice ),
            itemRoot.get(Item_.itemName) ,
            itemRoot.get(Item_.description)
    ));
    TypedQuery<WinningBidWrapper> q = entityManager.createQuery(query);
    return q.getResultList();
}
```

Метод `getWinningBid` создает запрос, выполняющий соединение выбранных сущностей и возвращающий объект-обертку. Объект-обертка – это синтетический объект, используемый как хранилище извлекаемых полей. По сути, он является простым объектом передачи данных (DTO). Подробнее об объектах-обертках будет рассказываться в разделе 11.2.6, где обсуждается инструкция `SELECT`.

В этом примере используется внутреннее соединение. Чтобы выполнить внешнее соединение, методу `join` нужно явно передать аргумент типа `JoinType`. Тип `JoinType` является перечислением, включающим три разновидности соединений: `INNER` (внутреннее, по умолчанию), `RIGHT` (внешнее) и `LEFT` (внешнее).

Объект соединения, возвращаемый методом `join`, можно использовать для определения `on`-условия. Метод `on` может принимать объект выражения или предиката, сконструированный с помощью `CriteriaBuilder`. Ниже приводится пример определения `on`-условия:

```
j2.on(builder.like(itemRoot.get(Item_.itemName),"boat"));
```

В Criteria API также поддерживается выборка по соединению (fetch join), благодаря чему можно организовать одновременное извлечение связанных сущностей в одном запросе. Например, заказ содержит ссылку на ставку; чтобы извлечь обе сущности одновременно, можно сконструировать следующий запрос:

```
orderRoot.fetch(Order_.bid,JoinType.INNER);
```

Теперь, после знакомства с понятием корня запроса, можно переходить к обсуждению предложения `FROM`.

11.2.5. Предложение FROM

Предложение `FROM` создается динамически прикладным интерфейсом Criteria API, на основе созданных перед этим корней запроса и соединений. То есть, вам не придется явно определять предложение `FROM` – все необходимое будет сделано автоматически. Механизм JPA найдет корни запроса и соединения, и на их основе сконструирует предложение `FROM`.

11.2.6. Предложение SELECT

Предложение `SELECT` управляет результатами, возвращаемыми запросом. Оно предоставляет несколько различных способов извлечения данных, чем обеспечивает существенную гибкость. Настройка этого предложения осуществляется вызовом метода `select` объекта `CriteriaQuery`. Обманчиво простой метод `select` принимает единственный параметр типа `Selection`. Как будет показано далее, используя довольно ограниченный круг подклассов, наследующих `Selection`, можно конструировать весьма сложные запросы. Предложению `SELECT` можно посвятить целую главу. Соответственно в этом небольшом разделе мы сможем рассмотреть лишь самые основы.

Метод `select` позволяет извлекать данные в разных представлениях, в зависимости от потребностей. При этом вы не ограничены извлечением только лишь сущностей JPA – вы можете извлекать сущности, одиночные значения, множества значений и, как было показано выше, синтезировать новые объекты из результатов запросов. Вы можете также работать с кортежами – упорядоченными списками значений – если не желаете тратить время на обдумывание и определение объектов-оберток.

Выборка сущностей

Самый простой случай использования метода `select` – выборка сущностей. Чтобы выбрать сущность, методу `select` нужно передать объект `Root` для сущности. Как это сделать, показано в следующем фрагменте:

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();
CriteriaQuery<Item> query = builder.createQuery(Item.class);
Root<Item> root = query.from(Item.class);
query.select(root);
TypedQuery<Item> tq = entityManager.createQuery(query);
```

Этот фрагмент не требует подробных разъяснений. Объект `Root` передается в вызов метода `select` – в результате мы получаем список объявлений без необходимости выполнять приведение типов, благодаря использованию `TypedQuery`.

Выборка одного значения

С помощью метода `select` можно выбирать определенные значения. В листинге 11.5 приводится метод `getAllItemsNames()`, извлекающий список названий товаров из всех объявлений, посредством прямого запроса атрибута `itemName` с использованием корня запроса. Немного запутанный синтаксис использования метамодели гарантирует, что запрашиваемое значение будет доступно в корне запроса. Вы не сможете запросить имя пользователя, если корнем запроса будет объявление, потому что объявление не содержит имени пользователя. Ниже приводятся соответствующие строки из листинга 11.5:

```
Root<Item> root = query.from(Item.class);
query.select(root.get(Item_.itemName));
```

Выборка нескольких значений

Следующий логический шаг – выборка нескольких значений. Часто бывает необходимо извлечь не одно, а несколько значений – обычно значение и ключ, с которым связано это значение. Для этого нужно создать экземпляр `CriteriaQuery` с параметром типа `Object[]`. Затем с помощью экземпляра `CritieriaBuilder` создать экземпляр `CompoundSelection` и указать значения, какие требуется извлечь. Значениями являются атрибуты любых извлекаемых экземпляров – если выполняется соединение нескольких таблиц, значения можно брать из соединения. Запрос возвращает массив объектов, обращаться с которым следует крайне осторожно, потому что ошибки обработки полученного массива обнаруживаются только на этапе выполнения. Чтобы получить более полное представление, ниже приводится фрагмент, возвращающий `itemId` вместе с `itemName` – что гораздо полезнее, чем извлечение одного только названия:

```
CriteriaQuery<Object[]> query = builder.createQuery(Object[].class);
Root<Item> root = query.from(Item.class);
query.select(builder.array(root.get(Item_.itemId),
root.get(Item_.itemName)));
TypedQuery<Object[]> tq = entityManager.createQuery(query);
```

Выборка объектов-оберток

Работа с массивами объектов представляет довольно сложную проблему, хотя бы потому, что ошибки обнаруживаются только во время выполнения. Кроме того, по ошибке можно обратиться не к тому элементу массива. Например, в зависимости от типа данных, при извлечении ставок в некотором диапазоне дат, можно выбрать не тот элемент массива из-за «ошибки смещения индекса». Используя объекты-обертки можно конструировать объекты специально для обработки результатов запроса. В примере, приводившемся ранее (листинг 11.7), мы исполь-

зовали объект-обертку для выборки всех значений, возвращаемых выражением JOIN, в виде единственного объекта, представляющего сводную информацию о выигравшей ставке. Соответствующий фрагмент показан в листинге 11.8.

Листинг 11.8. Конструирование класса WinningBidWrapper на основе сложного запроса

```
query.select(
    builder.construct(
        // Класс, на основе которого создается экземпляр
        // для каждой строки в результате
        WinningBidWrapper.class,
        // Первый параметр конструктора
        userRoot.get( BazaarAccount_.username ),
        // Второй параметр конструктора
        bidRoot.get( Bid_.bidPrice ),
        // Третий параметр конструктора
        itemRoot.get(Item_.itemName) ,
        // Четвертый параметр конструктора
        itemRoot.get(Item_.description)
));
```

Как показывает этот фрагмент, метод construct экземпляра CriteriaBuilder принимает имя класса, а также столбцы, присутствующие в возвращаемом наборе данных, значения которых будут передаваться конструктору указанного класса. Соответственно список параметров, следующий за именем класса, служит двум целям: он определяет извлекаемые столбцы и параметры конструктора класса объекта-обертки. Указываемый класс является обычным POJO – он не обязан быть сущностью JPA.

Выборка кортежей

Классы-обертки очень удобны, но определение большого числа таких классов может привести загромождению кода и усложнить его сопровождение. Чтобы избежать этого и вместе с тем сохранить все преимущества типизированных запросов, Criteria API предоставляет поддержку извлечения результатов в виде кортежей. Кортеж – это упорядоченный список элементов, в данном случае – результатов. Используя метамодель, вы определяете, какие значения должны быть выбраны, и затем извлекаете отдельные значений из кортежа. Вы можете также извлекать значения по строковым ключам, идентификаторам, и так далее. Метод getWinningBidTuple в листинге 11.9 извлекает все ту же сводную информацию о выигравшей ставке, но для получения результатов использует кортеж.

Листинг 11.9. Получение сводной информации по выигравшей ставке в виде кортежа

```
public List<Tuple> getWinningBidTuple(Long itemId) {
    CriteriaBuilder builder = entityManager.getCriteriaBuilder();
    // Создать новый экземпляр CriteriaBuilder? типизированный как кортеж
    CriteriaQuery<Tuple> query = builder.createTupleQuery();
    Root<Item> itemRoot = query.from(Item.class);
    Root<Order> orderRoot = query.from(Order.class);
```

```
    Root<Bid> bidRoot = query.from(Bid.class);
    Root<BazaarAccount> userRoot = query.from(BazaarAccount.class);
    Join<Order,Bid> j1 = orderRoot.join(Order_.bid);
    Join<Order,Item> j2 = orderRoot.join(Order_.item);
    Join<Order,BazaarAccount> j3 = orderRoot.join(Order_.bidder);
    Path<Long> itemIdPath = itemRoot.get(Item_.itemId);
    Predicate itemPredicate = builder.equal(itemIdPath,itemId);
    // Выбрать атрибуты для кортежа
    query.multiselect(
        userRoot.get( BazaarAccount_.username ),
        bidRoot.get( Bid_.bidPrice ),
        itemRoot.get(Item_.itemName) ,
        itemRoot.get(Item_.description));
    // Создать кортеж TypedQuery
    TypedQuery<Tuple> q = entityManager.createQuery(query);
    query.where(itemPredicate);
    // Выполнить запрос и получить результаты в виде кортежа
    List<Tuple> results = q.getResultList();
    for(Tuple result : results) {
        logger.log(Level.INFO, "Item: {0}",
            // Извлечь значение из кортежа
            result.get(itemRoot.get(Item_.itemName)));
    }
    return q.getResultList();
}
```

Как демонстрирует этот пример, использование кортежа не представляет особых сложностей и мало чем отличается от использования объекта-обертки. К элементам кортежа можно обращаться с помощью метамодели, по индексу или по строковому ключу. Для иллюстрации в конце метода показано, как можно обращаться к отдельным значениям.

11.3. Низкоуровневые запросы

Что в данном случае подразумевается под «низкоуровневыми запросами»? Это – запросы на языке SQL, понятном конкретному серверу баз данных: Oracle, MySQL, Derby, и так далее. До настоящего момента мы конструировали запросы либо на языке JPQL, либо с использованием Criteria API, которые затем преобразовывались в запросы на языке SQL. Этот промежуточный слой обеспечивает независимость от особенностей конкретных баз данных и дает возможность работать в терминах объектов, а не реляционной модели. Однако промежуточный слой абстракции – обоюдоострое оружие. Один из его недостатков заключается в том, что он не позволяет использовать какие-либо специфические возможности баз данных. Низкоуровневые запросы, напротив, дают возможность использовать конкретный диалект языка SQL и отказаться от промежуточного слоя, осуществляющего трансляцию.

Чтобы наглядно увидеть, какие выгоды несет поддержка низкоуровневых SQL-запросов, допустим, что нам требуется сгенерировать иерархический список категорий с подкатегориями. Решить эту задачу на языке JPQL невозможно, потому

что JPQL не поддерживает рекурсивные соединения (recursive joins), реализованные в таких базах данных, как Oracle. Из этого следует, что нам придется использовать низкоуровневые запросы на языке SQL.

Допустим, что мы используем базу данных Oracle и нам требуется извлечь все подкатегории для заданной категории с применением рекурсивных соединений в форме `START WITH ... CONNECT BY ...`:

```
SELECT CATEGORY_ID, CATEGORY_NAME
FROM CATEGORY
START WITH parent_id = ?
CONNECT BY PRIOR category_id = category_id
```

В идеале следует избегать пользоваться низкоуровневыми SQL-запросами, которые нельзя выразить на языке JPQL (как, например, наш SQL-запрос, в котором используются специфические возможности базы данных Oracle). Но, в целях демонстрации, в следующем разделе мы рассмотрим пример простого SQL-запроса, который можно использовать с большинством реляционных баз данных.

> **Примечание.** *Механизм JPA выполняет SQL-запросы, как инструкции JDBC, и не следит за тем, изменились ли данные, которые также хранятся в каких-либо активных сущностях. Желательно избегать использовать SQL-инструкции* `INSERT`, `UPDATE` *и* `DELETE` *в низкоуровневых запросах, потому что механизм хранения не имеет возможности узнать о произведенных изменениях в базе данных, что может привести к рассогласованию данных в сущностях, если ваша реализация JPA использует кэширование.*

Подобно JPQL-запросам, SQL-запросы могут быть динамическими и именованными. При этом не следует забывать о тонких различиях между JPQL и SQL. Запросы JPQL возвращают сущности или множества, а запросы SQL возвращают записи в базе данных. Соответственно SQL-запросы могут возвращать больше чем сущности, осуществляя соединение таблиц. Давайте посмотрим, как пользоваться низкоуровневыми запросами на языке SQL, динамическими и именованными.

11.3.1. Динамические SQL-запросы

Создать динамический SQL-запрос можно с помощью метода `createNativeQuery` интерфейса `EntityManager`, как показано ниже:

```
Query q = em.createNativeQuery("SELECT user_id, first_name, last_name "
          + " FROM users WHERE user_id IN (SELECT seller_id FROM "
          + "items GROUP BY seller_id HAVING COUNT(*) > 1)",
            actionbazaar.persistence.User.class);

return q.getResultList();
```

Как видно в этом примере, метод `createNativeQuery` принимает два параметра: SQL-запрос и класс сущности, которая должна быть возвращена. Это обстоятельство может превратиться в проблему, если потребуется вернуть несколько сущностей разных классов. В таких ситуациях JPA позволяет использовать аннотацию `@SqlResultSetMapping` в паре с методом `createNativeQuery`. Аннотация

`@SqlResultSetMapping` описывает отображение получаемых результатов в одну или более сущностей.

Например, ниже показано, как с помощью аннотации `@SqlResultSetMapping` определить отображение для сущности `User`:

```
@SqlResultSetMapping(name = "UserResults",
    entities = @EntityResult(
        entityClass = actionbazaar.persistence.User.class))
```

и использовать его в низкоуровневом запросе:

```
Query q = em.createNativeQuery("SELECT user_id, first_name, last_name "
          + " FROM users WHERE user_id IN (SELECT seller_id FROM "
          + "items GROUP BY seller_id HAVING COUNT(*) > 1)",
            "UserResults");

return q.getResultList();
```

Этот прием пригодится, когда потребуется написать SQL-запрос, возвращающий более одной сущности. Реализация механизма доступа к базе данных автоматически определит, какие сущности возвращаются запросом по объявлению в аннотации `@SqlResultSetMapping`, создаст требуемые экземпляры и инициализирует их полученными данными.

После того, как запрос будет создан, порядок извлечения результатов для запросов SQL и JPQL ничем не отличается.

11.3.2. Именованные SQL-запросы

Своими особенностями использования именованные SQL-запросы очень похожи на именованные запросы JPQL. Прежде чем использовать именованный запрос, его необходимо создать. Определить именованный SQL-запрос можно с помощью аннотации `@NamedNativeQuery`:

```
public @interface NamedNativeQuery {
    String name();
    String query();
    QueryHint[] hints() default {};
    Class resultClass() default void.class;
    String resultSetMapping() default ""; // имя SQLResultSetMapping
}
```

Тип результата можно определить как класс сущности или как отображение, объявленное с помощью аннотации `@NamedNativeQuery`. Допустим, что нам потребовалось преобразовать запрос из примера выше в именованный SQL-запрос. Первым шагом нужно определить именованный запрос, возвращающий сущность `User`:

```
@NamedNativeQuery(
    name = "findUserWithMoreItems",
    query = "SELECT user_id , first_name , last_name, birth_date
             FROM users
```

```
            WHERE user_id IN
                ( SELECT seller_id
                  FROM items
                  GROUP BY seller_id HAVING COUNT(*) > ?)",
    hints = {@QueryHint(name = "toplink.cache-usage",
        value="DoNotCheckCache")},
    resultClass = actionbazaar.persistence.User.class)
```

Затем, если запрос возвращает более сущности разных классов, нужно определить отображение `SqlResultSetMapping` в атрибуте `resultSetMapping`, как показано ниже:

```
@NamedNativeQuery(
    name = "findUserWithMoreItems",
    query = "SELECT user_id , first_name , last_name, birth_date
             FROM users
             WHERE user_id IN
                (SELECT seller_id
                 FROM items
                 GROUP BY seller_id
                 HAVING COUNT(*) > ?)",
    resultSetMapping = "UserResults")
```

В атрибуте `hints` можно передать дополнительные параметры, зависящие от производителя. Этот атрибут напоминает одноименный атрибут класса `NamedQuery`, обсуждавшегося в разделе 10.3.2.

> **Примечание.** *Порядок использования именованных запросов SQL ничем не отличается от использования именованных запросов JPQL, за исключением того, что спецификация JPA не требует реализации поддержки именованных параметров в низкоуровневых SQL-запросах.*

Чтобы проиллюстрировать, насколько похожи в использовании запросы JPQL и SQL, ниже показано, как вызвать именованный SQL-запрос `findUserWithMoreItems` (определенный ранее в методе сеансового компонента):

```
return em.createNamedQuery("findUserWithMoreItems")
                    .setParameter(1, 5)
                    .getResultList();
```

Эта инструкция сначала создает экземпляр именованного низкоуровневого запроса `findUserWithMoreItems`. Затем определяются позиционные параметры запроса. Наконец, запрос выполняется и возвращается полученный набор результатов.

11.3.3. Хранимые процедуры

В Java EE 7 появилась поддержка вызова хранимых процедур. Хранимая процедура – это, по сути, сценарий, который выполняется внутри базы данных. Язык создания хранимых процедур часто имеет специфические особенности, в зависимости от базы данных – в Oracle используется язык PL/SQL, в Microsoft SQL

Server – Transact-SQL, в PostgreSQL поддерживается собственный язык pgSQL, а также pl/perl и pl/PHP, и так далее. Некоторые базы данных, такие как Oracle, Informix и DB2, поддерживают возможность создания хранимых процедур на языке Java. Хранимые процедуры близко напоминают функции – они точно так же могут принимать параметры и возвращать данные. Данные могут возвращаться в виде возвращаемого значения или в параметрах. Параметры могут быть типа `IN`, `OUT` или `INOUT` – параметры типа `IN` являются входными параметрами, параметры типа `OUT` служат для возврата данных, а параметры типа `INOUT` одновременно являются входными и выходными значениями.

Причин использования хранимых процедур бесчисленное множество: это и высокая производительность, и возможность журналирования, и уменьшение объема сетевого трафика для сложных запросов, и инкапсуляция прикладной логики, и обработка привилегий, и многие другие причины. Хранимые процедуры могут вызываться посредством запросов или запускаться автоматически триггерами. Эти два способа вызова хранимых процедур являются наиболее часто используемыми, но некоторые базы данных предоставляют и другие методы, например с использованием протокола HTTP. Механизм JPA также поддерживает возможность вызова хранимых процедур и получения результатов, как для самых обычных запросов.

Поддержка хранимых процедур реализована в JPA в виде объекта `javax.persistence.StoredProcedureQuery`. Этот класс наследует интерфейс `javax.persistence.Query`, с которым вы уже неоднократно встречались на протяжении этой главы. Создать экземпляр `StoredProcedureQuery` можно вызовом одного из трех методов объекта `EntityManager`:

```
StoredProcedureQuery createStoredProcedureQuery(String procedureName);
StoredProcedureQuery createStoredProcedureQuery(
    String procedureName, Class... resultClasses);
StoredProcedureQuery createStoredProcedureQuery(
    String procedureName, String... resultSetMappings);
```

Эти методы принимают имя процедуры а также типы результатов или отображения. Имя процедуры – это имя процедуры, как оно определено в базе данных. В табл. 11.13 перечислены наиболее важные методы интерфейса `StoredProcedureQuery`. Эти методы используются для настройки параметров, передаваемых процедуре. Как упоминалось выше, параметры могут использоваться для передачи значений (`IN`), получения данных (`OUT`) или одновременно для передачи и получения данных (`INOUT`). Тип `ParameterMode` – это перечисление, определяющее направление передачи параметра.

Таблица 11.13. Ключевые методы интерфейса StoredProcedureQuery

Метод	Описание
`setParameter(Parameter<T> param, T value)`	Устанавливает значение параметра

Таблица 11.13. (окончание)

Метод	Описание

`setParameter(`
`  Parameter<Calendar> param,`
`  Calendar value,`
`  TemporalType temporalType)`
	Устанавливает значение параметра типа `Calendar`

`setParameter(Parameter<Date> param,`
`  Date value,`
`  TemporalType temporalType)`
	Устанавливает значение параметра типа `Date`

`setParameter(String name,` `  Object value)`	Устанавливает значение параметра по его имени в виде строки
`setParameter(String name,` `  Calendar value, TemporalType temporalType)`	Устанавливает значение параметра типа `Calendar` по его имени в виде строки

`setParameter(String name,`
`  Date value,`
`  TemporalType temporalType)`
	Устанавливает значение параметра типа `Date` по его имени в виде строки

`setParameter(int position,` `  Object value)`	Устанавливает значение параметра по номеру позиции
`setParameter(int position,` `  Calendar value, TemporalType temporalType)`	Устанавливает значение параметра типа `Calendar` по номеру позиции

`setParameter(int position,`
`  Date value,`
`  TemporalType temporalType)`
	Устанавливает значение параметра типа `Date` по номеру позиции

`Object getOutputParameterValue(` `  int position)`	Извлекает значение выходного параметра `INOUT` или `OUT` по номеру позиции
`Object getOutputParameterValue(` `  String parameterName)`	Извлекает значение выходного параметра `INOUT` или `OUT` по имени в виде строк

`registerStoredProcedureParameter(`
`  String parameterName,`
`  Class type, ParameterMode mode)`
	Регистрирует параметр хранимой процедуры

`int getUpdateCount()`	Возвращает счетчик обновлений или `-1`, если нет ожидающих результатов
`boolean execute()`	

Чтобы получить более полное представление об особенностях использования хранимых процедур, рассмотрим простой пример из приложения ActionBazaar. Предлагаемая вашему вниманию хранимая процедура (см. листинг 11.10) написана на языке pgSQL. Язык pgSQL – это один из нескольких языков сце-

нариев, поддерживаемых базами данных PostgreSQL – он очень похож на язык PL/SQL. Конкретный язык сейчас для нас не имеет значения, но обратите внимание, что, при использовании хранимых процедур, приложение оказывается тесно связанным с определенной базой данных, что иногда может вызывать проблемы.

Листинг 11.10. Хранимая процедура, возвращающая число ставок для заданного пользователя (pgSQL)

```
-- Определение новой функции PostgreSQL, принимающей один параметр
CREATE FUNCTION getQtyOrders(userId int) RETURNS int AS $$
DECLARE
    qty int;
BEGIN
    SELECT COUNT(*) INTO qty FROM BID WHERE bazaaraccount_user_id = userId;
    -- Вернуть целочисленный результат
    RETURN qty;
END;
$$ LANGUAGE plpgsql;
```

В листинге 11.10 определяется хранимая процедура `getQtyOrders`, принимающая идентификатор пользователя и возвращающая число ставок, сделанных этим пользователем. То есть, процедура имеет один входной параметр и один результат. Входной параметр – целое число. Результат – так же целое число. В листинге 11.11 показано, как осуществляется вызов этой хранимой процедуры из программы.

Листинг 11.11. Вызов хранимой процедуры с помощью JPA

```
// ❶ Создать новый экземпляр StoredProcedureQuery
StoredProcedureQuery spq =
    entityManager.createStoredProcedureQuery("getQtyOrders");
// ❷ Зарегистрировать параметр хранимой процедуры
spq.registerStoredProcedureParameter("param1", Integer.class,
    ParameterMode.IN);
// ❸ Установить значение параметра
spq.setParameter("param1",userID);
Object[] count = (Object[])spq.getSingleResult();
```

Код в листинге 11.11 создает экземпляр `StoredProcedureQuery` ❶ и затем регистрирует целочисленный параметр ❷. Параметр регистрируется под именем `param1`, которое затем используется для установки значения ❸. После этого производится выполнение хранимой процедуры, как самого обычного запроса, и извлечение результатов. Как видите, вызов хранимой процедуры не особенно отличается от выполнения обычного запроса.

На этом мы завершаем наш тур по JPQL, Criteria API и низкоуровневым SQL-запросам. Теперь вспомним основные темы, обсуждавшиеся в этой главе, а затем перейдем к CDI.

11.4. В заключение

В этой главе мы познакомились с JPQL, Criteria API и низкоуровневыми SQL-запросами. Три разных способа извлечения данных предназначены для решения разных задач. JPQL – это язык запросов, синтаксис которого близко напоминает синтаксис SQL, за исключением того, что он работает с объектами. Запросы на этом языке все еще оформляются в виде строк и работа с ними очень похожа на использование `PreparedStatements` из интерфейса JDBC. Прикладной интерфейс Criteria API полностью отличается – он использует метамодель, отражающую сущности JPA и позволяет конструировать типизированные запросы в коде на языке Java. Код получается достаточно громоздким и трудночитаемым, зато он позволяет не волноваться по поводу появления синтаксических ошибок, которые обнаруживаются только во время выполнения. Последний способ, который мы рассмотрели в этой главе, – низкоуровневый язык SQL. Поддержка SQL-запросов дает возможность использовать некоторые особенности, присущие конкретным базам данных. При использовании первых двух подходов, JPA преобразует запросы, написанные на языке JPQL или на Java с применением Criteria API, в низкоуровневый код на SQL. Используя язык SQL напрямую, вы минуете этот промежуточный этап и можете использовать те или иные особенности баз данных. Начиная с версии Java EE 7, появилась также возможность вызова хранимых процедур из JPA, что открывает еще более широкие возможности.

В следующей главе мы перейдем к изучению CDI – мощной технологии внедрения зависимостей, появившейся в Java EE 6 и продолжающей играть важную роль в Java EE 7.

ГЛАВА 12.
Использование CDI в EJB 3

Эта глава охватывает следующие темы:

- внедрение зависимостей для POJO;
- область видимости и жизненный цикл компонентов;
- основные конструкции CDI;
- продолжительные взаимодействия.

Контексты и внедрение зависимостей (Context and Dependency Injection, CDI) – это новая, потрясающая особенность, появившаяся в Java EE 6 и получившая дальнейшее развитие в Java EE 7. Механизм CDI привнес полноценную поддержку внедрения зависимостей и контекстов в платформу Java EE. В более ранних главах вы уже видели примеры внедрения ресурсов и других компонентов EJB с использованием аннотаций `@PersistenceUnit`, `@Resource` и `@EJB`. Мы также коснулись темы интерцепторов, на которых основана поддержка аспектно-ориентированного программирования (AOP). Эти новые возможности вместе с другими инновациями, включая JPA, значительно упростили разработку приложений Java EE.

Но инновации, появившиеся в Java EE 5 вместе с EJB 3, в первую очередь ограничивались компонентами EJB. Чтобы получить возможность пользоваться компонентами EJB из веб-слоя, все еще требовалось писать шаблонный код, извлекающий компонент из JNDI. Компоненты JSF находились в зачаточном состоянии и могли предложить совсем немного. Все эти недостатки пытались восполнить различные решения, такие как JBoss Seam, реализующие улучшенные контейнеры компонентов, связывающие миры EJB и POJO, а также предоставляющие дополнительные возможности для не-EJB компонентов. В сообществе Java обратили внимание на JBoss Seam и этот проект стал основой для разработки механизма CDI, который был введен Java EE 6 как одна из основных технологий и получил дальнейшее развитие в Java EE 7.

В этой главе рассматриваются все основные понятия CDI, а также основные конструкции. На всем ее протяжении мы будем использовать CDI для создания

надежного интерфейса к приложению ActionBazaar и связывания JSF-слоя представления с компонентами EJB. Начнем с того, что познакомимся с происхождением механизма CDI – технологиями, повлиявшими на его развитие.

12.1. Введение в CDI

Первоначально спецификация CDI была разработана в рамках процесса Java Community Process (JCP) как JSR-299. Изначально JSR-299 называлась WebBeans, но быстро стало очевидно, что особенности, описываемые спецификацией JSR-299, далеко вышли за рамки простой замены JSF-компонентов. Прародителем CDI был открытый проект Seam, разрабатываемый компанией JBoss. Целью проекта Seam было упрощение разработки приложений на платформе Java EE за счет поддержки прямого обращения к компонентам EJB из JSF, а так же предоставления компонентов – в основном компонентов Java – которые определялись бы и внедрялись с помощью аннотаций. Проект Seam включал также поддержку создания и отправки электронных писем, интеграции прикладных процессов и контекстов. В число поддерживаемых контекстов входил контекст взаимодействий, значительно упрощавший разработку веб-приложений для веб-браузеров с вкладками. Когда проект Seam только начал развиваться, в Java EE имелась весьма незначительная поддержка внедрения зависимостей (Dependency Injection, DI). В то же время фреймворк Spring произвел революцию в традиционном программировании на Java EE и бросил вызов традиционным стандартным контейнерам Java EE. Этот фреймворк наглядно продемонстрировал мощь внедрения зависимостей и необходимость реализации подобного механизма в Java EE.

Несмотря на то, что Seam существенно упростил разработку на платформе Java EE и сделал возможным непосредственное использование компонентов EJB из JSF, этот проект так и остался простым дополнением. Чтобы задействовать возможности Seam, все еще требовалось загружать и настраивать его для каждого конкретного контейнера. Кроме того, с некоторыми контейнерами имелись проблемы совместимости, поэтому необходимо было проверять работоспособность с определенным контейнером и, возможно, решать проблемы, характерные для этого контейнера. Ценность возможностей, предоставляемых проектом Seam, особенно на фоне конкуренции со стороны Spring и Google Guice, оказалась слишком высокой, чтобы оставлять их на уровне внешнего дополнения. В результате было инициировано создание спецификации JSR и многие базовые особенности Seam были интегрированы в Java EE 6 под видом CDI.

Для интеграции базовых особенностей Seam требовалось явно определить понятие компонента в контексте Java EE. До версии Java EE 6 такого определения не существовало. На тот момент имелось две разновидности компонентов: JSF-компоненты и компоненты Enterprise Java Beans. Чтобы выработать обобщенное определение, было введено понятие управляемого компонента с соответствующей спецификацией. Управляемый компонент – это обычный POJO, управляемый контейнером, который предоставляет базовый набор услуг, включая внедрение ресурсов, события жизненного цикла и интерцепторы. Согласно спецификации

управляемый компонент должен иметь конструктор без аргументов, не должен поддерживать сериализацию и иметь уникальное имя. Управляемые компоненты определяются посредством аннотации `@ManagedBean` и могут иметь методы обратного вызова для обработки событий жизненного цикла, отмеченные аннотациями `@PostConstruct` и `@Destroy`. Некоторые требования, такие как наличие конструктора без аргументов, могут ослабляться применением расширений. Компоненты EJB, JSF и CDI – все это расширения управляемых компонентов.

CDI – это не просто расширение для управляемых компонентов, это полноценный контейнер, который можно использовать независимо от Java EE. Например, CDI можно использовать в автономных настольных приложениях на основе JavaFX. С помощью CDI практически любой POJO можно превратить в управляемый компонент; в том числе и объекты, не имеющие конструкторов без аргументов и не требующие применения специальных фабричных функций для их создания. CDI не навязывает собственную модель компонентов. Что еще более важно, CDI унифицирует модели компонентов JSF и EJB, позволяя из JSF-компонентов напрямую использовать компоненты EJB. Фактически, этот фреймворк можно назвать средством интеграции прикладной логики с пользовательским интерфейсом. На рис. 12.1 показано место CDI в окружении других технологий Java EE.

Как показано на рис. 12.1, CDI выполняет функцию моста между JSF и управляемыми компонентами. С выходом Java EE 7, функции CDI определяются спецификацией JSR-346.

Рис. 12.1. CDI в контексте услуг

12.1.1. Службы CDI

CDI предоставляет ряд базовых услуг, основанных на таких понятиях, как контексты и внедрение зависимостей. Контексты, подробное определение которых будет дано ниже, можно интерпретировать как четко определенные области действия, связанные с жизненным циклом. В предыдущих главах вы уже видели, как действует внедрение зависимостей, но CDI поднимает DI на новый уровень. Прежде мы использовали механизм внедрения зависимостей, встроенный в EJB 3, для внедрения объектов подключения к базе данных, контекстов хранения JPA и ссылок на другие компоненты EJB. Делали мы это с помощью специальных аннотаций, таких как `@Resource`, `@PersistenceContext` и `@EJB`. В EJB 3 механизм внедрения зависимостей ограничивается только компонентами EJB – вы не сможете выполнить внедрение чего-либо в произвольный POJO или внедрить простой объект в компонент EJB. В CDI подобные ограничения отсутствуют, в чем вы вскоре убедитесь.

CDI – это контейнер объектов, который можно использовать отдельно или внутри имеющегося контейнера Java EE. Прежде мы уже затрагивали две основные особенности CDI, однако коме них, контейнер CDI поддерживает возможность управления жизненным циклом объектов с сохранением состояния, связывания объектов с четко определенными контекстами, осуществления типизированного внедрения зависимостей, уведомления о событиях, а также надежные интерцепторы. Давайте рассмотрим каждую из перечисленных особенностей в отдельности.

Жизненный цикл объектов с сохранением состояния

Контейнер CDI поддерживает четко определенный жизненный цикл для подконтрольных ему компонентов. CDI – это расширение спецификации управляемых компонентов, которая предусматривает возможность определения обработчиков для управления созданием новых объектов и уничтожением существующих. Однако возможности, предусмотренные спецификацией управляемых компонентов, весьма ограниченны – спецификация просто по-иному сформулировала ограничения, существующие для JSF-компонентов, но она допускает возможность расширения и усовершенствования.

CDI расширяет спецификацию управляемых компонентов, добавляя новые способы их создания. В частности, смягчено требование к наличию конструктора без аргументов. Под управлением CDI компоненты могут создаваться с использованием конструкторов, принимающих аргументы, при этом параметры будут «внедряться» в вызовы конструкторов. Кроме того, предусматривается возможность определения специальных методов для создания экземпляров компонентов. Это открывает возможность использования шаблона проектирования «Фабричный метод» для организации создания компонентов и существенно увеличивает гибкость.

Контексты

Понятие контекста выглядит достаточно простым, если рассматривать его на примере типичного веб-приложения электронной коммерции. Большинство веб-

приложений имеют, по крайней мере, два контекста, или области видимости: приложение и сеанс. Данные, хранящиеся в *области видимости приложения*, являются общими для всех – они не связаны с каким-то определенным пользователем. Примером могут служить данные, кэшированные для отображения на главной странице. Эта страница доступна всем пользователям и нет никакого смысла всякий раз обращаться к базе данных за одной и той же информацией для обслуживания каждого посетителя. *Область видимости сеанса*, напротив, связана с определенным пользователем – обычно с окном браузера. Типичным примером области видимости сеанса может служить виртуальная корзина покупателя. Корзина содержит покупки для каждого отдельно взятого посетителя. Данные, связанные с конкретным пользователем, передаются между браузером и сервером либо в виде cookies, либо в виде параметров строки запроса в URL. Области видимости приложения и сеанса – это разные типы контекстов.

Мы только что рассмотрели два разных контекста, имеющиеся в типичном веб-приложении. Но, если отступить немного назад, можно заметить, что существует еще несколько контекстов. С появлением вкладок в браузерах, пользователи получили возможность одновременно просматривать разные разделы одного веб-сайта. Например, на сайте туристического агентства пользователь может сравнивать стоимость туристической поездки в Рим в два разных дня или условия проживания в двух разных отелях. То есть, для одного и того же пользователя может быть создано несколько разных контекстов. Кроме того, пользователь может создавать вложенные контексты, запустив в одном окне процедуры выбора отеля и оформления проката автомобиля. Как видите, области видимости приложений и области видимости сеанса недостаточно – необходимы дополнительные контексты.

Решить эту проблему поможет CDI. Эта технология поддерживает дополнительные контексты и упрощает их создание. А так как CDI является еще и контейнером, появляется возможность управления объектами в контекстах, ассоциированных с ними. Объекты и их жизненные циклы оказываются неразрывно связанными с контекстами. Это чрезвычайно мощная концепция; теперь большая часть рутинных задач, которые раньше приходилось решать в приложениях, перекладывается на плечи контейнера.

CDI поддерживает четыре встроенных контекста, или области видимости:

- приложения;
- диалога;
- запроса;
- сеанса.

Помимо этих четырех контекстов существует еще два псевдоконтекста: область видимости одиночного (singleton) объекта и область видимости зависимости (dependent). Область видимости одиночного объекта – это контекст компонента-одиночки. Область видимости зависимости – это область видимости, назначаемая компоненту по умолчанию, если явно не была присвоена иная область видимости. Объект, принадлежащий области видимости зависимости, создается в момент создания объекта-владельца области видимости, и уничтожается вместе с уничтоже-

нием объекта-владельца. Это пояснение кому-то может показаться непонятным, но все прояснится, если вы продолжите чтение этой главы.

Однако вы не ограничены этими шестью областями видимости, или контекстами. CDI имеет расширяемую архитектуру и написав совсем немного кода, можно реализовать поддержку собственных контекстов. Например, в CDI можно добавить поддержку область видимости представления JSF.

Типизированное внедрение зависимостей

В отличие от других фреймворков DI, CDI не использует строковые идентификаторы, чтобы выяснить, какой объект требуется внедрить. Для этого используется информация о типах, которая поддерживается объектно-ориентированной моделью языка Java. В ситуациях, когда выбор оказывается неоднозначным из-за наличия нескольких объектов, соответствующих заданному типу, для уточнения выбора можно использовать квалифицирующие аннотации. Поскольку CDI использует систему типов и аннотации, никогда не возникает сомнений в том, что будет внедрено в том или ином случае, и всегда гарантируется соответствие типов.

Уведомления о событиях

Одна из опасностей, подстерегающих крупные приложения с богатыми функциональными возможностями, – тесная взаимозависимость их компонентов. С развитием приложения увеличивается риск превращения его в трудно управляемую «кашу» из кода. Применение механизма внедрения для регистрации обработчиков не решает проблему, а только усугубляет ее, потому что в серверном приложении могут запускаться новые сеансы, а объекты могут сериализоваться для передачи между слоями. Несмотря на то, что внедрение упрощает процедуру приобретения ссылки на объект, одного этого оказывается недостаточно. В действительности нам необходимо иметь возможность объявить, что компонент обрабатывает определенные события и позволить фреймворку самому заниматься их маршрутизацией. Добиться этого можно добавлением аннотаций к методам, занимающимся обработкой событий, а все остальное взвалить на плечи контейнера, по аналогии с библиотекой Swing, при использовании которой программист отмечает метод, как обработчик событий типа `ActionListenerEvent`, а регистрация обработчика выполняется автоматически.

Интерцепторы

Вы уже знакомы с некоторыми особенностями интерцепторов в EJB. Поддержка интерцепторов в обеих технологиях, EJB и CDI, определяется спецификацией «JSR-381 Interceptors 1.1». Однако в CDI интерцепторы можно определять не только для компонентов EJB, но и для любых других компонентов, управляемых контейнером CDI. В CDI интерцепторы можно использовать с прикладными методами компонентов, а также с методами обработки событий жизненного цикла и таймеров. Когда мы приступим к изучению тонкостей реализации, вы увидите, как использовать аннотации CDI для создания интерцепторов методов и классов.

В CDI появилась новая конструкция, тесно связанная с интерцепторами, которая называется *декораторы* (*decorators*). Декоратор – это тот же интерцептор, но связанный с определенным интерфейсом. Декоратор реализует интерфейс, методы которого будут перехватываться. То есть, декоратор – это интерцептор, обеспечивающий перехват определенной прикладной логики. Декоратор, в отличие от интерцептора, опирается на точное знание класса, для которого предусматривается перехват методов. Интерцептор – это универсальное решение, предназначенное для реализации сквозной функциональности, независимой от классов, а декоратор – решение для реализации сквозной функциональности в конкретной иерархии классов.

Теперь, когда вы получили некоторое представление об основных особенностях CDI, переключим наше внимание на взаимосвязь CDI и EJB 3. В конце концов, данная книга посвящена EJB, соответственно вас наверняка интересует, как эти две технологии дополняют друг друга.

12.1.2. Отношения между CDI и EJB 3

Отношения между EJB и CDI могут показаться странными на первый взгляд. Казалось бы, оба являются контейнерами объектов и, соответственно, в чем-то дублируют функциональность друг друга. Однако это не совсем так. Компоненты EJB по-прежнему находятся под управлением контейнера EJB. То есть, контейнер EJB решает все задачи, связанные с поддержкой транзакций, конкурентного выполнения и всех остальных функциональных особенностей, о которых рассказывалось в предыдущих главах, а CDI управляет собственными компонентами и предоставляет дополнительные услуги компонентам EJB. Эти две технологии не конкурируют между собой, а дополняют друг друга. Компоненты CDI можно рассматривать как контейнеры объектов, поддерживающие внедрение, события, интерцепторы и контексты для объектов, которым не нужен полный комплекс услуг, предоставляемых контейнером EJB. В этой книге мы уже очертили круг применения компонентов EJB – там, где требуется поддержка транзакций, безопасности, и так далее. CDI – это контейнер для POJO.

Технологии CDI и EJB полностью интегрируются друг с другом. Это означает, что компоненты EJB могут пользоваться всеми возможностями, обсуждавшимися выше, включая внедрение зависимостей, уведомление о событиях, интерцепторами и декораторами. Эти возможности доступны компонентам EJB всех типов: одиночкам, с сохранением состояния, без сохранения состояния и MDB. Поскольку функциональные возможности этих контейнеров все же в чем-то дублируются, возникает законный вопрос: когда следует пользоваться услугами контейнера EJB (внедрение, интерцепторы, и так далее), а когда – аналогичными услугами контейнера CDI. Ответ прост: всегда используйте возможности CDI, потому что они более мощные, универсальные и не ограничиваются только поддержкой компонентов EJB. Как будет показано далее, вы с легкостью сможете использовать аннотацию `@Inject` вместо `@EJB`.

Совместимость этих двух технологий значительно упрощает использование компонентов одиночек, с сохранением состояния и без сохранения состояния. Ме-

ханизм внедрения зависимостей в CDI поддерживает внедрение компонентов EJB в любые компоненты CDI и избавляет от необходимости писать код, извлекающий компоненты из JNDI. Просто добавьте аннотацию `@Inject`, а все остальное сделает контейнер CDI. Использование компонентов EJB никогда не было таким простым, как сейчас, – вы можете рассматривать компоненты EJB как самые обычные POJO. Кроме того, если в компоненте CDI потребуется воспользоваться дополнительными возможностями, можно просто добавить аннотацию `@Stateless`, `@Singleton` или `@Stateful` и превратить его в компонент EJB.

Технология CDI не является заменой EJB – это мощное расширение, причем, не только для EJB но и для JSF.

12.1.3. Отношения между CDI и JSF 2

Мы уже касались ключевых аспектов отношений между CDI и JSF 2. Технология CDI обеспечивает надежную замену компонентам JSF. Улучшенные компоненты CDI обладают намного более богатыми возможностями, в сравнении с компонентами JSF. Компоненты CDI гораздо гибче – их не нужно определять в файле *faces-config.xml*. Благодаря CDI, практически любой объект Java можно превратить в компонент, и использовать несколько разных способов для создания экземпляров. Компоненты CDI можно без каких-либо затруднений использовать как замену компонентам JSF.

Как уже отмечалось выше, когда мы обсуждали отношения между CDI и EJB, технология CDI оказывает большое влияние на возможность использования компонентов EJB из JSF. Благодаря CDI, компоненты JSF с легкостью могут вызывать методы компонентов EJB непосредственно – вам больше не нужно писать код, который связывал бы компоненты JSF и EJB. То есть, из реализации страницы JSF можно непосредственно вызвать метод компонента-одиночки, или сеансового компонента. Доступ к компонентам EJB можно организовать непосредственно в выражениях на языке Expression Language (EL).

Подчеркнем еще раз полную совместимость CDI и JSF, но имейте в виду, что технологии CDI и JSF не имеют тесной связи. Компоненты JSF могут пользоваться всеми преимуществами CDI, но сама технология CDI совершенно нейтральна к выбору веб-фреймворка. Более того, поддержка CDI несомненно будет реализована и в других веб-фреймворках. В настоящее время уже существуют расширения для интеграции CDI и Struts 2. Так как теперь CDI является одной из основных составляющих Java EE 6 и 7, поддержка CDI будет только расширяться. Теперь, когда вы получили концептуальное представление о CDI, можно переходить к знакомству с компонентами CDI.

12.2. Компоненты CDI

В отличие от EJB, CDI не имеет своей модели компонентов. Компонентом CDI может быть управляемый компонент (JSF), компонент Enterprise Java Bean или POJO. Все эти разновидности объектов могут пользоваться возможностями CDI.

В этой главе под словами «компонент CDI» мы обычно будем подразумевать POJO, если явно не будет указано иное. Таким образом, компоненты CDI – более широкое понятие, не ограничивающееся каким-то одним классом компонентов, таким как JSF или EJB.

Хотя технология CDI никак не связана с JSF, вместо компонентов JSF лучше использовать компоненты CDI (POJO). Это избавит вас от необходимости определять компоненты в файле *faces-config.xml* или использовать аннотацию `@ManagedBean`.

Каждый компонент CDI связан с контекстом, имеет тип и может быть квалифицирован. Контекст определяет жизненный цикл компонента, то есть, будет ли он существовать только во время обработки одного запроса или станет частью более продолжительного диалога или процесса. Одной из особенностей CDI является использование системы типов для внедрения. Как результат, там, где другие контейнеры используют строковые имена, CDI использует типы Java. Так как компонент может иметь только один тип, CDI включает механизм квалификации, который позволяет безопасно различать экземпляры одного типа. А теперь, имея определение компонента CDI, посмотрим, как пользоваться компонентами CDI.

12.2.1. Как пользоваться компонентами CDI

Создатели CDI выбрали подход на основе предпочтений соглашений перед настройками. Это означает, что вы можете приступать к использованию компонентов CDI немедленно. Если ваш контейнер поддерживает CDI (а такая поддержка имеется во всех современных контейнерах, совместимых с Java EE 6), вам достаточно будет просто добавить в приложение файл *beans.xml*. Этот файл служит двум целям: он хранит настройки для CDI и служит маркером, что контейнер CDI должен исследовать JAR-файлы на наличие компонентов. В листинге 12.1 показано содержимое пустого файла с настройками. Этот файл следует поместить в каталог *META-INF* любого архива, содержащего компоненты.

Листинг 12.1. Пустой файл с настройками beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
       http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
</beans>
```

Компонентом CDI может быть любой POJO, присутствующий в JAR-архиве. Обычно экземпляр компонента создается путем внедрения экземпляра или ссылки на него из страницы JSF. Контейнер CDI сам позаботится о создании экземпляра компонента. Важно помнить, что компоненты CDI обладают большей гибкостью, чем управляемые компоненты JSF или компоненты EJB. Создавая компоненты на основе POJO, можно объявлять конструкторы с параметрами. Объявить можно только один конструктор с аргументами, а его параметрами должны быть другие

компоненты CDI (POJO, управляемые компоненты JSF или компоненты EJB). Кроме того, CDI поддерживает создание компонентов с помощью фабричных методов, что особенно полезно в ситуациях, когда требуется сохранить ссылку на компонент или получить более полный контроль над процедурой создания компонента. Мы рассмотрим дополнительные способы создания компонентов далее в этой главе. Но сначала исследуем приемы именования компонентов.

12.2.2. Именование компонентов и их разрешение в выражениях EL

В отличие от других фреймворков DI, контейнер CDI не использует строковые идентификаторы. Вместо этого он опирается на систему типов Java, благодаря чему внедрение зависимостей выполняется с учетом типов объектов.

Однако в выражениях на унифицированном языке EL отсутствует информация о типах. Чтобы решить эту проблему, CDI предоставляет аннотацию `@Named`, которую следует помещать перед классами или фабричными методами (описываются ниже). Данная аннотация определяет имя, которое затем можно использовать в выражениях EL. Определение аннотации приводится ниже:

```
package javax.inject;
@Qualifier
@Documented
@Retention(RUNTIME)
public @interface Named {
    String value() default "";
}
```

Эта аннотация имеет необязательный атрибут `value`, значение которого используется как имя компонента. Если значение атрибута `value` не указано, в качестве имени будет использоваться имя класса. Аннотация `@Named` не превращает класс в компонент CDI; она просто определяет имя для класса, которое можно будет использоваться в выражениях EL, как показано в листинге 12.2.

Листинг 12.2. Класс Employee, отмеченный для использования в EL

```
@Named           // ❶ Присвоить имя в виде строки Employee
@RequestScoped   // Ограничить область видимости класса запросом
public class Employee extends User implements Serializable {

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
    ...
}
```

В листинге 12.2 приводится определение класса `Employee` из приложения ActionBazaar. Аннотация `@Named` применяется к классу ❶. Соответственно этот класс будет доступен для использования в выражениях EL. Кроме того, область видимости экземпляров класса `Employee` ограничена запросом – для обработки каждого запроса будет создаваться новый экземпляр. В листинге 12.3 показано, как сослаться на экземпляр `Employee` из файла *editEmployee.xhtml*.

Листинг 12.3. Ссылка на экземпляр Employee из файла editEmployee.xhtml

```
<!DOCTYPE html>
<HTML
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
        <title>Edit Employee</title>
    </h:head>
    <h:body>
        <h:form id="accountForm">
            <h:panelGrid columns="2">
            <h:outputLabel for="title" value="Title"/>
<!-- Обращение к свойству title экземпляра Employee в выражении EL -->
            <h:inputText id="title" value="#{employee.title}"/>
            <h:outputLabel for="username" value="User name"/>
<!-- Обращение к свойству username экземпляра Employee в выражении EL -->
            <h:inputText id="username" value="#{employee.username}"/>
        ...
            </h:panelGrid>
        ...
        </h:form>
    </h:body>
</HTML>
```

Как показано в листинге 12.3, компонент `Employee` получил имя `employee`, в соответствии с соглашениями, принятыми в Java, – первая буква имени в нижнем регистре. Без аннотации `@Named` фреймворк JSF не смог бы найти компонент с именем `employee`.

Теперь перейдем к исследованию областей видимости. Мы уже говорили об областях видимости и контекстах выше в этой главе. В листинге 12.2 присутствует аннотация `@RequestScoped`, назначение которой кому-то может показаться непонятным.

12.2.3. Области видимости компонентов

Выше мы уже обсудили понятия контекстов и областей видимости – эти два термина могут использоваться взаимозаменяемо. Контейнер CDI поддерживает четыре контекста и два псевдоконтекста. Все компоненты связаны с определенными контекстами, определяющими их жизненные циклы. Когда контекст уничтожается, уничтожаются и соответствующие компоненты. Область видимости (контекст) компонента определяется с помощью аннотаций, которые могут помещать-

ся перед классами или фабричными методами (о которых рассказывается ниже). Аннотации для определения разных контекстов перечислены в табл. 12.1.

Таблица 12.1. Области видимости/контексты в CDI

Контекст	Описание
`@ApplicationScoped`	Экземпляр создается только один раз с момента запуска приложения и уничтожается при его завершении.
`@Dependent`	Экземпляр создается всякий раз, когда выполняется внедрение. Это – контекст по умолчанию и используется в подавляющем большинстве случаев.
`@ConversationScoped`	Новый тип контекстов, добавленный в CDI, но поддерживавшийся в JBoss Seam. Это контекст, управляемый программно. Он распространяется на несколько запросов, но короче контекста сеанса. В веб-приложениях контекст диалога (conversation) используется для выполнения операций, включающих несколько запросов. Например, к таким ситуациям относится оформление пользователем двух разных заказов в разных вкладках браузера.
`@RequestScoped`	Этот контекст соответствует стандартному HTTP-запросу. Он создается при получении запроса и уничтожается после отправки ответа.
`@SessionScoped`	Этот контекст соответствует стандартному HTTP-сеансу. Ссылки, созданные в этом контексте, остаются действительными до конца сеанса.

В отсутствие аннотаций перед компонентом, он автоматически получает область видимости зависимости (dependent). Это означает, что новый экземпляр компонента будет создаваться всякий раз, когда выполняется внедрение этого компонента. Большинство компонентов в приложениях обычно получают контекст зависимости. Компонентам JSF в веб-приложениях следует присваивать либо контекст запроса, либо контекст диалога (conversation).

Контекст зависимости и CDI

Частой ошибкой при использовании CDI является применение области видимости зависимости к компонентам JSF-страниц. Проблема состоит в том, что при таком подходе для каждого вызова будет создаваться новый компонент. То есть, когда пользователь отправит форму, в ответ будет создан новый экземпляр, который будет выглядеть, как если бы пользователь ничего не присылал. Всегда используйте хотя бы аннотацию `@RequestScoped` или более привычную `@Model`.

При использовании CDI, все компоненты связаны с определенными контекстами, определяющими их жизненные циклы.

Контекст диалога

Контекст диалога (conversation) заслуживает обсуждения в отдельном разделе. В отличие от других областей видимости, контекст диалога – совершенно новый

зверь. Если вам приходилось пользоваться предшественником CDI, JBoss Seam (версии 1 или 2), понятие диалога наверняка будет вам не в новинку. Диалог – это область видимости, более короткая, чем сеанс, но более длинная, чем запрос. Это показано на рис. 12.2. Управление контекстом диалога осуществляется программно; программный код сам определяет, когда начинается диалог и когда он заканчивается.

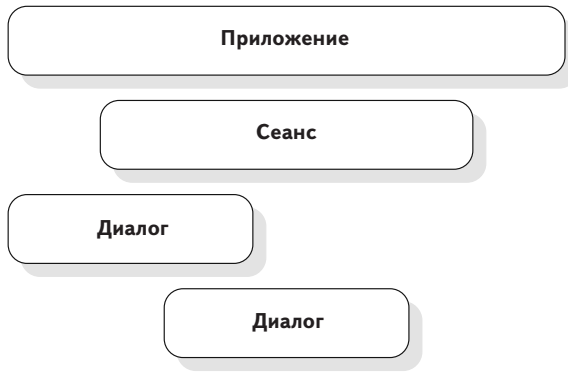

Рис. 12.2. Области видимости приложения, сеанса и диалога

Чтобы проще было понять, что такое диалог представьте, как пользователь взаимодействует с веб-приложением при помощи современного браузера. Покупатель может подключиться к приложению ActionBazaar и открыть в разных вкладках два объявления, заинтересовавшие его. Однако, даже при том, что покупатель просматривает два объявления в разных вкладках, с точки зрения приложения пользователь действует в рамках единственного сеанса. Как результат, информацию о текущем просматриваемом объявлении нельзя хранить на уровне сеанса, потому что сеанс делится между двумя вкладками. А теперь представьте, что произойдет, если пользователь в одной вкладке будет просматривать объявление о продаже лодки, а в другого – о продаже бриллианта в пять карат. Когда пользователь переключится во вкладку с объявлением о продаже лодки и щелкнет на кнопке **Place Bid** (Сделать ставку), он будет удивлен, если на странице подтверждения ему будет предложено подтвердить ставку на покупку бриллианта. Эту проблему решает контекст диалога.

Чтобы отметить компонент, как принадлежащий диалогу, нужно добавить аннотацию `@ConversationScoped` к его определению. По умолчанию диалог ассоциируется с текущим контекстом запроса и завершается вместе с контекстом запроса. Чтобы продлить срок существования контекста диалога, его нужно превратить в контекст *продолжительного диалога*. Такое превращение осуществляется программно. Для этого следует получить текущий контекст диалога через внедрение и вызвать метод `begin`. После этого контекст диалога может быть завершен программно или спустя установленный период времени. При этом он не может продолжаться дольше сеанса.

Мы еще вернемся к контексту диалога далее в этой главе, после знакомства с основами CDI. Контекст диалога является одной из самых сложных особенностей. Контекст диалога – это пример того, как развивается стандарт EE под влиянием технологий, разрабатываемых частными компаниями. Контекст диалога впервые был реализован в проекте JBoss Seam и позднее интегрирован в Java EE 6.

12.3. Следующее поколение механизмов внедрения зависимостей

Технология внедрения зависимостей существует уже достаточно давно. Она получила известность еще в коммерческих фреймворках. Разработчики CDI не только повторили успех этих фреймворков, но и пошли на шаг дальше, обеспечив типизированное внедрение. Для разрешения зависимостей в CDI используются не строки в свободной форме, а система типов Java, что в сочетании с жизненным циклом компонентов обеспечило следующий логический шаг на пути развития внедрения зависимостей. Итак, начнем со знакомства с аннотацией `@Inject`, которую вам придется использовать довольно часто.

12.3.1. Внедрение с помощью @Inject

Аннотация `@Inject` составляет основу CDI. Эта аннотация отмечает точку, куда должен быть внедрен экземпляр компонента. Она может помещаться или перед переменной экземпляра, или перед конструктором. Когда контейнер CDI создает экземпляр класса, содержащий поле с аннотацией `@Inject`, он сначала проверяет наличие готового экземпляра, и только потом создает новый экземпляр и сохраняет в поле. Важно также отметить, что если компонент не отмечен аннотацией области видимости (например, `@Conversation`, `@RequestScoped` или `@SessionScoped`), ему по умолчанию присваивается *область видимости зависимости*, которую можно присвоить явно, с помощью аннотации `@Dependent`. Контейнер CDI определяет тип точки внедрения и с помощью возможных дополнительных квалифицирующих аннотаций выясняет, какой объект требуется внедрить или сначала создать, а затем внедрить. Все эти операции выполняются при создании первого экземпляра компонента.

Рассмотрим пример использования аннотации `@Inject` в ActionBazaar. В этом приложении есть компонент `LandingController`, реализующий главную страницу. Он имеет область видимости приложения и кэширует последние объявления для отображения на главной странице. Кэширование объявлений осуществляется с целью избежать обращений к базе данных при каждом запросе главной страницы. Обращения к базе данных по каждому запросу могли бы значительно ухудшить масштабируемость сайта из-за большого объема совершенно ненужных операций ввода/вывода. Реализация компонента `LandingController` показана в листинге 12.4.

Листинг 12.4. Компонент главной страницы

```
@Named               // Превращает LandingController в JSF-компонент
@ApplicationScoped   // Будет создан единственный экземпляр LandingController
public class LandingController {

    // ❶ Внедрить itemManager
    @Inject
    private ItemManager itemManager;

    private List<Item> newestItems;

    // ❷ Вызывать после внедрения
    @PostConstruct
    public void init() {
        newestItems = itemManager.getNewestItems();
    }

    public List<Item> getNewestItems() {
        return newestItems;
    }
}
```

В центре внимания этого примера – аннотация `@Inject` перед полем `itemManager` ❶. Когда контейнер CDI приступит к созданию экземпляра `LandingController`, он извлечет экземпляр сеансового компонента без сохранения состояния `ItemManager` из контейнера EJB, а после внедрения вызовет метод с аннотацией `@PostConstruct` ❷. Обратите внимание, что в данном конкретном случае с тем же успехом можно было бы использовать аннотацию `@EJB`. Но аннотация `@Inject` позволяет внедрять любые другие POJO, в чем вы убедитесь далее в этой главе. В листинге 12.5 демонстрируется еще один способ применения аннотации `@Inject` – для внедрения параметров конструктора.

Листинг 12.5. Внедрение параметров конструктора

```
@Named
@ApplicationScoped
public class LandingController {

    // ❶ Переменная экземпляра больше не отмечена аннотацией
    private ItemManager itemManager;

    private List<Item> newestItems;

    // ❷ Конструктор по умолчанию для контейнера
    //    (для выполнения сериализации и т.д.)
    protected LandingController() {}

    // ❸ Конструктор с параметром отмечен аннотацией @Inject
    @Inject
    // Принимает параметр типа ItemManager
    public LandingController( ItemManager itemManager) {
```

```
        this.itemManager = itemManager;
    }
    ...
}
```

Использование внедрения с конструктором дает еще один способ управления созданием компонентов ❸. Аннотацией `@Inject` может быть отмечен только один конструктор; если аннотировать несколько конструкторов, контейнер не сможет понять, какой из них следует использовать. В этом определении `LandingController` мы убрали аннотацию, находившуюся перед полем ❶ и добавили параметр в конструктор. Конструктор по умолчанию остался ❷; он часто требуется для нужд контейнеров, потому что контейнеры имеют обыкновение обертывать компоненты прокси-объектами для поддержки интерцепторов.

В первом примере мы легко могли бы использовать аннотацию `@EJB`. Истинная мощь аннотации `@Inject` стала очевидна только после аннотирования конструктора. Это наиболее естественное решение – оно избавляет от необходимости использовать аннотацию `@PostConstruct` для определения операций настройки, выполняемых после создания компонента и внедрения зависимостей. Возможность применения механизма внедрения к конструкторам позволяет компонентам действовать подобно обычным POJO – конструктор получает параметры и инициализирует объект, как это принято в объектно-ориентированном мире. Вам не нужно выполнять внедрение и использовать метод-обработчик для инициализации компонента.

Во всех примерах, рассматривавшихся до настоящего момента, все компоненты создавались контейнером. Но нередко бывает желательно иметь возможность создавать и уничтожать компоненты вручную. Реализовать такую возможность можно с помощью фабричных методов, давайте посмотрим как.

12.3.2. Фабричные методы

В предыдущем разделе мы узнали, как организовать внедрение зависимостей в параметры конструкторов, чтобы создание компонентов выглядело более естественным и похожим на создание простых POJO. Но нередко возникают ситуации, когда необходимо иметь полный контроль над созданием компонента. Например, экземпляр компонента может создаваться на основе записи в базе данных, или может потребоваться выполнить некоторые дополнительные операции по настройке компонента, или необходимо предварительно выбрать тип создаваемого компонента. Для удовлетворения этого требования в CDI поддерживается понятие *метода-продюсера* (*producer method*), или *фабричного метода*. Фабрика – это метод или поле экземпляра, к которому обращается контейнер CDI для создания экземпляра. Метод или поле экземпляра должен быть отмечен аннотацией `@Producer`. Так как контейнер CDI выполняет типизированное внедрение, по типу поля или возвращаемого значения он определяет тип создаваемого компонента. Фабрика может быть аннотирована дополнительным квалификатором, но мы пока отложим обсуждение квалификаторов и вернемся к ним ниже, в этой же главе. Чтобы

лучше разобраться с фабриками, рассмотрим пример из приложения ActionBazaar, представленный в листинге 12.6.

Листинг 12.6. Фабрика, возвращающая экземпляры "currentUser"

```
@Named
public class CurrentUserBean implements Serializable {

    @EJB
    private UserService userService;

    private User user;

    @Produces      // ❶ Отмечает метод как фабрику
    @SessionScoped // ❷ Новые экземпляры получают область видимости сеанса
    @Named("currentUser") // ❸ Экземпляр будет доступен в JSF
    public User getCurrentUser() {
        if(user == null || user.isAnonymous()) {
            user = userService.getAuthenticatedUser();
        }
        return user;
    }

    public boolean isAuthenticated() {
        return userService.isAuthenticated();
    }
}
```

Здесь определен фабричный метод, который должен создать и вернуть компонент `User`, представляющий текущего пользователя, посетившего вебсайт ActionBazaar. Этот метод отмечен аннотацией `@Produces` ❶. Аннотация `@SessionScoped` ❷ информирует контейнер CDI, что экземпляр должен сохраняться на протяжении всего сеанса. как результат, этот метод будет вызываться только один раз для каждого сеанса; созданный экземпляр будет помещен в кэш сеанса. Аннотация `@Named` ❸ делает экземпляр доступным для JSF – присвоение строкового имени позволяет ссылаться на компонент из выражений EL. В листинге 12.7 показано, как могла бы выглядеть страница JSF, использующая этот компонент `User`.

Листинг 12.7. Вызов фабричного метода в index.xhtml

```
<HTML xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html">
    <h:body>
        <h:form id="itemForm">
            <!-- ❶ Проверить, был ли аутентифицирован пользователь -->
            Hi! <h:outputText rendered="#{currentUserBean.authenticated}"
                <!-- ❷ Персонализировать страницу именем пользователя -->
                value="#{currentUser.username}"/>
            <h:link rendered="#{!currentUserBean.authenticated}"
                value="Sign in" outcome="login"/>
```

```
            <!-- ❸ Вывести ссылку на форму входа,
                    если пользоватедь пока не аутентифицирован -->
            <h:commandLink rendered="#{currentUserBean.authenticated}"
                immediate="true"
                action="#{logoutController.logout()}">Log-out</h:commandLink>
        </h:form>
    </h:body>
</HTML>
```

Фабричный метод, представленный в этом листинге, вызывается косвенно. Так как возвращаемый экземпляр будет кэшироваться в сеансе, фабричный метод должен вызываться только после аутентификации пользователя и имеется объект, представляющий пользователя с ролью продавца или покупателя. Поэтому сначала проверяется, был ли аутентифицирован пользователь ❶. Если пользователь аутентифицирован ❷, вычисляется выражение, возвращающее имя пользователя, которое либо извлекает экземпляр компонента текущего пользователя из объекта сеанса, либо вызывает фабричный метод, возвращаемое значение которого будет кэшировано в сеансе. Если посетитель еще не аутентифицирован, отображается ссылка на форму аутентификации ❸.

Обратите внимание, что в этом примере контейнер CDI просканирует все классы в JAR-файле, если он содержит файл *beans.xml*. Он запомнит все фабричные методы, найденные при анализе классов. Когда приложение запросит экземпляр компонента – в данном случае экземпляр `User` – контейнер CDI вызовет фабричный метод, если экземпляр компонента еще не был создан в текущей области видимости.

Фабричные методы и JPA

Возможно, вам будет любопытно узнать, как CDI творит все это волшебство. Чтобы обеспечить все эти возможности, CDI обертывает компоненты прокси-объектами. Если передать оператору `instanceof` внедренный компонент и его тип, он вернет `true`. В большинстве случаев тот факт, что внедренный компонент является прокси-объектом, не представляет никаких проблем. Но многие реализации JPA заставляют задуматься над этим. Если попытаться передать экземпляр компонента, созданный контейнером CDI, в JPA, JPA сообщит, что компонент не может быть сохранен и имеет неизвестный тип. Чтобы исправить эту проблему, необходимо использовать фабричный метод и сохранять ссылку на оригинальный POJO. Все ссылки, приобретаемые путем внедрения, ссылаются на прокси-объекты.

В любом компоненте внутри приложения вы теперь сможете использовать следующий код для получения ссылки на текущего пользователя:

```
@Inject
private User currentUser;
```

Просматривая такой код, не забывайте обращать внимание на тип. Для контейнера CDI имя поля не имеет никакого значения – он выполняет внедрение, опираясь на тип. Но иногда может возникнуть потребность внедрить другой экземпляр `User`, например, представляющий пользователя, обслуживаемого в настоящий мо-

мент сотрудником отдела поддержки. В следующем разделе вы узнаете, как пользоваться квалификаторами для добавления к точке внедрения дополнительной идентификационной информации.

12.3.3. Квалификаторы

Внедрение зависимостей исключительно на основе информации о типе уже открывает широкие возможности, но здесь возникает интересный вопрос: действительно ли мы ограничены единственным экземпляром заданного типа? Представьте, что в примере с текущим пользователем из предыдущего раздела потребовалось внедрить другой экземпляр, представляющий пользователя, обслуживаемого в настоящий момент сотрудником отдела поддержки. Контейнер CDI имеет решение этой проблемы, которое называется *квалификатором*. Квалификатор дает возможность квалифицировать определенные точки внедрения, как использующие разные экземпляры компонента. Таким способом можно пометить объект `User`, как *текущий пользователь* или как *пользователь, обслуживаемый сотрудником отдела поддержки*. Квалификатор задается с использованием собственной аннотации.

Чтобы определить квалификатор, нужно определить новую аннотацию и снабдить это определение аннотацией `@Qualifier`. Причиной, почему вместо строковых имен для квалификации предлагается определять и использовать новые аннотации, является стремление сохранить строгую типизацию. Это одна из отличительных особенностей CDI, которая уменьшает вероятность появления ошибок во время выполнения. После определения, новые аннотации следует помещать перед точками внедрения и перед фабричными методами.

Так, в примере с внедрением текущего аутентифицированного пользователя можно было бы определить квалификатор такого пользователя. Это улучшило бы читаемость кода и позволило бы внедрять другие экземпляры. Следующий фрагмент определяет квалификаторы `@AuthenticatedUser` и `@Seller`:

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

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

После определения квалификатора его можно добавить к фабричному методу:

```
@Produces @SessionScoped @AuthenticatedUser @Named("currentUser")
public User getCurrentUser() {
    ...
}
```

После этого можно сообщить контейнеру CDI, какой экземпляр следует внедрить, как показано в листинге 12.8.

Листинг 12.8. Управление выбором экземпляра для внедрения с помощью квалификатора

```
@Named
@RequestScoped
public class BidController implements Serializable {

    @Inject @Seller       // ❶ Внедрить продавца, к объявлению
    private User seller;  //    которого делается ставка

    @Inject @AuthenticatedUser // ❷ Внедрить аутентифицированного
    private User user;         //    пользователя делающего ставку

    public String placeOrder() {
        ...
    }
}
```

В этом листинге можно видеть применение квалификатора. Здесь осуществляется внедрение двух экземпляров `User`. Первый экземпляр ❶ представляет продавца, к объявлению которого вы делаете ставку. Второй экземпляр – аутентифицированный пользователь, делающий ставку ❷. В обоих случаях предоставляется дополнительная информация, чтобы контейнер CDI смог определить, какой именно экземпляр следует внедрять.

В примере демонстрируется применение единственного квалификатора, но это не означает, что нельзя применить сразу несколько квалификаторов. Дополнительные квалификаторы позволяют уточнить выбор при необходимости. Теперь, когда вы познакомились с квалификаторами, перейдем к исследованию методов уничтожения.

12.3.4. Методы уничтожения

Метод уничтожения (disposer method) выполняет операции, связанные с удалением компонента. Хотя концептуально эти методы напоминают деструкторы в C++ или методы-финализаторы в Java, все же они имеют свои уникальные черты. Во-первых, метод уничтожения находится в том же классе, что и фабричный метод, а не в классе, экземпляры которого он должен уничтожать. Во-вторых, он вызывается, когда прекращается действие контекста и контейнер освобождает компонент. Например, если компонент принадлежит контексту сеанса, метод уничтожения не будет вызван, пока не истечет время действия сеанса.

Метод уничтожения отличается от обычного метода наличием аннотации `@Disposes` перед параметром. При этом могут использоваться дополнительные квалификаторы. Методу могут передаваться также другие параметры; контейнер CDI будет пытаться интерпретировать их как компоненты, используя квалификаторы, если они указаны. Метод уничтожения может закрыть соединение с базой данных и выполнить другие заключительные операции. В листинге 12.9 наглядно показано, насколько проста в использовании аннотация `@Disposes`.

Листинг 12.9. Метод уничтожения освобождает экземпляр объекта, представляющего аутентифицированного пользователя

```
@Named
@SessionScoped
public class CurrentUserBean implements Serializable {
    ...
    // ❶ Фабричный метод должен находиться
    //    в одном классе с методом уничтожения
    @Produces @SessionScoped @AuthenticatedUser @Named("currentUser")
    public User getCurrentUser() {
        ...
    }

    public void logout(
            // ❷ Метод уничтожения должен находиться
            //    в одном классе с фабричным методом
            @Disposes
            @AuthenticatedUser
            User user) {
        // Заключительные операции и выход
    }
}
```

Этот листинг демонстрирует пример объявления метода уничтожения. Как уже говорилось, метод уничтожения должен объявляться в одном классе с фабричным методом, но, в отличие от последнего ❶, он не должен принимать дополнительные параметры. Метод уничтожения ❷ осуществляет уничтожение объекта `User`, когда истечет предельное время нахождения сеанса в неактивном состоянии. Внутри этого метода можно выполнять любые заключительные операции.

Вы уже видели, как совместно использовать квалификаторы и фабричные методы для создания разных экземпляров объектов для внедрения. Но иногда бывает желательно подменить внедряемый экземпляр. Об этой возможности рассказывается далее.

12.3.5. Определение альтернатив

Приложения часто предусматривают несколько сценариев развертывания. Различия могут быть обусловлены использованием разных баз данных, например MySQL или Oracle, или разных систем управления информационным наполнением, например Alfresco или Documentum. В каждом случае требуется развертывать разный код. До сих пор, рассматривая внедрение зависимостей, мы подразумевали, что выбор внедряемого компонента производится на этапе разработки. Но, при наличии разных сценариев развертывания, необходим некоторый механизм настройки альтернатив для точек внедрения. Очевидно, что этот механизм должен использовать конфигурационные файлы и не требовать изменений в программном коде.

Для решения этой проблемы в CDI существует понятие *альтернатив*. Альтернатива – это дополнительная реализация компонента, в той же иерархии наследования, которая может использоваться взамен основной. Чтобы пометить компонент

как альтернативу, нужно добавить аннотацию `@Alternative` к объявлению класса. Чтобы задействовать альтернативу, нужно добавить ее в файл *beans.xml*. В зависимости от целевого окружения, можно использовать разные файлы *beans.xml* или динамически генерировать файл с нужным содержимым.

Приложение ActionBazaar предусматривает несколько разных сценариев развертывания, включая развертывание в эксплуатационной среде, в окружении для разработки и в окружении для тестирования (QA). Когда приложение ActionBazaar развертывается в контейнере, создается компонент, который автоматически запускается и наполняет систему некоторыми начальными данными. При развертывании в окружении для разработки выполняется воссоздание базы данных и ее наполнение тестовыми данными с начальным набором учетных записей пользователей. В окружении для тестирования имеющаяся база данных обновляется до текущей версии и в нее выгружается тестовое множество учетных записей пользователей, если они еще отсутствуют, чтобы лица, занимающиеся тестированием, могли выполнять свои тестовые сценарии. В эксплуатационном окружении компонент начальной загрузки не делает ничего – в этом случае не требуется вносить какие-либо изменения в фактические данные или в учетные записи, а обновлением базы данных займется администратор.

Иерархия классов, реализующих начальные операции, выполняемые на запуске, показана на рис. 12.3. Здесь `SystemStartup` – это базовый класс, используемый для инициализации приложения в эксплуатационном окружении. Два других подкласса, оба снабженные аннотацией `@Alternative`, используются в окружении для разработки и тестирования, соответственно.

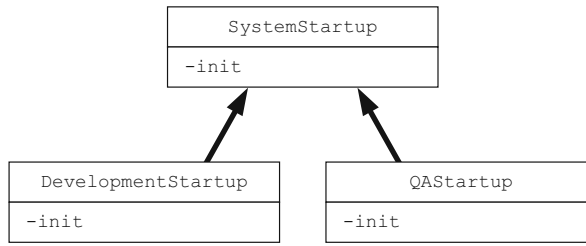

Рис. 12.3. Иерархия классов инициализации

В листинге 12.10 осуществляется внедрение экземпляра компонента `SystemStartup` и его вызов из метода, отмеченного аннотацией `@PostConstruct`. В эксплуатационном окружении будет выполнено внедрение реализации по умолчанию `SystemStartup`.

Листинг 12.10. Класс инициализации приложения

```
// Инициализацию выполняет компонент-одиночка, создаваемый
// во время запуска приложения
@Singleton
@Startup
```

```
public class Bootstrap {

    // Внедрить экземпляр SystemStartup
    @Inject
    private SystemStartup systemStartup;

    @PostConstruct
    public void postConstruct() {
       // Экземпляр SystemStartup вызывается в postConstruct
       systemStartup.init();
    }
}
```

В листинге 12.11 демонстрируется альтернативная реализация `SystemStartup` для использования во время разработки.

Листинг 12.11. Альтернативный класс SystemStartup, имеющий иную реализацию

```
// Класс отмечен как альтернативная реализация
@Alternative
// Наследует базовый класс SystemStartup
public class DevelopmentStartup extends SystemStartup {

    @Inject
    private UserService userService;

    @Inject
    private ItemManager itemManager;

    @Override
    public void init() {
        // подготовка тестовых данных и учетных записей
    }
}
```

Здесь класс отмечен аннотацией `@Alternative` и наследует базовый класс `SystemStartup`. Класс `DevelopmentStartup` используется, только если это разрешено. Чтобы разрешить использование альтернативы, класс должен быть указан в файле *beans.xml*. Как уже упоминалось, нужно либо включать в пакет разные файлы *beans.xml*, либо динамически генерировать нужное содержимое для файла во время сборки. Чтобы задействовать класс `DevelopmentStartup`, нужно добавить следующий элемент в файл *beans.xml*:

```
<alternatives>
    <class>com.actionbazaar.setup.DevelopmentStartup</class>
</alternatives>
```

Теперь у вас должно сложиться достаточно полное представление об особенностях внедрения и жизненном цикле компонентов и пришло время обратить внимание на интерцепторы и декораторы. А потом мы посмотрим, как использовать стереотипы, чтобы избежать взрывоподобного увеличения числа применяемых аннотаций.

12.4. Интерцепторы и декораторы

Ранее в этой книге мы уже исследовали поддержку интерцепторов в EJB 3, однако она ограничивается только компонентами EJB. Поддержку интерцепторов в CDI можно применять к любым компонентам, включая и компоненты EJB. Для управления реализацией интерцепторов CDI используются аннотации, как и в EJB 3. Несмотря на то, что поддержка в CDI не имеет таких же широких возможностей, как AspectJ, она все же намного проще в использовании.

Поддержка интерцепторов в CDI имеет две разновидности: традиционные интерцепторы и декораторы. Традиционные интерцепторы можно применять к любым методам любых компонентов. Интерцепторы не предъявляют никаких специфических требований к к компонентам или методам. Традиционный интерцептор – это именно то, что подразумевается под термином *интерцептор*. *Декораторы* – это новый тип интерцепторов. Декоратор реализует интерфейс перехватываемого класса, вследствие чего они оказываются тесно связанными с реализацией прикладных методов.

Декораторы используются с целью реализации прикладной логики для интерфейса, абстрагирующего сквозную функциональность. Традиционные интерцепторы, с другой стороны, реализуют универсальную логику сквозной функциональности. На первый взгляд такое определение может показаться странным, но по мере чтения следующих разделов и кода примеров различия будут становиться все более очевидными. Итак, начнем с традиционных интерцепторов.

12.4.1. Привязка интерцепторов

Создание интерцептора в CDI сопряжено с решением четырех разных задач программирования. И хотя их решения не выглядят так же элегантно, как определения точек сопряжения в AspectJ, они все же обладают высокой степенью предсказуемости во время выполнения. Вот эти задачи:

1. Создать аннотацию интерцептора.
2. Создать реализацию интерцептора, отмеченную аннотацией интерцептора.
3. Отметить целевые экземпляры аннотацией интерцептора.
4. Включить интерцептор в файл *beans.xml*.

Существует три вида интерцепторов, применяемых к разным типам методов, которые перечислены в табл. 12.2. В этой главе мы будем рассматривать только интерцепторы для прикладных методов, поскольку остальные два вида интерцепторов не имеют существенных отличий.

Таблица 12.2. Виды интерцепторов

Вид интерцептора	Аннотация	Описание
Для прикладных методов	`@AroundInvoke`	Перехватывают вызовы прикладных методов.

Таблица 12.2. (окончание)

Вид интерцептора	Аннотация	Описание
Для обработчиков событий жизненного цикла	`@PostConstruct/@PreDestroy`	Перехватывают вызовы обработчиков событий жизненного цикла.
Для методов, вызываемых таймерами EJB	`@AroundTimeout`	Перехватывают вызовы методов, осуществляемые таймерами EJB.

Чтобы определить интерцептор, необходимо сначала определить аннотацию интерцептора. Аннотация интерцептора служит для того, чтобы связать перехватываемый класс или метод с реализацией интерцептора. Поэтому, аннотация помещается и перед реализацией интерцептора, и перед классом или методом, обращение к которому должно перехватываться. По умолчанию интерцептор неактивен; его необходимо включить в файле *beans.xml*. Чтобы лучше понять суть, рассмотрим пример из приложения ActionBazaar.

Во многих приложениях производительность является важным аспектом. Поэтому в ходе тестирования бывает желательно собрать информацию о производительности ключевых интерфейсов, чтобы можно было составить представление о быстродействии приложения. Как отмечалось в предыдущем разделе, для тестирования будет использоваться отдельный файл *beans.xml*. Интерцептор производительности будет измерять время выполнения и сохранять его в файл на жестком диске сервера. Здесь мы сосредоточимся на применении интерцептора к классу `LandingController`. Класс `LandingController` реализует главную страницу приложения ActionBazaar. Если эта страница будет работать слишком медленно, пользователи могут отказаться от посещения других разделов сайта.

Ниже показано определение аннотации `PerformanceMonitor`:

```
@InterceptorBinding
@Retention(RUNTIME)
@Target({TYPE})
public @interface PerformanceMonitor {}
```

Эта аннотация сама отмечена аннотацией `@InterceptorBinding`. Когда контейнер CDI встретит эту аннотацию, он поймет, что вы просите его обернуть метод или методы интерцептором и создаст экземпляр интерцептора. После определения аннотации интерцептора необходимо определить его реализацию, как показано в листинге 12.12.

Листинг 12.12. Реализация интерцептора измерения производительности

```
// ❶ Связать @PerformanceMonitor с PerformanceInterceptor
@PerformanceMonitor
@Interceptor // ❷ Отметить класс как интерцептор
public class PerformanceInterceptor {

    // Отметить метод, который будет перехватывать вызовы других методов
```

```
    @AroundInvoke
    // InvocationContext хранит контекстную информацию
    public Object monitor(InvocationContext ctx) throws Exception {
        long start = new Date().getTime();
        try {
            // Вызвать метод proceed для вызова фактического метода
            return ctx.proceed();
        } finally {
            long elapsed = new Date().getTime() - start;
            Logger.getLogger("PerformanceInterceptor").log(Level.INFO,
                "Elapsed time: {0}", elapsed);
        }
    }
}
```

В этом листинге определена реализация интерцептора. Интерцептор необходимо снабдить аннотацией ❶, чтобы связать их друг с другом. Реализация должна быть также отмечена аннотацией `@Interceptor`, чтобы отметить этот класс как интерцептор ❷. Аннотация `@AroundInvoke` отмечает метод, который фактически осуществляет перехват. Посредством `InvocationContext` внутри этого метода можно получить информацию о перехваченном методе, а также его параметрах. С помощью метода `proceed` осуществляется вызов перехваченного метода, впрочем, при желании перехваченный метод можно и не вызывать.

В табл. 12.3 перечислены некоторые методы класса `InvocationContext`. Этот класс позволяет получить полезные сведения о перехваченном методе, о его параметрах и другую контекстную информацию.

Таблица 12.3. Методы InvocationContext

Метод	Описание
`Object getTarget();`	Возвращает экземпляр объекта, для которого осуществляется перехват.
`Object getTimer();`	Возвращает объект таймера, который вызывает перехватываемый метод.
`Method getMethod();`	Возвращает перехватываемый метод.
`Object[] getParameters();`	Возвращает параметры перехватываемого метода.
`void setParameters(Object[] os);`	Изменяет параметры перехватываемого метода.
`Map<String, Object> getContextData();`	Возвращает контекстные данные, связанные с этим вызовом – информация из аннотации.
`Object proceed() throws Exception;`	Вызывает перехваченный метод.

После определения интерцептора его можно использовать. Аннотацией интерцептора можно отмечать целые классы или отдельные методы. Ниже показано, как применить интерцептор к классу `LandingController`:

```
@Named
@ApplicationScoped
@PerformanceMonitor
```

```
public class LandingController {
    ...
}
```

Простое добавление аннотации к классу `LandingController` не включает перехват по умолчанию. Интерцептор нужно включить в файле *beans.xml*. Следующий фрагмент включает интерцептор:

```
<interceptors>
    <class>com.actionbazaar.util.PerformanceInterceptor</class>
</interceptors>
```

После включения интерцептора, он будет отслеживать все вызовы всех методов класса `LandingController`. При необходимости можно отметить аннотацией интерцептора и другие классы. При одновременном использовании нескольких интерцепторов, они будут выполняться в том же порядке, в каком определяются в файле *beans.xml*. Дополнительные интерцепторы включаются добавлением новых элементов `<class></class>`.

Теперь, после знакомства с традиционными интерцепторами, можно переходить к исследованию декораторов CDI. В отличие от интерцепторов декораторы тесно связаны с определенным интерфейсом.

12.4.2. Декораторы

Декораторы очень похожи на интерцепторы. Декораторы точно так же осуществляют перехват вызовов методов компонентов. Но, в отличие от интерцепторов, декораторы переопределяют перехватываемые методы. Декоратор наследует или реализует интерфейс компонента. Как результат, декоратор оказывается тесно связанным с компонентом и может реализовать прикладную логику. Поскольку фактически выполняется переопределение методов, вы получаете простой доступ к их параметрам.

Чтобы лучше понять назначение декораторов, рассмотрим процедуру торга в приложении ActionBazaar. Одна из больших проблем подобных сайтов – мошенничество. Перебивая цену, предложенную другим пользователем, любой потенциальный покупатель хотел бы быть уверенным, что он не состязается с продавцом, создавшим поддельную учетную запись специально, чтобы устроить гонку предложений на свой товар. Для защиты от такого рода мошенничества необходим хотя бы самый простенький механизм определения состояния гонки. Однако, хотя определение состояния гонки и является частью процесса торгов, оно не является частью логики обработки ставок. Это – сквозная функция, подобная функции журналирования, только более узкоспециализированная. Следовательно, ее можно реализовать с помощью декоратора, как показано в листинге 12.13.

Листинг 12.13. Декоратор, реализующий BidManager и перехватывающий вызовы методов

```
@Decorator // ❶ Отметить класс как декоратор
// ❷ Декоратор - абстрактный
```

```
public abstract class BidManagerFraudDetector implements BidManager {

    @Inject               // ❸ Внедрить обертываемый экземпляр bidManager
    @Delegate             // ❹ Отметить экземпляр как делегат
    @BidManagerQualifier // ❺ Квалификатор экземпляров bidManager
    private BidManager bidManager;
    @Override
    // ❻ Реализация перехватываемого метода
    public void placeBid(Bid bid) {
        ...
    }
}
```

В этом листинге определяется декоратор. Чтобы создать декоратор, нужно отметить класс аннотацией `@Decorator` ❶ и реализовать или унаследовать декорируемый класс ❷. Затем следует внедрить экземпляр ❸ декорируемого класса и отметить его, как экземпляр делегата ❹. Чтобы гарантировать выбор нужного экземпляра здесь используется квалификатор ❺. Наконец, переопределяется перехватываемый метод ❻. Поскольку в данном примере нас интересует только метод `placeBid`, мы пометили класс как абстрактный.

Декораторы, как и интерцепторы, по умолчанию выключены. Чтобы активизировать предыдущий декоратор, нужно добавить следующий фрагмент в файл *beans.xml*:

```
<decorators>
    <class>com.actionbazaar.buslogic.BidManagerFraudDetector</class>
</decorators>
```

Так же, как в случае с интерцепторами к одному классу можно применить несколько декораторов. Порядок следования декораторов в файле *beans.xml* определяет порядок их выполнения. Теперь, после знакомства с интерцепторами и декораторами, можно посмотреть, как уменьшить число аннотаций в коде.

12.5. Стереотипы

По мере знакомства с новыми особенностями CDI мы накопили большое число аннотаций. В некоторых случаях последовательность аннотаций перед методами оказывается длиннее сигнатуры метода. Если взглянуть внимательнее на все эти методы, можно заметить, что часто к ним применяются одни и те же аннотации. Такой избыточный код усложняет его сопровождение. К счастью, разработчики CDI предвидели эту проблему и добавили конструкцию, получившую название *стереотип* (stereotype).

Вас не должно удивлять, что стереотип – это еще одна аннотация. Но в отличие от аннотаций, встречавшихся нам до сих пор, стереотип является объединением нескольких аннотаций. Стереотип создается объявлением новой аннотации и добавлением в нее других аннотаций. Давайте начнем с того, что рассмотрим комплекс аннотаций, применяемых к методу `getCurrentUser()` в классе

CurrentUserBean. Сигнатура метода выглядит короче, чем список аннотаций, применяемых к нему:

```
@Named @SessionScoped
public class CurrentUserBean implements Serializable {
    @Produces @SessionScoped @AuthenticatedUser @Named
    public User getCurrentUser() {
        ...
    }
}
```

С помощью стереотипа, определение которого приводится ниже, можно существенно уменьшить число аннотаций, применяемых к методу:

```
@SessionScoped
@AuthenticatedUser
@Named
@Stereotype
@Target( { TYPE, METHOD, FIELD })
@Retention(RUNTIME)
public @interface CurrentUser {}
```

В новой аннотации @CurrentUser мы объединили аннотации @SessionScoped, @AuthenticatedUser и @Named. В следующем фрагменте показано, насколько эта аннотация упрощает определение класса CurrentUserBean:

```
@Named @SessionScoped
public class CurrentUserBean implements Serializable {
    @Produces @CurrentUser
    public User getCurrentUser() {
        ...
    }
}
```

Существует несколько основных правил определения стереотипов. Стереотипы могут состоять из аннотаций CDI, определяющих:

- область видимости (контекст);
- привязки интерцепторов;
- имена для интеграции с JSF;
- альтернативы.

В стереотипы нельзя включать аннотации JPA или EJB. Также нельзя включать аннотацию @Produces и другие аннотации, определяющие обработчики событий жизненного цикла.

В CDI имеется встроенный стереотип @Model для создания компонентов, играющих роль модели в шаблоне «модель-представление-контроллер» (model-view-controller). Определение аннотации @Model приводится ниже:

```
@Named
@RequestScoped
@Documented
@Stereotype
```

```
@Target(value = {ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
@Retention(value = RetentionPolicy.RUNTIME)
public @interface Model {}
```

С ее помощью можно заменить аннотации `@Named` и `@RequestScoped`, как показано в следующем фрагменте:

```
@Model
public class ItemController implements Serializable {
    ...
}
```

Стереотипы значительно упрощают чтение определений компонентов CDI. В следующем разделе мы рассмотрим поддержку событий в CDI.

12.6. Внедрение событий

Внедрение зависимостей – чрезвычайно удобный способ получения ссылок на другие компоненты. Однако механизм внедрения зависимостей в CDI поддерживает также *внедрение событий*. Это позволяет организовать доставку событий другим компонентам без образования зависимостей на этапе компиляции. Гениальность такого подхода заключается в простом использовании аннотаций для определения методов, которые будут принимать события. Чтобы обеспечить доставку события компоненту, вам не придется прибегать к дополнительным ухищрениям. Вам не нужно реализовать интерфейс или регистрировать класс как приемник событий – всю «грязную» работу за вас выполнит контейнер. Возбудить событие так же просто; объект, отвечающий за отправку событий, просто внедряется в класс, нуждающийся в этой функции.

Аннотация `@Observes` помещается перед одним или несколькими параметрами метода, принимающего событие. Тип параметра определяется типом события. Например, если параметр имеет тип `ActionEvent`, тогда метод будет получать все события `ActionEvent`. С помощью дополнительных квалификаторов перед точкой внедрения можно обеспечить более точную классификацию событий. Дополнительные параметры метода интерпретируются контейнером как компоненты, которые нужно найти или создать.

Отправка события компонентом осуществляется с использованием объекта `Event`, параметризованного типом объекта, который нужно отправить. Он внедряется с помощью аннотации `@Inject` и может иметь дополнительные аннотации-квалификаторы, сужающие круг потенциальных целей события. Фактическая отправка осуществляется вызовом метода `fire` объекта `Event`.

Чтобы лучше понять, как действует механизм событий в CDI, обратимся к примеру из приложения ActionBazaar. В этом приложении, когда добавляется новое объявление, необходимо известить множество компонентов. Состав компонентов, реагирующих на это событие, может изменяться с течением времени, к тому же в приложение могут добавляться новые модули с новыми компонентами, и едва ли кто-то будет гореть желанием постоянно изменять `ItemController`. Поэто-

му ItemController возбуждает событие, которое получают другие компоненты, включая LandingController и TwitterNotifier. Применение разновидности шаблона проектирования «Наблюдатель» (Observer), реализованной в CDI, позволяет писать слабо связанный код – ItemController не имеет ссылок на другие компоненты, а другим компонентам, ожидающим указанное событие, не требуется получать ссылку на ItemController. На рис. 12.4 изображен процесс передачи события.

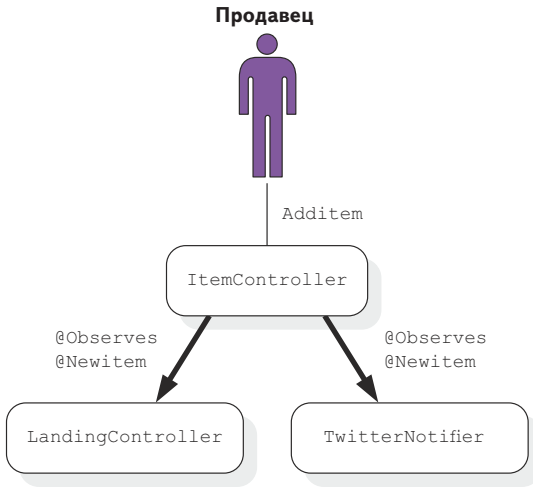

Рис. 12.4. Передача события, возбуждаемого методом AddItem

Начнем с метода addItem в классе ItemController. В листинге 12.14 приводится код, имеющий отношение к отправке события.

Листинг 12.14. ItemController и возбуждение событий

```
@Model
public class ItemController implements Serializable {

    @Inject   // ❶ Внедрить объект события для отправки
    @NewItem  // ❷ Дополнительный квалификатор объекта события
    // Внедряемый объект события параметризованный типом объявления
    private Event<Item> itemNotifier;

    public String add() {
        // Сначала сохранить объявление, потом отправить с событием
        itemNotifier.fire(item); // ❸ Отправить событие всем приемникам
        ...
        return NavigationRules.HOME.getRule();
    }
}
```

В этом листинге компонент ItemController отправляет события, извещающие о добавлении нового объявления. Сначала в компонент внедряется объект события ❶, который дополнительно квалифицируется аннотацией @NewItem ❷, чтобы

ограничить круг целевых компонентов, заинтересованных именно в событии добавления нового объявления. Сам объект события использует механизм обобщенных типов Java Generics, чтобы гарантировать передачу в вызов `fire` объекта нужного типа. Возбуждение события осуществляется достаточно просто – простым вызовом метода `fire` объекта события и передачей ему нового объявления `item` ❸. Реализация обработки события показана в листинге 12.15.

Листинг 12.15. Изменения в LandingController, реализующие кэширование по событию `NewItem`

```
@Named
@ApplicationScoped
@PerformanceMonitor
public class LandingController {

    public void onNewItem(@Observes // ❶ Пометить метод как приемник события
                          @NewItem Item item) {// ❷ Обрабатывает только
                                               // события добавления нового
                                               // объявления
        newestItems.add(0,item);
    }
}
```

Как показано в этом листинге, контейнер CDI выполняет регистрацию события автоматически. Вам не нужно предусматривать реализацию интерфейса или получать ссылку на `ItemController`, чтобы зарегистрировать компонент `LandingController` как приемник события. Метод `onNewItem` интерпретируется как приемник события, потому что его параметр `item` отмечен аннотацией `@Observes` ❶. Кроме того, благодаря квалификатору `@NewItem` ❷, метод будет вызываться только по событиям добавления нового объявления.

Итак, мы познакомились со всеми основными особенностями CDI. Теперь, опираясь на полученные знания, можно повторно вернуться к теме организации диалогов, чтобы лучше понять, как они действуют и как ими пользоваться.

12.7. Диалоги

Выше в этой главе мы уже касались темы диалогов. CDI поддерживает понятие области видимости (или контекста), с которым тесно связаны жизненные циклы компонентов. Если компонент ограничен контекстом запроса, он существует только в период обработки запроса – к нему нельзя будет обратиться во время обработки последующих запросов. Компонент, ограниченный контекстом диалога, существует только в течение диалога, протяженность которого короче протяженности сеанса, но длиннее протяженности обработки одного запроса. То есть, контекст диалога распространяется на несколько запросов и представляет некоторую единицу работы.

Диалоги являются важным инструментом организации взаимодействий пользователей с веб-браузерами. В наши дни стало обычным делом, когда пользова-

тель открывает несколько вкладок в браузере. Вкладки могут ссылаться на одно и то же веб-приложение. Соответственно, вкладки будут использовать один и тот же контекст сеанса. Например, пользователь может в одной вкладке с сайтом турагентства планировать отпуск, а в другой – поездку в деловую командировку. Для веб-приложений критически важно изолировать вкладки, чтобы в результате щелчка на кнопке **Submit** (Отправить) во вкладке, где планируется деловая командировка, не произошло оформление заказа на поездку в отпуск, или, хуже того, не произошел заказ билетов на самолет для деловой командировки и бронирование отеля для поездки в отпуск. Такое возможно, когда в веб-приложении сохраняется только информация для последней открытой вкладки или когда смешиваются данные из разных вкладок. Решить эту проблему можно с помощью абстракции поверх сеанса, позволяющей отслеживать идентификаторы для каждой открытой вкладки. Фактически именно это решение предлагает CDI, но за счет управления жизненным циклом компонентов, позволяя вам сосредоточиться исключительно на прикладной логике.

Чтобы указать, что компонент имеет область видимости диалога, его нужно отметить аннотацией `@ConversationScoped`. Но по умолчанию контекст диалога распространяется только на один запрос, если явно не *продлить* его. Диалог, который не был продлен, называют *переходным* диалогом (transient conversation). Явное продление диалога необходимо, чтобы контейнер знал, когда начинается продолжительный диалог, а когда обрабатывается короткая реплика. Контейнеру нужно так же знать, когда заканчивается диалог. Явный запуск и явное завершение диалога используются для управления ресурсами. Чтобы запустить или остановить диалог, необходимо обратиться к объекту `javax.enterprise.context.Conversation`. Получить ссылку на него можно путем внедрения. Методы этого интерфейса перечислены в табл. 12.4.

Таблица 12.4. Методы объекта Conversation

Метод	Описание
`void begin()`	Преобразует текущий переходный диалог в продолжительный диалог.
`void begin(String id)`	Преобразует текущий переходный диалог в продолжительный диалог с указанным идентификатором `id`.
`void end()`	Завершает диалог и преобразует его в переходный диалог.
`String getId()`	Возвращает идентификатор текущего диалога.
`long getTimeout()`	Возвращает таймаут текущего диалога в миллисекундах.
`boolean isTransient()`	Возвращает `true`, если если текущий диалог является переходным и будет завершен после обработки запроса.
`void setTimeout(long timeout)`	Устанавливает указанный таймаут для текущего диалога; эта операция выполняется, только если диалог не является переходным.

Чтобы лучше понять суть механизма диалогов, рассмотрим относительно простой случай его применения в приложении ActionBazaar. Этот пример демонстрирует большую часть особенностей CDI, с которыми мы уже познакомились в этой главе, включая фабричные методы и квалификаторы. Мы рассмотрим путь, который проходит пользователь от просмотра списка объявлений до фактического размещения ставки, как показано на рис. 12.5.

Пользователь начинает работу с приложением, открывая начальную страницу со списком объявлений – в данном случае объявлений о продаже парусных лодок. На этой странице выводятся фотографии лодок и краткие описания. Пользователь щелкает на объявлении, заинтересовавшем его, в результате чего открывается страница с более подробным описанием и с полем, где можно ввести предлагаемую цену. Открывая страницу с подробной информацией, пользователь может щелкнуть на ссылке правой кнопкой мыши и выбрать пункт контекстного меню **Open Link in New Tab** (Открыть ссылку в новой вкладке), что обычно делается с целью сравнить несколько объявлений. Если пользователь делает ставку, он переносится на страницу подтверждения, где должен подтвердить свое намерение. После того как пользователь подтвердит свое желание, он переносится обратно на главную страницу с коротким сообщением, подтверждающим, что его ставка принята.

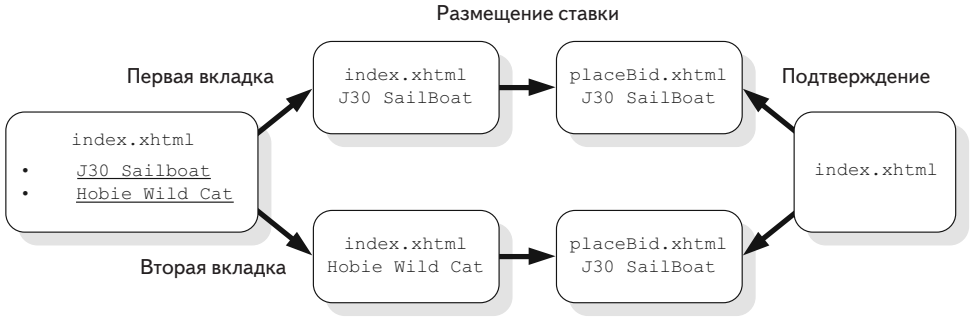

Рис. 12.5. Порядок размещения ставки в приложении ActionBazaar

Для целей этого примера, предположим, что пользователь открывает два объявления в двух дополнительных вкладках. Всего получается три открытые вкладки: главная страница (*index.xhtml*), страница с объявлением «J30 Sailboat» (*item.xhtml*) и страница с объявлением «Hobie Wild Cat» (*item.xhtml*). В двух вкладках отображается информация о двух выбранных лодках и все действия, выполняемые в этих вкладках, должны относиться к соответствующим объявлениям. Таким образом каждая вкладка представляет отдельный диалог.

В листинге 12.16 приводится упрощенная версия главной страницы, отображающей список новых объявлений. Покупатель может щелкнуть на одном из этих объявлений, чтобы перейти на страницу с подробным описанием. Ссылка – это команда JSF, которая генерирует запрос `POST`, для обработки которого вызывается метод `startConversation` компонента `ItemController`.

Листинг 12.16. Главная страница со списком объявлений

```
<!DOCTYPE html>
<HTML
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Welcome to Action Bazaar</title>
  </h:head>
  <h:body>
    <h:form id="itemForm">
      <!-- Таблица для отображения списка объявлений -->
      <h:dataTable value="#{landingController.newestItems}" var="itr">
        <h:column>
          <!-- Команда для открытия страницы с объявлением -->
          <h:commandLink
            <!-- Метод, инициирующий диалог -->
            action="#{itemController.startConversation(itr)}">
              #{itr.itemName}
          </h:commandLink>
        </h:column>
      </h:dataTable>
      <h:button outcome="addItem" value="Add Item"/>
    </h:form>
  </h:body>
</HTML>
```

В листинге 12.17 показано, как запускается продолжительный диалог.

Листинг 12.17. ItemController – начало диалога

```
@Model
// ❶ Компонент ItemController имеет область видимости запроса
public class ItemController implements Serializable {

    @Inject
    private Conversation conversation; // ❷ Внедрить объект Conversation

    private BigDecimal bidAmount;

    private Item item;                 // ❸ Выбранное объяаление кэшируется
                                       //    для фабричного метода
    // ❹ startConversation вызывается щелчком на ссылке
    public String startConversation(Item item) {
        conversation.begin(); // ❺ Запуск диалога
        this.item = item;     // ❻ Кэширование выбранного объявления
        return NavigationRules.ITEM.getRule();
    }

    // ❼ Фабричный метод, возвращающий выбранное объявление
    @Produces @SelectedItem @Named("selectedItem") @ConversationScoped
    public Item getCurrentItem() {
        if(item == null) {
            item = new Item();
```

```
        }
        return item;
    }
}
```

В этом листинге приводится код реализации класса `ItemController` и в частности – метод `startConversation` ❹, на который ссылается форма в листинге 12.16. Важно отметить следующее об этом классе:

- он доступен в выражениях JSF EL, как показывает листинг 12.16;
- в своей работе опирается на внедренный объект `Conversation`;
- «производит» объект текущего выбранного объявления, который затем хранится в кэше диалога.

Аннотация `@Model` ❶ указывает, что данный компонент доступен из страниц JSF и имеет область видимости запроса. Аннотация `@Model` – это стереотип, объединяющий аннотации `@RequestScoped` и `@Named`. Это весьма распространенная комбинация. Прежде чем метод `startConversation` будет вызван из страницы JSF, контейнер CDI внедрит объект диалога ❷. Чтобы запустить диалог, необходимо явно сообщить контейнеру об этом, что делается вызовом метода `startConversation` ❺. Одновременно в свойстве `item` ❸ сохраняется объявление, выбранное пользователем ❻. Теперь диалог запущен и выбранное объявление сохранено в кэше.

Последнее, что делает метод `startConversation`, – переход к странице с объявлением. Ответ формируется в контексте текущего запроса, поэтому кэшированное объявление все еще доступно. Страница, в которой отображается выбранное объявление, показана в листинге 12.18. При отображении этой страницы, объект `selectedItem` будет извлечен методом `getCurrentItem` ❼. Это – фабричный метод, возвращающий выбранное объявление, кэшированное в контексте диалога и, соответственно, продолжающее существовать до завершения этого диалога. Диалог может быть завершен программно или спустя установленный период отсутствия активности.

Листинг 12.18. Страница с объявлением – для просмотра и предложения цены

```
<!DOCTYPE html>
<HTML
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html">
  <h:body>
    <h:form>
      <!-- ❶ Вывод названия товара из выбранного объявления -->
      Item Name: #{selectedItem.itemName}<br/>
      <!-- ❷ Сумма ставки -->
      Bid: <h:inputText value="#{currentBid.bidPrice}" size="5"/><br/>
      <h:commandButton value="Place Bid"
        <!-- ❸ Команда размещения ставки -->
        action="#{bidController.placeOrder()}"/>
    </h:form>
  </h:body>
</HTML>
```

Эта страница отображает название товара из выбранного объявления ❶. Для этого производится вызов фабричного метода компонента `ItemController`, представленного в листинге 12.17. Предлагаемая цена вводится в поле ввода ❷. Щелчок на кнопке **Place Bid** (Сделать ставку) ❸ приводит к вызову метода `placeOrder` и переходу к странице *placeBid.xhtml* с информацией о ставке и кнопкой подтверждения. Реализация метода `placeOrder` компонента `BidController` приводится в листинге 12.20.

Листинг 12.19. placeBid.xhtml – страница подтверждения ставки

```
<HTML
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html">
  <h:body>
    <h:form>
      <!-- ❶ Вывести объект объявления с областью видимости диалога -->
      Confirm Item: #{selectedItem.itemName}<br/>
      Amount: #{currentBid.bidPrice}<br/>
      <h:commandButton value="Confirm Bid"
        <!-- ❷ Подтверждает ставку -->
        action="#{bidController.confirmBid()}"/>
    </h:form>
  </h:body>
</HTML>
```

В листинге 12.19 представлен код страницы подтверждения ставки. Здесь снова отображается выбранное объявление ❶, хранящееся в контексте диалога. Щелчок на кнопке **Confirm Bid** (Подтвердить ставку) ❷ сохраняет ставку, вызывая метод `confirmBid` компонента `BidController`, как показано в листинге 12.20.

Листинг 12.20. BidController – сохранение ставки

```
@Model
public class BidController implements Serializable {

    @Inject @SelectedItem // ❶ Внедрить выбранное объявление из диалога
    private Item item;

    @Inject
    private Conversation conversation; // ❷ Внедрить контекст диалога

    public String confirmBid() {
        logger.log(Level.INFO, "Conversation ID: {0}",
        conversation.getId()); // ❸ Записать ID текущего контекста диалога
        bidManager.placeBid(currentBid);
        conversation.end();    // ❹ Завершить диалог
        return NavigationRules.HOME.getRule();
    }

    public String placeOrder() {
        // выполнить некоторые операции
        return NavigationRules.PLACE_BID.getRule();
```

```
    }
}
```

Выбранное объявление внедряется в этот компонент ❶. Также внедряется контекст диалога ❷. Метод `confirmBid` выводит идентификатор диалога для целей отладки ❸ и после успешной регистрации ставки завершает диалог ❹. После этого все объекты, хранящиеся в контексте диалога, будут уничтожены.

В этом примере из приложения ActionBazaar выбранное объявление сохраняется в контексте диалога фабричным методом. После этого оно используется в двух страницах: в странице с подробной информацией об объявлении, позволяющей пользователю прочитать описание и сделать ставку, и в странице подтверждения, где пользователь подтверждает свой выбор. Это лишь один пример использования контекста диалога – такой подход намного проще, чем пытаться разработать собственное решение.

Мы исследовали основные возможности технологии CDI. Пришло время завершить дискуссию некоторыми замечаниями по эффективному ее использованию в EJB 3.

12.8. Эффективное использование CDI в EJB 3

После знакомства с такими захватывающими возможностями CDI возникает законный вопрос: как эта технология укладывается в общую картину EJB 3 и как наиболее эффективно использовать CDI совместно с компонентами EJB? У кого-то даже может возникнуть вопрос: насколько вообще актуальна технология EJB 3 и возможно ли использовать компоненты CDI взамен компонентов EJB? В данном разделе мы попытаемся ответить на эти вопросы и дать некоторые рекомендации по проектированию приложений Java EE.

Как отмечалось ранее, технология CDI была добавлена в Java EE 6 и она продолжила развитие в Java EE 7. EJB и CDI – это взаимодополняющие технологии, усиливающие друг друга и упрощающие создание приложений Java EE. Технология CDI значительно упрощает использование компонентов EJB – она избавляет от необходимости непосредственно обращаться к каталогу JNDI или писать собственный слой абстракции доступа к JNDI. CDI добавляет в POJO поддержку внедрения зависимостей и контекстов, а также некоторые расширения. Компоненты CDI не могут служить заменой компонентам EJB – компоненты EJB незаменимы там, где требуется поддержка безопасности, транзакций, удаленных взаимодействий, планирования, асинхронных операций и блокировок. Но не забывайте, что компоненты EJB выполняются в контейнере CDI, поэтому вы можете решать любые возникающие проблемы, как если бы они были компонентами CDI, и при необходимости использовать службы, предоставляемые контейнером EJB.

Эффективные приемы совместного использования CDI и EJB были наглядно продемонстрированы в примерах в этой главе. Даже при том, что ActionBazaar – достаточно маленькое приложение и многие примеры кода были сжаты для эко-

номии места, они все же достаточно наглядно иллюстрируют некоторые эффективные приемы интеграции двух технологий. Компоненты CDI с успехом могут использоваться взамен компонентов JSF и поддерживать интерфейс JSF. Обратите внимание, что технология CDI никак не связана с технологией JSF – CDI совершенно нейтральна к выбору веб-фреймворка. Она позволяет изолировать прикладную логику в собственном слое, отдельно от реализации пользовательского интерфейса. Кроме того, эта технология прекрасно работает с веб-службами SOAP и RESTful – совершенно необязательно создавать компоненты EJB, чтобы обеспечить поддержку веб-службы SOAP или возврат данных в формате JSON из веб-службы RESTful. Приложения с многослойной архитектурой проще реализуются с применением технологии CDI, потому что она делает интеграцию слоев бесшовной.

Так как компоненты EJB одновременно являются компонентами CDI, старайтесь всегда использовать аннотацию `@Inject` вместо `@EJB`. Возможность конфигурирования, поддерживаемая контейнером CDI, значительно упрощает тестирование приложений. С помощью конфигурационного файла *beans.xml* можно подменять внедряемые экземпляры для нужд модульного и интеграционного тестирования. Для проверки устойчивости приложения к ошибкам и исключениям, возникающих при работе с базой данных, можно внедрить специально подготовленный фиктивный объект. Кроме того, для аннотации `@Inject` совершенно неважно, является ли объект компонентом EJB или простым POJO – благодаря этому в будущем вы легко сможете превратить POJO в компонент EJB.

Поддержка декораторов в CDI дает возможность реализовать сквозную функциональность, более мощную и узкоспециализированную, чем позволяют традиционные интерцепторы EJB. Декораторы помогают отдельно реализовать сквозную логику, тесно связанную с прикладной логикой. Хорошим примером может служить реализация механизма определения мошенничества в ActionBazaar. Даже при том, что он тесно связан с логикой добавления ставки, это все же самостоятельный механизм, отдельный от механизма регистрации ставок, и наверняка будет совершенствоваться с течением времени. Реализацию этого механизма желательно держать отдельно – если вы соберетесь реализовать новый алгоритм определения мошенничества, вам не придется копаться в логике обработки ставки. Изменение алгоритма не должно дестабилизировать работу кода обработки ставки и усовершенствование такой важной сквозной функции проще будет подвергнуть всестороннему регрессионному тестированию.

CDI предоставляет мощную поддержку событий, помогающую уменьшить связанность кода, как в компонентах JSF, так и в компонентах EJB. Используйте их для устранения зависимостей между компонентами, чтобы одни компоненты не хранили ссылки на другие. Тесно связанные системы чрезвычайно сложны в развитии, так как даже небольшое изменение может повлечь лавину сопутствующих изменений. Поддержка событий в CDI не требует от отправителя знать что-либо об объектах, которые будут принимать их, а от приемника – хранить ссылку на отправителя. Она значительно уменьшает запутанность кода. Однако события CDI не могут распространяться за пределы JVM. Это означает, что если приложение

выполняется одновременно на нескольких серверах с балансировкой нагрузки, вам придется проявлять осторожность. Пример использования событий для извещения `LandingController` о появлении нового объявления, представленный в этой главе, не будет работать в среде из нескольких серверов. Каждый сервер будет иметь свой список новых объявлений. В подобных ситуациях на роль механизма поддержки событий больше подходит JMS. Но события CDI можно использовать совместно с JMS. Список новых объявлений мог бы приниматься компонентом, управляемым сообщениями, и отправляться с помощью событий CDI всем компонентам внутри сервера.

Существует одна область, где компоненты CDI вытесняют компоненты EJB – сеансовые компоненты с сохранением состояния. Компоненты с областью видимости диалога намного удобнее и прекрасно подходят для большинства приложений. В прошлом сеансовые компоненты с сохранением состояния чаще всего использовались для реализации продолжительных прикладных процедур, которые мы называем диалогами. Диалоги – это только начало; CDI поддерживает возможность создания собственных областей видимости, что дает дополнительную гибкость и позволяет находить элегантные решения, более простые в сопровождении.

12.9. В заключение

В этой главе мы исследовали основные возможности CDI, узнали, как создавать компоненты CDI и увидели, как связаны между собой компоненты CDI, EJB и JSF. Мы охватили разные области видимости CDI, включая область видимости запроса, сеанса и диалога. Хотя эта тема не обсуждалась в данной главе, тем не менее, заметим, что CDI поддерживает возможность определения собственных областей видимости, которые могут пригодиться для реализации специфических бизнес-процессов. Мы подробно осветили одну из важнейших особенностей: внедрение зависимостей. Механизм внедрения зависимостей в CDI использует информацию о типах, вместо строковых идентификаторов. Внедрение зависимостей в CDI находит более широкое применение, чем аналогичный механизм в EJB – его, к примеру, можно применять к конструкторам. Мы рассмотрели фабричные методы с позиции взаимодействий между CDI и JPA. Исследовали интерцепторы и декораторы. Декораторы – это специализированные интерцепторы, связанные с определенными интерфейсами. Также мы познакомились с квалификаторами и их применением для различения экземпляров одного типа – например, нового и текущего объявлений. Квалификаторы CDI можно использовать, чтобы точно определить, какой экземпляр должен быть внедрен. Затем, опираясь на полученные знания, мы познакомились с событиями и рассмотрели сложный пример организации диалога.

В заключение хотелось бы подчеркнуть пару важных моментов. Технология CDI не является заменой для EJB и совершенно нейтральна к выбору веб-фреймворка. Несмотря на то, что компоненты CDI с успехом можно использовать взамен

компонентов JSF, сама технология CDI никак не связана с JSF. Аннотация `@Named` помогает присвоить строковые имена компонентам CDI для их использования другими технологиями, такими как JSF, поэтому для организации взаимодействий между фреймворком и CDI потребуется совсем немного промежуточного кода.

Часть IV

ВВОД EJB В ДЕЙСТВИЕ

В этой части вы познакомитесь с рекомендациями по вводу EJB 3 в действие в вашей компании. Сначала, в главе 13, мы рассмотрим порядок упаковки компонентов EJB и сущностей для развертывания на сервере. Затем познакомимся с системой модулей Java EE и особенностями загрузки классов в приложениях EE. В главе 14 будут представлены веб-сокеты и их взаимоотношения с компонентами EJB, а также возможность асинхронного выполнения прикладной логики с применением утилит EJB. В главе 15 вы познакомитесь с разными стратегиями тестирования, включая модульное и интеграционное тестирование, без необходимости развертывать тестируемое приложение на действующем сервере.

ГЛАВА 13. Упаковка приложений EJB 3

Эта глава охватывает следующие темы:

- система модулей Java EE;
- загрузка классов в приложениях EE;
- упаковка компонентов EJB и MDB;
- упаковка сущностей JPA;
- упаковка компонентов CDI.

В предыдущих главах вы узнали, как формировать слой прикладной логики с помощью сеансовых компонентов и компонентов MDB, и как использовать сущности для поддержки сохранения данных. Но, что делать дальше со всем этим кодом?

Эта глава начинается с обсуждения приемов упаковки и развертывания приложения, которые необходимо знать, чтобы подготовить ваше корпоративное приложение к работе. Мы посмотрим, как совмещаются разные модули (JAR, EJB-JAR, WAR, CDI и EAR) в приложениях EE и как они взаимодействуют при развертывании на сервере EE. Мы обсудим, как осуществляется настройка приложений EE с помощью аннотаций и файлов XML. Наконец, мы познакомимся с некоторыми приемами развертывания и общими проблемами, которые обычно возникают при развертывании одного и того же приложения на разных реализациях серверов EE. Начнем с обзора поддержки модулей EE и приемов их упаковки.

13.1. Упаковка приложений

Корпоративное приложение на языке Java может содержать сотни классов. Вы будете разрабатывать код разных типов: компоненты EJB, сервлеты, компоненты JSF, сущности JPA, а также вспомогательные классы и утилиты. Этот код, в свою очередь, может зависеть от десятков внешних библиотек, что приведет к увеличению общего числа классов еще на несколько сотен. Все эти классы являются частью приложения. Кроме того, приложения обычно содержат код, написанный на

других языках, таких как JSP, HTML, CSS, JavaScript, и статические файлы, такие как изображения. В некоторый момент вам потребуется объединить все это вместе и развернуть на сервере EE. Эта процедура так и называется: *развертывание*.

Как уже говорилось в главе 1, контейнер EJB является частью сервера EE и отвечает за управление компонентами EJB и MDB. Чтобы развернуть эти компоненты на сервере EE, их следует упаковать в модуль EJB-JAR. А чтобы понять, как выполняется упаковка модулей EJB-JAR, необходимо посмотреть, какое место они занимают в общей картине пакетов Java EE и из каких элементов состоит законченное корпоративное приложение на Java.

До настоящего момента основное внимание в этой книге уделялось использованию сеансовых компонентов EJB и компонентов MDB, для реализации прикладной логики, а также сущностей JPA, для организации сохранения данных. Но приложение не будет полным без слоя представления, обращающегося к прикладной логике, реализованной в компонентах EJB. Например, компоненты EJB, созданные для приложения ActionBazaar, будут лежать мертвым грузом, если клиентское приложение не сможет обращаться к ним. Наиболее вероятно вы будете использовать стандартные технологии, такие как JSF, для создания веб-слоя в своих приложениях. Такие веб-приложения вместе с компонентами EJB образуют корпоративное приложение, которое можно развернуть на сервере приложений.

Чтобы развернуть приложение и ввести его в действие, необходимо упаковать модули EJB-JAR и WAR в общий модуль EAR и развернуть его на сервере приложений. Спецификация Java EE определяет стандартный способ упаковки этих модулей в файл формата JAR. Одно из преимуществ этого формата, также определяемого спецификаций, состоит в том, что он совместим с разными серверами приложений.

В табл. 13.1 перечислены модули, поддерживаемые сервером EE. Каждый модуль обычно объединяет в себе схожие элементы приложения. Например, у вас может быть несколько модулей EJB-JAR, каждый из которых включает разные части прикладной логики. Аналогично у вас может быть несколько модулей WAR, реализующих уникальные пользовательские интерфейсы. Модуль EAR служит супермодулем, содержащим все остальные модули, поэтому в конечном итоге вам придется развертывать только один файл. Сервер приложений исследует содержимое модуля EAR и развернет его. Порядок загрузки модуля EAR сервером мы рассмотрим в разделе 13.1.2.

Таблица 13.1. Модули, поддерживаемые сервером EE

Описание	Дескриптор	Содержит
EJB Java Archive (EJB-JAR)	`META-INF/ejb-jar.xml` или `WEB-INF/ejb-jar.xml`	Сеансовые компоненты и компоненты MDB. Может также включать сущности JPA и компоненты CDI.
Web Application Archive (WAR)	`WEB-INF/web.xml`	Артефакты веб-приложения, такие как сервлеты, страницы JSP и JSF, изображения и так далее. Может также включать компоненты EJB и CDI, сущности JPA.

Таблица 13.1. (окончание)

Описание	Дескриптор	Содержит
Enterprise Application Archive (EAR)	`META-INF/application.xml`	Другие модули Java EE, такие как EJB-JAR и WAR.
Resource Adapter Archive (RAR)	`META-INF/ra.xml`	Адаптеры ресурсов.
Client Application Archives (CAR)	`META-INF/application-client.xml`	Автономные Java-клиенты для компонентов EJB.
Java Persistence Archive (JPA)	`META-INF/persistence.xml` или `WEB-INF/persistence.xml`	Объектно-реляционные отображения между приложениями и базами данных в стандарте Java EE. Может включаться в состав архивов EJB-JAR, WAR, EAR и CAR.
Context Dependency and Injection Bean Archive (CDI)	`META-INF/beans.xml` или `WEB-INF/beans.xml`	Компоненты CDI в стандарте Java EE. Может включаться в состав архивов EJB-JAR, WAR, EAR и CAR.

Корпоративные приложения на Java должны быть сначала собраны в модули определенного типа, а затем из них собирается главный модуль EAR, который затем может быть развернут на сервере. Типы модулей определяются спецификацией Java EE.

Все эти модули сохраняются в виде файлов в формате JDK-JAR. Вы можете собрать их вручную, с помощью утилиты `jar`, входящей в состав JDK. Однако в действительности все рутинные операции, связанные со сборкой модулей, выполняются либо интегрированной средой разработки, либо специализированными инструментами (такими как Maven и ANT). Но в любом случае для каждого модуля вам придется написать программный код и дескриптор развертывания (deployment descriptor) с настройками. После сборки всех модулей выполняется последний шаг – сборка главного модуля (то есть, EAR) для развертывания.

Дескрипторы развертывания и аннотации

В большинстве случаев дескрипторы развертывания не требуются в модулях EE. Принцип преимущества соглашений перед настройками, а также аннотации в программном коде часто избавляют от необходимости определять настройки в виде дескрипторов развертывания. Но иногда дескрипторы не только полезны, но и необходимы. Например, когда используются возможности CDI, архив должен содержать файл *META-INF/beans.xml* или *WEB-INF/beans.xml*, даже если он пустой, так как он служит индикатором использования CDI. Это именно тот случай, когда дескриптор развертывания должен присутствовать обязательно. Другой пример: связывание удаленных компонентов EJB в разных окружениях и разных серверах. Можно конечно жестко «зашить» информацию об удаленном сервере в аннотации, но гораздо удобнее создать еще один дескриптор развертывания для архивов, предназначенных для разных окружений.

В этой главе мы сосредоточимся в основном на модулях EJB-JAR и EAR, но также коснемся модулей JPA и CDI. Сущности JPA занимают особое положение. Для них не определен отдельный тип модуля, поэтому они включаются в модули других типов. Например, возможность упаковывать сущности в модули WAR позволяет использовать технологию EJB 3 JPA в веб-приложениях. Сущности могут также включаться в модули EJB-JAR, позволяя прикладным компонентам извлекать и изменять хранимые данные. Автономные клиенты EJB также могут включать сущности, хотя использование сущностей в них обычно ограничивается временем выполнения клиента и конфигурационными данными – они не имеют доступа к хранимым данным, так как для этого существуют компоненты EJB. Сущности не могут включаться в архивы RAR.

Если сущности занимают особое положение, почему тогда в спецификации Java EE для них не определен собственный тип модулей? В конце концов, JBoss поддерживает тип модулей Hibernate Archive (HAR) для упаковки хранимых объектов фреймворка Hibernate. Если вы знакомы с хронологией развития спецификации EJB 3, ответ на этот вопрос должен быть для вас очевиден. Всех остальных мы сейчас угостим «Байками от группы экспертов» (здесь должна звучать музыка, нагнетающая страх).

В процессе работы над спецификацией EJB 3, в ее черновой вариант был введен тип архивов Persistence Archive (PAR), который таинственным образом исчез из окончательного варианта. Группы экспертов EJB и Java EE вели жаркие, эмоциональные битвы за введение отдельного типа модулей для хранимых сущностей в Java EE, и искали поддержки в сообществе в целом и на форумах разработчиков в частности. Многие разработчики высказались против введения нового типа модулей для хранимых сущностей, потому что сущности поддерживаются не только внутри контейнеров, но и за их пределами. Учитывая, что доступ к данным необходим в большинстве приложений, гораздо больше смысла организовать поддержку упаковки сущностей в модули наиболее распространенных типов, чем вводить новый тип. Теперь, когда вы знаете, какие типы модулей поддерживаются и немного о том, как они создаются, пришла пора заглянуть под капот модулей EAR.

13.1.1. Строение системы модулей Java EE

Система модулей Java EE основана на файле EAR, представляющем модуль верхнего уровня, который содержит все остальные модули Java EE, подлежащие развертыванию. Поэтому, чтобы понять, как действует механизм развертывания, познакомимся поближе с файлом EAR и начнем с примера из приложения ActionBazaar.

Приложение ActionBazaar содержит модуль EJB-JAR, веб-модуль, файл JAR со вспомогательными классами и модуль клиентского приложения. Ниже показана структура модуля EAR для приложения ActionBazaar:

```
META-INF/application.xml
actionBazaar-ejb.jar
actionBazaar.war
```

```
actionBazaar-client.jar
lib/actionBazaar-commons.jar
```

Файл *application.xml* – это дескриптор развертывания, описывающий стандартные модули, упакованные в файл EAR. Содержимое этого файла приводится в листинге 13.1.

Листинг 13.1. Дескриптор развертывания приложения ActionBazaar в модуле EAR

```
<application>
  <module>
    <!-- ❶ Модуль EJB -->
    <ejb>actionBazaar-ejb.jar</ejb>
  </module>
  <module>
    <!-- ❷ Веб-модуль -->
    <web>
      <web-uri>actionBazaar.war</web-uri>
      <context-root>ab</context-root>
    </web>
  </module>
  <module>
    <!-- ❸ Модуль клиентского приложения -->
    <java>actionBazaar-client.jar</java>
  </module>
</application>
```

Как показано в листинге 13.1. дескриптор развертывания в модуле EAR явно идентифицирует каждый артефакт, как модуль определенного типа. Модуль EJB ❶ содержит компоненты EJB с прикладной логикой. Веб-модуль ❷ – веб-версию приложения. Модуль клиентского приложения ❸ – еще одну версию приложения, тонкого клиента с графическим интерфейсом. При развертывании этого модуля EAR на сервере приложений, сервер будет использовать информацию и дескриптора для развертывания модулей каждого типа.

В Java EE 5.0 дескриптор развертывания был сделан необязательным, даже для модулей EAR, тогда как в предыдущих версиях Java EE дескриптор являлся обязательной частью любого модуля. Серверы приложений, совместимые со спецификацией Java EE 5.0, автоматически определяют необходимые параметры развертывание, опираясь на стандартные соглашения по именованию или сканируя содержимое архивов. За дополнительной информацией по соглашениям обращайтесь по адресу: http://www.oracle.com/technetwork/java/namingconventions-139351.html. А теперь посмотрим, как серверы приложений осуществляют развертывание модуля EAR.

13.1.2. Загрузка модулей Java EE

В процессе развертывания сервер приложений определяет типы модулей, проверяет их и выполняет соответствующие операции, чтобы сделать приложение доступным для пользователей. Несмотря на то, что конечная цель одна и та же, разные производители серверов приложений могут по-разному реализовать раз-

вертывание. Однако все они стремятся обеспечить максимальную скорость этой процедуры.

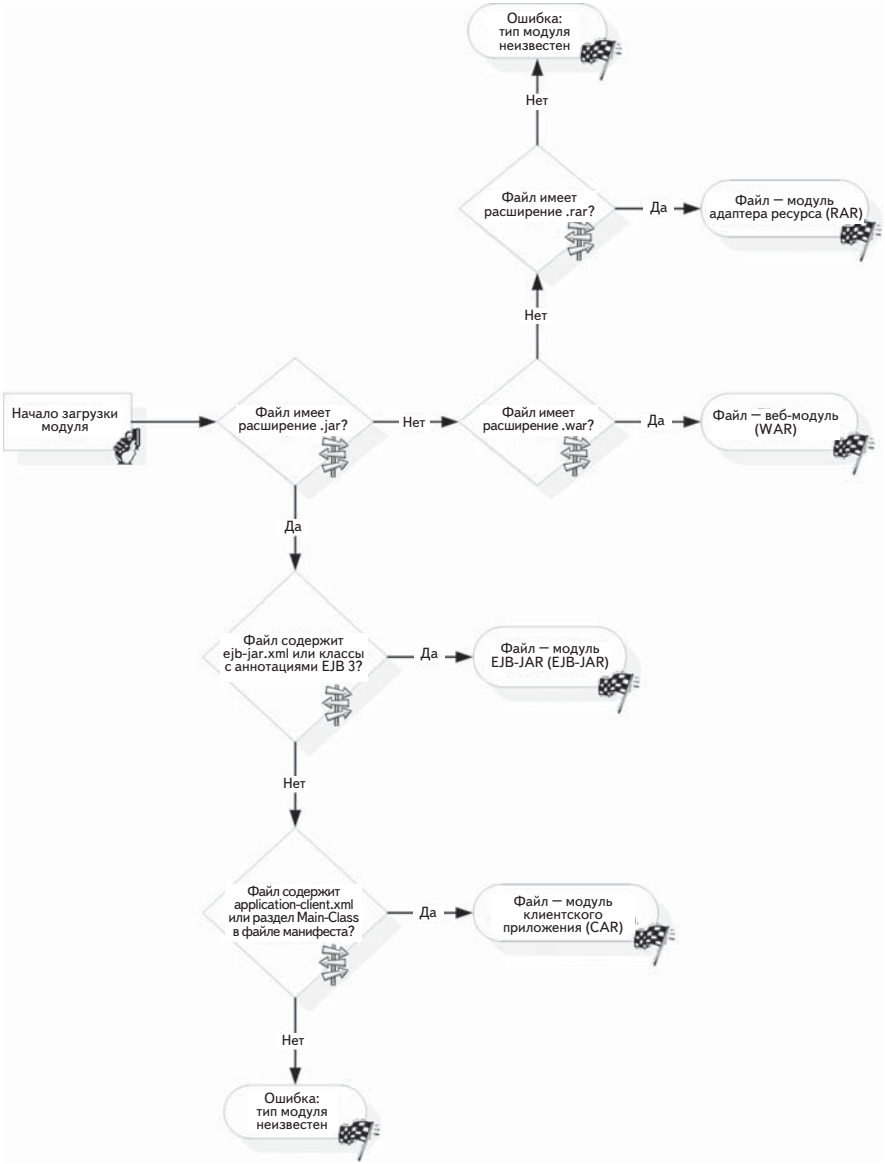

Рис. 13.1. Алгоритм, которому следуют серверы приложений при развертывании модуля EAR. Java EE не требует наличия дескриптора развертывания в модуле EAR, идентифицирующего типы модулей в пакете, поэтому сервер должен уметь автоматически определять типы модулей, опираясь на соглашения об именовании (расширения) и их содержимое. Реализуется это с использованием данного алгоритма

Даже при том, что каждый производитель волен оптимизировать свою реализацию как угодно, все они следуют стандартным правилам, определяемым спецификацией, когда дело доходит до поддерживаемых особенностей и порядка загрузки модулей. Это означает, что любой сервер приложений придерживается алгоритма, изображенного на рис. 13.1, когда пытается загрузить файл EAR, содержащий модули или архивы, перечисленные в табл. 13.1.

Модуль EAR с множеством модулей EJB-JAR, WAR, RAR и других, легко может содержать тысячи отдельных классов. Все эти классы должны быть проанализированы, загружены и переданы под управление серверу EE, что не так-то просто, когда разные модули WAR могут содержать разные версии одних и тех же классов. Далее мы обсудим, как сервер EE решает проблему этого «ада JAR».

13.2. Загрузка классов

Прежде чем приступить к исследованию порядка загрузки классов сервером EE, мы сначала узнаем, как действуют загрузчики классов Java, затем рассмотрим типичные стратегии загрузки классов, используемые серверами EE и, в заключение, узнаем, как разрешаются зависимости между модулями EE.

13.2.1. Основы загрузки классов

Загрузка классов в языке Java реализована в виде иерархической структуры загрузчиков классов, в соответствии с которой каждый загрузчик отвечает за загрузку определенных классов и построен поверх предыдущего. Эта структура изображена на рис. 13.2.

Загрузчики классов в Java следуют модели «сначала родитель». Это означает, что независимо от того, какой загрузчик выполняется, он сначала предложит загрузить класс своему родителю. Родитель предложит загрузить класс своему родителю, и так до тех пор, пока не будет достигнут базовый загрузчик классов (bootstrap class loader). Если базовый загрузчик не сможет загрузить класс, он вернет управление вниз по цепочке, и тогда каждый загрузчик получит шанс попытаться загрузить требуемый класс и, либо завершить процесс загрузки в случае успеха, либо передать управление на уровень ниже. Если самый первый (нижний) загрузчик потерпит неудачу, он возбудит исключение `java.lang.ClassNotFoundException`. Как только класс будет найден и загружен, загрузчик сохранит его в локальном кэше для быстрого доступа. Теперь, когда мы освежили в памяти основы загрузки классов, рассмотрим более сложную схему загрузки классов в приложениях Java EE.

13.2.2. Загрузка классов в приложениях Java EE

Модули EAR обычно содержат множество модулей EJB-JAR, WAR и RAR, а также сторонние библиотеки поддержки, включающие тысячи классов, и именно на сервер Java EE возлагается обязанность реализовать такую стратегию загрузки классов, гарантировала бы всем приложениям доступность необходимых им классов.

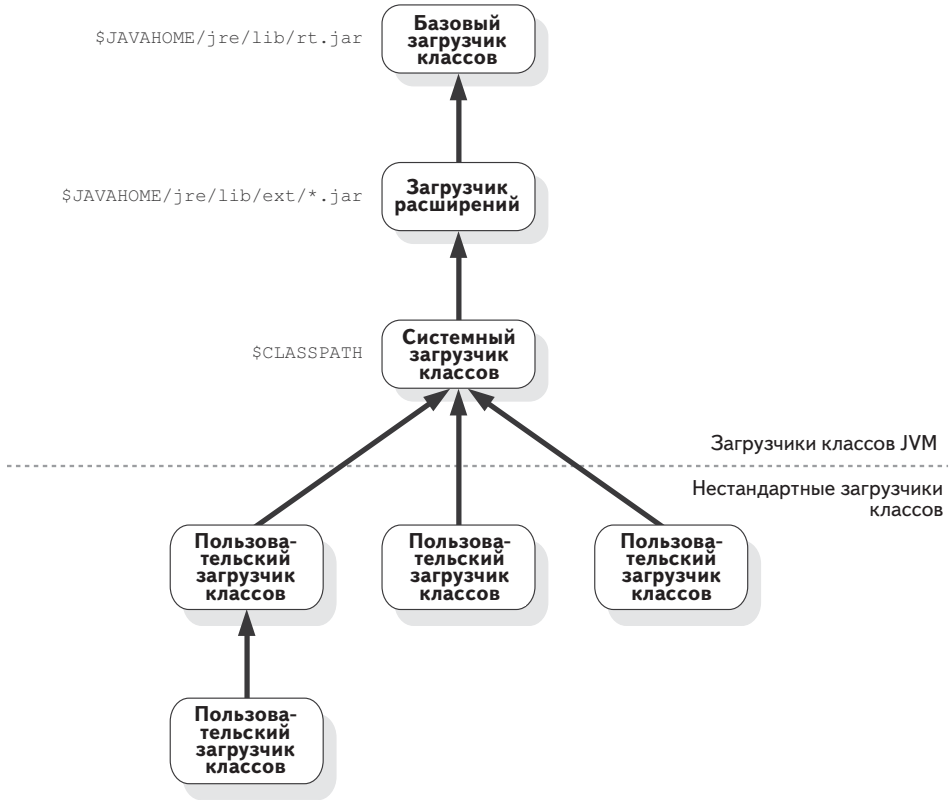

Рис. 13.2. Простая иерархия загрузчиков классов в приложениях на Java

Несмотря на важность загрузки классов, спецификация Java EE не определяет стандартную реализацию загрузчиков классов. Создание реализации полностью возлагается на плечи производителей серверов EE. Однако спецификация Java EE определяет стандартные требования к видимости и совместному использованию классов различными модулями внутри EAR. С этими требованиями мы познакомимся в следующем разделе, но прежде взгляните на рис. 13.3, где схематически изображена стратегия загрузки классов, реализованная в большинстве серверов EE.

Как показано на рис. 13.3, загрузчик классов сервера приложений загружает архивы JAR из каталога */lib* сервера приложений, где находятся все библиотеки, которые должен предоставлять сервер приложений, такие как Java EE API.

Наследником загрузчика классов сервера приложений является загрузчик классов из модуля EAR. Этот загрузчик загружает классы, развертываемые на уровне модуля EAR. По умолчанию эти классы хранятся в архивах JAR внутри каталога */lib* модуля EAR, однако имя каталога */lib* можно изменить с помощью элемента `library-directory` в дескрипторе развертывания *application.xml*.

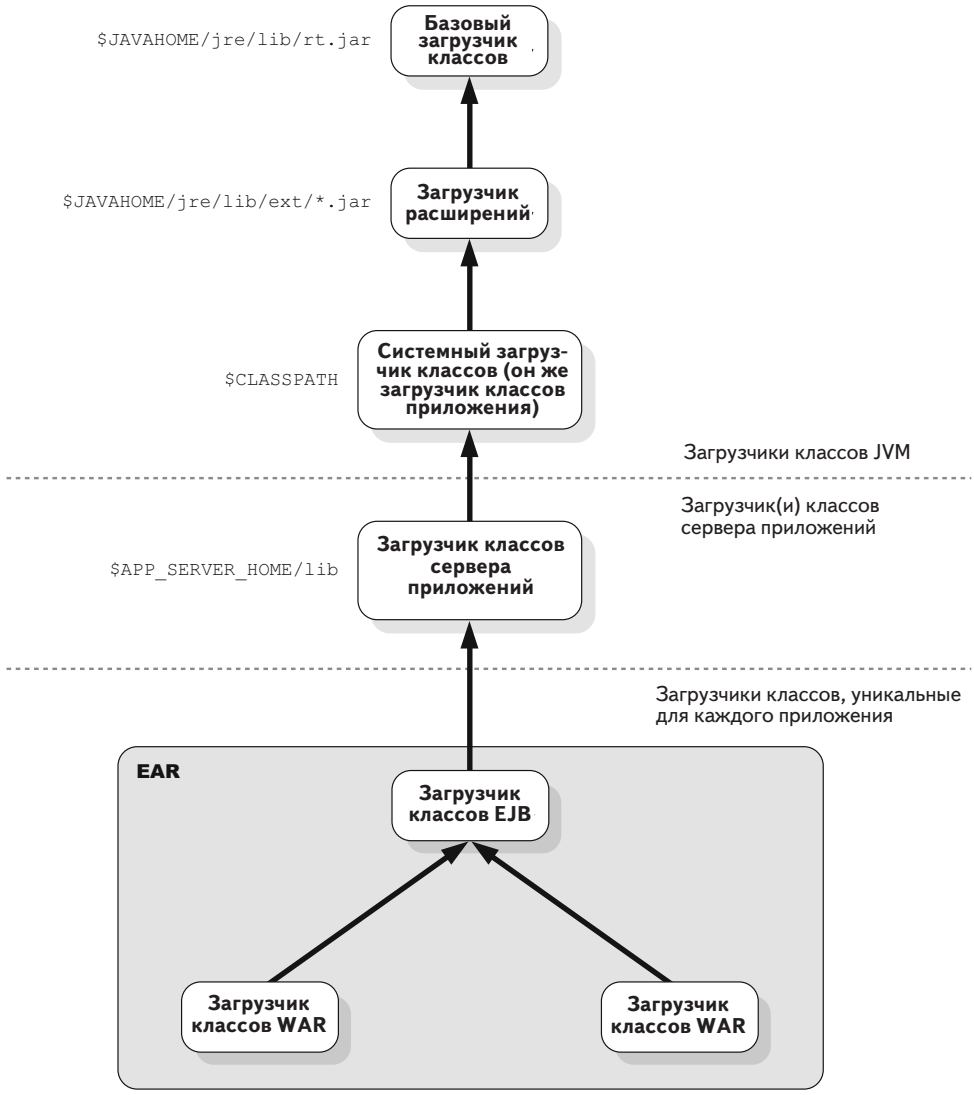

Рис. 13.3. Типичная структура загрузчиков классов для приложений Java EE

Наследником загрузчика классов из модуля EAR является загрузчик классов EJB. Даже при том, что модуль EAR может содержать множество модулей EJB-JAR, как правило существует только один загрузчик классов EJB, загружающий все модули EJB-JAR. Наконец, наследником загрузчика классов EJB является загрузчик классов WAR. Наследуя загрузчик EJB, WAR получает доступ ко всем компонентам EJB, развернутым в ходе загрузки EAR. Каждый модуль WAR имеет собственный загрузчик классов.

Все вышеописанное дает общее представление о том, как серверы EE обычно реализуют загрузку классов для корпоративных приложений. Такой порядок загрузки определяется требованиями стандартов EE к видимости классов для разных модулей внутри EAR, и именно поэтому он реализован в серверах EE. Давайте рассмотрим эти требования.

13.2.3. Зависимости между модулями Java EE

Спецификация Java EE определяет стандартные требования к видимости классов для всех модулей EE. Например, спецификация требует, чтобы классы были видимы модулю WAR, но не определяет, как загрузчики классов сервера EE должны обеспечивать эту видимость. В разделе 13.2.2 мы узнали, как серверы EE обычно реализуют загрузку классов.

Допустим, что листинг 13.2 описывает содержимое модуля EAR, который развертывается на сервере EE. На примере этого листинга мы исследуем наиболее общие требования к видимости классов для модулей EJB-JAR и WAR. Информацию о дополнительных требованиях к видимости или требованиях к видимости для других модулей можно найти в спецификации платформы Java EE.

Листинг 13.2. Пример содержимого модуля EAR, развертываемого на сервере EE

```
EE Server:
    resource-adapter1.rar:
        ❶ Содержимое и Class-Path адаптеров внешних ресурсов
        /com/**/*.class
        /META-INF/MANIFEST.MF Class-Path:

    resource-adapter2.rar:
        /com/**/*.class
        /META-INF/MANIFEST.MF Class-Path:

    action-bazaar.ear:
        /lib/*.jar
            ❷ Содержимое и Class-Path библиотек в каталоге /lib
            /com/**/*.class
            /META-INF/MANIFEST.MF Class-Path:

            ejb-jar1.jar
            ❸ Содержимое и Class-Path модуля EJB-JAR
            /com/**/*.class
            /META-INF/MANIFEST.MF Class-Path:

        ejb-jar2.jar
            ❹ Содержимое и Class-Path модулей EJB-JAR,
               развертываемых с модулем EAR
            /com/**/*.class
            /META-INF/MANIFEST.MF Class-Path:

        web-app1.war
            /WEB-INF/lib/*.jar
            ❺ Содержимое и Class-Path библиотек в каталоге /WEB-INF/lib
```

```
        /com/**/*.class
            /META-INF/MANIFEST.MF Class-Path:
        ❻ Содержимое и каталогf /WEB-INF/classes
        /WEB-INF/classes/**/*.class
        ❼ Class-Path модуля WAR
        /META-INF/MANIFEST.MF Class-Path:

    resource-adaptor-3.rar
        ❽ Содержимое и Class-Path адаптеров ресурсов,
           развертываемых с модулем EAR
        /com/**/*.class
        /META-INF/MANIFEST.MF Class-Path:
```

EJB-JAR

Исходя из примера содержимого модуля EAR, представленного в листинге 13.2, модуль EJB-JAR должен иметь доступ:

- к содержимому и значению `Class-Path` любых адаптеров внешних ресурсов ❶;
- к содержимому и значению `Class-Path` любых библиотек в каталоге */lib* модуля EAR ❷;
- к содержимому и значению `Class-Path` самого модуля EJB-JAR ❸;
- к содержимому и значению `Class-Path` дополнительных модулей EJB-JAR, развертываемых с модулем EAR ❹;
- к содержимому и значению `Class-Path` любых адаптеров ресурсов, развертываемых с модулем EAR ❽.

WAR

Исходя из примера содержимого модуля EAR, представленного в листинге 13.2, модуль WAR должен иметь доступ:

- к содержимому и значению `Class-Path` любых адаптеров внешних ресурсов ❶;
- к содержимому и значению `Class-Path` любых библиотек в каталоге */lib* модуля EAR ❷;
- к содержимому и значению `Class-Path` любых библиотек в каталоге */WEB-INF/lib* модуля WAR❺;
- к содержимому каталога */WEB-INF/classes* модуля WAR ❻;
- к значению `Class-Path` модуля WAR ❼.
- к содержимому и значению `Class-Path` всех модулей EJB-JAR, развертываемых с модулем EAR ❸,❹;
- к содержимому и значению `Class-Path` любых адаптеров ресурсов, развертываемых с модулем EAR ❽.

Как видите, загрузчику классов сервера EE приходится проделывать огромный объем работ. При большом количестве классов и большом числе уровней конфликты версий классов становятся неизбежными. Понимание требований видимости и знание модели делегирования «сначала родитель», которую используют загрузчи-

ки для поиска классов, помогут вам эффективнее упаковывать свои приложения. В следующем разделе мы посмотрим, как упаковываются сеансовые компоненты и компоненты, управляемые сообщениями.

13.3. Упаковка сеансовых компонентов и компонентов, управляемых сообщениями

Прежде чем создавать модуль EAR для развертывания приложения на сервере EE, необходимо создать модули, которые войдут в состав единого модуля EAR. Теперь, когда мы выяснили требования к видимости классов для разных модулей, перейдем к исследованию особенностей их упаковки. Далее мы посмотрим, как упаковать сеансовые компоненты и компоненты, управляемые сообщениями. Спецификация Java EE дает возможность упаковывать сеансовые компоненты и компоненты, управляемые сообщениями в модули EJB-JAR или WAR. При упаковке в модули EJB-JAR компоненты будут выполняться в полноценном контейнере EJB. При упаковке в модули WAR, компоненты будут выполняться в облегченном контейнере EJB Lite, обладающим большинством возможностей (но не всеми) полноценного контейнера EJB. Итак, приступим к упаковке модулей EJB-JAR и WAR, рассмотрим достоинства и недостатки дескрипторов развертывания в сравнении с аннотациями и затем посмотрим на настройки по умолчанию для интерцепторов.

13.3.1. Упаковка EJB-JAR

Модуль EJB-JAR в действительности является самым обычным Java-архивом в формате JAR. Учитывая, что Java EE следует принципу преимущества соглашений перед настройками и позволяет использовать аннотации, модуль EJB-JAR может вообще не включать дескриптор развертывания *META-INF/ejb-jar.xml*. Когда сервер EE развертывает модуль EAR, он автоматически исследует содержимое архивов JAR и ищет в них аннотации EJB 3 или файл *META-INF/ejb.jar.xml*, по наличию которых он определяет, что архив JAR является модулем EJB-JAR (подробнее этот процесс описывается в разделе 13.1.2). Итак, чтобы создать модуль EJB-JAR, нужно создать обычный архив JAR со следующей структурой:

```
ActionBazaar.jar:
  com/actionbazaar/ejb/BidServiceEjb.class  ← Сеансовый компонент
  com/actionbazaar/mdb/BidServiceMdb.class  ← Компонент MDB
  META-INF/ejb.jar.xml                      ← Дескриптор развертывания
```

Сама процедура создания архива JAR во многом зависит от используемой технологии. Разработчикам приложений на Java доступна масса возможностей, поэтому вы всегда сможете найти такой способ, который лучше всего соответствует вашим потребностям. Далее мы остановимся на вариантах, предоставляемых NetBeans и Maven.

Netbeans

В NetBeans создайте новый модуль EJB и добавьте в него исходный код своих сеансовых компонентов и компонентов MDB. На рис. 13.4 показано как может выглядеть вкладка представления **Projects** (Проекты) модуля.

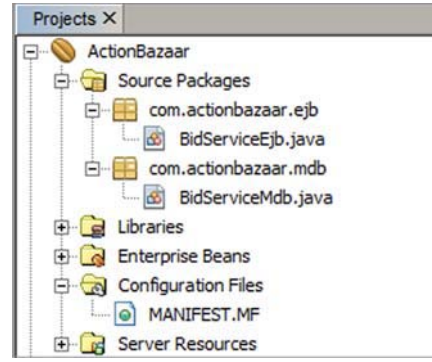

Рис. 13.4. Модуль EJB в представлении **Projects** (Проекты) Netbeans

После окончания разработки своих компонентов, щелкните правой кнопкой мыши на проекте и выберите пункт **Build** (Собрать) контекстного меню. NetBeans автоматически скомпилирует исходный код и сгенерирует модуль EJB-JAR. Найдите модуль EJB-JAR в каталоге */dist* проекта. Заглянув внутрь файла JAR (рис. 13.5), можно увидеть классы и конфигурационные файлы на своих местах.

```
c:\>jar -tf ActionBazaar.jar
META-INF/
META-INF/MANIFEST.MF
com/
com/actionbazaar/
com/actionbazaar/ejb/
com/actionbazaar/mdb/
com/actionbazaar/ejb/BidServiceEjb.class
com/actionbazaar/mdb/BidServiceMdb.class
```

Рис. 13.5. Содержимое модуля EJB-JAR

Вы можете заметить отсутствие файла *META-INF/ejb-jar.xml*. Это обусловлено тем, что он отсутствует в проекте. Чтобы добавить этот файл, щелкните правой кнопкой мыши на проекте, выберите пункт **New** (Создать) в контекстном меню и затем выберите пункт **Standard Deployment Descriptor** (Стандартный дескриптор развертывания). NetBeans добавит в проект пустой файл *ejb-jar.xml*. Если теперь пересобрать проект, файл *META-INF/ejb-jar.xml* автоматически будет добавлен в модуль EJB-JAR.

Maven

Чтобы создать модуль EJB-JAR с помощью Maven, добавьте POM-тип упаковки `<packaging>ejb</packaging>`. Для настройки конфигурации модуля EJB-JAR, чтобы изменить параметры создания файла JAR, используйте `maven-ejb-plugin`. В листинге 13.3 показан минимальный файл POM, создающий модуль EJB-JAR.

Листинг 13.3. Файл POM для сборки модуля EJB-JAR

```
<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>
  <groupId>com.actionbazaar</groupId>
  <artifactId>actionbazzar-ejb</artifactId>
  <version>1.0.0</version>
  <!-- Тип упаковки "ejb" создаст модуль EJB-JAR -->
  <packaging>ejb</packaging>
```

```
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <!-- Использовать maven-ejb-plugin для
             настройки создания модуля EJB-JAR -->
        <artifactId>maven-ejb-plugin</artifactId>
        <version>2.3</version>
        <configuration>
          <ejbVersion>3.1</ejbVersion>
          <archive>
            <addMavenDescriptor>false</addMavenDescriptor>
            <manifest>
              <!-- Этот параметр добавит Class-Path в META-INF/MANIFEST.MF -->
              <addClasspath>true</addClasspath>
            </manifest>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>javax</groupId>
      <artifactId>javaee-api</artifactId>
      <version>6.0</version>
      <type>jar</type>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</project>
```

Maven создаст модуль EJB-JAR в каталоге */target*, причем этот модуль будет создан без файла *META-INF/ejb-jar.xml*. Чтобы добавить дескриптор развертывания, можно создать файл */src/main/resources/META-INF/ejb-jar.xml*. После того, как файл дескриптора развертывания будет помещен в стандартный каталог ресурсов Maven, он автоматически будет включен в модуль EJB-JAR при следующей сборке.

Мы познакомились с двумя способами сборки модуля EJB-JAR. А теперь посмотрим, как упаковать сеансовые компоненты и компоненты MDB в модуль WAR.

13.3.2. Упаковка компонентов EJB в модуль WAR

Спецификация Java EE 6 позволяет включать сеансовые компоненты и компоненты, управляемые сообщениями, в модуль WAR, вместо того, чтобы развертывать их из отдельного модуля EJB-JAR. Однако, компоненты, включенные в модуль WAR, будут выполняться под управлением контейнера EJB Lite. Это означает, что они будут лишены некоторых функциональных возможностей полноценного контейнера EJB, но для простых приложений или прототипов это ограничение является несущественным.

Модуль WAR – это архив JAR с содержимым, удовлетворяющим требованиям к модулям WAR, предъявляемым спецификацией Java EE. Обычно модули WAR содержат классы и другие ресурсы в подкаталоге */classes*, сторонние зависимости в подкаталоге */lib* и настройки в файле */WEB-INF/web.xml*. Структура типичного модуля WAR имеет следующий вид:

```
ActionBazaar.war:
  classes/
    com/actionbazaar/web/ejb/BidServiceEjb.class ← Сеансовый компонент в WAR
    com/actionbazaar/web/mdb/BidServiceMdb.class ← Компонент MDB в WAR
  WEB-INF/web.xml ← Дескриптор развертывания модуля WAR (необязательный)
```

По аналогии с модулями EJB-JAR, процедура создания модуля WAR во многом зависит от технологии, используемой в вашем коллективе. Далее мы рассмотрим примеры использования NetBeans и Maven.

Netbeans

В NetBeans создайте новый модуль веб-приложения и добавьте в него исходный код своих сеансовых компонентов и компонентов MDB. На рис. 13.6 показано как может выглядеть вкладка представления **Projects** (Проекты) модуля.

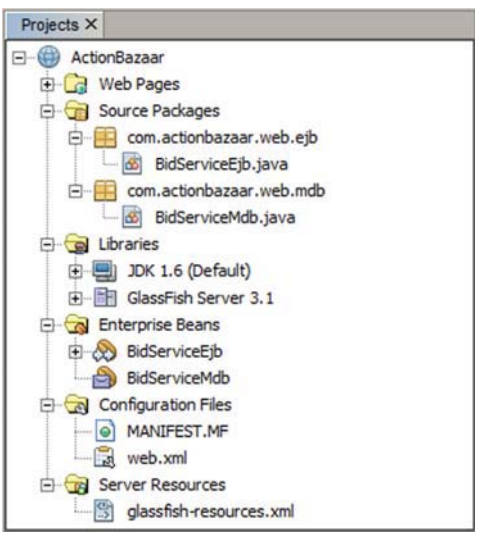

Рис. 13.6. Модуль Web Application в представлении **Projects** (Проекты) Netbeans

Этот веб-модуль приложения ActionBazaar содержит сеансовый компонент и компонент MDB. Чтобы собрать проект, щелкните на нем правой кнопкой мыши и выберите пункт **Build** (Собрать) контекстного меню. NetBeans создаст модуль WAR и поместит его в подкаталог */dist*. На рис. 13.7 показана структура получившегося модуля WAR.

Как и при создании модуля EJB-JAR, по умолчанию NetBeans не включает файл *ejb-jar.xml*. Это объясняется отсутствием данного файла в проекте. Чтобы добавить этот файл, щелкните правой кнопкой мыши на проекте, выберите пункт **New** (Создать) в контекстном меню и затем выберите пункт **Standard Deployment Descriptor** (Стандартный дескриптор развертывания). NetBeans добавит в проект пустой файл *ejb-jar.xml*. При упаковке сеансовых компонентов и компонентов MDB в модули WAR, в отличие от EJB-JAR, NetBeans помещает дескриптор развертывания *ejb-jar.xml* в подкаталог *WEB-INF*, а не *META-INF*.

```
c:\>jar -tf ActionBazaar.war
META-INF/
META-INF/MANIFEST.MF
WEB-INF/
WEB-INF/classes/
WEB-INF/classes/com/
WEB-INF/classes/com/actionbazaar/
WEB-INF/classes/com/actionbazaar/web/
WEB-INF/classes/com/actionbazaar/web/ejb/
WEB-INF/classes/com/actionbazaar/web/mdb/
WEB-INF/classes/com/actionbazaar/web/ejb/BidServiceEjb.class
WEB-INF/classes/com/actionbazaar/web/mdb/BidServiceMdb.class
WEB-INF/web.xml
index.xhtml

c:\>
```

Рис. 13.7. Содержимое модуля WAR

Maven

Чтобы создать модуль WAR с помощью Maven, добавьте POM-тип упаковки `<packaging>war</packaging>`. Для настройки параметров создания модуля WAR используйте `maven-war-plugin`. В листинге 13.4 показан минимальный файл POM, создающий модуль WAR.

Листинг 13.4. Файл POM для сборки модуля WAR

```
<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>
  <groupId>com.actionbazaar</groupId>
  <artifactId>actionbazzar-web</artifactId>
  <version>1.0.0</version>
  <!-- Тип упаковки "war" создаст модуль WAR -->
  <packaging>war</packaging>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <!-- Использовать maven-war-plugin для
             настройки создания модуля WAR -->
        <artifactId>maven-war-plugin</artifactId>
        <version>2.1</version>
        <configuration>
          <!-- Этот параметр присвоит модулю WAR имя ActionBazaar.war -->
          <warName>ActionBazaar</warName>
          <archive>
            <addMavenDescriptor>false</addMavenDescriptor>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
```

```
    <dependency>
      <groupId>javax</groupId>
      <artifactId>javaee-api</artifactId>
      <version>6.0</version>
      <type>jar</type>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</project>
```

Maven создаст модуль WAR в каталоге */target*. Этот модуль будет содержать файл *WEB-INF/web.xml*, но не *ejb-jar.xml*. Чтобы добавить файл *ejb-jar.xml*, можно вручную создать файл */src/main/webapp/WEB-INF/ejb-jar.xml*. После этого Maven автоматически включит его в модуль WAR при следующей сборке.

Использование удаленных компонентов EJB в модулях WAR

Поддержка прямого включения компонентов EJB в модули WAR, появившаяся в Java EE 6, – не единственный способ, хотя и очень удобный. Приложения (особенно широко используемые и высоконадежные) обычно нуждаются в более значительных вычислительных мощностях для выполнения прикладной логики, чем необходимо для отображения веб-интерфейса. Поэтому архитектурно такие приложения нередко делятся на два кластера: небольшой кластер веб-серверов, где выполняются модули WAR, и более мощный кластер вычислительных серверов, где выполняются модули EJB. При такой организации модули WAR взаимодействуют с модулями EJB удаленно, посредством компонентов, снабженных аннотацией `@Remote`.

Если ваше приложение организовано подобным способом, в конфигурацию Maven для расширения `maven-ejb-plugin` можно включить параметр `<generateClient>true</generateClient>`. Обнаружив этот параметр, Maven упакует все интерфейсы для компонентов EJB в отдельный файл JAR. Модули WAR смогут затем использовать эту зависимость, включающую только интерфейсы, вместо полноценных компонентов EJB.

Мы познакомились с двумя способами сборки модуля WAR. А теперь посмотрим, в каких ситуациях не обойтись без дескрипторов развертывания.

13.3.3. XML против аннотаций

Дескриптор развертывания EJB (*ejb-jar.xml*) описывает содержимое модуля EJB-JAR, используемые им ресурсы, а также настройки транзакций и системы безопасности. Дескриптор развертывания записывается на языке XML, а поскольку является внешним по отношению к байт-коду Java, с его помощью можно определять разные настройки для окружения разработки и развертывания.

Дескриптор развертывания не является обязательным и его можно заменить аннотациями, но мы не рекомендуем стремиться использовать аннотации повсеместно. Аннотации прекрасно подходят для разработки, но они могут не соответствовать окружениям развертывания с часто изменяющимися настройками. Для больших компаний характерно привлечение разных людей, делающих свою работу в разных окружениях, таких как окружение разработки, тестирования, эксплуатации и так далее. Например, приложение может нуждаться в таких ресурсах,

как объекты DataSource или JMS, а имена этих ресурсов в каталоге JNDI могут различаться для разных окружений. Поэтому было бы бессмысленно определять подобные имена в программном коде с помощью аннотаций. Дескриптор развертывания позволит специалистам, осуществляющим развертывание, принять правильное решение для каждого конкретного случая. Имейте в виду, даже при том, что дескриптор развертывания является необязательным, некоторые настройки, такие как интерцепторы по умолчанию для модуля EJB-JAR, могут указываться только в дескрипторе развертывания (см. раздел 13.3.5). Модуль EJB-JAR может содержать:

- дескриптор развертывания (*ejb-jar.xml*);
- специализированный дескриптор развертывания, необходимый для настройки специфических параметров конкретного контейнера EJB.

Многие из вас будут рады узнать, что настройки в аннотациях можно смешивать с настройками в дескрипторах, определяя одни настройки в аннотациях, а другие в дескрипторе развертывания. Запомните, что дескриптор развертывания рассматривается как истина в последней инстанции – настройки, определяемые в нем, имеют более высокий приоритет, чем настройки, определяемые с помощью аннотаций. Поясним: с помощью аннотации можно присвоить атрибуту `TransactionAttribute` для метода EJB значение `REQUIRES_NEW`, но, если в дескрипторе развертывания присвоить этому же атрибуту значение `REQUIRED`, будет действовать значение `REQUIRED`.

Давайте разберем несколько коротких примеров и посмотрим, как могут выглядеть дескрипторы развертывания для модуля EJB. В листинге 13.5 приводится пример простого дескриптора развертывания для EJB-модуля BazaarAdmin.

Нестандартные дескрипторы развертывания

В дополнение к стандартному дескриптору развертывания (*ejb-jar.xml*) большинство серверов приложений имеют также расширения поддержки специфических параметров настройки. Например, GlassFish поддерживает собственный файл дескриптора *glassfish-ejb-jar.xml*, а WebLogic – файл *weblogic-ejb-jar.xml*. Важно понимать, что они не определяются стандартами Java EE. В них можно указывать только настройки для конкретных серверов приложений. Обычно эти файлы с настройками открывают прямой доступ к нестандартным особенностям, дополняющим стандартные возможности серверов EE. Часто эти особенности оказываются весьма полезными, но их применение ухудшает совместимость приложений с разными серверами, а иногда вообще делает их несовместимыми с другими серверами.

Листинг 13.5. Простой дескриптор развертывания ejb-jar.xml

```
<!-- ❶ Версия спецификации EJB, в данном случае 3.2 -->
<ejb-jar version="3.2">
  <enterprise-beans>
    <session>
      <!-- ❷ Идентификатор компонента EJB -->
      <ejb-name>BazaarAdmin</ejb-name>
```

```
      <remote>actionbazaar.buslogic.BazaarAdmin</remote>
      <ejb-class>actionbazaar.buslogic.BazaarAdminBean</ejb-class>
      <!-- ❸ Тип компонента -->
      <session-type>stateless</session-type>
      <!-- ❹ Тип транзакции для компонента -->
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
...
  <assembly-descriptor>
    <!-- ❺ Содержит настройки для транзакций, управляемых контейнером -->
    <container-transaction>
      <method>
        <ejb-name>BazaarAdmin</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
    <security-role>
      <!-- ❻ Роли, имеющие право доступа к компоненту BazaarAdmin -->
      <role-name>users</role-name>
    </security-role>
  </assembly-descriptor>
</ejb-jar>
```

Используя дескрипторы развертывания для своих компонентов EJB, убедитесь, что установлена версия 3.2 для `ejb-jar` ❶, потому что она будет использоваться сервером Java EE для определения версии компонентов EJB в архиве. Элемент `name` ❷ определяет компонент EJB с тем же именем в атрибуте `name` аннотации `@Stateless`. Эти имена должны совпадать, если вы переопределяете какие-либо атрибуты, указанные в аннотации. Элемент `session-type` ❸ определяет тип сеансового компонента. Этот элемент может иметь значение `stateless` или `stateful`. С помощью элемента `transaction-type` ❹ можно указать, будет ли компонент использовать транзакции, управляемые контейнером (`Container`) или компонентом (`Bean`). Дополнительные настройки транзакций, системы безопасности и других механизмов можно определить в элементе `assembly-descriptor` ❺ ❻.

В табл. 13.2 перечислены наиболее часто используемые аннотации и соответствующие им элементы дескриптора. Имейте в виду, что для каждой аннотации имеется соответствующий ей элемент. Вам потребуются только те, которые имеют смысл в вашем окружении разработки. Некоторые элементы дескриптора наверняка понадобятся для определения ссылок на ресурсы, привязок интерцепторов и настроек системы безопасности. Мы настоятельно рекомендуем вам самим заняться исследованием других элементов.

Таблица 13.2. Аннотации и соответствующие им элементы дескриптора

Аннотация	Тип	Атрибут аннотации	Элемент дескриптора
`@Stateless`	Компонент EJB	`name`	`<session-type>Stateless ejb-name`

Таблица 13.2. (продолжение)

Аннотация	Тип	Атрибут аннотации	Элемент дескриптора
`@Stateful`	Компонент EJB	`name`	`<session-type>Stateful ejb-name`
`@MessageDriven`	Компонент EJB	`name`	`message-driven ejb-name`
`@Remote`	Интерфейс		`remote`
`@Local`	Интерфейс		`Local`
`@TransactionManagement`	Тип управления транзакциями		`transaction-type`
`@TransactionAttribute`	Способ управления транзакциями		`container-transaction trans-attribute`
`@Interceptors`	Интерцепторы		`interceptor-binding interceptor-class`
`@ExcludeClassInterceptors`	Интерцепторы		`exclude-classinterceptor`
`@ExcludeDefaultInterceptors`	Интерцепторы		`exclude-defaultinterceptors`
`@AroundInvoke`	Собственные интерцепторы		`around-invoke`
`@PostActivate`	Обработчики событий жизненного цикла		`post-activate`
`@PrePassivate`	Обработчики событий жизненного цикла		`pre-passivate`
`@DeclareRoles`	Настройки безопасности		`security-role`
`@RolesAllowed`	Настройки безопасности		`method-permission`
`@PermitAll`	Настройки безопасности		`unchecked`
`@DenyAll`	Настройки безопасности		`exclude-list`
`@RunAs`	Настройки безопасности		`security-identity run-as`
`@Resource`	Ссылки на ресурсы (DataSource, JMS, переменные окружения, почта и др.)		`resource-ref resource-env-ref message-destination-ref env-ref`
	Внедрение ресурсов	Метод записи/ поле	`injection-target`
`@EJB`	Ссылки для компонентов EJB		`ejb-ref ejb-local-ref`

Таблица 13.2. (окончание)

Аннотация	Тип	Атрибут аннотации	Элемент дескриптора
`@PersistenceContext`	Ссылка на контекст хранения		`persistence-context-ref`
`@PersistenceUnit`	Ссылка на единицу хранения		`persistence-unit-ref`

Схему XML для дескриптора развертывания в EJB 3 можно найти по адресу: http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd.

13.3.4. Переопределение настроек, указанных в аннотациях

Как уже отмечалось выше, вы можете определять настройки в аннотациях и переопределять их в дескрипторах развертывания. Но имейте в виду, чем больше таких настроек у вас будет, тем выше вероятность допустить ошибку и превратить отладку в кошмар.

> **Примечание.** *Главное правило, которое следует помнить, – значения атрибутов* `name` *в аннотациях определения сеансовых компонентов и компонентов MDB должны совпадать с соответствующими элементами* `ejb-name` *в дескрипторе. Если в аннотациях атрибут* `name` *не указан, в качестве значений соответствующих элементов* `ejb-name` *следует указывать имена классов. Это означает, что переопределяя настройки, указанные в аннотации, в дескрипторе развертывания, элемент* `ejb-name` *должен соответствовать имени класса компонента.*

Пусть имеется сеансовый компонент без сохранения состояния, снабженный следующими аннотациями:

```
@Stateless(name = "BazaarAdmin")
public class BazaarAdminBean implements BazaarAdmin {
    ...
    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public Item addItem() {...}
}
```

Атрибуту `name` здесь присвоено значение `BazaarAdmin`, и это же значение указано в элементе `ejb-name` внутри дескриптора развертывания:

```
<ejb-name>BazaarAdmin</ejb-name>
```

Если не определить атрибут `name`, контейнер будет использовать в качестве имени компонента имя класса `BazaarAdminBean` и тогда, чтобы переопределить настройки, указанные в аннотации, необходимо будет использовать это имя в дескрипторе развертывания:

```
<ejb-name>BazaarAdminBean</ejb-name>
```

Также в примере выше метод компонента отмечен транзакцией `@TransactionAttribute` с параметром `REQUIRES_NEW`. Переопределить параметр на `REQUIRED` можно с помощью следующего фрагмента в дескрипторе:

```
<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>BazaarAdmin</ejb-name>          <!-- ❶ Идентификатор EJB -->
      <method-name>getUserWithItems</method-name>
      <method-params></method-params>
    </method>
    <trans-attribute>Required</trans-attribute> <!-- ❷ Уровень транзакции -->
  </container-transaction>
</assembly-descriptor>
```

В этом примере элемент `assembly-descriptor` определяет параметр транзакции ❷. Кроме того, элемент `ejb-name` в дескрипторе сборки соответствует оригинальному имени, указанному в аннотации `@Stateless`.

13.3.5. Определение интерцепторов по умолчанию

Интерцепторы (как вы наверняка помните из главы 5) позволяют реализовать сквозную функциональность наиболее простым способом. Интерцептор можно определить как для всего класса, так и для отдельных методов. Имеется также возможность определить интерцептор по умолчанию для всего модуля, для всех классов компонентов EJB в модуле EJB-JAR. Мы уже знаем, что интерцепторы по умолчанию можно определять только в дескрипторе развертывания *ejb-jar.xml*. В листинге 13.6 показано, как определить интерцепторы по умолчанию для модуля EJB.

Листинг 13.6. Настройка интерцепторов по умолчанию в ejb-jar.xml

```
<interceptor-binding>     <!-- ❶ Определение привязки для интерцептора -->
  <ejb-name>*</ejb-name> <!-- ❷ Применяется ко всем компонентам EJB -->
  <interceptor-class>
    actionbazaar.buslogic.CheckPermissionInterceptor
  </interceptor-class>
  <interceptor-class>
    actionbazaar.buslogic.ActionBazaarDefaultInterceptor
  </interceptor-class>
</interceptor-binding>
```

Элемент `interceptor-binding` ❶ определяет привязки интерцепторов для конкретного компонента EJB с именем, указанным в элементе `ejb-name`. Для интерцепторов по умолчанию, которые применяются ко всем компонентам EJB, находящимся в модуле EJB-JAR, в качестве значения элемента `ejb-name` следует указать `*` ❷. С помощью элемента `<interceptor-class>` указывается класс интерцептора. Как показывает листинг 13.6, в одной и той же привязке можно

определить несколько интерцепторов, а порядок их следования в дескрипторе развертывания будет определять порядок вызова. В данном случае при обращении к методу компонента EJB сначала будет вызываться `CheckPermissionInterceptor`, а потом `ActionBazaarDefaultInterceptor`.

Если вы хотите освежить свои знания интерцепторов, прочитайте еще раз главу 5 и затем возвращайтесь к нам сюда. Мы подождем.

13.4. Упаковка сущностей JPA

Прикладной интерфейс Java Persistence API (JPA) прежде был частью спецификации EJB 3, но в версии EJB 3.2 его определение было выделено в отдельную спецификацию (http://jcp.org/en/jsr/detail?id=317). Главная причина – область применения JPA не ограничивается платформой Java EE. Его можно также использовать в приложениях на платформе Java Standard. Так как теперь интерфейс JPA определяется собственной спецификацией, мы подчеркнем здесь наиболее важные его элементы, связанные с EJB. За более подробной информацией обращайтесь к главе 9.

13.4.1. Модуль доступа к хранимым данным

Спецификация позволяет использовать сущности JPA в любых приложениях на Java, поэтому в приложениях Java EE сущности JPA просто упаковываются в модули EJB-JAR или WAR. Однако сущности JPA можно упаковывать в обычные архивы JAR и включать их непосредственно в главный модуль EAR. Ключевое значение для развертывания сущностей JPA имеет файл *META-INF/persistence.xml*, описывающий архивы JAR, модули EJB-JAR и модули WAR, содержащие одну или более единиц хранения. Далее мы коротко рассмотрим, как правильно упаковывать классы сущностей JPA и составлять файл *persistence.xml*.

JAR

В листинге 13.7 показано, как размещаются сущности JPA в простом архиве JAR.

Листинг 13.7. Структура архива JAR, содержащего сущности JPA

```
ActionBazaar.jar:
  META-INF/
    persistence.xml       ← ❶ Конфигурационный файл JPA
  actionbazaar/
    persistence/          ← ❷ Классы сущностей JPA в модуле
      Category.class
      Item.class
```

Настройки хранения данных находятся в файле *META-INF/persistence.xml* ❶, а классы сущностей – в подкаталоге *persistence* ❷.

EJB-JAR

Упаковка сущностей JPA в модуль EJB-JAR выполняется точно так же, как упаковка в обычный архив JAR. В листинге 13.8 показано, как располагаются сущности JPA внутри модуля EJB-JAR.

Листинг 13.8. Структура модуля EJB-JAR, содержащего сущности JPA

```
ActionBazaar-ejb.jar:
  META-INF/
    persistence.xml          ← ❶ Конфигурационный файл JPA
  actionbazaar/
    buslogic/
      BazaarAdminBean.class
    persistence/             ← ❷ Классы сущностей JPA в модуле
      Category.class
      Item.class
      BazaarAdmin.class
```

Настройки хранения данных находятся в файле *META-INF/persistence.xml* ❶, а классы сущностей – в собственном подкаталоге ❷.

WAR

Упаковка сущностей JPA в модуль WAR выглядит немного сложнее. Сделать это можно двумя способами. Первый и самый простой способ: поместить архив JAR с сущностями JPA в модуль WAR, в каталог *WEB-INF/lib*. В листинге 13.9 показано, как располагаются сущности JPA внутри модуля WAR (предполагается, что используется файл JAR, структура которого приводится в листинге 13.7).

Листинг 13.9. Структура модуля WAR, содержащего сущности JPA, упакованные в архив JAR

```
ActionBazaar-web.war:
  WEB-INF/
    classes/
      ...
    lib/
      ActionBazaar.jar
    web.xml
      ...
```

В листинге 13.10 показано, как располагаются сущности JPA внутри модуля WAR, если они не упакованы в отдельный архив JAR, а непосредственно входя в состав модуля WAR.

Листинг 13.10. Структура модуля WAR, содержащего сущности JPA

```
ActionBazaar-web.war:
  WEB-INF/
    classes/
      META-INF/
```

```
    persistence.xml ← ❶ Конфигурационный файл JPA
  persistence/      ← ❷ Классы сущностей JPA в модуле
    Category.class
    Item.class
    BazaarAdmin.class
lib/
  ...
web.xml
```

Файл *persistence.xml* в этом случае должен находиться в каталоге */classes/META-INF/* ❶. Сущности JPA копируются как файлы **.class* в подкаталог отдельного пакета ❷, что вполне типично для модулей WAR. Теперь, после знакомства с особенностями включения сущностей в модули, узнаем чуть больше о содержимом файла *persistence.xml*.

Области видимости единиц хранения

Единицы хранения можно определять в модулях WAR, EJB-JAR или JAR, на уровне модуля EAR. Если единица хранения определяется в модуле WAR или EJB-JAR, она будет доступна только в данном конкретном модуле. А если единица хранения упакована в архив JAR и находится в корне или в подкаталоге *lib* модуля EAR, она автоматически будет доступна всем модулям в EAR. При этом следует помнить, что в случае конфликта имен между единицами хранения в каком-нибудь модуле и на уровне EAR, будет использоваться единица хранения из модуля.

Допустим, что имеется файл EAR со следующим содержимым:

```
lib/actionBazaar-common.jar
actionBazaar-ejb.jar
actionBazaar-web.war
```

где *actionBazaar-common.jar* включает единицу хранения с именем `ActionBazaar` и архив *actionBazaar-ejb.jar* так же включает единицу хранения с именем `ActionBazaar`.

В этом случае в веб-модуле автоматически будет доступна глобальная единица хранения `actionBazaar`, и ее можно использовать, как показано ниже:

```
@PersistenceUnit(unitName = "actionBazaar")
private EntityManagerFactory emf;
```

Но этот же код в модуле EJB будет использовать локальную единицу хранения. Если потребуется организовать доступ к глобальной единице хранения, объявленной на уровне модуля EAR, для ссылки на нее необходимо использовать специально сформированное имя:

```
PersistenceUnit(unitName ="lib/actionBazaar-common.jar#actionBazaar")
private EntityManagerFactory emf;
```

13.4.2. Описание модуля доступа к хранимым данным в persistence.xml

В главе 9 мы видели, как группировать сущности в единицы хранения и как определять эти единицы в файле *persistence.xml*. Теперь, когда мы узнали, как упаковывать сущности, пришло время детальнее исследовать файл *persistence.xml* – де-

скриптор, описывающий преобразование любого модуля JAR в модуль доступа к хранимым данным. Следует так же упомянуть, что *persistence.xml* является единственным обязательным дескриптором развертывания.

В листинге 13.11 приводится пример содержимого простого файла *persistence.xml* для приложения ActionBazaar. Он благополучно развертывает модуль доступа к хранимым данным в любом контейнере Java EE, поддерживающем JPA.

Листинг 13.11. Пример файла persistence.xml

```
<persistence>
  <!-- ❶ Определяет единицу хранения JPA -->
  <persistence-unit name = "actionBazaar" transaction-type = "JTA">
    <provider>
      <!-- ❷ Полное квалифицированное имя провайдера -->
      oracle.toplink.essentials.PersistenceProvider
    </provider>
    <!-- ❸ Имя источника данных ОВИС в каталоге JNDI -->
    <jta-data-source>jdbc/ActionBazaarDS</jta-data-source>
    <mapping-file>secondORMap.xml</mapping-file>
    <!-- ❹ Список сущностей, входящих в данную единицу хранения -->
    <class>ejb3inaction.persistence.Category</class>
    <class>ejb3inaction.persistence.Bid</class>
    ...
    <!-- ❺ Список файлов JAR с сущностями в данной единице хранения -->
    <jar-file>entities/ShippingEntities.jar</jar-file>
    <properties>
      <!-- ❻ Настройки провайдера, реализующего доступ к данным -->
      <property name = "toplink.ddl-generation"
        value = "drop-and-create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

Давайте пройдемся по содержимому этого файла. С помощью элемента `<persistence-unit>` определяется единица хранения ❶. При желании можно указать необязательный фабричный класс провайдера доступа к данным ❷. Если провайдер не указан, будет использоваться провайдер по умолчанию, настроенный в сервере приложений. Далее определяется имя источника данных для провайдера ❸, чтобы с его помощью механизм JPA мог подключиться к базе данных. Если в одном архиве имеется несколько единиц хранения, может потребоваться идентифицировать классы сущностей, составляющие их ❹. Если имеются сущности, хранящиеся в другом файле JAR, которые требуется включить в состав данной единицы хранения, с помощью. элемента `<jar-file>` ❺ можно определить путь к файлу JAR, относительно данного файла JAR. При необходимости в элементе `properties` ❻ можно также определить дополнительные настройки.

Так как теперь интерфейс JPA определяется собственной спецификацией, мы не будем продолжать углубляться в исследование его особенностей. Мы рассказали вам о требованиях к упаковке сущностей JPA в приложениях EE и познакомили с основными элементами файла *persistence.xml*. А за дополнительной информа-

цией обращайтесь к спецификации JPA или к другим книгам, посвященным этой теме.

Объектно-реляционные отображения

Обычно все объектно-реляционные отображения определяются с помощью аннотаций JPA. Но иногда аннотации оказываются не лучшим решением, например, когда в разных окружениях используются разные модели объектов. В таких случаях объектно-реляционные отображения можно определять в файлах *orm.xml*. При упаковке приложения EE файл *orm.xml* сохраняется рядом с файлом *persistence.xml*. Однако его можно также сохранить отдельно, в другом каталоге и с другим именем, указанными элементом `<mapping-file>` в *persistence.xml*. Как и для любых других дескрипторов развертывания, если этот файл существует, настройки в нем имеют преимущество перед настройками в аннотациях.

13.5. Упаковка компонентов CDI

Как и для сущностей JPA, для компонентов CDI не определяется какой-то специальный модуль, потому что область применения механизма внедрения зависимостей не ограничена окружением Java EE. Определение CDI 1.1 для Java EE 7 можно найти в документе JSR 346 (http://jcp.org/en/jsr/detail?id=346), являющемся продолжением JSR 299 и JSR 330. Спецификация CDI 1.1 добавляет дополнительные требования к внедрению зависимостей в окружении Java EE. Но внедрение зависимостей можно также использовать в приложениях на платформе Java Standard. Поэтому для компонентов CDI, как и для сущностей JPA, не предусматривается отдельный модуль EE – они просто включаются в состав других модулей. В этом разделе мы посмотрим, как компоненты CDI упаковываются в другие модули EE, такие как архивы JAR, модули EJB-JAR и WAR. Более подробная информация о CDI приводится в главе 12.

13.5.1. Модули CDI

В процессе развертывания приложения EE выполняется процедура поиска компонентов CDI, в ходе которой внутри приложения EE выявляются артефакты, использующие CDI. Эти артефакты содержат компоненты, которыми должен управлять сервер EE. Артефакт может быть обычным архивом JAR или модулем любого типа (см. табл. 13.1). Суть поиска заключается в определении наличия файла *beans.xml* в следующих местоположениях:

- *META-INF/beans.xml* в модулях JAR, EJB-JAR, RAR или в модуле JAR клиента, а также в архивах JAR или в каталогах, на которые ссылаются элементы `Class-Path` в файлах *META-INF/MANIFEST.MF*;
- *WEB-INF/beans.xml* в модулях WAR;
- *META-INF/beans.xml* в любых каталогах в пути к классам JVM.

Если в любом из перечисленных мест присутствует файл *beans.xml*, сервер EE проверит все классы в архиве и возьмет под свое управление все управляемые компоненты, которые обнаружит.

13.5.2. Дескриптор развертывания beans.xml

Механизм CDI и файл *beans.xml* подробно рассматриваются в главе 12, поэтому мы не будем повторно описывать их здесь, а только посмотрим, как использовать файл *beans.xml*, чтобы упаковать компоненты CDI в архивы JAR и модули EJB-JAR и WAR.

JAR

Допустим, что мы занимаемся разработкой некоторого приложения, не связанного с платформой EE, или вспомогательного проекта для поддержки приложения EE. В любом случае нам может понадобиться создать обычный архив JAR для передачи своего кода. Если в архиве имеются классы, использующие механизм CDI, в него необходимо добавить файл *META-INF/beans.xml*. Пример структуры такого архива приводится в листинге 13.12.

Листинг 13.12. Структура архива JAR, содержащего компоненты CDI

```
ActionBazaar-Utils.jar:
  META-INF/
    beans.xml ← ❶ Признак наличия компонентов CDI и настройки для них
com/actionbazaar/util/BidServiceUtil.class ← ❷ Классы, использующие CDI
```

Присутствие файла *beans.xml* ❶ в архиве служит признаком наличия компонентов CDI; `BidServiceUtil` ❷ – это класс, который будет проверен контейнером CDI и, при наличии соответствующих аннотаций, взят им под управление.

EJB-JAR

Для модулей EJB-JAR действуют те же правила, что и для архивов JAR. Если в модуле имеются компоненты CDI, в него нужно добавить файл *METAINF/beans.xml*, как показано в листинге 13.13.

Листинг 13.13. Структура модуля EJB-JAR, содержащего компоненты CDI

```
ActionBazaar.jar:
  META-INF/
    bean.xml ← ❶ Признак наличия компонентов CDI и настройки для них
  com/actionbazaar/ejb/BidServiceEjb.class ← ❷ Классы, использующие CDI
  com/actionbazaar/dao/BidDao.class        ← ❸ Классы, использующие CDI
  com/actionbazaar/dao/BidDaoJdbc.class    ← ❹ Классы, использующие CDI
```

Присутствие файла *beans.xml* ❶ в модуле служит признаком наличия компонентов CDI; `BidServiceUtil`, `BidDao` и `BidDaoJdbc` ❷ ❸ ❹ – это классы, которые будут проверены контейнером CDI и, при наличии соответствующих аннотаций, взяты им под управление.

WAR

В модули WAR необходимо добавить файл *WEB-INF/beans.xml*, как показано в листинге 13.14.

Листинг 13.14. Структура модуля WAR, содержащего компоненты CDI

```
ActionBazaar-web.war:
  WEB-INF/
    beans.xml ← ❶ Признак наличия компонентов CDI и настройки для них
    classes/
      persistence/
        Category.class     ← ❷ Классы, использующие CDI
        Item.class         ← ❸ Классы, использующие CDI
        BazaarAdmin.class  ← ❹ Классы, использующие CDI
```

Присутствие файла *beans.xml* ❶ в модуле служит признаком наличия компонентов CDI; `Category`, `Item` и `BazaarAdmin` ❷ ❸ ❹ – это классы, которые будут проверены контейнером CDI и, при наличии соответствующих аннотаций, взяты им под управление. Модуль WAR содержит также каталог *WEB-INF/lib/*, где могут находиться различные архивы JAR. Не забывайте, что эти архивы так же могут содержать файл *beans.xml*, служащий признаком наличия компонентов CDI, как показано в листинге 13.12.

Файл *beans.xml* играл важную роль в настройке механизма CDI вплоть до версии Java EE 7. Спецификация Java EE 7 сделала его необязательным – его можно заменить атрибутом `bean-discovery-mode` элемента `<beans>` в дескрипторе развертывания. Познакомимся с этим атрибутом в следующем разделе.

13.5.3. Атрибут bean-discovery-mode

Как только выяснилось, что файл *beans.xml* может оставаться пустым и служить лишь признаком использования механизма CDI, тут же стали высказываться предложения вообще избавиться от него. Эта возможность была предусмотрена в спецификации Java EE 7, в виде атрибута `bean-discovery-mode`, который может определять три режима исследования классов в приложениях Java EE 7 контейнером CDI. Эти режимы перечислены в табл. 13.3.

Таблица 13.3. Значения атрибута bean-discovery-mode

Значение	Описание
`ALL`	Обрабатываются все классы. Это поведение соответствует обычной практике включения файла *beans.xml* в приложения Java EE 6.
`ANNOTATED`	Обрабатываются только классы с аннотациями, объявляющими их компонентами. Это – поведение по умолчанию для Java EE 7.
`NONE`	Все классы в архиве (JAR) игнорируются.

В приложениях на платформе Java EE 7 отпала необходимость включать файл *beans.xml*. В отсутствие этого файла поиск компонентов CDI будет выполняться,

как для значения ANNOTATED в атрибуте bean-discovery-mode (см. табл. 13.3). Это поведение по умолчанию можно переопределить включением файла *beans.xml* или присваиванием другого значения атрибуту bean-discovery-mode, как показано в листинге 13.15.

Листинг 13.15. Присваивание значения атрибуту bean-discovery-mode

```
<?xml version="1.0" encoding="UTF-8"?>
<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"
       bean-discovery-mode="all"> <!-- ❶ Изменить значение атрибута -->
</beans>
```

В этом листинге атрибуту bean-discovery-mode присваивается значение "all" ❶, вследствие чего отменяется поведение, принятое в Java EE 7 по умолчанию, и разрешается обработка всех классов, как в Java EE 6.

Упаковка компонентов CDI в архивы JAR или модули EE выполняется достаточно просто. Далее мы познакомимся с некоторыми эффективными приемами упаковки и проблемами, с которыми вы можете столкнуться.

13.6. Эффективные приемы и типичные проблемы развертывания

После прочтения данной главы может сложиться ощущение, что для развертывания компонентов EJB 3 требуется масса всякой всячины. Возможно, это ощущение недалеко от истины. Однако в реальности от вас не требуется уделять пристальное внимание всем мелочам – серверы предоставляют массу вспомогательных инструментов и связующего кода, которые позволят автоматизировать рутину. При этом, независимо от того, какие компоненты используются в приложении и на каком сервере оно будет развертываться, следует запомнить несколько основных принципов.

13.6.1. Эффективные приемы упаковки и развертывания

Ниже перечислены наиболее эффективные приемы, применение которых упрощает сборку и развертывание приложений:

- *двигайтесь вперед короткими шажками.* Даже если вы опытный разработчик приложений EE, старайтесь идти вперед короткими шажками при разработке процедуры упаковки и развертывания приложения. Не стремитесь работать целый месяц над созданием сотен компонентов, а затем пытаться с первого раза упаковать и развернуть приложение. Возникающие проблемы проще решать, если двигаться вперед постепенно, короткими шагами;

- *используйте однородное серверное окружение*. Многие выбирают EE-сервер GlassFish, но существует масса других вариантов. Всегда, когда это возможно, старайтесь, чтобы все члены вашей команды использовали для разработки один и тот же сервер EE, одной и той же версии. Если разные разработчики вынуждены использовать разные версии серверов EE, отделяйте код от упаковки, чтобы избежать проблем при упаковке;
- *отделяйте код от конфигурации*. Часто приложения включают программный код и файлы конфигурации, что нередко вызывает проблемы при создании пакетов для окружений разработки, тестирования и эксплуатации, которые могут отличаться. Чтобы избежать этих проблем, создавайте отдельные проекты для упаковки с разными конфигурациями (дескрипторами развертывания!) в этих проектах. Если вы пользуетесь инструментами сборки, такими как Maven, объявление зависимостей и комбинирование проектов облегчится. Кому-то создание разных проектов для упаковки приложения под разные окружения может показаться напрасной тратой сил, но это не так – в долгосрочной перспективе этот прием поможет вам сэкономить массу времени на разрешении проблем, связанных с упаковкой;
- *внимательно изучите приложение и его зависимости*. Убедитесь, что все ресурсы настроены, прежде чем пытаться развертывать приложение в целевом окружении. Если приложение требует большого количества ресурсов, подумайте о возможности использования дескриптора развертывания, чтобы с его помощью сообщить о зависимостях тому, кто будет заниматься развертыванием. Неправильная упаковка классов и библиотек вызывает массу проблем при загрузке классов. Поэтому вам так же следует разобраться с зависимостями от вспомогательных классов и сторонних библиотек, и упаковать их соответствующим образом. Избегайте дублирования библиотек в разных местах. Вместо этого найдите такой вариант упаковки приложения и настройки сервера приложений, который позволит множеству модулей в приложении совместно использовать общие библиотеки;
- *старайтесь не использовать нестандартные API и аннотации*. Не используйте нестандартные элементы и аннотации, реализованные производителем конкретного сервера приложений, если только они не являются единственным способом решения ваших задач. Тщательно оценивайте, настолько ли велики преимущества, чтобы ради них можно было пожертвовать переносимостью. Если вы зависите от каких-то нестандартных особенностей, проверьте – есть ли возможность воспользоваться нестандартным дескриптором развертывания;
- *используйте опыт и знания вашего администратора баз данных*. Привлекайте вашего администратора баз данных к работе над автоматизацией создания схемы базы данных для вашего приложения. Не полагайтесь на механизмы автоматического создания таблиц для сущностей, потому что результаты их работы могут не соответствовать вашим требованиям. Убедитесь, что база данных настроена правильно и не является узким местом для вашего приложения. По своему опыту мы можем смело утверждать,

что дружба с администратором баз данных окажется существенным подспорьем для ваших проектов! Если вашему приложению требуются другие ресурсы, такие как провайдеры JMS или LDAP, обращайтесь к соответствующим администраторам за помощью в их настройке. И снова, использование файлов XML для описания объектно-реляционных отображений и зависимостей от ресурсов поможет вам решать проблемы, связанные с настройками, без необходимости вмешиваться в программный код;

- *используйте инструменты сборки*. Наиболее вероятно, что для сборки своих приложений вы будете использовать Maven, но какие бы инструменты вы не использовали, вы должны знать, как ими пользоваться, и понимать все их преимущества. Старайтесь не выполнять вручную никаких операций в процессе сборки приложения. Если вдруг обнаружится, что какие-то этапы приходится выполнять вручную, это может служить сигналом, что либо вы используете не все возможности инструмента, либо выбор инструмента не соответствует вашим задачам и и его следует заменить чем-то другим, лучше соответствующим вашим потребностям.

Итак, мы познакомились с наиболее эффективными приемами, но как быть, если их все же оказывается недостаточно? Ниже мы раскроем вам несколько секретов из сокровищницы, которые помогут вам в решении проблем, связанных с развертыванием.

13.6.2. Решение типичных проблем развертывания

В этом разделе описываются некоторые типичные проблемы, возникающие на этапе развертывания, с которыми вы можете столкнуться. Большинство из них решается проаильной сборкой приложения:

- `ClassNotFoundException` – это исключение возникает при попытке динамически загрузить ресурс, который не может быть найден. Причиной этого исключения может быть отсутствие библиотеки на нужном уровне загрузки – файла JAR, содержащего класс, который не удалось найти. Если вы загружаете в своем приложении ресурс или файл свойств, убедитесь, что используете `Thread.currentThread().getContextClassLoader().getResourceAsStream()`;
- `NoClassDefFoundException` – это исключение возбуждается при попытке создать объект или когда зависимости прежде загруженного класса не могут быть разрешены. Обычно эта проблема возникает, когда не все зависимые библиотеки загружаются одним и тем же загрузчиком классов;
- `ClassCastException` – это исключение обычно является результатом дублирования классов на разных уровнях. Оно возникает в ситуациях, когда один и тот же класс загружается загрузчиками разных уровней; то есть, при попытке выполнить приведение к классу, загруженному загрузчиком классов L1, экземпляра того же класса, загруженного загрузчиком классов L2;

- `NamingException` – это исключение обычно возбуждается при неудачной попытке найти объект в JNDI, когда контейнер пытается внедрить в компонент EJB несуществующий ресурс. Трассировка стека этого исключения поможет узнать, какой именно объект не удалось найти. Обязательно проверяйте, все ли зависимости от источников, компонентов EJB и других ресурсов разрешаются должным образом;
- `NotSerializableException` – это исключение возбуждается, когда возникает необходимость преобразовать некоторый объект, хранящийся в памяти, в некоторое представление в виде `byte[]`, но он не поддерживает такое преобразование. Такое может случиться при попытке пассивировать (passivate) сеансовый компонент без сохранения состояния и сохранить его на диске для освобождения оперативной памяти или, если обращение к сеансовому компоненту выполняется удаленным клиентом и возвращаемые им объекты требуется передать по сети. Какова бы ни была причина, если объект не поддерживает сериализацию, вы получите это исключение. Лучший способ избежать его – добавить тест JUnit для проверки сериализуемости объектов. Обычно на первых порах объекты поддерживают возможность сериализации, но со временем разработчик может вносить в них изменения, приводящие к потере сериализуемости;
- попытка развернуть приложение может потерпеть неудачу из-за ошибки в дескрипторе развертывания. Проверьте соответствие дескрипторов схеме XML. Сделать это можно, попробовав собрать приложение с помощью интегрированной среды разработки (IDE), вместо того, чтобы вручную править XML-файлы дескрипторов.

13.7. В заключение

Искусство сборки и упаковки является одной из основ создания приложений Java EE. В этой главе мы узнали, как загружаются классы и как спецификация Java EE определяет зависимости между классами внутри корпоративных приложений. Мы узнали, как правильно упаковывать компоненты EJB в собственные модули EJB-JAR и в модули WAR, и как обеспечить включение в пакет дескриптора развертывания для модуля EJB-JAR. После этого мы перешли к упаковке сущностей JPA и компонентов CDI, для которых спецификация EE не определяет отдельные модули и потому они должны упаковываться в существующие модули EE. В заключение мы рассмотрели некоторые наиболее эффективные приемы упаковки, а так же типичные проблемы развертывания и способы их решения.

ГЛАВА 14. Использование веб-сокетов с EJB 3

Эта глава охватывает следующие темы:

- основы веб-сокетов;
- интеграция веб-сокетов с Java EE;
- создание конечных точек программно и с помощью аннотаций.

В этой главе мы погрузимся в изучение удивительной новой технологии, добавленной в Java EE 7 для поддержки HTML5: веб-сокеты (WebSockets). Веб-сокеты – это низкоуровневые сокеты поддерживающие по-настоящему двустороннюю связь между веб-браузером и сервером. Посредством веб-сокетов можно передавать данные веб-браузеру со стороны сервера и по его инициативе, не дожидаясь запросов со стороны клиента.

14.1. Ограничения схемы взаимодействий «запрос/ответ»

В традиционной модели взаимодействий по протоколу HTTP клиентский браузер открывает соединение с HTTP-сервером и запрашивает выполнение операции, такой как `GET`, `POST`, `PUT`, `DELETE` и других. HTTP-сервер выполняет операцию и возвращает результат, обычно в виде разметки HTML. Соединение с сервером может поддерживаться открытым для отправки дополнительных запросов, чтобы при необходимости получить несколько ресурсов с сервера не открывать дополнительные сокеты для передачи дополнительных запросов. Учитывая сложность некоторых современных веб-страниц, в этом есть определенный смысл – страницы с большим объемом содержимого, такие как eBay.com, оказались бы слишком медленными, если бы для загрузки каждого изображения требовалось выполнять полную процедуру открытия/закрытия сокета. Но в любом случае инициатором

в этих взаимодействиях является клиент – клиент выполняет запрос, а сервер отвечает на него. Сервер никогда не отправляет данные клиенту по своей инициативе.

Модель «запрос/ответ» прекрасно подходит для веб-сайтов с обычным информационным наполнением, таких как новостные сайты, электронные энциклопедии, академические журналы, хранилища файлов, и так далее. Но она совершенно не годится для игровых сайтов, сайтов обмена мгновенными сообщениями, сайтов мониторинга биржевых котировок и любых других сайтов, где требуется передавать информацию клиенту по инициативе сервера в асинхронном режиме. Парадигма «запрос/ответ» только создает дополнительные помехи для таких сайтов – клиент всегда вынужден посылать инициирующий запрос. Существуют разные подходы к решению этой проблемы.

Один из таких подходов, призванных преодолеть ограничения протокола HTTP, заключается в использовании расширений к браузерам, таких как Java Applets, Flash, ActiveX, и так далее. Используя эти расширения, можно делать почти все, что угодно, потому что имеется возможность открывать низкоуровневые сокеты для асинхронного получения потоковых данных. Но они имеют несколько существенных недостатков. Конечные пользователи должны устанавливать расширения в свои браузеры, и некоторые расширения, такие как ActiveX, не являются кросс-платформенными. Кроме того, многие мобильные браузеры имеют ограниченную поддержку или вообще не поддерживают расширения; например, Flash не поддерживается стандартным браузером в iOS. Расширения часто требуют значительного времени для загрузки и инициализации во время запуска браузера. И хотя для современных мощных компьютеров и высокоскоростных соединений с Интернетом это не такая большая проблема, она все же остается заметной. Интеграция расширений в страницы не всегда выполняется чисто и может быть выполнена с ошибками. Но самый большой недостаток – все эти технологии не используют стандартные веб-протоколы и его обработка на сервере должна быть написана вручную. Во многих корпоративных сетях используются прокси-серверы, которые, благодаря поддержке аутентификации и блокировки портов, составляют часть корпоративной системы безопасности. Такими серверами нередко блокируется весь не-HTTP трафик, поступающий из-за пределов компании, а это означает, что Java-апплеты не смогут напрямую взаимодействовать с сервером через порты, отличные от стандартных портов HTTP. Это лишь часть проблем, мешающих использованию решений на основе расширений для браузеров.

Преодолеть ограничения модели «запрос/ответ» призваны два других подхода: AJAX и Comet. В следующих разделах мы сравним каждый из них с веб-сокетами, но и они не дают полноценного решения. Технология AJAX позволяет выполнять асинхронные запросы со стороны клиента уже после загрузки страницы, чтобы получать части страниц или данные без полной перезагрузки страницы. Данные при этом часто возвращаются в формате записи объектов JavaScript – JSON (JavaScript Object Notation). Но клиенты, реализованные на основе AJAX, все еще вынуждены посылать инициирующий запрос. Чтобы создать ощущение, что сервер по собственной инициативе отправляет данные клиенту, во многих

веб-приложениях прежде использовался прием опроса с помощью механизмов AJAX, что неплохо подходит, например, для программ чтения электронной почты, но совершенно не годится для организации взаимодействий в режиме реального времени.

Технология Comet поддерживает возможность передачи данных по инициативе сервера. При ее использовании HTTP-соединение остается открытым и поддерживается открытым максимально долго, чтобы данные можно было асинхронно передавать непрерывным потоком. Comet связывает соединение в браузере. Так как этот прием не является стандартным, для него отсутствует стандартная серверная реализация, к тому же поддержка долгоживущих сетевых соединений может привести к конфликтам с корпоративными брандмауэрами.

Теперь, когда мы познакомились с некоторыми ограничениями модели «запрос/ответ», посмотрим на эту проблему с другой стороны – со стороны нового стандарта Java EE, через призму веб-сокетов.

14.2. Введение в веб-сокеты

Технология WebSockets – это решение для реализации в веб-приложениях по-настоящему асинхронных взаимодействий без применения расширений, приемов опроса на основе AJAX или нестандартных решений Comet. Она стандартизована и является частью стандарта HTML5, а это означает, что все браузеры, включая мобильные, должны поддерживать ее. Так как эта технология основана на использовании стандартного протокола HTTP, брандмауэры и браузеры отлично поддерживают ее. Технология WebSockets использует также преимущества Web Worker – рабочих потоков выполнения, которые являются еще одной новинкой, появившейся в стандарте HTML5. Технология Web Workers стала первой, обеспечивающей рудиментарную поддержку многопоточного выполнения в JavaScript, необходимую для организации асинхронных взаимодействий.

Для начала мы познакомимся поближе с технологией WebSockets, чтобы потом проще было разбираться с ее поддержкой в Java EE 7. Если вы уже владеете приемами использования веб-сокетов, можете сразу прейти к разделу, следующему далее.

Чтобы лучше понять суть технологии WebSockets, мы сначала исследуем ее в контексте HTML5, не касаясь серверного компонента. Мы познакомимся с основами протокола, а также с приемами взаимодействий с веб-сокетами из программного кода на JavaScript. После того, как вы получите представление о технологии с точки зрения HTML5, мы сравним ее с AJAX и Comet. Технология AJAX все еще остается востребованной и действительно полезной, чего нельзя сказать о Comet, так как в сравнении с ней технология WebSockets выглядит намного лучше.

14.2.1. Основы веб-сокетов

WebSockets – это новая технология транспортировки, основанная на использовании протокола HTTP. Сначала соединение открывается, как соединение HTTP

или HTTPS, а затем оно преобразуется в соединение WebSocket с использованием специальной процедуры. Обмен информацией через веб-сокеты происходит с использованием протоколов HTTP/HTTPS, поэтому есть возможность использовать сложившуюся инфраструктуру. Оба протокола, HTTP и HTTPS, отлично обслуживаются серверным программным обеспечением и промежуточной инфраструктурой, такой как брандмауэры и прокси-серверы. Таким образом, веб-сокеты оказываются обратно совместимыми с существующей сетевой архитектурой. Вам не придется открывать дополнительные порты в настройках брандмауэров или изменять настройки прокси-серверов.

Чтобы открыть веб-сокет в браузере, необходимо передать конструктору веб-сокета URL с использованием схемы WebSocket. Ниже приводится фрагмент кода на JavaScript из приложения ActionBazaar, открывающий соединение со службой чата:

```
var webSocket = new WebSocket('ws://127.0.0.1:8080/chapter14/chat');
```

Адрес URL начинается с префикса `ws:`, сообщающего браузеру, что он должен использовать протокол WebSocket. Чтобы установить защищенное соединение, следует указать префикс `wss:`, но для этого сервер должен иметь подписанный сертификат SSL. В листинге 14.1 показано, как выглядит запрос, отправляемый браузером.

Листинг 14.1. HTTP-запрос на открытие соединения по протоколу WebSocket

```
GET /chapter14/chat HTTP/1.1    ← ❶ Стандартный заголовок запроса HTTP GET
Upgrade: websocket              ← ❷ Требование выполнить переход
Connection: Upgrade                 на протокол WebSocket
Host: 127.0.0.1:8080
Origin: http://localhost:8040
Cookie: JSESSIONID=df1d67da4bc7eae66f50a05a9d05 ← ❸ Ключ сеанса Java EE
Sec-WebSocket-Key: ddrh4Ti+hXSYfRS14HJodA==     ← Идентификатор для проверки
Sec-WebSocket-Version: 13     ← ❹ Версия спецификации WebSocket
Sec-WebSocket-Extensions: x-webkit-deflate-frame ← ❺ Дополнительные настройки
HTTP Handshake Request                              сжатия, поддерживаемые
                                                    браузером
```

В листинге 14.1 показан стандартный HTTP-запрос `GET` ❶. Как видите, клиент запрашивает преобразование HTTP-соединения в соединение WebSocket ❷. Клиент передает ключ, для использования сервером в ответах ❸, и идентификатор, по которому будет проверяться – действительно ли HTTP-соединение было преобразовано в соединение WebSocket. Дополнительно серверу передаются: версия ❹ и параметры оптимизации сжатия ❺.

В ответ на этот запрос сервер возвращает подтверждение, сообщающее, что соединение было преобразовано и клиент может приступать к передаче дополнительных данных; сервер так же может передавать данные через это соединение, не ожидая запросов со стороны клиента. Ответ сервера показан в листинге 14.2.

Листинг 14.2. HTTP-ответ, подтверждающий открытие соединения по протоколу WebSocket

```
HTTP/1.1 101 Web Socket Protocol Handshake ← HTTP 101 - обозначение протокола
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open Source Edition 4.0
Java/Oracle Corporation/1.7)
Server: GlassFish Server Open Source Edition 4.0
Connection: Upgrade
Sec-WebSocket-Accept: 5+Xng8HF5TYinHaBSIR1kt4n0vA= ← Ключ, подтверждающий
                                                     преобразование протокола
Upgrade: websocket ← Признак успешного преобразования в протокол WebSocket
```

После того, как соединение будет преобразовано, клиент и сервер могут приступать к обмену сообщениями. Содержимое сообщений определяется приложениями. Это могут быть двоичные данные, XML, простой текст, JSON, и так далее. В наши дни в большинстве случаев используется формат JSON, как наиболее компактный и простой для использования в сценариях JavaScript. Формат XML, несмотря на то, что он хорошо документирован, слишком расточителен и требует дополнительных накладных расходов на транспортировку и обработку. Эти накладные расходы непосредственно сказываются на производительности, наиболее ощутимые на мобильных устройствах.

Чтобы отправить данные из JavaScript на сервер, необходимо вызвать метод `send` объекта `WebSocket`. Существует несколько разных способов отправки данных, в зависимости от их типа:

```
void send(DOMString data);
void send(Blob data);
void send(ArrayBuffer data);
void send(ArrayBufferView data);
```

В этой главе мы в основном сосредоточимся на отправке текстовых данных в формате JSON, но точно так же можно передавать двоичные данные. Текст передается в виде последовательности символов Юникода – символы в других кодировках, преобразуются в символы Юникода. Структура сообщения может быть любой – вы вольны выбрать любой формат, лучше всего соответствующий вашим потребностям. Однако чаще всего вам придется пользоваться форматом JSON. JSON – это компактный формат, и прекрасно подходящий для использования в JavaScript. Кроме того, в Java EE 7 поддерживается возможность непосредственного преобразования JSON в объекты Java с использованием специального API для обработки JSON (JSR-353). Рассмотрим простой пример отправки сообщения в формате JSON через веб-сокет, как показано в листинге 14.3.

Листинг 14.3. Отправка данных в формате JSON с использованием веб-сокетов

```
// Создать объект JavaScript и заполнить его
var msg = {
    type: "message",
    text: "Hello World",
    date: Date.now()
};
```

```
// Отправить объект JavaScript
webSocket.send(JSON.stringify(msg));
```

Здесь создается объект JavaScript `msg` с несколькими свойствами: `type`, `text` и `date`.

Отправка двоичных данных может пригодиться, например, для выгрузки больших файлов на сервер. Метод `send` способен передавать данные, только если веб-сокет открыт; в противном случае будет сгенерирована ошибка. В случае ошибки будет вызван обработчик `onerror`, о котором рассказывается ниже.

Кроме передачи данных на сервер, веб-сокеты могут также принимать данные. Для этого объекты `WebSocket` поддерживают несколько обработчиков:

```
webSocket.onopen = function(evt) { /* обработчик события открытия сокета */};
webSocket.onclose = function(evt) { /* обработчик события заткрытия сокета */};
webSocket.onmessage = function(evt) { /* обработчик приема сообщения */ };
webSocket.onerror = function(evt) { /* обработчик ошибки */ };
```

Эти обработчики охватывают все события жизненного цикла объектов `WebSocket`. Первый обработчик, `onopen`, вызывается, когда устанавливается соединение с сервером и веб-сокет становится готовым к отправке/приему данных. Метод `onclose` вызывается после закрытия соединения. Соединение может быть закрыто с любой из сторон, сервера или клиента. Обработчик `onerror` вызывается в случае обнаружения ошибки, как, например, попытка отправки данных в закрытое соединение. Обработчик `onmessage` вызывается, когда приходит сообщение с сервера. Сервер может отправить сообщение клиенту в любой момент – это не обязательно будет ответ на запрос, как это происходит при использовании технологии AJAX. Содержимое сообщения может быть объектом типа `ArrayBuffer`, `Blob` или `String`. Получить содержимое сообщения можно обращением к свойству `data` параметра `evt`.

Чаще всего данные передаются в формате JSON. Это намного более простой и компактный формат, чем XML. Его проще анализировать и он генерирует меньший объем трафика. Эти две характеристики особенно важны для мобильных устройств. Формат JSON имеет две основные структуры: пары имя/значение и упорядоченный список значений. В листинге 14.3 показано, как выглядит формат простого сообщения для чата в приложении ActionBazaar.

Листинг 14.4. Пример сообщения для чата ActionBazaar

```
{
    "type": "ChatMessage",
    "user": "admin",
    "message", "I've forgotten my password"
}
```

В этом листинге представлено очень простое сообщение для чата. Это – массив из трех свойств, или значений. Это сообщение гораздо компактнее, чем эквивалентное сообщение в формате XML:

```
<chat-message>
  <type>ChatMessage</type>
```

```
  <user>admin</user><message>![CDATA[I've forgotten my password]]></message>
</chat-message>
```

Объем сообщения в формате JSON составляет 79 символов, против 126 символов в формате XML. То есть при переходе от формата JSON к формату XML объем сообщения увеличился на 59% и этот объем необходимо передавать и размещать в памяти. С другой стороны формат XML намного очевиднее и в сочетании с XML Schema сообщения в этом формате легко могут быть проверены. Это невозможно при использовании формата JSON, хотя определенные наработки в этом направлении все-таки существуют.

Сценарий на JavaScript легко сможет преобразовать сообщение из листинге 14.4 в объект и обратиться к его свойствам, как показано в листинге 14.5, где приводится реализация обработчика событий `onMessage`.

Листинг 14.5. Преобразование сообщения JSON в объект JavaScript

```
function onMessage(evt) { // ❶ Обработчик, вызываемый веб-сокетом
    var msg = eval('(' + evt.data + ')'); // Преобразовать сообщение в объект
    console.log('Type: ' + msg.type);     // Вывести свойство объекта
    console.log('User: ' + msg.user);
    console.log('Message: ' + msg.message);
}
```

В этом листинге представлен обработчик, которому передается принятое сообщение в формате JSON ❶. Чтобы извлечь данные из сообщения, его необходимо преобразовать в объект JavaScript. После этого можно просто обращаться к свойствам полученного объекта. Далее в этой главе мы покажем, как выполняется преобразование сообщений JSON в Java-объекты. Поддержка JSON была добавлена в Java EE 7 в виде JSR-353.

Итак, мы рассказали вам об основных особенностях веб-сокетов. Теперь давайте сравним веб-сокеты с технологиями, существовавшими до появления стандарта HTML5. Ни одна из них не была полностью стандартизована.

14.2.2. Веб-сокеты и AJAX

AJAX (Asynchronous JavaScript and XML – асинхронный JavaScript и XML) – это довольно старый способ обмена данными между клиентом и сервером. Как и при использовании веб-сокетов, клиент инициирует обмен отправкой запроса AJAX. Но, в отличие от веб-сокетов, запрос отправляется асинхронно – клиент посылает сообщение серверу, а сервер отвечает на него. После того, как клиент примет сообщение от сервера, соединение закрывается. С точки зрения пользователя обмен сообщениями выполняется асинхронно, потому что запрос AJAX отправляется отдельным потоком выполнения, не «замораживая» пользовательский интерфейс.

Технология AJAX часто используется, чтобы избежать необходимости полной перезагрузки страницы. Рассмотрим в качестве примере получение страницы с информацией о доставке в приложении ActionBazaar. Когда ставка пользователя выигрывает, он должен указать свой адрес и выбрать способ доставки, напри-

мер, воспользовавшись услугами FedEx, UPS или USPS[1]. Стоимость доставки зависит от конечного адреса. Чтобы избежать необходимости перехода на другую веб-страницу, и, соответственно, полной загрузки новой страницы, приложение выполняет запрос AJAX, чтобы получить примерную стоимость доставки после ввода допустимого почтового индекса. Это уменьшает нагрузку на сервер, снижает объем передаваемых данных и увеличивает удобство процедуры заполнения формы с информацией о доставке.

Несмотря на то, что в аббревиатуре AJAX присутствует символ X, расшифровывающийся как XML, вместо формата XML может использоваться формат JSON. Технология AJAX позволяет передавать данные в формате JSON, XML и даже HTML. Все взаимодействия выполняются по протоколу HTTP или HTTPS. С точки зрения сервера, запрос AJAX выглядит как любой другой HTTP-запрос.

Основой технологии AJAX является объект `XMLHttpRequest`. Это встроенный объект JavaScript, обладающий методами для открытия соединения, отправки данных и так далее. Пример использования этих методов приводится в листинге 14.6.

Листинг 14.6. Отправка запроса AJAX из JavaScript

```
var request = new XMLHttpRequest();        // ❶ Создать объект XMLHttpRequest
request.onreadystatechange = function() { // ❷ Установить обработчик
    if(request.readState == 4 &&
        request.status = 200) {            // ❸ Проверить успех HTTP-вызова
            // обработка request.responseXML;
        }
    }
}
request.open('GET',url); // ❹ Подготовить HTTP-запрос GET
request.send();          // ❺ Послать запрос серверу
```

Код в этом листинге демонстрирует основные этапы выполнения запросов AJAX. Сначала создается объект `XMLHttpRequest` ❶, а затем устанавливается обработчик ответа сервера ❷. Когда сервер пришлет ответ, будет вызвана функция-обработчик, она проверит успешность выполнения запроса ❸. Чтобы выполнить вызов, необходимо сначала подготовить сам запрос ❹, а затем выполнить его отправку ❺.

Java Server Faces и AJAX

Реализация JSF предоставляет методы-обертки для упрощения вызовов AJAX. Если вы используете JSF в качестве технологии отображения, вы почти наверняка будете использовать тег `<f:ajax/>` для выполнения вызовов AJAX. Этот тег осуществляет вызов метода компонента Java на сервере. JSF-тег помогает легко интегрировать поддержку AJAX в приложение без необходимости осуществлять рутинные операции вручную. Кроме того, инфраструктура JSF AJAX автоматически обновляет другие компоненты JSF, избавляя от необходимости писать код, выполняющий итерации по дереву DOM.

[1] Известные в США компании, специализирующиеся на доставке почтовых отправлений. – *Прим. перев.*

Программный код JavaScript в этом листинге мало чем отличается от программного кода, использующего веб-сокет, представленного в листинге 14.3. Основные отличия заключаются в доступных функциональных возможностях. Технология AJAX использует модель «запрос/ответ». Она не предусматривает для сервера возможности передавать сообщения клиенту по собственной инициативе, без запроса со стороны клиента. Например, в реализации чата с применением технологии AJAX клиент должен постоянно проверять появление новых сообщений на сервере. Этот лишний сетевой трафик увеличивает нагрузку на сервер, замедляя работу чата. Технология AJAX отлично подходит в случаях, когда клиенту требуется извлекать данные с сервера для выполнения проверок или частичной загрузки страниц. Она не годится в ситуациях, когда сервер может генерировать данные асинхронно или когда требуется передать серверу несколько сообщений до того, как будет получен ответ.

Технология Comet является родственной к технологии AJAX. Ее основная цель – реализовать WebSocket-подобную функциональность с применением технологии AJAX.

14.2.3. Веб-сокеты и Comet

Подход на основе AJAX, обсуждавшийся в предыдущем разделе, отлично подходит в ситуациях, когда клиенту требуется запросить информацию с сервера, но он не годится, когда информацию необходимо передать по инициативе сервера. Его можно использовать для опроса сервера, но это далеко не идеальное решение по множеству причин, в том числе из-за увеличения трафика и неизбежных задержек. Comet – это не стандартное решение, а скорее уловка, основанная на использовании существующих веб-технологий, чтобы дать серверу возможность отправлять данные клиенту по своей инициативе, в пределах ограничений HTTP 1.1. Одним из существенных ограничений HTTP 1.1 является отсутствие поддержки долгоживущих соединений. Это одна из проблем, которые решают веб-сокеты.

Существует несколько подходов к реализации Comet, которые условно можно разделить на две группы: на основе потоковой передачи данных и опросов с ожиданием. Об этом детально рассказывается в статье на IBM developerWorks[2]. В статье подробно описывается, как действуют эти реализации и приводятся примеры кода. Далее мы расскажем в общих чертах о двух разных подходах и сравним их с технологией WebSockets. Мы не будем подробно обсуждать реализации решения Comet, так как это далеко выходит за рамки данного раздела, а кроме того, после сравнения вы наверняка отдадите предпочтение веб-сокетам.

Прием потоковой передачи данных (streaming) основан на создании долгоживущего HTTP-соединения и передаче сообщений со стороны сервера через него. Существует два способа создания таких долгоживущих соединений: Forever IFrames (на основе скрытых фреймов `<iframe>`) и объект `XMLHttpRequest` с составными сообщениями (multipart XMLHttpRequest).

---

[2] Матье Карбу (Mathieu Carbou). «Reverse Ajax: Часть 1. Знакомство с концепцией Comet.», IBM developerWorks, 19 июля 2011, http://www.ibm.com/developerworks/ru/library/wa-reverseajax1/.

При использовании реализации Forever IFrames на страницу помещается скрытый фрейм – тег `<iframe>` – с атрибутом `src`, запрашивающим содержимое. Когда сервер получает запрос для этого фрейма, он сохраняет соединение открытым и передает через него код, заключенный в теги `<script></script>`, с фактическим сообщением. Браузер выполняет код, который возвращает содержащееся в нем сообщение.

Другой подход к потоковой передаче данных, на основе объекта `XMLHttpRequest` с составными сообщениями, заключается в создании на стороне клиента объекта `XMLHttpRequest` с взведенным флагом `multipart`. Соединение в этом случае остается открытым, и сервер передает каждое сообщение как часть составного ответа. Если вы знакомы с особенностями отправки форм с составным содержимым – здесь, по сути, происходит то же самое, только наоборот.

Оба способа – Forever IFrames и объект `XMLHttpRequest` с составными сообщениями – реализуют потоковую передачу данных. Другой способ реализации технологии Comet – опрос с ожиданием. В этой реализации клиент выполняет запрос AJAX, а сервер удерживает соединение, пока не появится ответ. После передачи ответа клиенту соединение закрывается. Затем клиент выполняет следующий запрос AJAX, для организации ожидания следующего сообщения.

Оба способа – потоковая передача данных и опрос с ожиданием – фактически являются грубыми приемами. Метод Forever IFrames не дает простой возможности обнаружения ошибок и восстановления после них. Метод на основе объектов `XMLHttpRequest` с составными сообщениями поддерживается не всеми браузерами. В опросе с ожиданием есть риск потери сообщений. Представьте ситуацию, когда сервер сгенерировал новое сообщение после того, как отправил клиенту предыдущее и закрыл соединение, но до того, как клиент успел открыть следующее соединение. Сервер не может знать, восстановит ли клиент соединение или отключился навсегда, поэтому он не может сохранить сообщение в кэше. Применение объекта `XMLHttpRequest` – для реализации метода на основе составных сообщений или опроса с ожиданием – связывает соединение с сервером, в том смысле, что нигде на этой странице уже нельзя будет выполнить запрос AJAX. Кроме того, могут возникать проблемы с прокси и при выполнении запросов к поддоменам.

Веб-сокеты предоставляют фактически ту же функциональность, что и Comet, только без осложнений. С появлением веб-сокетов отпала необходимость использовать скрытые фреймы или писать код JavaScript для обработки составных ответов. Благодаря веб-сокетам, клиент и сервер могут одновременно посылать друг другу данные в простом текстовом виде, в формате JSON, XML, двоичном или любом другом. Существует четко определенный JavaScript API, поддерживаемый большинством браузеров.

Одно из самых больших отличий технологии WebSockets от Comet заключается в поддержке на стороне сервера. Спецификация Java EE 7 включает стандартное определение WebSockets для платформы Java, в том числе и прикладной интерфейс, который, по сути, повторяет интерфейс клиентов JavaScript. Для технологии Comet отсутствует стандартный серверный API, отчасти потому, что сама техно-

логия Comet не была стандартизована. В следующих двух разделах мы займемся исследованием Java EE WebSocket API.

14.3. Веб-сокеты и Java EE

Теперь, когда вы получили общее представление о технологии WebSockets и о том, как она соотносится с другими технологиями, можно переходить к изучению API. Как уже говорилось выше, поддержка веб-сокетов в Java определяется спецификацией JSR-356. Эта спецификация определяет клиентский и серверный API. Клиентский API может использоваться в любых программах на языке Java, что дает им возможность взаимодействовать с конечными точками веб-сокетов. Серверный API доступен только в контейнерах Java EE.

Реализация WebSocket API доступна в виде двух пакетов: `javax.websocket.*` и `javax.websocket.server.*`, для клиента и сервера, соответственно. Основное внимание мы будем уделять серверному API.

Прежде чем погрузиться в исследование серверного API, уделим несколько минут обсуждению поддержке многопоточности и безопасности. В отличие от сервлетов, для каждого клиента, инициировавшего соединение, создается своя конечная точка веб-сокета. Соответственно конечная точка хранит информацию о своем состоянии и обслуживает единственного клиента. Такое поведение можно изменить, реализовав объект `ServerEndpointConfi.Configurator` с методом `getEndPointInstance()`. Этот метод возвращает экземпляр конечной точки веб-сокета для использования с новым соединением.

Существует две важные проблемы, связанные с безопасностью, которые следует учитывать при работе с веб-сокетами: ограничение доступа для неаутентифицированных пользователей и шифрование трафика. Конечная точка веб-сокета определяется адресом URL и, соответственно, ее безопасность зависит от настроек в *web.xml*. Например, в листинге 14.7 представлены настройки конечной точки веб-сокета для службы поддержки клиентов. Серверная конечная точка настраивается помощью аннотации `@ServerEndpoint(value="/support")`.

Листинг 14.7. Настройки безопасности для конечной точки веб-сокета

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>
      Limit CSR
    </web-resource-name>
    <!-- ❶ Ограничить возможность подключениея адресом /support/* -->
    <url-pattern>/support/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <!-- ❷ Разрешается подключаться только пользователям с ролью CSR -->
    <role-name>csr</role-name>
  </auth-constraint>
</security-constraint>
```

Настройки в этом листинге разрешают доступ к конечной точке веб-сокета только для запросов по адресу `/support` ❶ и только аутентифицированным пользователям с ролью CSR ❷. Это не является абсолютной защитой конечной точки. С настройками по умолчанию все данные передаются через конечную точку веб-сокета в открытом текстовом виде. Любой, воспользовавшись программой перехвата трафика, сможет читать передаваемую информацию и даже внедрять свои пакеты. Для обеспечения безопасности обмена необходимо использовать защищенные сокеты. Для этого клиент должен открывать соединение с применением префикса `wss:` вместо `ws:`. В результате этого вместо соединения HTTP будет открыто соединение HTTPS. Чтобы задействовать защищенные сокеты, в настройки из листинга 14.7 следует добавить следующий фрагмент:

```
<user-data-constraint>
  <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
```

Теперь, после знакомства с основными настройками безопасности веб-сокетов, рассмотрим два типа конечных точек.

14.3.1. Конечные точки веб-сокетов

Существует два типа конечных точек веб-сокетов: создаваемые программно и с помощью аннотаций.

Конечные точки, создаваемые программно

Конечные точки, создаваемые программно, делят ответственность за обработку событий жизненного цикла соединения и сообщений между разными классами. Программная конечная точка должна наследовать класс `javax.websocket.Endpoint` для реализации обработки начального запроса на соединение. В нем так же предусмотрены методы обратного вызова для обработки ошибок и события закрытия соединения. После открытия соединения, с помощью `javax.websocket.MessageHandler` можно зарегистрировать экземпляры для обработки входящих сообщений.

Несмотря на то, что создание конечной точки программным способом выглядит сложнее, чем с применением аннотаций, они обеспечивают большую гибкость с архитектурной точки зрения. Обработчики сообщений могут одновременно использоваться несколькими конечными точками, из чего следует, что они не привязаны к конкретным URI.

Конечные точки, создаваемые с помощью аннотаций

Конечные точки, создаваемые с помощью аннотаций, как можно догадаться, определяются и настраиваются с применением аннотаций. Аннотированной конечной точкой может быть любой класс Java; в отличие от программных конечных точек, аннотированные конечные точки не должны наследовать какие-либо классы или интерфейсы. С функциональной точки зрения между программными

и аннотированными конечными точками нет никаких различий, поэтому выбор между ними делается исключительно из архитектурных соображений. В аннотированных конечных точках методы обработки событий и сообщений определяются с помощью аннотаций.

Использование веб-сокетов в EJB

Существует два способа интеграции веб-сокетов с компонентами EJB: экспортирование компонента EJB в виде конечной точки веб-сокета и вызов компонента из конечной точки. Первый способ недостаточно хорошо определен в спецификации Java EE 7. Несмотря на то, что концептуально конечная точка веб-сокета может быть сеансовым компонентом с сохранением состояния или компонентом-одиночкой, явная поддержка таких конечных точек отсутствует. Некоторые контейнеры, такие как GlassFish 4.0, использующие реализацию Tyrus WebSocket, поддерживают возможность экспортирования компонентов-одиночек в виде конечных точек веб-сокетов, но эта возможность до сих пор не стандартизована. В других контейнерах ситуация с поддержкой может отличаться.

Второй способ поддерживается всеми контейнерами. С помощью аннотации `@EJB` можно внедрять ссылки на компоненты EJB в конечные точки или обработчики сообщений. Этот прием выглядит гораздо лучше с архитектурной точки зрения, потому что обработчики сообщений веб-сокетов обычно не соответствуют отдельным прикладным методам. Веб-сокеты в корне отличаются от таких технологий, как JAX-WS, которые имеют четко определенные сообщения и обычно используются для экспортирования прикладной функциональности для использования внешними системами.

Теперь, когда мы добавили в копилку знаний еще ряд базовых сведений, рассмотрим интерфейс `Session`, перед тем, как погрузиться в тонкости конечных точек, создаваемых программно или с помощью аннотаций.

14.3.2. Интерфейс Session

Интерфейс `javax.websocket.Session` является одним из самых важных для классов веб-сокетов. Этот интерфейс используется обеими разновидностями конечных точек, создаваемыми программно или с помощью аннотаций. Он представляет соединение клиента с сервером и служит абстракцией низкоуровневого сетевого сокета. Работая с веб-сокетами, вы не сможете получить доступ к соответствующему экземпляру `java.net.Socket`; интерфейс `Session` является ближайшим его заменителем.

Экземпляр интерфейса `Session` создается в момент, когда клиент благополучно устанавливает соединение с сервером. Этот экземпляр имеет множество удобных методов, включая методы отправки сообщений обратно клиенту, сохранения состояния и получения дополнительной информации о соединении. В зависимости от разновидности конечной точки, программной или аннотированной, можно получить ссылку на сеанс или на обработчики `onOpen` и `onMessage`, соответственно. Интерфейс `Session` предоставляет также ряд других вспомогательных методов.

Отправка сообщений

Интерфейс веб-сокетов содержит два метода, которые можно использовать для отправки сообщений клиенту:

- `getBasicRemote()` – отправляет сообщения синхронно;
- `getAsyncRemote()` – отправляет сообщения асинхронно.

Выбор метода зависит от способа отправки сообщения – синхронно или асинхронно. Метод асинхронной отправки возвращает экземпляр `RemoteEndpoint.Async`, а метод синхронной отправки – экземпляр `RemoteEndpoint.Basic`. Оба реализуют интерфейс `RemoteEndpoint`, определение которого приводится в листинге 14.8.

Листинг 14.8. Интерфейс RemoteEndpoint

```
public interface RemoteEndpoint {
    // ❶ Интерфейс для отправки асинхронных сообщений
    public static interface Async extends RemoteEndpoint {
        long getSendTimeout();
        void setSendTimeout(long l);
        void sendText(String string, SendHandler sh);
        Future<Void> sendText(String string);
        Future<Void> sendBinary(ByteBuffer bb);
        void sendBinary(ByteBuffer bb, SendHandler sh);
        Future<Void> sendObject(Object o);
        void sendObject(Object o, SendHandler sh);
    }
    // ❷ Интерфейс для отправки синхронных сообщений
    public static interface Basic extends RemoteEndpoint {
        void sendText(String string) throws IOException;
        void sendBinary(ByteBuffer bb) throws IOException;
        void sendText(String string, boolean bln) throws IOException;
        void sendBinary(ByteBuffer bb, boolean bln) throws IOException;
        OutputStream getSendStream() throws IOException;
        Writer getSendWriter() throws IOException;
        void sendObject(Object o) throws IOException, EncodeException;
    }
    void setBatchingAllowed(boolean bln) throws IOException;
    boolean getBatchingAllowed();
    void flushBatch() throws IOException;
    void sendPing(ByteBuffer bb) throws IOException,IllegalArgumentException;
    void sendPong(ByteBuffer bb) throws IOException,IllegalArgumentException;
}
```

Мы не будем погружаться в особенности каждого метода, но отметим, что асинхронный ❶ и синхронный ❷ интерфейсы обладают одинаковыми основными возможностями. Оба могут отправлять текст, массивы байтов и объекты Java. Когда выполняется отправка объекта, для его преобразования в представление, которое может быть передано через веб-сокет, используются кодерами (encoders), зарегистрированные в конечной точке.

В приложении ActionBazaar отправка сообщений осуществляется с помощью асинхронного интерфейса, как показано ниже:

```
ChatMessage cm = new ChatMessage(username,message);
csrSession.getAsyncRemote().sendObject(cm);
```

Синхронная и асинхронная отправка сообщений

Кого-то из вас может заинтересовать вопрос выбора способа отправки сообщений: синхронного или асинхронного. В большинстве случаев предпочтительнее использовать асинхронный способ. Синхронная отправка сообщений замедляет приложение из-за необходимости ожидания реакции медленного клиента. Если приложение осуществляет обход списка объектов `Session` и выполняет массовую рассылку сообщений, один медленный клиент замедлит обслуживание всех клиентов. Кроме того, если синхронная отправка сообщения осуществляется в середине транзакции, это приведет к блокировке ресурса до окончания передачи сообщения. Поэтому старайтесь избегать блокирования доступа к таблицам базы данных из-за медленных соединений. Использование синхронного способа отправки влечет ухудшение масштабируемости приложения.

Синхронные сообщения не могут служить решением проблем, связанных с конкуренцией. Проектируйте серверные и клиентские приложения с прицелом на использование асинхронного способа. Помните, что клиент может иметь медленное соединение посредством сотовой связи в формате 2G.

Закрытие соединения

После завершения использования веб-сокета важно закрыть его. Веб-сокеты могут находиться в открытом состоянии очень долго, если позабыть о них, и потребляют немало ресурсов, даже при небольшом объеме передаваемых данных. Закрыть веб-сокет можно с помощью двух методов объекта `Session`:

```
close() throws IOException;
close(CloseReason closeReason) throws IOException;
```

Первый метод `close` не имеет входных параметров и закрывает соединение с нормальным кодом состояния. Второй метод `close` позволяет указать причину закрытия соединения. В приложении ActionBazaar, когда производится завершение его работы или выполняется остановка сервера (в нормальном режиме), все открытые веб-сокеты закрываются и клиентам передается следующее сообщение для информирования пользователя:

```
session.close(new CloseReason(CloseReason.CloseCodes.GOING_AWAY,
    "Server is going down for maintenance."));
```

Получение основных сведений о веб-сокете

Интерфейс `Session` позволяет также получить информацию о текущем клиенте и типе соединения. Ниже перечислены некоторые, наиболее интересные методы:

- `getId()` – возвращает строку с уникальным идентификатором данного сеанса;
- `getPathParameters()` – возвращает параметры, переданные в URL;
- `getQueryString()` – возвращает строку запроса из URL (`?vacation=yes`);

- `getRequestParameterMap()` – возвращает строку запроса, преобразованную в ассоциативный массив;
- `getRequestURI()` – возвращает URI, для которого был открыт данный сеанс;
- `isOpen()` – возвращает `true`, если соединение остается действующим;
- `isSecure()` – возвращает `true`, если соединение использует защищенные сокеты;
- `getUserPrincipal()` – возвращает аутентифицированного пользователя или `null`;
- `getUserProperties()` – возвращает ассоциативный массив `Map<String, String>`, который можно использовать для хранения данных;
- `addMessageHandler()` – регистрирует обработчик сообщений для использования в конечных точках, созданных программно.

Если пользователь аутентифицировался в приложении посредством механизма HTTP-аутентификации, метод `getUserPrincipal()` вернет объект `Principal`. Этот метод используется в примере из приложения ActionBazaar, представленном в следующем разделе. Метод `getId()` можно использовать для идентификации конкретного сеанса; не забывайте, что один клиент может открыть множество сеансов (`getOpenSessions()` возвращает список сеансов), поэтому не следует полагать, что клиенту соответствует единственный сеанс, или пытаться идентифицировать конечного пользователя по IP-адресу.

Достоверность результата метода isOpen. *Используя метод* `isOpen()`, *помните, что его результат достоверен только в момент вызова. Возвращаемое им значение* `true` *не означает, что соединение будет открыто к моменту ближайшей попытки отправить сообщение. В этот момент клиент может выполнять процедуру закрытия соединения и спустя миллисекунду соединение может оказаться закрытым. Доверять результату метода* `isOpen()` *можно, только если он возвращает* `false`. *Не делайте никаких предположений на основании результата метода* `isOpen()`!

В следующем разделе мы познакомимся с кодерами и декодерами, которые потребуются для обработки сообщений.

14.3.3. Кодеры и декодеры

Интерфейс веб-сокетов, с которым вы познакомитесь далее в этой главе, позволяет регистрировать кодеры и декодеры, используемые для преобразования объектов Java в исходящие сообщения и входящих сообщений в объекты Java. Строго говоря, для реализации конечных точек веб-сокетов кодеры/декодеры не обязательны. Но, если только вы не предполагаете обрабатывать простые текстовые сообщения, не имеющие определенной структуры, или двоичные данные, кодеры и декодеры позволят отделить обработку сообщений от их кодирования/декодирования.

Прикладной интерфейс JAVA для обработки формата JSON. *Хотя это и выходит за рамки данного раздела, тем не менее, отметим, что при реализации кодеров и декодеров вы почти наверняка будете использовать API, определяемый спецификацией*

SR-353 и являющийся частью стандарта Java EE 7. Этот API упрощает создание и анализ данных в формате JSON; он широко используется в примерах программного кода в этой книге.

Для начала познакомимся с декодерами.

Декодеры

Декодер вызывается перед тем, как сообщение будет передано для обработки вашему коду. Декодер должен реализовать один из вложенных интерфейсов в `javax.websocket.Decoder`. На выбор имеется несколько таких интерфейсов:

- `TextStream` – используется для обработки потоков (`java.io.Reader`);
- `Text` – используется для обработки строк, целиком размещенных в памяти;
- `BinaryStream` – используется при обработке с использованием `InputStream`;
- `Binary` – анализируемое сообщение представлено объектом `ByteBuffer`.

Выбор интерфейса зависит от типа передаваемых данных. Если данные поступают в формате JSON, для реализации их декодирования используется интерфейс `TextStream` или `Text`. Для реализации декодера двоичных данных, таких как изображения, используется интерфейс `BinaryStream` или `Binary`. В листинге 14.9 представлен декодер `CommandMessageDecoder`.

Листинг 14.9. Декодер CommandMessageDecoder

```
public class CommandMessageDecoder
    // ❶ Наследует интерфейс Decoder.Text
    implements Decoder.Text<AbstractCommand> {

    // ❷ Инициализация декодера
    @Override
    public void init(EndpointConfig config) {}

    // ❸ Преобразует текст сообщения в объект AbstractCommand
    @Override
    public AbstractCommand decode(String message) throws DecodeException {
        ...
    }

    // ❹ Возвращает true, если декодер в состоянии разобрать текст
    @Override
    public boolean willDecode(String message) {
        ...
    }

    // ❺ Освобождает любые использовавшиеся ресурсы
    @Override
    public void destroy() {}
}
```

Декодер в этом листинге относительно прост. Класс объявляется декодером ❶ и параметризуется типом `AbstractCommand` – он будет преобразовывать входящее текстовое сообщение в объект `AbstractCommand` (тип данных в ActionBazaar). В классе предусмотрены два метода-обработчика событий жизненного цикла, `init()` ❷ и `destroy()` ❺. Веб-сокет первым вызывает метод `willDecode()` ❹, чтобы определить возможность обработки сообщения с помощью этого декодера. Если он вернет `true`, будет вызван метод `decode()` ❸.

Кодеры

Кодеры близко напоминают декодеры. Кодер должен реализовать один из вложенных интерфейсов в `javax.websocket.Endcoder`. Единственное отличие – возвращаемое значение и входной параметр в методе `encode()` переставлены местами. В листинге 14.10 представлен кодер `CommandMessageEncoder`.

Листинг 14.10. Кодер CommandMessageEncoder

```
public class CommandMessageEncoder implements
    // ❶ Наследует интерфейс Encoder.Text
    Encoder.Text<AbstractCommand>{

    // ❷ Инициализация кодера
    @Override
    public void init(EndpointConfig config) {}

    // ❸ Вызывается перед уничтожением экземпляра кодера
    @Override
    public void destroy() {}

    // ❹ Преобразует объект в строку
    @Override
    public String encode(AbstractCommand commandMessage)
            throws EncodeException {
        StringWriter writer = new StringWriter();
        ...
        return writer.toString();
    }
}
```

Класс `CommandMessageEncoder` имеет практически ту же структуру, что и класс `CommandMessageDecoder`, только вместо декодирования текста в объект он выполняет обратное преобразование – объекта в текст. Класс наследует интерфейс `Encoder.Text` ❶ и параметризован типом `AbstractCommand` – он будет преобразовывать объект `AbstractCommand` в исходящее текстовое сообщение типа `String`. Метод инициализации ❷ будет вызван сразу после создания экземпляра. При уничтожении экземпляра кодера будет вызван метод `destroy()` ❸, что позволяет организовать освобождение любых занятых ресурсов. Наконец, метод `encode()` ❹ выполняет преобразование объекта в строку. А теперь вернемся к нашему приложению ActionBazaar и посмотрим, как действуют веб-сокеты!

14.4. Веб-сокеты в приложении ActionBazaar

Веб-сокеты в приложении ActionBazaar используются для организации поддержки пользователей и информационной панели для сотрудников. Чат позволяет сотрудникам отдела поддержки оказывать содействие конечным пользователям в режиме реального времени, а информационная панель предоставляет сотрудникам самую свежую информацию о количестве пользователей, ожидающих помощи, дает возможность обмениваться служебной информацией, проверять присутствие друг друга на сайте и справляться друг у друга, не возникает ли одна и та же проблема у других пользователей.

Реализация двух разных сценариев поможет нам продемонстрировать два разных подхода к реализации конечных точек веб-сокетов. Первая, реализация чата поддержки конечных пользователей, демонстрирует использование конечных точек, создаваемых программно. Вторая, реализация информационной панели, демонстрирует использование конечных точек, создаваемых с помощью аннотаций. Между этими двумя сценариями существует одно важное отличие. В первом случае веб-сокет используется для соединения двух пользователей. Во втором случае осуществляется рассылка событий всем активным сотрудникам отдела поддержки. Вы можете использовать исходный код этих двух примеров как шаблон для реализации своих конечных точек. Оба примера прекрасно работают в окружении с балансировкой нагрузки, когда приложение выполняется сразу на нескольких серверах.

На рис. 14.1 изображена схема реализации чата поддержки. Пользователь (покупатель или продавец) открывает страницу чата. Сценарий JavaScript в этой странице автоматически открывает соединение с конечной точкой `ClientChatEndpoint` (`/chat`), которая, в свою очередь, вызывает компонент-одиночку `ChatServer`. Сотрудник отдела поддержки, напротив, открывает служебную страницу поддержки. Сценарий JavaScript в этой странице автоматически открывает соединение с конечной точкой `SupportChatEndpoint` (`/admin/support`), которая так же вызывает компонент-одиночку `ChatServer`. Таким образом, компонент `ChatServer` оказывается связующим звеном, отвечающим за маршрутизацию сообщений, для чего он кэширует `javax.websocket.Session` конечных точек клиентов и сотрудников. Реализация компонента `ChatServer` предтавлена в листинге 14.11.

Листинг 14.11. Компонент-одиночка ChatServer, связывающий клиентов и сотрудников отдела поддержки

```
@Singleton
public class ChatServer {
    // ❶ Ожидающие клиенты и сотрудники
    private final Stack<Session> availableReps = new Stack<>();
    private final Stack<Session> waitingClients = new Stack<>();
    // ❷ Все открытые сеансы веб-сокетов клиентов и сотрудников
    private final Set<Session> csrSessions;
```

```
    private final Set<Session> clientSessions;
    // ❸ Активные диалоги
    private final Map<String,SupportConversation> conversations;

    // ❹ Вспомогательные методы для сотрудников отдела поддержки
    public void customServiceRepConnected(Session csrSession) {...}
    public void customServiceRepDisconnected(Session csrSession) {...}
    // ❺ Вспомогательные методы для клиентов
    public void addClientSession(Session clientSession) {...}
    public void removeClientSession(Session clientSession) {...}
    // ❻ Осуществляет отправку сообщения
    public void sendMessage(Session sourceSession, String message) {...}
    // ❼ Закрывает все диалоги
    public void shutdown() {...}
    // ❽ Выполняет различные команды, такие как завершение диалога
    public void performCommand(AbstractCommand command) {...}
}
```

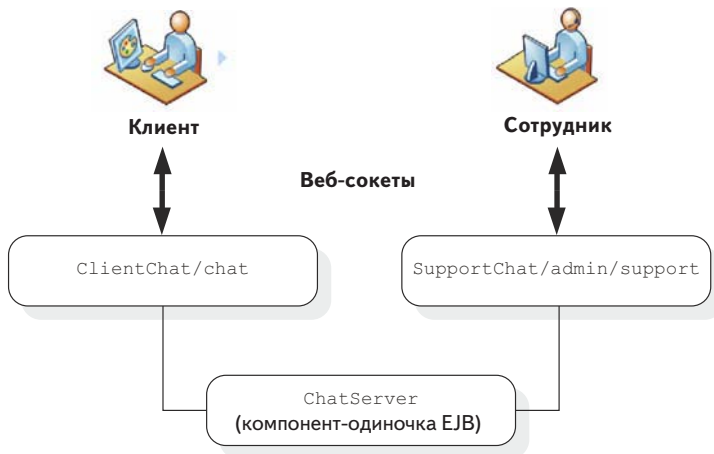

Рис. 14.1. Реализация чата на основе веб-сокетов в приложении ActionBazaar

Код в этом листинге выглядит относительно просто. Компонент `ChatServer` хранит список сотрудников отдела поддержки и клиентов, пока не участвующих в диалоге ❶. Он также хранит список всех сеансов, независимо от того, участвуют ли они в диалоге или еще ожидают его начала ❷. Наконец, он хранит список клиентов и сотрудников, между которыми уже установился диалог ❸. Компонент имеет методы для подключения и отключения сотрудников ❹ и клиентов ❺. Когда вызывается метод `onMessage` конечной точки, он, в свою очередь, вызывает метод `sendMessage` ❻ компонента, который отправляет сообщение обеим сторонам. Если сервер завершает работу, что определяется посредством `ServletContextListener`, метод `shutdown()` ❼ завершает все активные диалоги, сообщая пользователям об остановке сервера. Последний метод, `performCommand` ❽, реализует шаблон команд для выполнения операций, запрошенных клиентами или сотрудниками.

В текущей реализации поддерживается единственная операция – завершения чата по желанию клиента, когда он удовлетворен ответом, или по желанию сотрудника, когда он считает, что дал исчерпывающий ответ.

Последовательность действий, описываемая здесь, выглядит более сложной. На рис. 14.2 представлена схема взаимодействий между различными частями. Забегая вперед, скажем, что конечные точки, создаваемые программным путем, используют `MessageHandler` для обработки входящих сообщений. На этой диаграмме оба, клиент и сотрудник, инициируют создание экземпляров веб-сокетов, в результате чего вызывается метод `onOpen` в соответствующих им конечных точках. Затем создаются два экземпляра одного и того же класса обработчика сообщений. Обработчик сообщений, представленный на диаграмме сокращенным именем `CmdMsgHndlr`, полученным из полного имени `CommandMessageHandler`, передает сообщения компоненту `ChatServer`. Затем `ChatServer` отправляет сообщение обратно клиенту и сотруднику. Нужно отметить, что отправка сообщений от клиента серверу и от сервера клиенту выполняется асинхронно. Клиент и сервер не ожидают, пока их сообщения будут благополучно доставлены. Если возникают какие-либо ошибки, вызывается метод `onError` (описывается далее).

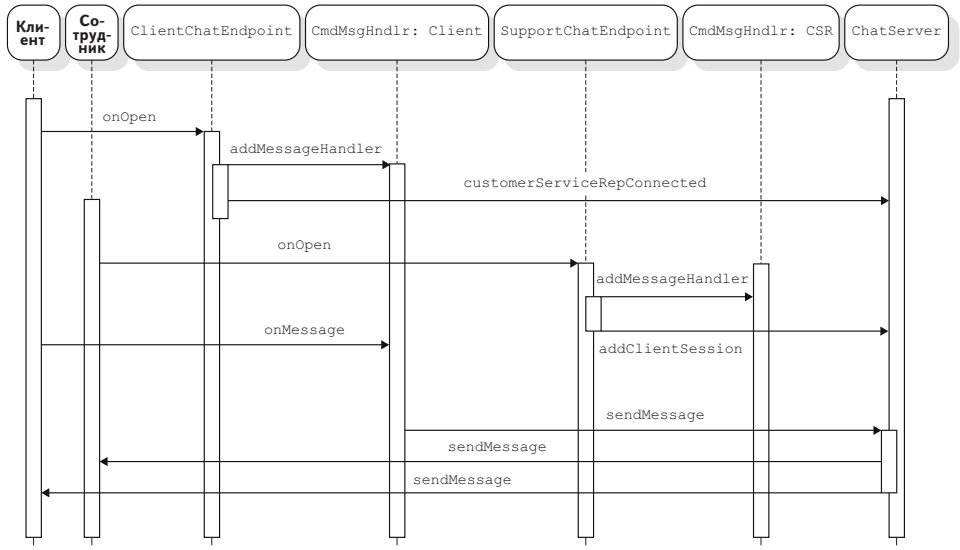

Рис. 14.2. Схема взаимодействий между конечной точкой, обработчиком сообщений и ChatServer

Клиент и сервер обмениваются сообщениями в формате JSON. На стороне сервера за кодирование и декодирование сообщений отвечают `CommandMessageEncoder` и `CommandMessageDecoder`. Для реализации обоих классов использован прикладной интерфейс, описываемый спецификацией JSR-353. Формат JSON намного более компактный, чем формат XML, и проще в использовании в JavaScript.

На рис. 14.3 представлена функциональная схема реализации службы информационной панели. Она намного проще, чем чат обслуживания клиентов, о котором рассказывалось выше. Эта служба позволяет сотрудникам отдела поддержки видеть очередь клиентов, ожидающих ответа на вопрос, а также рассылать широковещательные сообщения другим сотрудникам.

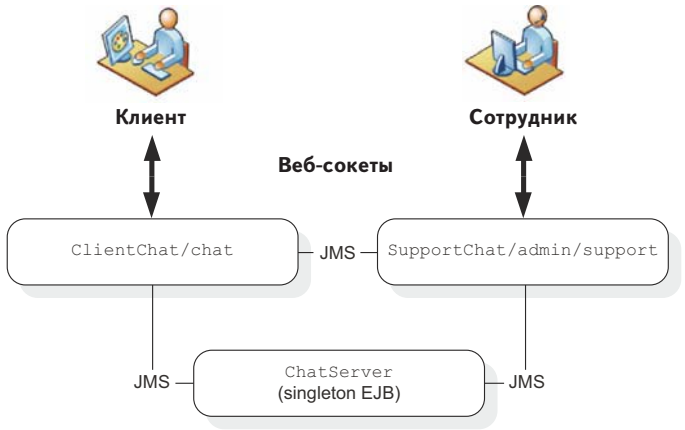

Рис. 14.3. Overview of the support bulletin

Конечная точка веб-сокета информационной панели реализована с применением аннотаций. Это означает, что события жизненного цикла и сообщения обрабатываются одним объектом. Сообщения, отправляемые с помощью информационной панели, рассылаются всем конечным точкам `BulletinService` посредством темы JMS. Широковещательные сообщения, посылаемые сотрудниками, передаются в одну и ту же тему JMS. Здесь принято решение использовать JMS вместо CDI потому, что в окружениях с балансировкой нагрузки экземпляры `BulletinService` могут быть разбросаны по нескольким серверам. Все сообщения посылаются в формате JSON.

Теперь, когда вы знаете, как используются веб-сокеты в приложении ActionBazaar, вернемся к чату и посмотрим, как он реализован с применением программных конечных точек.

14.4.1. Использование программных конечных точек

Как уже упоминалось выше, существует два способа создания конечных точек: программный и с помощью аннотаций. В этом разделе мы обсудим программный способ. Несмотря на кажущуюся сложность, программный способ создания конечных точек ничуть не сложнее способа с применением аннотаций. Просто программные конечные точки позволяют управлять процедурой регистрации обработчиков сообщений. Программные конечные точки дают два основных преимущества, позволяющих:

- отделять обработчики сообщений от реализации конечной точки;
- регистрировать разные обработчики сообщений, опираясь на информацию о сеансе.

Отделение реализации конечной точки от обработчиков сообщений дает возможность повторно использовать одни и те же обработчики в разных конечных точках. Конечная точка не обязательно должна быть связана с обработчиком сообщений отношением «один к одному». Конечная точка обслуживает события жизненного цикла и связана с URL. Возьмем в качестве примера реализацию чата в приложении ActionBazaar – в ней используются две отдельные конечные точки, одна создается для покупателя/продавца, а другая – для сотрудника отдела поддержки. В обоих случаях обработчик сообщений передает входящие сообщения компоненту `SupportServer` – компоненту-одиночке, осуществляющему маршрутизацию сообщений. Так как обеими конечными точками используется одна и та же логика обработки сообщений, не имеет смысла предусматривать две разные реализации.

Второе преимущество использования программных конечных точек не выглядит таким очевидным. Разные обработчики сообщений могут выбираться, исходя из параметров, передаваемых при открытии сеанса. Например, вы можете организовать поддержку параметра в строке соединения, определяющего тип используемого протокола. Тип `Java` мог бы, к примеру, определять необходимость передачи данных в виде сериализованных объектов Java вместо формата JSON. То есть, для URL `http://actionbazaar/chat?type=Java` мог бы выбираться обработчик сообщений, принимающий данные типа `java.io.InputStream/byte[]` и осуществляющий десериализацию этих данных в объекты Java.

Обработчики сообщений

Владение интерфейсом `javax.websocket.MessageHandler` играет ключевую роль в использовании конечных точек, создаваемых программным способом. Фактически от вас не требуется реализовать этот интерфейс непосредственно; вместо этого вы должны реализовать один из двух вложенных интерфейсов: `Partial` или `Whole`. Как следует из их названий, вы можете обрабатывать сообщение фрагментами, по мере их получения, или дождаться окончания приема и обработать сообщение целиком. Выбор зависит от того, как предполагается обрабатывать данные. Будете ли вы принимать потоковые данные, такие как изображения или видео, или дискретные сообщения, такие как команды в видеоигре. В листинге 14.12 приводится определение интерфейса `MessageHandler`.

Листинг 14.12. Интерфейс `MessageHandler`

```
public interface MessageHandler {
    // Интерфейс обработки сообщений фрагментами
    public static interface Partial<T extends Object>
    extends MessageHandler {
        public void onMessage(T partialMessage, boolean last);
```

```
    }

    // Интерфейс сообщений целиком
    public static interface Whole<T extends Object>
    extends MessageHandler {
        public void onMessage(T message);
    }
}
```

В этом листинге можно видеть два вложенных интерфейса, которые можно реализовать. Оба интерфейса имеют метод `onMessage`. Интерфейс использует механизм параметризации типов в Java, благодаря чему обеспечивается возможность выбора типа, передаваемого реализации. Если тип объекта-сообщения определяется в приложении, для его обработки будет использоваться декодер, зарегистрированный в конечной точке. Обработчик в интерфейсе `Partial` включает флаг, позволяющий отличать последний фрагмент сообщения.

Необходимость реализации интерфейса `MessageHandler` имеет также свои недостатки. Забегая вперед, отметим, что при использовании аннотации `@OnMessage` можно организовать получение экземпляра `javax.websocket.Session` вместе с данными, с помощью которого можно отправить сообщение обратно клиенту. Кроме того, так как экземпляр `MessageHandler` не является управляемым объектом, вы не сможете внедрить в него ссылки на компоненты EJB, диспетчеры сущностей и другие объекты, управляемые контейнером.

В листинге 14.13 демонстрируется реализация `ChatMessageHandler`. Этот обработчик сообщений используется обеими конечными точками, созданными для клиента и сотрудника отдела поддержки.

Листинг 14.13. Реализация обработчика сообщений ChatMessageHandler в приложении ActionBazaar

```
// ❶ Реализует интерфейс MessageHandler.Whole
public class ChatMessageHandler implements MessageHandler.Whole<ChatMessage>
{
    private final ChatServer chatServer;
    private final Session session;

    public ChatMessageHandler(ChatServer chatServer, Session session) {
        // ❷ Кэшировать ссылки на сеанс веб-сокета и на компонент ChatServer
        this.chatServer = chatServer;
        this.session = session;
    }

    @Override
    public void onMessage(ChatMessage message) {
        chatServer.handleMessage(session, message.getMessage());
    }
}
```

Код в этом листинге реализует интерфейс `MessageHandler.Whole` ❶ для обработки сообщений целиком. Конструктор кэширует ссылки на компонент-оди-

ночку `ChatServer` и сеанс веб-сокета ❷. Эти ссылки можно использовать во время обработки сообщений ❸. При исследовании этого примера важно помнить, что:

- для каждого соединения с веб-сокетом создается собственный экземпляр `ChatMessageHandler`;
- метод `onMessage` принимает `ChatMessage`; веб-сокеты используют декодер `ChatMessageDecoder` для преобразования данных в формате JSON в объект Java.

В следующем разделе мы создадим конечные точки и зарегистрируем обработчики сообщений.

Использование программных веб-сокетов

Теперь, когда у нас имеется обработчик сообщений, пришло время определить конечную точку. Для этого необходимо:

1. Определить класс, наследующий `javax.websocket.Endpoint`.
2. Реализовать метод `onOpen`.
3. Зарегистрировать обработчик сообщений в методе `onOpen`.
4. Зарегистрировать конечную точку.

Класс `javax.websocket.Endpoint`, который требуется унаследовать, – это абстрактный класс. Он имеет следующее определение:

```
public abstract class Endpoint {
    public abstract void onOpen(Session session, EndpointConfig config);
    public void onClose(Session session, CloseReason closeReason) {
        // пустой
    }

    public void onError(Session session, Throwable thr) {
        // пустой
    }
}
```

Вам требуется реализовать только метод `onOpen`. Внутри этого метода необходимо зарегистрировать обработчик сообщений.

Чтобы создать веб-сокет программным способом, требуется унаследовать `javax.websocket.Endpoint` и реализовать метод `onOpen`. Для настройки конечной точки, чтобы она смогла принимать соединения, необходимо отметить ее аннотацией `@ServerEndpoint`, которую мы обсудим в следующем разделе, когда будем знакомиться с аннотированными конечными точками. Регистрация обработчиков сообщений в методе `onOpen` осуществляется вызовом метода `addMessageHandler` объекта сеанса веб-сокета. Для каждого типа сообщений можно зарегистрировать только один обработчик. К типам сообщений относятся текст, двоичные данные и сообщения-объекты, как описывалось выше. После регистрации обработчиков приложение готово принимать сообщения от клиента.

14.4.2. Использование аннотированных конечных точек

Аннотированные конечные точки проще, но они не такие гибкие. При использовании аннотированных конечных точек все настройки выполняются посредством аннотаций. С помощью аннотаций отмечаются: класс, реализующий конечную точку, методы обработки событий жизненного цикла и методы обработки сообщений. Вся функциональность, связанная с конечной точкой, сосредоточена в одном объекте Java.

Создание экземпляров аннотированных конечных точек осуществляется реализацией веб-сокетов. От вас не требуется управлять созданием экземпляров этого класса – экземпляр будет создан автоматически при создании нового соединения и уничтожен после того, как соединение закроется. Требований к классу реализации конечной точки не так много: это должен быть неабстрактный, конкретный класс с общедоступным конструктором, не имеющим аргументов. Внедрение зависимостей выполняется после создания экземпляра класса.

@ServerEndpoint

Аннотация `@ServerEndpoint` превращает класс в конечную точку веб-сокета. Во время запуска приложения контейнер проверит наличие в пути поиска классов, отмеченных этой аннотацией. Помимо идентификации класса как конечной точки веб-сокета, данная аннотация определяет URI конечной точки, а также кодеры и декодеры, применяемые при обработке сообщений. Ниже приводится определение аннотации:

```
@Retention(value = RetentionPolicy.RUNTIME)
@Target(value = {ElementType.TYPE})
public @interface ServerEndpoint {
    public String value();
    public String[] subprotocols() default {};
    public Class<? extends Decoder>[] decoders() default {};
    public Class<? extends Encoder>[] encoders() default {};
    public Class<? extends ServerEndpointConfig.Configurator> configurator()
        default ServerEndpointConfig.Configurator.class;
}
```

Параметры аннотации, используемые для настройки конечной точки:

- `value` – идентификатор URI конечной точки;
- `subprotocols` – упорядоченный список протоколов уровня приложения;
- `decoders` – упорядоченный список подклассов `javax.websocket.Decoder`;
- `encoders` – упорядоченный список подклассов `javax.websocket.Encoder`;
- `configurator` – дополнительный класс, осуществляющий дополнительную настройку конечной точки.

Обязательным является только параметр `value`, определяющий идентификатор URI конечной точки. Все остальные параметры можно опустить. URI – это относительный адрес, откладываемый от корня контейнера, содержащего веб-сокет. Так

как контейнером веб-сокетов часто служат контейнер веб-приложения, корень совпадает с корнем контейнера веб-приложения. Чтобы лучше понять принцип использования этой аннотации, рассмотрим пример реализации `BulletinService`, которая представлена в листинге 14.14.

Листинг 14.14. Настройка конечной точки BulletinService

```
// ❶ Отметить класс как конечную точку веб-сокета
@ServerEndpoint(
    // ❷ Определить URI для конченой точки BulletinService
    value="/admin/bulletin",
    // ❸ Определить декодер для сообщений
    decoders = {BulletinMessageDecoder.class},
    // ❹ Определить кодер для сообщений
    encoders = {BulletinMessageEncoder.class})
public class BulletinService {
    ...
}
```

Здесь определяется класс `BulletinService`, отмеченный аннотацией `@ServerEndpoint`, которая превращает его в конечную веб-сокета ❶. Параметр `value` определяет URI сервера ❷. Если предположить, что веб-приложение развертывается, как было показано в главе 14, полный путь к службе будет иметь вид: `http://<адрес>:<порт>/chapter14/admin/bulletin`. Параметр `encoders` определяет, что кодирование исходящих сообщений будет выполняться с использованием `BulletinMessadeEncoder` ❹, а параметр `decoders` определяет декодер для входящих сообщений ❸. Несмотря на то, что в каждом из этих параметров можно указать несколько кодеров/декодеров, для преобразований будет использоваться только первый.

@PathParam

Аннотация `@PathParam` несет ту же функциональную нагрузку, что и аннотация `javax.ws.rs.PathParam`, с которой мы познакомились при обсуждении веб-служб RESTful в главе 8. С ее помощью можно отображать переменные из URI в параметры методов, отмеченных аннотациями `@OnMessage`, `@OnError`, `@OnOpen` и `@OnClose`. Это дает возможность передавать параметры в URL и тем самым уменьшить сложность сообщений, передаваемых через веб-сокеты. Аннотация имеет следующее определение:

```
@Target(value = {ElementType.PARAMETER,
                 ElementType.METHOD, ElementType.FIELD})
@Retention(value = RetentionPolicy.RUNTIME)
@Documented
public @interface PathParam {
    public String value();
}
```

Как видно из этого определения, аннотация `@PathParam` имеет единственный параметр – переменная, которая должна присутствовать в адресе URI, опреде-

ляемом аннотацией `@ServerEndpoint`. Реализация веб-сокета будет пытаться преобразовать строку, полученную из URL, в данные с типом параметра метода. Поддерживаются только `String` и простейшие типы. В случае ошибки преобразования или использования параметра недопустимого типа будет возбужден исключение `DecodeException`, в результате чего произойдет вызов метода, отмеченного аннотацией `@OnError`.

Чтобы лучше понять, как действует эта аннотация, рассмотрим пример ее применения в приложении ActionBazaar. Так как за помощью к сотрудникам отдела поддержки могут обращаться пользователи при помощи разных клиентов, необходимо определить тип клиента на этапе открытия соединения. Приложение ActionBazaar различает следующие типы клиентов: `desktop`, `mobile-web` и `mobile-native`. Эти типы определены в ActionBazaar и используются для создания специализированных сообщений для разных клиентов, как показано в листинге 14.15.

Листинг 14.15. Использование переменных в URL

```
// ❶ Определение переменной части пути в URL
@ServerEndpoint(value="/admin/bulletin/{clientType}",
    decoders ={BulletinMessageDecoder.class},
    encoders = {BulletinMessageEncoder.class})
public class BulletinService {
    @OnOpen
    public void onOpen(
            // ❷ Отобразить переменную часть пути из URL в параметр метода
            @PathParam("clientType") String clientType) {
        ...
    }
}
```

В этом листинге переменная часть пути `clientType` из URI ❶ отображается в параметр метода `onOpen` ❷. Мобильное приложение ActionBazaar для iPad, написанное на Objective-C, будет конструировать следующий URL: `http://<адрес>:<порт>/chapter14/admin/bulletin/mobile-native`. Фрагмент `mobile-native` в адресе URL будет передан методу `onOpen`, где его можно обработать и использовать при конструировании последующих ответов.

@OnOpen

Аннотация `@OnOpen` отмечает метод конечной точки, который будет вызываться при открытии соединения впервые. Она имеет следующее определение:

```
@Retention(value = RetentionPolicy.RUNTIME)
@Target(value = {ElementType.METHOD})
public @interface OnOpen {}
```

Как видно из этого определения, данная аннотация не имеет параметров и выглядит очень простой. Но первое впечатление обманчиво – аннотированный ею метод может принимать несколько разных параметров и в разном порядке. Контейнер веб-сокетов использует механизм рефлексии Java для исследования па-

раметров и вызывает метод с указанными параметрами. Аннотированный метод может принимать параметры следующих типов:

- `javax.websocket.Session` – сеанс, вновь созданный для данного соединения;
- `javax.websocket.EndpointConfig` – настройки для конечной точки, включая списки кодеров и декодеров, а также ассоциативный массив с пользовательскими параметрами;
- параметры, отмеченные аннотацией `@PathParam` – отображение переменных из URI в параметры метода.

Метод, отмеченный аннотацией `@OnOpen`, вызывается только один раз, в момент открытия соединения. Если метод принимает объект `Session`, его можно кэшировать, чтобы в дальнейшем с его помощью можно было посылать сообщения обратно клиенту. Реализация `BulletinService` кэширует объект `Session` и использует его для отправки сообщений клиентам в другом методе, осуществляющем мониторинг очереди JMS.

@OnClose

Аннотация `@OnClose` отмечает метод, который будет вызываться при закрытии соединения любой из сторон, клиентом или сервером. Подобно методу, отмеченному аннотацией `@OnOpen`, метод с аннотацией `@OnClose` может принимать необязательные параметры, такие как `Session`, `EndpointConfig` и параметры, отмеченные аннотацией `@PathParam`. В этом методе можно организовать освобождение ресурсов. Например, конечная точка `BulletinService` в приложении ActionBazaar регистрируется как потребитель JMS, а метод с аннотацией `@OnClose` удаляет регистрацию в JMS, чтобы позволить сборщику мусора утилизировать ресурсы, ставшие ненужными. Объявление этого метода в ActionBazaar показано в листинге 14.16.

Листинг 14.16. Использование аннотации @OnClose

```
@OnClose // ❶ Отметить метод, как обработчик события закрытия соединение
public void cleanup() {
    ...
}
```

Здесь метод `cleanup()` отмечен, как обработчик события закрытия соединения ❶. Важно отметить, что в методе, отмеченном аннотацией `@OnClose`, нельзя послать сообщение клиенту, потому что соединение действительно закрыто. Кроме того, если объект `Session` был кэширован и используется асинхронно, вызовы методов, осуществляющих отправку данных, могут потерпеть неудачу, так как соединение может быть закрыто клиентом или сервером во время обработки отправляемых данных. Будьте готовы обработать состояние, когда соединение фактически уже было закрыто (удаленной стороной), а метод `@OnClose` еще не вызывался. Учтите, что использование ключевого слова `synchronized` в этой ситуации не поможет – только тщательная обработка ошибок и поддержка выполнения в многопоточной среде помогут гарантировать корректную работу приложения.

@OnMessage

Аннотация `@OnMessage` отмечает метод, который будет вызываться для обработки входящих сообщений. В одной конечной точке может присутствовать несколько методов, отмеченных аннотацией `@OnMessage`. Каждый такой метод должен обрабатывать сообщения определенного типа. Например, Нельзя объявить два метода с аннотацией `@OnMessage`, обрабатывающие сообщение типа `String`, иначе контейнер не сможет определить, какой из методов следует вызвать.

Аннотация `@OnMessage` имеет следующее определение:

```
@Retention(value = RetentionPolicy.RUNTIME)
@Target(value = {ElementType.METHOD})
public @interface OnMessage {
    public long maxMessageSize() default -1L;
}
```

Как видно из этого определения, данная аннотация имеет единственный параметр – максимальный размер сообщения. По умолчанию размер сообщения не ограничивается. Но на практике всегда следует определять верхний предел, чтобы защитить приложение от злонамеренного клиентского кода, с помощью которого злоумышленники могут попытаться вызвать переполнение памяти на сервере. Если размер сообщения превысит ограничение, соединение автоматически будет закрыто, а клиенту будет возвращен код 1009 (too big – слишком большой объем данных).

Типы данных, которые могут передаваться методу, делятся на три группы: текстовые данные, двоичные данные и служебные сообщения, как показано в табл 14.1.

Таблица 14.1. Типы параметров методов @OnMessage

Группа	Тип параметра	Обработка сообщений фрагментами	Примечания
Текст	`String`	Нет	
Текст	`int`, `long`, `float`, `double`, и т. д.	Нет	
Текст	`String`, `boolean`	Да	Если в параметре типа `boolean` передано значение `true`, значит был получен последний фрагмент сообщения.
Текст	Пользовательский объект	Нет	Необходимо зарегистрировать `Decoder.Text`.
Двоичные данные	`byte[]`	Нет	
Двоичные данные	`byte[]`, `boolean`	Да	Если в параметре типа `boolean` передано значение `true`, значит был получен последний фрагмент сообщения.

Таблица 14.1. (окончание)

Группа	Тип параметра	Обработка сообщений фрагментами	Примечания
Двоичные данные	`ByteBuffer`	Нет	
Двоичные данные	`ByteBuffer`, `boolean`	Да	Если в параметре типа `boolean` передано значение `true`, значит был получен последний фрагмент сообщения.
Двоичные данные	`InputStream`	Да	
Двоичные данные	Пользовательский объект	Нет	Необходимо зарегистрировать `Decoder.Binary`.
Служебный объект	`PongMessage`	Нет	

Если предполагается передавать в сообщениях пользовательские объекты, как показано в табл. 14.1, необходимо зарегистрировать декодеры в `@ServerEndpoint`. В дополнение к типам параметров, перечисленных в табл. 14.1, методы с аннотацией `@OnMessage` могут принимать следующие параметры:

- `Session` – объект `Session`, связанный с клиентом;
- `@PathParam` – отображение переменных из URI в параметры метода.

Параметры могут следовать в произвольном порядке – назначение параметров будет определено контейнером автоматически с помощью механизма рефлексии. В отличие от интерфейса `MessageHandler`, с которым мы познакомились выше, методы с аннотацией `@OnMessage` могут возвращать данные. Если тип возвращаемого значения отличается от `String` и не является простым типом Java, в `@ServerEndpoint` необходимо зарегистрировать кодер для этого типа, иначе будет генерироваться ошибка.

Теперь, после знакомства с теорией применения аннотаций, рассмотрим практический пример из приложения ActionBazaar, представленный в листинге 14.17.

Листинг 14.17. Обработчики сообщений в BulletinService из приложения ActionBazaar

```
@ServerEndpoint(value="/admin/bulletin/{clientType}",
    decoders = {BulletinCommandDecoder.class,BulletinMessageDecoder.class},
    encoders = {BulletinMessageEncoder.class,CommandResultEncoder.class})
public class BulletinService{

    // ❶ Отметить метод, как обработчик сообщений
    @OnMessage
    // ❷ Принимает простую строку и объект Session
    public void processMessage(String message, Session session) {
        ...
    }

    // ❸ Обработчик сообщений с возвращаемым значением
    @OnMessage
    public CommandResult processCommand(
```

```
            // ❹ Отображение переменной части пути из URI в параметр метода
            @PathParam("clientType") String clientType,
            // ❺ Принимает пользовательский объект - используется декодер
            BulletinCommand command,
            Session session) {
        ...
    }
}
```

В этом листинге демонстрируются два метода конечной точки `BulletinService` ❶. Первый метод принимает сообщение типа `String` и ссылку на сеанс веб-сокета ❷. Он реализует прием сообщений для информационной панели. Второй метод гораздо интереснее – он возвращает экземпляр `CommandResult` ❸, который будет преобразовываться в формат JSON экземпляром `CommandResultEncoder`. Он так же принимает тип клиента, полученный из переменной части пути URI ❹. И, наконец, он принимает объект `BulletinCommand`, который декодируется из формата JSON с помощью `BulletinCommandDecoder` ❺.

Как видите, обработка сообщений реализуется довольно просто. Прикладной интерфейс аннотаций обладает большей гибкостью с точки зрения параметров. Программный способ с использованием интерфейса `MessageHandler` не позволяет организовать получение параметров из URL или копировать объект `Session`.

@OnError

Обработка ошибок имеет большое значение для конечных точек веб-сокетов. В любой момент может возникнуть логическая ошибка или произойти разрыв соединения с удаленным клиентом. Аннотация `@OnError` реализует механизм обработки таких неожиданных ситуаций. Она имеет следующее определение:

```
@Retention(value = RetentionPolicy.RUNTIME)
@Target(value = {ElementType.METHOD})
public @interface OnError {}
```

Этой аннотацией можно отметить только один метод. Сама аннотация не имеет параметров, но сам метод может принимать параметры следующих типов:

- `Throwable` – исключение, представляющее ошибку;
- `Session` – сеанс веб-сокета;
- `@PathParam` – аннотированные параметры.

Чтобы лучше понять, как действует этот обработчик, рассмотрим пример из конечной точки `BulletinService` в приложении ActionBazaar (листинг 14.18).

Листинг 14.18. Обработка ошибок веб-сокета в BulletinService

```
// ❶ Отметить handleFailure, как обработчик ошибок веб-сокета
@OnError
// ❷ Принимает параметр типа Throwable, представляющий ошибку
public void handleFailure(Throwable t,
// ❸ Сеанс, представляющий соединение с клиентом
Session session,
```

```
// ❹ Отображение переменной части пути из URI в параметр метода
@PathParam("clientType") String clientType) {
    ...
}
```

Аннотация `@OnError` в листинге 14.18 превращает метод `handleFailure` в обработчика ошибок веб-сокета ❶. Он принимает три параметра. Первый параметр ❷ – исключение, вызванное ошибкой. Второй параметр – сеанс веб-сокета ❸, и третий параметр – тип клиента, извлекаемый из переменной части пути в URL ❹.

Теперь вы знаете, как создавать конечные точки программным путем и с помощью аннотаций. Следующий вопрос, который нам предстоит исследовать, – порядок использования конечных точек в контексте EJB.

14.5. Эффективное использование веб-сокетов

Веб-сокеты являются мощным дополнением к стеку Java EE и дают возможность создавать приложения, принимающие данные от сервера в асинхронном режиме. Клиентам больше не нужно опрашивать сервер, проверяя наличие изменений, – сервер теперь сам может в нужные моменты посылать сообщения клиенту. В отличие от веб-служб RESTful и SOAP, конечные точки веб-сокетов сохраняют информацию о состоянии. В данном случае имеется два элемента, определяющих состояние: объекты, хранящиеся в памяти сервера (конечная точка), и открытое соединение с клиентом. Если страница открывает два соединения с сервером, как в приложении ActionBazaar (чат и информационная панель), создается два комплекта объектов и два дорогостоящих соединения. Веб-сокеты ухудшают масштабируемость приложений, но при этом делают возможным отправку извещений клиентам стандартным способом, превосходящим прием опроса с использованием технологии AJAX или Comet.

Ниже перечислены рекомендации по эффективному использованию веб-сокетов:

- применяйте веб-сокеты, только если решение на основе AJAX оказывается непригодным;
- храните состояние в пользовательских свойствах сеанса, доступных посредством `Session.getUserProperties()`;
- для доступа к ресурсам, требующим применения транзакций, таким как базы данных, используйте компоненты EJB;
- используйте подпротоколы для поддержки версий;
- осуществляйте обмен данными в формате JSON;
- ограничивайте связь с клиентом одним веб-сокетом;
- ограничивайте максимальные размеры сообщений;
- для защиты веб-сокетов используйте средства безопасности HTTP;
- используйте безопасные веб-сокеты HTML5 (WSS), когда это возможно.

Веб-сокеты – превосходная новая технология, появившаяся с выходом HTML5, но они не являются универсальным решением. Веб-сокеты должны использоваться, только когда возможность отправки сообщений клиентам по инициативе сервера является действительно необходимой. Чат в приложении ActionBazaar как раз является примером такой необходимости, потому что решение, основанное на опросе сервера клиентами, оказывается малоэффективным. Но веб-сокеты непригодны для организации проверок полей веб-форм, таких как существование имени пользователя при создании новой учетной записи, – в подобных ситуациях технология AJAX позволяет получить более удачное решение. Кроме того, применение веб-сокетов в главной странице приложения ActionBazaar было бы весьма неудачным решением, так как каждый просмотр главной страницы приводил бы к созданию соединения с сервером.

Информация о состоянии, имеющая отношение к веб-сокетам, должна храниться в ассоциативном массиве сеанса, доступном с помощью метода `Session.getUserProperties()`. В этом ассоциативном массиве должны храниться только объекты, поддерживающие сериализацию. Прикладная информация о состоянии не должна храниться в переменных-членах, статических переменных или в локальной файловой системе. В окружениях с балансировкой нагрузки, контейнер приложения может передавать обслуживание конечной точки между узлами. Поэтому на другом узле будут доступны только данные, хранящиеся непосредственно в сеансе. Приложение может действовать правильно, выполняясь на одном узле, но его поведение может измениться при развертывании на нескольких узлах.

Для доступа к ресурсам под управлением транзакций, таким как базы данных, конечные точки веб-сокетов должны использовать компоненты EJB. Избегайте внедрения источников данных в веб-сокеты и прямых обращений к базе данных из них. Во-первых, нежелательно, чтобы конечные точки веб-сокетов удерживали ресурсы, такие как соединения с базами данных, на протяжении всего времени существования веб-сокета. Во-вторых, веб-сокеты не должны содержать прикладную логику. Логика должна быть реализована либо в POJO, если она не использует ресурсы, доступ к которым осуществляется в рамках транзакций, либо в компонентах EJB.

Веб-сокеты предоставляют инфраструктуру для реализации ваших собственных протоколов. Соединения через веб-сокеты можно открывать не только в веб-клиентах на JavaScript, но также в приложениях на платформе iOS и Android. Поддержка приложений для iOS и Android отличается от поддержки веб-клиентов. Так как программный код на JavaScript всегда загружается веб-браузером с сервера, веб-клиенты обычно получают самую свежую реализацию. Но в случае с клиентами других типов, у пользователей может иметься несколько версий клиента. Как следствие, необходимо изначально предусматривать поддержку разных версий протокола. Если только развитие приложения не прекратится после выпуска первой версии, вне всяких сомнений появятся версии слоя сетевых взаимодействий, не обладающих обратной совместимости.

В настоящее время наиболее предпочтительным форматом обмена данными является формат JSON, особенно при взаимодействии с мобильными устройст-

вами. Хотя формат JSON не является самодокументируемым и испытывает нехватку многих особенностей, доступных в формате XML, включая возможность проверки с использованием XML Schema, тем не менее, он обладает такими преимуществами, как легковесность, и хорошо подходит при наличии ограничений на вычислительные ресурсы и пропускную способность. Формат XML не такой компактный и требует значительных затрат на обработку. В мобильных устройствах это приводит к увеличенному расходованию заряда аккумулятора и ухудшению отзывчивости приложений.

Веб-сокеты следует использовать очень экономно. Например, для администратора приложения ActionBazaar открывается два соединения с сервером через веб-сокеты. Мы использовали два соединения, только чтобы продемонстрировать два разных подхода к реализации конечных точек, в действующем же приложении следовало бы использовать одно соединение. Соединение через веб-сокет – весьма дорогостоящий ресурс и большое их количество может ухудшить масштабируемость приложения.

При разработке конечных точек большое внимание следует уделять их безопасности. Взлом веб-сокета может позволить злоумышленнику совершить вторжение в систему или выполнить атаку вида «отказ в обслуживании» (Denial Of Service, DOS). Важно не забывать определять максимальный размер сообщений. Злоумышленник может написать сценарий, пытающийся передать в конечную точку веб-сокета гигабайты случайных данных, что может вызвать отказ в обслуживании добропорядочных пользователей. Кроме того, если конечная точка не предназначена для общественного доступа, она должна быть защищена и требовать наличия аутентифицированного сеанса. Наконец, для передачи конфиденциальных данных следует использовать защищенные веб-сокеты (wss). В противном случае данные, передаваемые по сети в простом текстовом виде, легко можно перехватить, подменить и так далее. Даже если страница будет запрошена по протоколу HTTPS, открытие соединения через веб-сокет с использованием схемы `ws:` приведет к созданию незащищенного соединения. Теперь, когда вы познакомились с приемами эффективного использования веб-сокетов, пришло время завершить главу.

14.6. В заключение

Эта глава началась со знакомства с веб-сокетами и сравнения их с технологиями AJAX и Comet. В отличие от AJAX и Comet, веб-сокеты обеспечивают полноценную двунаправленную связь между сервером и клиентом. Поддержка веб-сокетов впервые появилась с выходом стандарта HTML5 и, в отличие от технологии Comet, которая опирается на существующие технологии, чтобы обеспечить возможность отправки сообщений со стороны сервера без инициирующего запроса со стороны клиента, веб-сокеты полностью стандартизованы и имеют самую широкую поддержку. Веб-сокеты используют имеющуюся инфраструктуру HTTP и потому совместимы с существующими брандмауэрами.

Спецификация Java EE 7 включает стандартное определении веб-сокетов в виде документа JSR-356. Веб-сокеты можно создавать программным способом или с применением аннотаций. Программный подход обладает большей гибкостью, он позволяет управлять созданием конечных точек, а также отделять друг от друга конечные точки и обработчики сообщений. Конечные точки, создаваемые с помощью аннотаций, проще в использовании. Используя этот подход, можно просто отметить POJO аннотацией `@ServerEndpoint` и реализовать хотя бы один метод с аннотацией `@OnMessage`.

Прикладной интерфейс веб-сокетов поддерживает регистрацию кодеров и декодеров. Кодеры и декодеры упрощают разработку и дают возможность обработчикам сообщений принимать объекты Java вместо двоичных потоков. Однако обработчики сообщений способны также принимать простой текст или потоки байтов. При желании можно также организовать обработку сообщений не только целиком, но и фрагментами. Используя специальный API для обработки формата JSON (JSR-353), определение которого было добавлено в спецификацию Java EE 7, легко можно обеспечить обработку и преобразование данных в/из формат JSON.

Спецификация веб-сокетов испытывает недостаток точных формулировок в части интеграции веб-сокетов и платформы Java EE. Некоторые реализации, такие как Tyrus (https://tyrus.java.net) поддерживают создание конечных точек веб-сокетов и обработчиков в виде компонентов. Но, учитывая особенности конечных точек, лучше использовать компоненты EJB из конечных точек веб-сокетов, а не экспортировать компоненты EJB как конечные точки.

Веб-сокеты открывают возможность создания совершенно нового класса приложений Java EE. Прежде платформа Java EE использовалась в основном для реализации традиционных веб-приложений, основанных на компонентах EJB без сохранения состояния. С появлением веб-сокетов в Java EE 7 стало возможным создавать одностраничные приложения и взаимодействовать с мобильными приложениями, выполняющимися под управлением iOS или Android.

ГЛАВА 15. Тестирование компонентов EJB

Эта глава охватывает следующие темы:

- разные стратегии тестирования;
- модульное тестирование компонентов EJB;
- интеграционное тестирование компонентов EJB с помощью встраиваемого контейнера `EJBContainer`;
- интеграционное тестирование компонентов EJB с помощью Arquillian;
- использование CDI для нужд тестирования.

В предыдущих главах мы занимались изучением различных технологий, предлагаемых EJB 3, и рассмотрением примеров их применения. Теперь, когда вы получили представление о том, как их использовать в своих приложениях, пришла пора узнать, как убедиться в безупречной, точной и безопасной работе этих технологий. Обычно для этого выполняют тестирование.

Эта глава охватывает некоторые основные приемы тестирования компонентов EJB. Сначала мы обсудим разные стратегии тестирования и как можно использовать их для тестирования компонентов EJB. Затем мы рассмотрим такие технологии, как встраиваемый контейнер `EJBContainer` и инструмент Arquillian. В заключение мы расскажем, как избежать некоторых наиболее типичных проблем тестирования, чтобы вы могли повысить эффективность тестирования своих программ. Итак, приступим к знакомству с основами тестирования.

15.1. Введение в тестирование

Тестирование программного обеспечения часто является причиной жарких споров, так как все согласны, что тестирование необходимо и выгодно, но не все схо-

дятся во мнении, какие приемы считать лучшими. В этой главе мы рассмотрим три основные стратегии тестирования программного обеспечения: модульное тестирование, интеграционное тестирование и функциональное тестирование. Каждая из этих стратегий представляет разные подходы к тестированию, и каждая имеет свои плюсы и минусы. Обычно разработка собственной стратегии тестирования в организациях заключается в поиске правильного баланса этих трех стратегий. Итак, давайте рассмотрим эти стратегии.

15.1.1. Стратегии тестирования

В этом разделе мы расскажем о трех разных стратегиях: модульное тестирование, интеграционное тестирование и функциональное тестирование, а так же покажем примеры их применения для тестирования компонентов EJB. Каждая стратегия преследует определенную цель, и все они используются для тестирования приложений разными способами. Ни одна стратегия не дает 100% гарантии обнаружения всех имеющихся ошибок. И даже комбинация всех трех стратегий не может дать такой гарантии. Но их комбинирование поможет вам убедиться, что приложение работает, как предполагалось. Итак, начнем с первой стратегии – модульного тестирования.

Модульное тестирование

Целью модульного тестирования является проверка работы прикладной логики приложения при разных исходных данных и исследование правильности результатов. Несмотря на то, что цель выглядит простой и понятной, реализация модульного тестирования может оказаться очень сложным и запутанным делом, особенно при наличии старого кода.

Существует большое разнообразие приемов тестирования. Однако все они опираются на следующие базовые принципы:

1. Подключение к внешним ресурсам (базам данных, веб-службам и другим) недопустимо.
2. Каждый тест проверяет только один класс.
3. Тестируется только публичный контракт класса (общедоступные методы или интерфейсы), а внутренний код тестируется за счет изменения входных данных.
4. Для получения данных, требуемых тестируемой логике, должны создаваться фиктивные зависимости.

Для тестирования Java-приложений часто используется фреймворк JUnit. Для создания фиктивных классов-зависимостей используется мощный и простой в использовании фреймворк Mockito. Эти две технологии, в сочетании с принципом преимущества умолчаний перед настройками и интерпретацией компонентов EJB как POJO, значительно упрощает модульное тестирование компонентов.

Хотя тестирование является отличным способом гарантировать качество кода, его нельзя рассматривать как панацею от всех бед. Когда приложение развертывается на сервере Java EE, оно превращается в нечто совершенно иное. Промыш-

ленные контейнеры создают прокси-объекты для доступа к вашим компонентам EJB, внедряют зависимости, используют конфигурационные файлы (*web.xml*, *ejb-jar.xml*) для выполнения дополнительных настроек, подключаются к базам данных и производят массу других действий. Чтобы убедиться, что после всего этого код функционирует правильно, необходимо вслед за модульным тестированием произвести интеграционное тестирование.

Интеграционное тестирование

Стратегия интеграционного тестирования позволяет тестировать прикладной код в окружении, близком к фактическому окружению, но не являющемся им. Ее главная цель – убедиться в правильности взаимодействий кода с внешними ресурсами и различных технологий в приложении между собой. Давайте посмотрим на простом примере, что это означает.

Для нормальной работы приложению необходима база данных, наполненная корректными данными. В интеграционном тестировании не требуется использовать фиктивные данные, как при модульном тестировании. Вместо этого в интеграционных тестах часто используются базы данных, находящиеся в памяти, которые легко можно создавать и уничтожать во время выполнения тестов. База данных в памяти – это самая настоящая база данных, что дает возможность проверить правильность работы сущностей JPA. Но все же эта база данных не совсем настоящая – она лишь имитирует настоящую базу данных для целей интеграционного тестирования.

Какое отношение это имеет к компонентам EJB? Так же как для проверки сущностей JPA необходима имитация базы данных, для проверки компонентов EJB необходима имитация контейнера. Подобно базам данных в памяти, таким как Derby, имитирующим «настоящие» базы данных (например, MySQL или Oracle), существует встраиваемый контейнер `EJBContainer`, имитирующий «настоящий» сервер Java EE, такой как GlassFish. Давайте познакомимся с этим встраиваемым контейнером.

Встраиваемый контейнер `EJBContainer` является частью спецификации EJB и обязательно должен присутствовать в полноценной реализации EJB. Мы уже упоминали встраиваемый контейнер `EJBContainer` в главе 5, как один из способов задействовать компоненты EJB внутри приложений Java SE. Однако контейнер `EJBContainer` можно также использовать для интеграционного тестирования. Интеграционный тест может запускать встраиваемый контейнер в памяти, развертывать в нем все компоненты EJB, которые будут найдены в пути поиска классов теста, и затем производить тестирование компонентов EJB. Несмотря на то, что встраиваемый контейнер является отличным ресурсом, его использование может оказаться по-настоящему непростым делом. В таких ситуациях нам на выручку приходят сторонние инструменты, такие как Arquillian.

Arquillian – это достаточно мощный и развитый инструмент интеграционного тестирования. Его можно считать своеобразной оберткой вокруг встраиваемого контейнера `EJBContainer`. Этот инструмент имеет простой интерфейс для настройки и запуска встраиваемого контейнера, что позволяет при создании интег-

рационных тестов сосредоточиться на тестировании компонентов EJB, не отвлекаясь особенно на управление ресурсами.

Несмотря на то, что интеграционное тестирование идет на шаг дальше, чем модульное тестирование, оно все еще имитирует внешние ресурсы, используемые тестируемым приложением. Для полноты проверки необходимо провести функциональное тестирование .

Функциональное тестирование

Функциональное тестирование – это стратегия тестирования, применение которой обычно влечет создание специальной группы специалистов, занимающихся тестированием. На этом уровне приложение развертывается в действующем окружении. Чтобы проверить правильную работу приложения, команда тестеров будет использовать комплекс автоматизированных и ручных тестов. Так как функциональное тестирование обычно выполняется отдельной группой специалистов, не занимающихся разработкой, мы не будет углубляться в изучение особенностей функционального тестирования и в оставшейся части главы сосредоточимся на модульном и интеграционном тестировании.

15.2. Модульное тестирование компонентов EJB

Давайте обратимся к примеру и посмотрим, как использовать фреймворки JUnit и Mockito для организации модульного тестирования компонентов EJB. В листинге 15.1 представлен простой сеансовый компонент без сохранения состояния из приложения ActionBazaar. Его цель – определить размер скидки за купленное членство. Тестирование компонента `DiscountManagerBean` жизненно важно, чтобы исключить возможность предоставления сумасшедших скидок, которые могут угрожать бизнесу.

Исходный код компонента `DiscountManagerBean` находится в пакете примеров, доступных для загрузки, в подмодуле Maven `chapter15-ejb`. Для начала рассмотрим реализацию компонента, чтобы понять, как его тестировать.

Листинг 15.1. DiscountManagerBean

```
@Stateless
public class DiscountManagerBean implements DiscountManager {

    @EJB
    // ❶ Зависимость, которую нужно сымитировать
    MembershipLevelManager membershipLevelManager;

    public DiscountManagerBean() {}

    // ❷ Конструктор, необходимый модульному тесту для внедрения имитации
    public DiscountManagerBean(MembershipLevelManager mock) {
```

```
        this.membershipLevelManager = mock;
    }

    @Override
    // ❸ Метод, который требуется протестировать
    public double findDiscount(Member member) {
        double discount = 0.0;
        MembershipLevel ml
            // ❹ Вызов метода должен возвращть фиктивные данные
            //    во время модульного тестирования
            = membershipLevelManager.findByMemberId(member.getId());
        if (ml != null) {
            switch (ml.getType()) {
                case SILVER:
                    discount = 0.05;
                    break;
                case GOLD:
                    discount = 0.10;
                    break;
                case PLATINUM:
                    discount = 0.12;
                    break;
            }
        }
        // ❺ Значение, возвращаемое методом,
        //    должно быть проверено тестом
        return discount;
    }
}
```

Компонент `DiscountManagerBean` зависит от компонента `MembershipLevelManager` ❶. Следуя фундаментальным принципам разработки модульных тестов, необходимо сымитировать эту зависимость. Имитация гарантирует независимость от внешних ресурсов, которые могут потребоваться компоненту `MembershipLevelManager` и, что тестированию будет подвергаться только `DiscountManagerBean`. Для внедрения имитации в `DiscountManagerBean` реализован конструктор ❷, который может использоваться модульным тестом. Метод `findDiscount()` ❸ содержит прикладную логику, которую необходимо протестировать. Как видите, результат, возвращаемый методом ❺, зависит от двух исходных значений. Первое – параметр `Member member` метода. Второе – объект `MembershipLevel` ❹, возвращаемый компонентом `MembershipLevelManager`. Полнота охвата этого метода модульным тестированием зависит от того, насколько полно будут сымитированы все возможные комбинации исходных значений.

Так как модульные тесты, как предполагается, должны выполняться быстро, они обычно включаются в тот же проект, что и тестируемые ими классы. Класс `DiscountManagerBeanTest`, представленный в листинге 15.2, можно найти в подмодуле Maven `chapter15-ejb`, в пакете примеров для данной главы. Теперь посмотрим, как пишутся модульные тесты.

Листинг 15.2. DiscountManagerBeanTest

```
public class DiscountManagerBeanTest {

    // ❶ Свойства уровня класса, используемые всеми тестами
    private Member member;
    private MembershipLevel membershipLevel;
    private MembershipLevelManager membershipLevelManagerMock;
    private DiscountManagerBean bean;

    @Before
    public void setUp() {
        // ❷ Создать экземпляр Member для использования тестами
        member = new Member();
        member.setId(11L);
        member.setUsername("junit123");

        // ❸ Создать экземпляр MembershipLevel для использования тестами
        membershipLevel = new MembershipLevel();
        membershipLevel.setId(44L);
        membershipLevel.setMember(member);

        // ❹ Создать имитацию MembershipLevelManager с помощью Mockito
        membershipLevelManagerMock = mock(MembershipLevelManager.class);
        // ❺ Определить, что должен имитировать фиктивный объект
        when(membershipLevelManagerMock.findByMemberId(
            // ❻ Определить, что должен возвращать фиктивный объект
            member.getId())).thenReturn(membershipLevel);

        // ❼ Создать экземпляр тестируемого компонента
        bean = new DiscountManagerBean(membershipLevelManagerMock);
    }

    // ❽ Модульный тест
    @Test
    public void userGetsGoldDiscount() {
        // ❾ Определить конкретные исходные значения для этого теста
        membershipLevel.setType(MembershipLevelType.GOLD);
        double discount = bean.findDiscount(member);
        // ❿ Проверить правильность результата
        assertEquals(0.10, discount, 0.0);
    }

    // остальные тесты опущены для экономии места

    @Test
    public void userGetsNoDiscountBecauseNotAMember() {
        when(membershipLevelManagerMock.findByMemberId(
            // ⓫ Изменить значение, возвращаемое фиктивным объектом
            member.getId())).thenReturn(null);
        double discount = bean.findDiscount(member);
        assertEquals(0.0, discount, 0.0);
    }
}
```

Первое, что сразу бросается в глаза в этом модульном тесте – наличие свойств уровня класса ❶. Эти свойства будут иметь одинаковые значения во всех тестах, поэтому мы вынесли их на уровень класса, чтобы избежать дублирования большого количества кода.

Далее следует метод, отмеченный аннотацией `@Before`, который отвечает за присваивание значений свойствам уровня класса перед началом фактического тестирования. Он создает экземпляр `Member` ❷ для тестирования. Аналогично создается экземпляр `MembershipLevel` ❸. Однако экземпляр `MembershipLevelManager` подвергается специальной обработке вызовом `Mockito.mock()` ❹, потому что вместо него в тестах должна использоваться имитация (фиктивный объект). Создание имитаций позволяет легко определять, как должен вести себя объект в процессе тестирования. Метод `Mockito.when()` ❺ используется, чтобы определить поведение объекта во время тестирования, то есть, в данном случае определяется, что при обращении к методу `findByMemberId()` с параметром `member.getId()` ❻, фиктивный объект, имитирующий `MembershipLevelManager`, должен вернуть экземпляр `MembershipLevel`. После создания фиктивного объекта создается экземпляр компонента `DiscountManagerBean` ❼, который является предметом тестирования.

Тестируйте классы по одному

Кто-то из вас может испытать беспокойство по поводу имитации `MembershipLevelManager`. В конце концов, если подменить `MembershipLevelManager` имитацией, его код не будет выполняться.

Не волнуйтесь, здесь все в порядке, потому что основной целью теста является `DiscountManagerBean`. Наш модульный тест должен проверить только `DiscountManagerBean` и подменить имитациями все его зависимости, предполагая при этом, что они работают правильно. Но как узнать, действительно ли они работают правильно?

Чтобы узнать это, нужно создать дополнительные тесты! По аналогии с тестом для компонента `DiscountManagerBean` вам потребуется создать тест для `MembershipLevelBean`. Запомните важное правило: если для какого-то прикладного класса в модульном тесте создается имитация, значит для этого класса следует создать собственный тест.

Далее следует один из тестовых методов, отмеченных аннотацией `@Test` – `userGetsGoldDiscount()` ❽. Имя метода указывает на цель теста – он должен проверить размер скидки, возвращаемой компонентом `DiscountManagerBean` для «золотого» члена. Внутри этого метода уровень членства устанавливается в значение `GOLD` ❾. Далее вызывается метод `DiscountManagerBean.findDiscount()` и его результат сравнивается с ожидаемым ❿. Так как тесты, проверяющие другие уровни членства, практически идентичны методу `userGetsGoldDiscount()`, мы не будем тратить время на них – вы можете загрузить примеры для этой главы и исследовать их самостоятельно. Но мы посмотрим на еще один тестовый метод – `userGetsNoDiscountBecauseNotAMember()`.

Как следует из имени этого метода, данный тест проверяет поведение компонента `DiscountManagerBean` в случае, если пользователь не приобрел платное членство. Для этого внутри теста вызывается метод `Mockito.when()` ⓫, что-

бы определить новое значение, которое должна возвращать имитация класса `MembershipLevelManager`. Этим новым значением является `null`. Заставив имитацию класса `MembershipLevelManager` возвращать `null`, мы получили возможность проверить правильность работы `DiscountManagerBean` в этой ситуации.

Модульное тестирование обладает очень широкими возможностями. Оно позволяет убедиться в правильности работы приложения, но не гарантирует полное отсутствие ошибок. После создания объектов сервером Java EE и связывания их между собой, многое может измениться и эти изменения невозможно учесть в модульных тестах. Это особенно характерно для приложений, зависящих от внешних ресурсов, таких как базы данных. Компонент `DiscountManagerBean` зависит от наличия данных о членстве пользователей в базе данных, но проверка возможности благополучного извлечения этих данных выходит за рамки модульного теста. Чтобы убедиться, что классы сущностей успешно могут работать с базой данных, необходимо подняться на следующий уровень тестирования. Этот следующий уровень – интеграционное тестирование, о котором рассказывается далее.

15.3. Интеграционное тестирование с использованием EJBContainer

Давайте посмотрим, как провести интеграционное тестирование того же компонента `DiscountManagerBean` из листинга 15.1. В предыдущем разделе мы рассмотрели пример модульного тестирования `DiscountManagerBean`. Мы показали, насколько просто с помощью фреймворков JUnit и Mockito создаются имитации с данными и используются для проверки результатов работы прикладных методов. В интеграционных тестах мы проверим те же самые результаты, но сделаем это совершенно иначе.

Наиболее важным аспектом интеграционного тестирования является максимально полная имитация среды выполнения, чтобы по результатам интеграционного тестирования можно было сделать выводы о том, как будет действовать приложение под управлением настоящего сервера. Добиться этого можно разными способами. В этом разделе мы сконцентрируемся на применении `EJBContainer` – встраиваемого контейнера, который легко можно запустить из любого приложения Java SE. Встраиваемый контейнер должен запускаться в начале процедуры тестирования вызовом метода-построителя класса `EJBContainer`, как показано ниже:

```
EJBContainer ejbContainer = EJBContainer.createEJBContainer();
```

После этого с помощью экземпляра `EJBContainer` можно получить контекст `Context`:

```
Context ctx = ejbContainer.getContext();
```

Имея контекст, без труда можно находить любые привязки для компонента в JNDI. Получив экземпляр компонента, вы сможете вызывать любые его методы, которые желаете протестировать:

```
DiscountManager manager = (DiscountManager) ctx.lookup(
    "java:global/chapter15-ejb-1.0/DiscountManagerBean");
```

Все вышеизложенное было лишь кратким введением в `EJBContainer`. Несмотря на то, что запуск и использование встраиваемого контейнера выглядит достаточно простым делом, на практике так бывает не всегда. Чтобы заставить встроенный контейнер функционировать правильно и поддерживать все ваши компоненты EJB, его необходимо настроить, что иногда может оказаться сложной задачей. Однако усилия окупятся сторицей, потому что интеграционное тестирование позволяет получить ценную информацию о приложении.

15.3.1. Настройка проекта

Прежде чем перейти к программному коду интеграционных тестов, необходимо создать и настроить проект, который будет запускать их. Но для этого следует сначала определиться, где будет размещаться код интеграционных тестов. В соответствии с соглашениями, реализация модульных тестов включается в один проект с тестируемыми классами, чтобы имелась возможность быстро выполнить тестирование и выявить возможные нарушения, которые могли быть внесены в код с последними изменениями. Но где должны храниться интеграционные тесты?

Как правило, для интеграционного тестирования требуется выполнить гораздо большее число операций по настройке окружения, а так как эти тесты выполняются под управлением встраиваемого контейнера, они выполняются значительно дольше. Из-за сложности и увеличенного времени, необходимого для выполнения, интеграционные тесты часто располагаются в отдельном проекте. Именно этим путем мы и пойдем в нашем примере интеграционного тестирования. Код в загружаемом пакете примеров для этой главы содержит подмодуль Maven с именем `chapter15-ejb-embedded-test`, куда мы поместили интеграционные тесты, использующие встраиваемый контейнер `EJBContainer`.

Будут ли мои интеграционные тесты запускаться?

При размещении интеграционных тестов в отдельный проект возникает проблема их доступности для запуска. Модульные тесты, находясь в том же проекте, выполняются инструментом Maven по умолчанию. Но, из-за того, что интеграционные тесты оформлены в виде отдельного проекта, разработчику потребуется выполнить ряд операций вручную, чтобы выполнить их, а это часто приводит к тому, что многие просто пропускают этот шаг.

Решить эту проблему помогут инструменты, такие как Bamboo, Jenkins и другие средства автоматизации сборки. Наиболее удачным считается решение настроить эти инструменты на автоматическое выполнение тестов после любых изменений в проекте. Как вариант, можно настроить выполнение интеграционного тестирования по расписанию, в нерабочие часы, например, вечером.

Следуя таким путем, вы снимаете груз ответственности за выполнение интеграционных тестов с команды разработчиков и перекладываете ее на средства автоматизации. Но, если в процессе тестирования будут выявлены ошибки, ответственность вновь перекладывается на разработчиков.

Теперь, когда мы определились, что код реализации интеграционных тестов будет размещаться в отдельном проекте, посмотрим, как настроить этот проект. Компоненты EJB, подлежащие тестированию, находятся в подмодуле Maven с именем `chapter15-ejb`. Все они достаточно просты, поэтому конфигурация контейнера `EJBContainer` не отличается сложностью, но и не так проста, как хотелось бы, поэтому есть смысл продемонстрировать ее. Сначала мы рассмотрим файл конфигурации *pom.xml* для Maven.

Maven

В файл */chapter15-ejb-embedded-test/pom.xml* необходимо внести множество изменений, чтобы получить возможность выполнять интеграционные тесты, использующие встраиваемый контейнер `EJBContainer`. Первое изменение – определение свойства с номером версии проекта `chapter15-ejb`, подлежащего интеграционному тестированию. В процессе развития компонентов EJB практически всегда появляется множество версий. С появлением новых версий тесты также должны изменяться, чтобы соответствовать тестируемому коду. Это свойство определяет, какая версия компонента EJB будет тестироваться:

```
<properties>
  <chapter15-ejb.version>1.0</chapter15-ejb.version>
</properties>
```

Причина определения версии в виде свойства состоит в том, что это значение потребуется еще в нескольких местах, в файле *pom.xml*. Дублирование версий в коде весьма нежелательно, поэтому было решено хранить требуемое значение в виде свойства. Первое место, где будет использовано это свойство, – определение проекта `chapter15-ejb` в зависимости. Добавление выполняется точно так же, как добавление любых других зависимостей Maven, только при этом используется свойство со значением версии:

```
<dependency>
  <groupId>com.actionbazaar<groupId>
  <artifactId>chapter15-ejb</artifactId>
  <version>${chapter15-ejb.version}</version>
  <scope>test</scope>
</dependency>
```

Это свойство потребуется еще в одном месте, в файле *pom.xml*. Нам необходимо настроить расширение `maven-surefire-plugin` и передать имя модуля в интеграционный тест. Зачем это нужно, вы узнаете чуть ниже, а пока взгляните, как выглядит конфигурация расширения `maven-surefire-plugin`:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.10</version>
      <configuration>
```

```
        <systemProperties>
          <property>
            <name>moduleName</name>
            <value>chapter15-ejb-${chapter15-ejb.version}</value>
          </property>
        </systemProperties>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Последнее изменение в файле *pom.xml* – определение зависимости от `glassfish-embedded-all`, реализации встраиваемого контейнера `EJBContainer` в GlassFish. Это самая обычная зависимость Maven:

```
<dependency>
  <groupId>org.glassfish.extras</groupId>
  <artifactId>glassfish-embedded-all</artifactId>
  <version>3.1</version>
  <scope>test</scope>
</dependency>
```

После добавления в проект зависимости от GlassFish следует определить конфигурацию для GlassFish, необходимую для интеграционного тестирования компонентов EJB в проекте.

GlassFish

Так как мы настроили Maven на использование реализации встраиваемого контейнера `EJBContainer` из GlassFish, нам также необходимо добавить в проект некоторые настройки для самого сервера GlassFish. Эти настройки находятся в подмодуле Maven `chapter15-ejb-embedded-test`, в каталоге */src/test/domain/config*. Файлы, находящиеся в этом каталоге, были скопированы из каталога *domain1/config* стандартной установки GlassFish. Мы внесли некоторые изменения, чтобы привести настройки в соответствие с проектом – изменения выполнялись только в файле *domain.xml*.

Во-первых, в файле *domain.xml* изменена настройка порта – перед номером по умолчанию была добавлена цифра 2, в надежде избежать во время тестирования конфликтов с уже занятыми портами. Например, по умолчанию прослушивается порт с номером 8080, но для интеграционного тестирования будет использоваться порт с номером 28080. Аналогично изменены все остальные номера портов. Во-вторых, во всех настройках `jdbc-connection-pool` в качестве источника данных настроено использование `org.apache.derby.jdbc.EmbeddedDataSource`. Для этого источника данных Derby автоматически будет создавать базу данных в памяти, освобождая от необходимости использовать внешний сервер баз данных. Наконец, специально для интеграционного тестирования добавлен новый параметр `jdbc-connection-pool` с настройками для сущностей JPA. Теперь, после настройки базы данных для использования в интеграционных тестах, давайте посмотрим, как настроить JPA на ее использование.

JPA

В нашем интеграционном тесте мы собираемся проверить компонент `DiscountManagerBean`, логика работы которого зависит от информации о членстве из сущности `MembershipLevel`. Механизм JPA должен иметь возможность взаимодействовать со встроенной базой данных Derby, создаваемой во время выполнения интеграционного тестирования. Для этого необходимо определить параметры в файле */src/test/resources/META-INF/persistence.xml*, настраивающие использование источника данных, ссылающегося на пул соединений со встраиваемой базой данных Derby:

```
<jta-data-source>jdbc/chapter15-ejb-embedded</jta-data-source>
```

Данная привязка JDBC предназначенная для настройки `jdbc-connection-pool`, добавленной в файл *domain.xml* для нужд интеграционного тестирования. В GlassFish имеется пул соединений по умолчанию, который можно использовать в интеграционном тестировании, но вообще использование ресурсов по умолчанию, как в данном случае, считается плохой практикой. С точки зрения интеграционного тестирования предпочтительнее создать новый ресурс и использовать его. В этом примере мы настроили новый источник данных в GlassFish для интеграционного тестирования, связанный со встроенной базой данных Derby, а теперь сконфигурировали JPA на его использование.

Итак, мы охватили вопросы настройки проекта. А теперь перейдем к более интересному – познакомимся с кодом реализации интеграционного теста.

15.3.2. Интеграционный тест

В листинге 15.3 представлен код реализации интеграционного теста. В нем, для развертывания компонентов EJB, используется встраиваемый контейнер `EJBContainer`.

Листинг 15.3. Тест DiscountManagerBeanEmbeddedTest, использующий встраиваемый контейнер EJBContainer

```
public class DiscountManagerBeanEmbeddedTest {

    private static Context ctx;
    private static EJBContainer ejbContainer;
    // ❶ Получить значение системного свойства для поиска в JNDI
    private static String moduleName = System.getProperty("moduleName");

    @BeforeClass
    // ❷ Запускает контейнер EJBContainer перед началом тестирования
    public static void setUpClass() {
        Map<String, Object> properties = new HashMap<>();
        // ❸ Подсказать, где искать domain.xml с настройками для тестов
        properties.put("org.glassfish.ejb.embedded.glassfish.instance.root",
            "./src/test/domain");
        // ❹ Запустить встраиваемый контейнер EJBContainer
        ejbContainer = EJBContainer.createEJBContainer(properties);
```

```
        ctx = ejbContainer.getContext();
    }

    @AfterClass
    public static void tearDownClass() {
        if (ejbContainer != null) {
            ejbContainer.close();
        }
    }

    // ❺ Метод интеграционного теста
    @Test
    public void userGetsGoldDiscount() throws NamingException {
        // ❻ Добавить информацию о членстве в базу данных
        Member member = new Member();
        member.setId(1L);
        member.setUsername("junitGoldMember");
        lookupMemberManager().insert(member);

        // ❼ Добавить информацию об уровне членства в базу данных
        MembershipLevel membershipLevel = new MembershipLevel();
        membershipLevel.setMember(member);
        membershipLevel.setType(MembershipLevelType.GOLD);
        lookupMembershipLevelManager().insert(membershipLevel);

        DiscountManager discountManager = lookupDiscountManager();
        // ❽ Интеграционное тестирование метода findDiscount()
        double discount = discountManager.findDiscount(member);
        assertEquals(0.10, discount, 0.0);
    }

    // ❾ Три вспомогательных метода для поиска компонентов EJB в JNDI
    static DiscountManager lookupDiscountManager()
            throws NamingException {
        return (DiscountManager)lookupEjb(EjbName.DiscountManagerBean);
    }

    static MemberManager lookupMemberManager()
            throws NamingException {
        return (MemberManager)lookupEjb(EjbName.MemberManagerBean);
    }

    static MembershipLevelManager lookupMembershipLevelManager()
            throws NamingException {
        return (MembershipLevelManager)lookupEjb(
            EjbName.MembershipLevelManagerBean);
    }

    /** Общий метод, используемый всеми тремя методами поиска в JNDI */
    // ❿ Перечисление, хранящее имена компонентов EJB для поиска в JNDI
    static enum EjbName {
        DiscountManagerBean, MemberManagerBean, MembershipLevelManagerBean
    }

    // ⓫ Общий метод, осуществляющий поиск в JNDI
```

```
    static Object lookupEjb(EjbName ejbName) throws NamingException {
        try {
            return ctx.lookup("java:global/"+moduleName+"/"+ejbName);
        } catch (Throwable ignore) {
            moduleName = "classes";
            return ctx.lookup("java:global/"+moduleName+"/"+ejbName);
        }
    }
}
```

Первое, на что следует обратить внимание в этой реализации интеграционного теста, – вызов `getProperty`, чтобы получить системное свойство `moduleName` ❶. Что это за свойство? Оно используется для поиска компонентов EJB в каталоге JNDI после запуска встраиваемого контейнера. В какой момент устанавливается это свойство? Вспомните изменения в конфигурации расширения `maven-surefire-plugin` в файле *pom.xml*. Мы использовали значение свойства `chapter15-ejb.version` в *pom.xml*, чтобы сгенерировать имя модуля `moduleName` для `maven-surefire-plugin`, а расширение `maven-surefire-plugin`, в свою очередь, передает `moduleName` интеграционному тесту в виде системного свойства `moduleName`. Такая схема была выбрана потому, что `moduleName` зависит от версии проекта `chapter15-ejb`. Чтобы избавиться от необходимости дублирования номера версии, мы передаем ее в виде свойства. Это существенно упрощает изменение номера версии в будущем.

Следующее, на что нужно обратит внимание в этом интеграционном тесте, – метод `setUpClass()` с аннотацией `@BeforeClass` ❷. Этот метод вызывается фреймворком JUnit перед созданием экземпляра класса теста для настройки любых статических ресурсов – в данном случае метод запускает встраиваемый контейнер `EJBContainer`. Для настройки `EJBContainer` создается ассоциативный массив и в него добавляется свойство `org.glassfish.ejb.embedded.glassfish.instance.root` со значением `./src/test/domain` ❸. Так настраивается каталог, где встраиваемый контейнер GlassFish будет искать конфигурационные файлы. Эта настройка непереносима и поддерживается только реализацией `EJBContainer` в GlassFish. После определения конфигурации метод `setUpClass()` запускает встраиваемый контейнер вызовом `EJBContainer.createEJBContainer(properties)` ❹.

К этому моменту самая сложная часть оказывается позади. Фактически, настройка и запуск встраиваемого контейнера оказываются намного сложнее, чем его использование в интеграционных тестах.

Теперь рассмотрим тестовый метод `userGetsGoldDiscount()` ❺. Метод с точно таким же именем уже был представлен в листинге 15.2 с реализацией модульного теста. Несмотря на то, что оба теста, модульный и интеграционный, по сути выполняют одну и ту же проверку, делают они это совершенно по-разному. Модульный тест использует фиктивные объекты, чтобы избежать использования контейнера EJB, и тестирует компонент `DiscountManagerBean` как простой POJO. Интеграционный тест, напротив, пытается опробовать работу компонента `DiscountManagerBean` во встраиваемом контейнере `EJBContainer`. Давайте посмотрим, что для этого должен сделать интеграционный тест.

Мы сконфигурировали базу данных Derby в памяти, в файле *domain.xml*, и связали ее с `jdbc/chapter15-ejb-embedded` в JNDI. Встраиваемая база данных создается каждый раз, когда запускается интеграционный тест, поэтому в нее необходимо добавить начальные данные. Метод `userGetsGoldDiscount()` создает сущность `Member`, записывает в нее данные и вставляет ее в базу данных ❻. Та же операция проделывается с сущностью `MembershipLevel` ❼. После добавления всех необходимых исходных данных вызывается метод `DiscountManagerBean.findDiscount()` ❽, а возвращаемый им результат сравнивается с ожидаемым значением.

Важно помнить, что вызывая метод `DiscountManagerBean.findDiscount()`, вы не просто вызываете метод POJO с имитациями зависимостей, как в модульном тестировании (см. листинг 15.2). В этом интеграционном тесте, когда вызывается метод `findDiscount()`, выполняется действительный метод компонента EJB из контейнера EJB, использующий действительную базу данных. Поэтому интеграционный тест позволяет убедиться не только в том, что `DiscountManagerBean` возвращает корректное значение, но и в том, что другие технологии – CDI, EJB, JPA, база данных – работают в комплексе, как ожидается.

Что еще имеется в классе интеграционного теста, кроме методов запуска встраиваемого контейнера и самого теста? Здесь есть три вспомогательных метода, используемых для извлечения компонентов из контейнера `EJBContainer` ❾: `lookupDiscountManager()`, `lookupMemberManager()` и `lookupMembershipLevelManager()`. Все они действуют совершенно одинаково. Они используют перечисление `EjbName` ❿, чтобы сообщить методу `lookupEjb()` ⓫, какой именно компонент требуется найти. Метод `lookupEjb()` сначала пытается выполнить поиск JNDI с использованием значения свойства `moduleName`, и в случае ошибки пытается повторить поиск с использованием жестко установленного значения `"classes"`. Оба варианта являются правильными, в зависимости от того, как инструмент Maven запускает тесты.

На этом завершается охват интеграционного тестирования с использованием встраиваемого контейнера `EJBContainer`. Как видно из короткого примера, встраиваемый контейнер `EJBContainer` является мощным инструментом, способным поднять тестирование приложений на новый уровень. Однако пользоваться им не всегда просто. Настройка этого инструмента сложна и может восприниматься как препятствие к его применению. Для устранения ненужных сложностей и упрощения интеграционного тестирования были созданы специализированные инструменты, такие как Arquillian, с которым мы познакомимся далее.

15.4. Интеграционное тестирование с применением Arquillian

В предыдущем разделе мы обсуждали интеграционное тестирование с использованием встраиваемого контейнера `EJBContainer`. Теперь рассмотрим тот же самый пример, но на этот раз воспользуемся фреймворком интеграционного тестирования с названием Arquillian. У многих наверняка появится вопрос: зачем

использовать Arquillian, если есть возможность использовать `EJBContainer`? Все просто, Arquillian делает интеграционное тестирование с применением встраиваемого контейнера намного проще, благодаря чему можно сосредоточиться на самих интеграционных тестах, переложив решение рутинных задач на фреймворк. Arquillian помогает также упростить сложные сценарии интеграционного тестирования. Например, Arquillian включает обертки для разных серверов приложений, что может пригодиться, если ваша организации находится на полпути переход от одного сервера приложений к другому, или если вы желаете организовать в своем продукте поддержку нескольких серверов приложений, чтобы расширить круг потенциальных клиентов.

Arquillian поддерживает также удаленное интеграционное тестирование. В некоторых случаях интеграционного тестирования с контейнерами и ресурсами в памяти оказывается недостаточно. В таких ситуациях может создаваться отдельное промышленное окружение, специально для интеграционного тестирования. Arquillian может подключаться к таким окружениям и выполнять ваши интеграционные тесты. Это лишь некоторые из причин, по которым может появиться желание использовать Arquillian. Давайте предположим, что мы также решили задействовать фреймворк Arquillian для интеграционного тестирования приложения ActionBazaar. Прежде всего, посмотрим, какие настройки при этом необходимо выполнить в проекте.

15.4.1. Настройка проекта

В загружаемых примерах для этой главы имеется подмодуль Maven с именем `chapter15-ejb-arquillian-test`. Этот проект содержит интеграционные тесты Arquillian для компонентов EJB в `chapter15-ejb`.

Как рассказывалось в предыдущем разделе, для интеграционного тестирования с применением встраиваемого контейнера `EJBContainer` необходимо решить, где будет находиться код интеграционных тестов. При использовании фреймворка Arquillian интеграционные тесты должны располагаться в отдельном внешнем проекте по тем же причинам, почему и интеграционные тесты, использующие `EJBContainer`. Интеграционные тесты обычно намного сложнее, требуют привлечения внешних ресурсов, и для их выполнения требуется больше времени, чем для выполнения модульных тестов. Поэтому интеграционные тесты для Arquillian принято помещать в отдельный проект. Инструменты непрерывного интеграционного тестирования, такие как Bamboo или Jenkins, можно настроить так, что они будут выполнять тестирование автоматически, если разработчики забудут сделать это перед тем, как отправить измененный код в репозиторий.

Настройка проекта на запуск интеграционных тестов под управлением Arquillian напоминает настройку применения встраиваемого контейнера `EJBContainer`. Maven, Arquillian, GlassFish и Derby – все эти компоненты требуется настроить для выполнения тестов. Для тестирования будет использоваться обертка Arquillian вокруг `EJBContainer` из GlassFish, именно поэтому необходимо настроить GlassFish. Но перед этим рассмотрим настройки Maven.

Maven

Сначала мы изменим настройки для Maven, добавив зависимости Arquillian в *pom.xml*. Первое изменение определяет, какую версию Arquillian использовать:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.arquillian</groupId>
      <artifactId>arquillian-bom</artifactId>
      <version>1.1.0.Final</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Затем нужно настроить интеграцию проекта Arquillian с фреймворком JUnit. Это позволит использовать жизненный цикл тестов Maven для запуска интеграционных тестов, как если бы они были модульными тестами. Имейте в виду: использование JUnit не превращает интеграционные тесты в модульные. Фреймворк JUnit в этом случае служит всего лишь инструментом для запуска Arquillian и интеграционных тестов:

```
<dependency>
  <groupId>org.jboss.arquillian.junit</groupId>
  <artifactId>arquillian-junit-container</artifactId>
  <scope>test</scope>
</dependency>
```

Наконец, требуется настроить обертки Arquillian вокруг GlassFish, потому что мы используем реализацию `EJBContainer` из GlassFish:

```
<dependency>
  <groupId>org.jboss.arquillian.container</groupId>
  <artifactId>arquillian-glassfish-embedded-3.1</artifactId>
  <version>1.0.0.CR3</version>
</dependency>
```

Мы охватили описание зависимостей от Arquillian в *pom.xml*. Но, нам нужно рассмотреть еще две зависимости: настройку местоположения компонентов EJB в проекте `chapter15-ejb` (который мы собираемся протестировать) и настройку расширения `glassfish-embedded-all`. Эти две зависимости настраиваются точно так же, как проект `chapter15-ejb-embedded-test`, обсуждавшийся в предыдущем разделе:

```
<dependency>
  <groupId>com.actionbazaar</groupId>
  <artifactId>chapter15-ejb</artifactId>
  <version>1.0</version>
  <scope>test</scope>
</dependency>

<dependency>
```

```
  <groupId>org.glassfish.extras</groupId>
  <artifactId>glassfish-embedded-all</artifactId>
  <version>3.1</version>
  <scope>test</scope>
</dependency>
```

Этими настройками завершается конфигурирование Maven. Обычно все, что нужно сделать, – это добавить описание зависимостей в файл *pom.xml*. После добавления зависимостей нужно настроить остальные технологии, используемые в интеграционном тестировании: Arquillian, Derby и GlassFish. Начнем с Arquillian, затем перейдем к настройке GlassFish и Derby.

Arquillian

Arquillian, как инструмент интеграционного тестирования с богатыми возможностями, требует некоторой настройки. Однако полное описание его настроек далеко выходит за рамки данной книги – мы рассмотрим лишь простейшую конфигурацию Arquillian, необходимую для интеграционного тестирования компонентов EJB из проекта `chapter15-ejb`. Настройки Arquillian хранятся в файле *src/test/resources/arquillian.xml*. Так как интеграционное тестирование будет выполняться с использованием GlassFish, нужно сообщить инструменту Arquillian, где искать настройки ресурсов GlassFish для интеграционного тестирования. Это простое свойство в *arquillian.xml*, указывающее на файл *glassfishresources.xml*:

```
<arquillian xmlns="http://jboss.org/schema/arquillian"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="
            http://jboss.org/schema/arquillian
            http://jboss.org/schema/arquillian/arquillian_1_0.xsd">
  <container qualifier="glassfish-embedded" default="true">
    <configuration>
      <property name="resourcesXml">
        src/test/glassfish/glassfish-resources.xml
      </property>
    </configuration>
  </container>
</arquillian>
```

Достаточно просто, не правда ли? Хотите верьте, хотите нет, но это все, что требуется для Arquillian. Теперь давайте посмотрим, что требуется для настройки GlassFish и Derby.

GlassFish и Derby

Так же как в разделе 15.3, где обсуждался пример интеграционного тестирования с непосредственным использованием встраиваемого контейнера `EJBContainer`, нам нужно настроить GlassFish, потому что в этом примере используется реализация `EJBContainer` из GlassFish. Для организации интеграционного тестирования с помощью Arquillian, следует создать файл *src/test/glassfish/glassfish-resources.xml*. В этом файле должен определяться пул соединений `jdbc-connection-pool` для GlassFish и его привязка в JNDI к `jdbc/chapter15-ejb-arquillian`:

```
<!DOCTYPE resources PUBLIC
   "-//GlassFish.org//DTD GlassFish Application Server 3.1
    Resource Definitions//EN"
   "http://glassfish.org/dtds/glassfish-resources_1_5.dtd">
<resources>
  <jdbc-resource pool-name="ArquillianEmbeddedDerbyPool"
    jndi-name="jdbc/chapter15-ejb-arquillian"/>
  <jdbc-connection-pool name="ArquillianEmbeddedDerbyPool"
    res-type="javax.sql.DataSource"
    datasource-classname="org.apache.derby.jdbc.EmbeddedDataSource"
    is-isolation-level-guaranteed="false">
    <property name="databaseName"
      value="target/derby/arquillian-integration-test"/>
    <property name="createDatabase" value="create"/>
  </jdbc-connection-pool>
</resources>
```

Обратите внимание, что настройки GlassFish при использовании Arquillian значительно отличаются от настроек при непосредственном использовании встраиваемого контейнера `EJBContainer`. При использовании `EJBContainer` обычно нужно определить настройки для всего домена GlassFish. При использовании Arquillian достаточно определить настройки только для нужных ресурсов – в данном примере, только пул соединений с базой данных Derby.

Последней составляющей, которую требуется настроить, является JPA. Давайте посмотрим, каие настройки необходимы.

JPA

Настройки JPA находятся в файле *src/test/resources/META-INF/persistence.xml*. Они фактически идентичны настройкам, что приводились в разделе 15.3, где описывается пример интеграционного тестирования с использованием встраиваемого контейнера `EJBContainer`. Единственное отличие – имя источника данных, искомое в JNDI. Для нужд интеграционного тестирования с применением Arquillian используется источник данных `jdbc/chapter15-ejb-arquillian`, соответственно файл *persistence.xml* содержит следующие строки:

```
<persistence version="2.1" 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">
  <persistence-unit name="ActionBazaar" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>jdbc/chapter15-ejb-arquillian</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
      <property
        name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

В данном случае необязательно было назначать источнику данных другое имя в каталоге JNDI – в обоих проектах, `chapter15-ejb-embedded-test` и `chapter15-`

ejb-arquillian-test, можно было бы использовать одно и то же имя. Так было бы проще, но проще не всегда лучше. На практике принято назначать более специфические имена. В этом случае выше вероятность получить правильную конфигурацию, и вы сами будете лучше понимать, как разные технологии взаимодействуют друг с другом.

На этом мы завершаем определение настроек, необходимых для проекта chapter15-ejb-arquillian-test. Как видите, они похожи на настройки для проекта chapter15-ejb-embedded-test, в том смысле что точно так же требуется настраивать основные технологии: Maven, GlassFish, Derby и JPA. Но при этом каждый проект имеет свои небольшие отличия. Теперь, после определения настроек для проекта, можно переходить непосредственно к тесту.

15.4.2. Интеграционный тест

Интеграционные тесты при использовании Arquillian получаются более простыми и прямолинейными, чем при использовании встраиваемого контейнера EJBContainer, что упрощает их разработку и сопровождение. В листинге 15.4 приводится код интеграционного теста для компонента DiscountManager.

Листинг 15.4. Arquillian-тест DiscountManagerBeanArquillianTest

```
// ❶ Сообщить фреймворку JUnit, что он должен делегировать
//    запуск этого теста инструменту Arquillian
@RunWith(Arquillian.class)
public class DiscountManagerBeanArquillianTest {

    // ❷ Аннотация Arquillian для сборки JavaArchive, подлежащего тестированию
    @Deployment
    public static JavaArchive createDeployment() {
        // ❸ Определить JavaArchive для сборки
        JavaArchive jar = ShrinkWrap.create(JavaArchive.class)
            // ❹ Добавить в JavaArchive все классы из com.actionbazaar.ejb
            .addPackage(DiscountManagerBean.class.getPackage())
            // ❺ Добавить в JavaArchive все классы из com.actionbazaar.entity
            .addPackage(Member.class.getPackage())
            // ❻ Добавить в JavaArchive файл persistence.xml
            .addAsResource("META-INF/persistence.xml");
        System.out.println(jar.toString(true));
        return jar;
    }

    // ❼ Внедрить компоненты EJB для использования в тесте
    @EJB
    MemberManager memberManager;
    @EJB
    MembershipLevelManager membershipLevelManager;
    @EJB
    DiscountManager discountManager;

    @Before
    public void insertTestData() {
```

```
        // ❽ Добавить информацию о членстве в базу данных
        Member member = new Member();
        member.setId(1L);
        member.setUsername("junitGoldMember");
        memberManager.insert(member);

        // ❾ Добавить информацию об уровне членства в базу данных
        MembershipLevel membershipLevel = new MembershipLevel();
        membershipLevel.setMember(member);
        membershipLevel.setType(MembershipLevelType.GOLD);
        membershipLevelManager.insert(membershipLevel);
    }

    @Test
    public void userGetsGoldDiscount() {
        Member member = memberManager.find(1L);
        // ❿ Вызвать тестируемый метод
        double discount = discountManager.findDiscount(member);
        // ⓫ Проверить правильность результата
        assertEquals(0.10, discount, 0.0);
    }
}
```

Так как запуск этого теста осуществляется с помощью Arquillian, мы можем использовать более совершенные технологии Java, такие как внедрение зависимостей, недоступные в тестах на основе встраиваемого контейнера `EJBContainer`. Это отличная новость для нас, как для разработчиков, потому что эти технологии позволяют сосредоточиться на самих тестах, а не на рутинных операциях, связанных с обеспечением их работы.

Начнем с аннотации `@RunWith(Arquillian.class)` ❶. Эта аннотация является особенностью фреймворка JUnit и позволяет переложить ответственность за выполнение теста на другой инструмент. В данном случае фреймворку JUnit предписывается, что он должен делегировать выполнение теста инструменту Arquillian. Благодаря этому вы получаете полный доступ ко всем возможностям Arquillian, таким как аннотация `@Deployment`.

Далее следует аннотация `@Deployment`, принадлежащая уже инструменту Arquillian, а не фреймворку JUnit. За счет возможности указать фреймворку JUnit, что он должен передать право выполнения теста инструменту Arquillian, вы получаете возможность использовать особенности инструмента Arquillian в тестах. Аннотация `@Deployment` отмечает метод `public static JavaArchive createDeployment()` ❷. Мы указали здесь полную сигнатуру метода, так как в отличие от аннотаций JUnit, аннотация `@Deployment` предполагает, что метод вернет реализацию `JavaArchive`. Объект `JavaArchive` можно рассматривать как своеобразное представление JAR, EJB-JAR, WAR или EAR. В главе 5 мы рассматривали структуру каждого из этих архивов. При использовании Arquillian, метод с аннотацией `@Deployment` отвечает за создание экземпляра `JavaArchive` и включения в него классов и ресурсов (файлов *.properties*, *.xml* и других), которые обычно входят в состав заключительного архива проекта. Так как `chapter15-ejb` является проектом EJB-JAR, а его окончательный архив – файлом *.jar*, в интеграционном тесте

мы выбрали тип архива `JavaArchive.class` ❸. Далее в архив `JavaArchive` добавляются все классы из пакетов `com.actionbazaar.ejb` ❹ и `com.actionbazaar.entity` ❺, а также файл *persistence.xml* ❻ с настройками для JPA. После создания архива метод возвращает его инструменту Arquillian.

За кулисами, после получения экземпляра `JavaArchive`, Arquillian запускает встраиваемый контейнер, выбранный для использования (в данном случае реализацию GlassFish), и развертывает `JavaArchive` в контейнер. Не забывайте, что `JavaArchive` является абстракцией. В данном примере интеграционного теста создается EJB-JAR, но можно также создать простой архив JAR, WAR, RAR или EAR. Так как Arquillian запускает контейнер, интеграционному тесту становятся доступны все преимущества технологий Java EE. Это можно наблюдать в виде аннотаций `@EJB`, используемых для внедрения компонентов `MemberManager`, `MembershipLevelManager` и `DiscountManager` ❼. В предыдущем примере, основанном на непосредственном использовании встраиваемого контейнера `EJBContainer`, необходимо было выполнять поиск в JNDI. Благодаря отсутствию необходимости выполнения операций с JNDI, разработчики могут больше внимания уделять самим тестам. Интеграционные тесты при использовании Arquillian получаются более простыми и прямолинейными, потому что появляется возможность использовать преимущества более совершенных технологий Java. После запуска встраиваемого контейнера и внедрения зависимостей в тестовый класс Arquillian готов к выполнению тестов – ну или почти готов. Перед запуском тестов требуется еще вызвать методы-обработчики событий жизненного цикла JUnit.

В методе `insertTestData()`, отмеченном аннотацией `@Before`, производится добавление данных о членстве ❽ и об уровне членства ❾ в базу. Когда JUnit делегирует запуск тестов инструменту Arquillian, последний прерывает стандартный жизненный цикл JUnit, чтобы вызвать собственный метод `@Deployment`. Теперь, после выполнения этого метода, Arquillian должен продолжить выполнение жизненного цикла JUnit, что мы и наблюдаем здесь. Для выполнения интеграционного тестирования нам нужно заполнить базу данных корректной информацией, что мы и делаем, добавляя данные о членстве и об уровне членства. В этот момент контейнер запущен, зависимости внедрены и необходимые данные добавлены в базу (это может служить прямым доказательством, что контейнер `EJBContainer` работает). Теперь осталось лишь выполнить тесты.

Тестирование в нашем примере осуществляет метод `userGetsGoldDiscount()`, отмеченный аннотацией `@Test`. Из имени этого метода явно следует, что именно он проверяет – действительно ли для «золотого» члена возвращается ожидаемая величина скидки. Для этого он вызывает метод `findDiscount(member)` ❿. Напомним, что будет вызван настоящий метод компонента EJB, использующий настоящие сущности JPA, взаимодействующие с настоящей базой данных. Интеграционный тест настолько близко имитирует фактическое окружение, что в случае успеха может служить гарантией безупречной совместной работы используемых технологий. Вслед за вызовом `findDiscount(member)` полученное значение сравнивается с ожидаемым ⓫. Если проверка увенчалась успехом, вы увидите зеленую полоску.

Вот и все! Мы благополучно протестировали компоненты EJB из проекта `chapter15-ejb` с помощью инструмента Arquillian. В этом примере мы показали только один метод `@Test`, охватывающий только один случай, из числа обрабатываемых методом `findDiscount(member)`. И сразу возникает вопрос: стоит ли добавлять дополнительные методы `@Test` для проверки других случаев. Скажем честно, этот вопрос порождает жаркие споры в сообществе. В общем случае проверять другие случаи не требуется, потому что это не является целью интеграционного тестирования. Тестирование обработки всех возможных комбинаций входных данных и корректную работу прикладной логики должны выполнять модульные тесты. А интеграционный тест должен проверить правильное взаимодействие (интеграцию) всех задействованных технологий (например, EJB, JPA и базы данных). Разные стратегии тестирования преследуют разные цели, и вам самим придется решать, что именно тестировать в своих проектах.

На этом мы завершаем изучение темы тестирования компонентов EJB. Мы рассказали вам, как осуществлять модульное тестирование компонентов с помощью JUnit и Mockito. Мы также показали, как реализовать интеграционное тестирование компонентов, с непосредственным применением встраиваемого контейнера `EJBContainer` или с помощью инструментов интеграционного тестирования, таких как Arquillian. Попутно мы познакомили вас с некоторыми приемами увеличения эффективности тестирования. В следующем разделе мы вновь вернемся к этим прием и расскажем о них немного подробнее.

15.5. Приемы эффективного тестирования

Для эффективного тестирования программного кода следует использовать многоуровневое тестирование. На каждом уровне должны проверяться определенные аспекты работы приложения. Многоуровневая организация тестов обеспечит эффективность вашей стратегии тестирования.

Первый уровень тестирования – модульное тестирование. Целью модульного тестирования является проверка правильной работы всей прикладной логики во всех классах. Основная стратегия модульного тестирования состоит в том, чтобы тестировать прикладные классы по одному, используя имитации любых зависимостей и проверяя все возможные комбинации входных параметров и данных, возвращаемых имитациями. Результаты, получаемые тестируемым классом, должны сравниваться с ожидаемыми значениями. Необходимость создания имитации для класса, являющегося частью приложения, свидетельствует о необходимости написать модульные тесты и для этого класса, чтобы в приложении не осталось классов, не охваченных тестированием. Модульные тесты не должны требовать сложных настроек и должны выполняться максимально быстро, чтобы нарушения в работе прикладной логики обнаруживались немедленно. Модульные тесты не должны использовать внешние ресурсы (базы данных, веб-службы, каталог имен и так далее); все взаимодействия с такими внешними ресурсами должны имитироваться с

помощью фиктивных объектов, возвращающих соответствующие данные. Хорошо продуманный комплект модульных тестов может обеспечить 100% охват кода и свидетельствовать о безупречной работе прикладной логики, но при этом приложение может быть полностью неработоспособным при выполнении под управлением сервера Java EE. В этом нет ничего страшного, потому что это не является целью модульного тестирования. Проверка работоспособности под управлением сервера является целью следующего уровня тестирования – интеграционного тестирования.

Целью интеграционного тестирования является проверка возможности взаимодействий всех технологий используемых приложением. Интеграционное тестирование отвечает, например, на такие вопросы: «Выполняется ли сохранение информации в базе данных?», «Выполняется ли успешное развертывание компонента EJB?», «Вызываются ли интерцепторы в ожидаемом порядке?». В процессе модульного тестирования проверяется правильность работы отдельных классов, тогда как в процессе интеграционного тестирования проверяется возможность взаимодействий отдельных классов между собой (внутри контейнера) и соответствие результатов ожиданиям. Для нужд интеграционного тестирования спецификация Java EE определяет встраиваемый контейнер `EJBContainer`, запускаемый любым кодом Java SE. Контейнер `EJBContainer` можно использовать непосредственно или косвенно, с помощью инструментов интеграционного тестирования, таких как Arquillian. В любом случае, настройка проекта и настройка всех задействованных технологий может оказаться очень непростым делом. По этой причине, в противоположность модульным тестам, интеграционные тесты желательно размещать в отдельном проекте. Инструмент Maven поможет обеспечить внедрение зависимостей в код, подлежащий интеграционному тестированию, а инструменты непрерывной сборки, такие как Bamboo или Jenkins, – выполнение интеграционных тестов. Несмотря на то, что интеграционное тестирование является большим шагом, оно все же не способно охватить все возможные ситуации. Последний уровень тестирования – функциональное тестирование.

Мы не рассматривали функциональное тестирование в этой главе, только лишь упомянули о существовании этого уровня. Функциональное тестирование обычно осуществляется отдельной командой специалистов, основной задачей которых является использование приложения и проверка правильности его работы. На этом уровне приложение развертывается в тестовом окружении, максимально полно дублирующем окружение промышленной эксплуатации. Модульное тестирование поможет узнать, правильно ли работает прикладная логика, функциональное тестирование – корректно ли взаимодействуют разные части приложения. Функциональное тестирование – это последняя линия обороны от логических проблем. Модульное и интеграционное тестирование могут не обнаруживать ничего криминального, но беда в том, что прохождение тестов не всегда означает, что код не содержит логических ошибок. Например, в приложении ActionBazaar в некоторой точке может обнаружиться, что «золотым» членам предоставляется 80% скидка. Все модульные и интеграционные тесты могут утверждать, что приложение работает правильно, а в процессе функционального тестирования, осуществляемого человеком, может быть выявлена проблема.

Тестирование никогда не дает 100% гарантии, но комбинирование всех трех стратегий поможет вам получить надежное приложение. Это все, что мы хотели рассказать о тестировании компонентов EJB. Давайте вкратце повторим все, что узнали в этой главе.

15.6. В заключение

Эта глава представляет собой краткое введение в тестирование компонентов EJB. Первой темой, которую мы рассмотрели, было описание разных стратегий тестирования, в частности: модульного, интеграционного и функционального тестирования. Модульное тестирование мы определили, как стратегию, предназначенную для проверки прикладной логики путем передачи ей комбинаций разных исходных данных и сопоставления полученных результатов с ожидаемыми. При этом была высказана рекомендация осуществлять тестирование прикладных классов по одному и использовать средства создания фиктивных объектов, такие как фреймворк Mockito, для имитации любых зависимостей и получения от них значений, соответствующих условиям тестов. Модульные тесты должны быть простыми в обращении, выполняться максимально быстро, не требовать излишне сложных настроек и не использовать внешних ресурсов.

Интеграционное тестирование мы определили, как стратегию, целью которой является проверка возможности работы как единого целого всех технологий и ресурсов, используемых в приложении. Обычно в интеграционном тестировании применяются встраиваемые технологии, чтобы избежать необходимости использовать внешние ресурсы. Мы показали, как для тестирования компонентов EJB можно использовать встраиваемую базу данных Derby и реализацию встраиваемого контейнера `EJBContainer` в GlassFish. Со встраиваемым контейнером `EJBContainer` можно работать непосредственно или посредством более мощных инструментов интеграционного тестирования, таких как Arquillian. Применение Arquillian дает разработчикам тестов доступ к более совершенным технологиям Java, таким как CDI, которая позволяет внедрять ресурсы из встраиваемого контейнера прямо в тесты, что позволяет программисту сосредоточиться на написании самих тестов, а не на рутинных операциях, связанных с обеспечением их работы.

Наконец, функциональное тестирование мы определили, как стратегию, целью которой является выявления логических ошибок в приложениях. Обычно функциональное тестирование выполняется отдельной командой специалистов по тестированию, которые могут принимать логические решения о правильности работы приложения.

Никакое тестирование не способно выявить все возможные ошибки. Но выбор надежной стратегии позволит минимизировать риск и поможет быстро устранять проблемы при их обнаружении.

ПРИЛОЖЕНИЕ А. Дескриптор развертывания, справочник

В этом приложении мы перечислим теги дескриптора развертывания EJB 3. Цель этого приложения – служить кратким справочником, к которому можно обращаться при создании дескриптора для своего корпоративного приложения. Каждый дескриптор определяется схемой XML, и мы опишем элементы схемы.

Как мы говорили на протяжении всей книги, для настройки корпоративных приложений можно использовать аннотации, дескрипторы XML или и то и другое. В наших примерах мы использовали в основном аннотации, поэтому здесь мы тоже будем указывать, какие аннотации соответствуют тегами дескриптора.

Данное приложение, в частности, является справочником по содержимому файла *ejb-jar.xml* – дескриптору развертывания для сеансовых компонентов и компонентов, управляемых сообщениями. Несмотря на то, что в этой книге имеется краткое введение в JPA, данное приложение не содержит никаких дополнительных подробностей о содержимом файла *persistence.xml*. Мы рекомендуем вам заняться исследованием этой темы самостоятельно.

Схему для EJB 3.2 можно найти по адресу: http://xmlns.jcp.org/xml/ns/javaee/ejb-jar_3_2.xsd. Это файл схемы, который мы будем обсуждать в данном приложении. Но файл схемы *ejb-jar_3_2.xsd* ссылается на другие схемы, как часть этого определения. Все файлы схем для EE 7 доступны для загрузки на сайте компании Oracle: http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/javaee/index.html#7.

A.1. ejb-jar.xml

Файл *ejb-jar.xml* – это необязательный дескриптор развертывания, включаемый в модуль EJB. Двумя основными элементами в *ejb-jar.xml* являются `enterprise-beans` (используемый для определения компонентов, ресурсов и служб) и `assembly-descriptor` (используемый для объявления ролей, прав доступа к методам, настроек декларативных транзакций и интерцепторов). В этом разделе мы коснемся только элементов, которые могут присутствовать в *ejb-jar.xml*, и не будем обсуждать элементы, единственной целью которых является обеспечение обрат-

ной совместимости с EJB 2. В *ejb-jar.xml* можно ссылаться на схему http://xmlns.jcp.org/xml/ns/javaee/ejb-jar_3_2.xsd.

Приступая к созданию нового файла *ejb-jar.xml*, начинайте с объявления корневого элемента `ejb-jar`, как показано ниже:

```
<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">
```

После объявления корневого элемента, можно приступать к заполнению документа, определяя настройки, которые будут иметь более высокий приоритет, чем настройки в аннотациях. Не забывайте, что файл *ejb-jar.xml* является необязательным. Аннотации считаются более предпочтительным способом настройки во время разработки. Любые настройки в файле *ejb-jar.xml*, соответствующие аннотациям, будут иметь более высокий приоритет. Однако вы можете не использовать аннотации и помещать все настройки в *ejb-jar.xml*, но так поступать не рекомендуется. А теперь перечислим теги, которые могут вкладываться в корневой тег `<ejb-jar...>` документа.

A.1.1. <module-name>

Тег `<module-name>` используется для определения имени модуля EJB после развертывания на сервере EE:

```
<ejb-jar...>
    <module-name>action-bazaar-ejb</module-name>
</ejb-jar>
```

Может применяться только для автономных модулей EJB-JAR или модулей EJB-JAR, упакованных в архив EAR. Игнорируется при использовании в файлах *.war* – в этом случае действуют стандартные правила именования модулей, принятые для файлов *.war*.

Несмотря на необязательность, тег `<module-name>` может оказаться весьма полезным, особенно при использовании для конструирования артефактов таких систем сборки, как Maven. Maven добавляет номера версий к именам артефактов (например: *action-bazaar-ejb-1.1.0.17.jar*) и, хотя это дает определенные удобства, так как позволяет сразу увидеть версию артефакта в приложении, с точки зрения развертывания наличие номеров версий осложняет поиск имен в JNDI (когда это необходимо), потому что все операции поиска придется изменить, чтобы включить номер версии. С помощью тега `<module-name>` можно стандартизовать имя модуля для развертывания, независимо от наличия номера версии в его исходном имени.

A.1.2. <enterprise-beans>

Тег `<enterprise-beans>` используется для определения компонентов EJB в модуле EJB-JAR. Для определения сеансовых компонентов или компонентов, уп-

равляемых сообщениями, можно использовать вложенные теги <session> или <message-driven>.

<session>

Тег <session> используется для определения сеансового компонента:

```
<ejb-jar...>
    <enterprise-beans>
        <session>...</session>
    </enterprise-beans>
</ejb-jar>
```

Ему соответствует аннотация @Session. В табл. А.1 перечислены теги для настройки <session>.

Таблица А.1. Теги для настройки <session>

Имя тега	Описание
ejb-name	Логическое имя сеансового компонента. Свойство "name" аннотаций @Stateless и @Stateful.
mapped-name	Используется некоторыми производителями как имя компонента. Свойство "mappedName" аннотаций @Stateless и @Stateful.
remote	Удаленный интерфейс для компонента EJB: @Remote.
local	Локальный интерфейс для компонента EJB: @Local.
service-endpoint	Интерфейс конечной точки веб-службы для компонента EJB. Применяется только к сеансовым компонентам без сохранения состояния: @WebService..
ejb-class	Полностью квалифицированное имя класса компонента.
session-type	Тип сеансового компонента: @Stateless, @Stateful, @Singleton.
stateful-timeout	Интервал времени, в течение которого сеансовый компонент с сохранением состояния может простаивать, прежде чем контейнер получит право удалить его: @StatefulTimeout.
timeout-method	Метод обратного вызова для таймеров, создаваемых программно: @Timeout.
timer	Таймер EJB. Создается контейнером во время развертывания. Методы обратного вызова определяются элементом timeout-method.
init-on-startup	Компонент-одиночка, который должен создаваться на этапе развертывания: @Startup.
concurrency-management-type	Определяет способ управления параллельным выполнением для компонентов-одиночек и сеансовых компонентов с сохранением состояния. Может принимать значение Bean или Container: @ConcurrencyManagement.

Таблица А.1. (окончание)

Имя тега	Описание
`concurrent-method`	Настраивает параллельное выполнение метода под управлением контейнера.
`depends-on`	Список из одного или более компонентов-одиночек, которые должны быть инициализированы перед данным компонентом-одиночкой. Применяется только к компонентам-одиночкам. Для определения зависимостей используется синтаксис `ejb-link`: `@DependsOn`.
`init-method`	Метод создания компонента EJB 3 в стиле EJB 2: `@Init`.
`remove-method`	Метод уничтожения компонента EJB 3 в стиле EJB 2: `@Remove`.
`async-method`	Определяет метод для асинхронного выполнения: `@Asynchronous`.
`transaction-type`	Определяет тип управления транзакциями для компонента EJB. Может принимать значение `Bean` или `Container`: `@TransactionManagement`.
`after-begin-method`	Метод обратного вызова для сеансового компонента с сохранением состояния, извещающий его о начале новой транзакции: `@AfterBegin`.
`before-completion-method`	Метод обратного вызова для сеансового компонента с сохранением состояния, извещающий его о начале подтверждения транзакции: `@BeforeCompletion`.
`after-completion-method`	Метод обратного вызова для сеансового компонента с сохранением состояния, извещающий его о завершении транзакции. Этот метод вызывается не только после подтверждения, но и после отката транзакции: `@AfterCompletion`.
`around-invoke`	Имя метода в классе, который должен быть вызван для перехвата вызова прикладного метода компонента EJB: `@AroundInvoke`.
`around-timeout`	Имя метода в классе, который должен быть вызван для перехвата вызовов методов EJB, осуществляемых таймерами: `@AroundTimeout`.
`post-activate`	Метод-обработчик, вызываемый после активизации: `@PostActivate`.
`pre-passivate`	Метод-обработчик, вызываемый перед пассивацией: `@PrePassivate`.
`security-role-ref`	Ссылки внутренних ролей на внешние роли.
`security-identity`	Используется, чтобы принудительно установить или переопределить фактический уровень привилегий вызывающего кода.
`passivation-capable`	Определяет возможность пассивации сеансового компонента с сохранением состояния.

<message-driven>

Тег <message-driven> используется для определения компонента, управляемого сообщениями (Message-Driven Bean, MDB):

```
<ejb-jar...>
    <enterprise-beans>
        <message-driven>...</message-driven>
    </enterprise-beans>
</ejb-jar>
```

Соответствует аннотации @MessageDriven. В табл. А.2 перечислены теги для настройки <message-driven>.

Таблица А.2. Теги для настройки <message-driven>

Имя тега	Описание
ejb-name	Логическое имя компонента MDB. Свойство "name" аннотации @MessageDriven.
mapped-name	Используется некоторыми производителями как имя компонента. Свойство "mappedName" аннотации @MessageDriven.
ejb-class	Полностью квалифицированное имя класса компонента.
messaging-type	Поддерживаемый тип механизма сообщений, то есть, интерфейс приемника сообщений MDB. Свойство "messageListenerInterface" аннотации @MessageDriven.
timeout-method	Метод обратного вызова для таймеров, создаваемых программно: @Timeout.
timer	Таймер EJB. Создается контейнером во время развертывания. Методы обратного вызова определяются элементом timeout-method.
transaction-type	Определяет тип управления транзакциями для компонента EJB. Может принимать значение Bean или Container: @TransactionManagement.
message-destination-type	Ожидаемый тип приемника, то есть, javax.jms.Queue.
message-destination-link	Используется некоторыми производителями для отображения логического приемника в фактический.
activation-config	Пары имя/значение, определяющие настройки MDB, которые используются контейнером EJB при создании компонента. Свойство "activationConfig" аннотации @MessageDriven.
around-invoke	Имя метода в классе, который должен быть вызван для перехвата вызова прикладного метода компонента EJB: @AroundInvoke.
around-timeout	Имя метода в классе, который должен быть вызван для перехвата вызовов методов EJB, осуществляемых таймерами: @AroundTimeout.
security-role-ref	Ссылки внутренних ролей на внешние роли.
security-identity	Используется, чтобы принудительно установить или переопределить фактический уровень привилегий вызывающего кода.

A.1.3. Интерцепторы

Для определения интерцепторов в модуле EJB используется тег `<interceptors>`:

```
<ejb-jar...>
    <interceptors>
        <interceptor>...</interceptor>
    </interceptors>
</ejb-jar>
```

Обычно интерцепторы EJB определяются в виде отдельных классов Java, методы которых отмечаются аннотациями `@AroundInvoke` и `@AroundTimeout`. Тег `<interceptor>` соответствует этим аннотациям. Кроме того, тег `<interceptor>` необходим для определения глобальных интерцепторов, поскольку для них отсутствует соответствующая аннотация. В табл. А.3 перечислены теги для настройки `<interceptor>`.

Таблица А.3. Теги для настройки <interceptor>

Имя тега	Описание
`description`	Описание интерцептора.
`interceptor-class`	Полностью квалифицированное имя класса интерцептора.
`around-invoke`	Имя метода в классе, который должен быть вызван для перехвата вызова прикладного метода компонента EJB: `@AroundInvoke`.
`around-timeout`	Имя метода в классе, который должен быть вызван для перехвата вызовов методов EJB, осуществляемых таймерами: `@AroundTimeout`.
`around-construct`	Имя метода в классе, который должен быть вызван в процессе конструирования объекта: `@AroundConstruct`.
`post-activate`	Метод-обработчик, вызываемый после активизации: `@PostActivate`.
`pre-passivate`	Метод-обработчик, вызываемый перед пассивацией: `@PrePassivate`.

A.1.4. <assembly-descriptor>

Тег `<assembly-descriptor>` применяется для определения декларативных транзакций, ролей и привилегий для методов, а также для привязки интерцепторов:

```
<ejb-jar...>
    <assembly-descriptor>...</assembly-descriptor>
</ejb-jar>
```

<security-role>

Тег `<security-role>` применяется для определения ролей, используемых в приложении:

```
<ejb-jar...>
    <assembly-descriptor>
        <security-role>...</security-role>
```

```
    </assembly-descriptor>
</ejb-jar>
```

Соответствует аннотации `@DeclareRoles`. В табл. А.4 перечислены теги для настройки `<security-role>`.

Таблица А.4. Теги для настройки <security-role>

Имя тега	Описание
`role-name`	Имя роли. Например: `@DeclareRoles({"Admin", "User"})`.

<method-permission>

Тег `<method-permission>` применяется для перечисления ролей, с привилегиями которых может вызываться метод компонента EJB:

```
<ejb-jar...>
    <assembly-descriptor>
        <method-permission>...</method-permission>
    </assembly-descriptor>
</ejb-jar>
```

Соответствует аннотации `@RolesAllowed`. В табл. А.5 перечислены теги для настройки `<method-permission>`.

Таблица А.5. Теги для настройки <method-permission>

Имя тега	Описание
`role-name`	Имя роли с привилегиями которой может вызываться метод. Допускается использовать либо только этот тег, либо тег `unchecked`, но не оба сразу.
`unchecked`	Указывает, что метод можно вызывать с привилегиями любых ролей. Допускается использовать либо только этот тег, либо тег `role-name`, но не оба сразу.
`method`	Определяет имя компонента EJB (как указано в `<session><ebj-name>`) и имя метода.

<container-transaction>

Тег `<container-transaction>` позволяет определять настройки транзакций для методов компонентов EJB:

```
<ejb-jar...>
    <assembly-descriptor>
        <container-transaction>...</container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

Соответствует аннотации `@TransactionAttribute`. В табл. А.6 перечислены теги для настройки `<container-transaction>`.

Таблица А.6. Теги для настройки <container-transaction>

Имя тега	Описание
`trans-attribute`	Определяет атрибуты транзакции для метода. Допустимыми значениями являются: `Required`, `RequiresNew`, `NotSupported`, `Supports`, `Never`, `Mandatory`.
`method`	Определяет имя компонента EJB (как указано в `<session><ebj-name>`) и имя метода.

<interceptor-binding>

Тег `<interceptor-binding>` используется для связывания интерцепторов с компонентами EJB, как на уровне класса, так и на уровне отдельных методов:

```
<ejb-jar...>
    <assembly-descriptor>
        <interceptor-binding>...</interceptor-binding>
    </assembly-descriptor>
</ejb-jar>
```

Соответствует аннотации `@Interceptors`. В табл. А.7 перечислены теги для настройки `<interceptor-binding>`.

Таблица А.7. Теги для настройки <interceptor-binding>

Имя тега	Описание
`ejb-name`	Определяет имя компонента EJB, как указано в `<session><ebj-name>` или в атрибуте `"name"` аннотаций `@Stateless` и `@Stateful`. Этот тег применит интерцептор ко всем методам класса. Значение `"*"` создаст интерцептор по умолчанию и свяжет его со всеми компонентами EJB и MDB в файле *ejb-jar.xml* или *.war*.

`<interceptor-order>`
`<interceptor-class>..</>`
`</interceptor-order>`
	Список классов интерцепторов, которые будут связаны с компонентом, определяемым тегом `ejb-name`. Тег `interceptor-order` является необязательным и при его наличии определяет порядок вызова интерцепторов. Аннотация, которая позволяла бы определять порядок вызова интерцепторов, отсутствует.

`exclude-default-interceptors`	Указывает, что интерцепторы по умолчанию не должны применяться к классу или методам компонента EJB: `@ExcludeDefaultInterceptors`.
`exclude-class-interceptors`	Определяет, какие классы интерцепторов не должны применяться к классу или методам компонента EJB: `@ExcludeClassInterceptors`.
`method`	Определяет метод.

<message-destination>

Тег <message-destination> определяет логическое имя приемника, которое позднее, на этапе развертывания будет отображено в имя фактического приемника:

```
<ejb-jar...>
    <assembly-descriptor>
        <message-destination>...</message-destination>
    </assembly-descriptor>
</ejb-jar>
```

В табл. А.8 перечислены теги для настройки <message-destination>.

Таблица А.8. Теги для настройки <message-destination>

Имя тега	Описание
message-destination-name	Определяет имя приемника сообщений. Имя должно быть уникально среди имен приемников сообщений в пределах файла *ejb-jar.xml* или *.war*.
mapped-name	Используется некоторыми производителями. Непереносимое (может не поддерживаться) имя, в которое должно отображаться логическое имя приемника.
lookup-name	Имя, используемое для поиска приемника сообщений в каталоге JNDI.

<exclude-list>

Тег <exclude-list> определяет список методов, доступ к которым будет запрещен всем пользователям:

```
<ejb-jar...>
    <assembly-descriptor>
        <exclude-list>...</exclude-list>
    </assembly-descriptor>
</ejb-jar>
```

Соответствует аннотации @DenyAll. В табл. А.9 перечислены теги для настройки <exclude-list>.

Таблица А.9. Теги для настройки <exclude-list>

Имя тега	Описание
method	Определяет имя компонента EJB (как указано в <session><ebj-name>) и имя метода.

<application-exception>

Тег <application-exception> определяет список прикладных исключений, которые могут возбуждаться прикладными компонентами EJB и MDB:

```
<ejb-jar...>
    <assembly-descriptor>
        <application-exception>...</application-exception>
    </assembly-descriptor>
</ejb-jar>
```

С помощью этого тега можно указать, какие прикладные исключения должны вызывать откат текущей транзакции. По умолчанию исключение `EJBException` будет вызывать откат, а прикладные исключения – нет. Соответствует аннотации `@ApplicationException`. В табл. А.10 перечислены теги для настройки `<application-exception>`.

Таблица А.10. Теги для настройки <application-exception>

Имя тега	Описание
`exception-class`	Полностью квалифицированное имя класса исключения.
`rollback`	Указывает, должен ли контейнер выполнять откат транзакции перед возбуждением исключения.

ПРИЛОЖЕНИЕ В. Введение в Java EE 7 SDK

Это приложение представляет собой обобщенное руководство по установке пакета Java EE 7 SDK. Мы дадим здесь основные инструкции по установке программного обеспечения и очень краткое введение в работу с консолью администратора GlassFish (GlassFish Administration Console). Мы не будем углубляться в подробности, потому что детальное описание любого сервера Java EE могло бы занять целую книгу. Вместо этого мы предлагаем вам самостоятельно посетить домашнюю страницу проекта GlassFish (https://glassfish.java.net/), где вы найдете дополнительную информацию о том, как использовать и настраивать GlassFish.

Примеры программного кода из программы ActionBazaar для этой книги предусматривают использование инструмента Maven для сборки, в ходе которой генерируются артефакты, пригодные для развертывания под управлением GlassFish. В этом приложении будет показано, как собрать и развернуть приложение «Hello World» из главы 1.

B.1. Установка Java EE 7 SDK

Компания Oracle предлагает комплект инструментов разработчика Java EE 7 SDK, включающий также установочный пакет JDK. Установка того пакета – самый простой и быстрый способ развернуть на своем рабочем месте все, необходимое для разработки приложений EJB 3. В состав пакета входят следующие компоненты:

- пакет JDK 7;
- сервер приложений GlassFish;
- примеры программного кода EE 7;
- документация EE 7 API Javadocs;
- руководство Oracle EE 7 Tutorial.

Пакет доступен для загрузки на веб-сайте Oracle, по адресу: http://www.oracle.com/technetwork/java/javaee/downloads/index.html. На выбор предлагаются версии пакета Java EE 7 SDK в комплекте с пакетом JDK и без него, а также версии пакета Java EE 7 Web Profile SDK, так же в комплекте с пакетом JDK и без него. Если вы сомневаетесь, какую версию выбрать, мы рекомендуем загрузить Java EE 7 SDK в комплекте с пакетом JDK. В результате вы получите полный устано-

вочный пакет с сервером EE и соответствующим пакетом JDK, необходимым для его работы. Загрузочная версия Web Profile SDK содержит облегченную версию сервера EE. И хотя им тоже можно успешно пользоваться, на первых порах лучше все-таки выбрать полную версию сервера EE, пока вы не освоитесь с технологией настолько, что сможете сделать обоснованный выбор. На момент написания этих строк самой свежей была версия Java EE 7 SDK в комплекте с пакетом JDK 7 Update 45 (загружаемый файл имел имя *java_ee_sdk-7-jdk7-windows.exe*). Для загрузки доступны так же версии для других основных платформ.

После загрузки установочного пакета выполните следующие действия:

1. Выполните двойной щелчок на выполняемом файле *java_ee_sdk-7-jdk7-windows.exe*. В результате должен появиться первый экран мастера установки (рис. В.1).
2. Щелкните на кнопке **Next** (Далее), чтобы запустить установку.
3. На следующем экране будет предложено выбрать тип установки (рис. В.2). Выберите параметр **Typical Installation** (Типичная установка) и щелкните на кнопке **Next** (Далее).
4. На следующем экране будет предложено выбрать каталог для установки GlassFish (рис. В.3). Примите предложенный по умолчанию каталог *C:\glassfish4*. Щелкните на кнопке **Next** (Далее).
5. На следующем экране будет предложено установить инструмент автоматического обновления (рис. В.4). Этот инструмент действует точно так же, как аналогичные инструменты обновления в любых других программных продуктах. Когда на сайте разработчика появятся обновления для сервера приложений GlassFish, вам будет предложено загрузить и установить их. Если вы находитесь за брандмауэром, можете указать имя хоста и номер порта своего прокси-сервера. Эти значения вы можете узнать у своего сетевого администратора. Щелкните на кнопке **Next** (Далее), чтобы перейти к последнему, обзорному экрану, предшествующему началу процесса установки.
6. На следующем экране будет предложено подтвердить сделанный выбор перед началом установки (рис. В.5). После ознакомления со списком компонентов, которые будут установлены, щелкните на кнопке **Install** (Установить).
7. Пока идет установка Java EE 7 SDK, на экране отображается индикатор процесса, чтобы пользователь мог видеть, что устанавливается и сколько времени осталось до окончания (рис. В.6).
8. После окончания установки на экране появится сообщение с отчетом об установке (рис. В.7). Мы рекомендуем скопировать эту информацию и сохранить на будущее. В ней вы найдете все порты и серверы, доступные на вашем сервере.

Вот и все. Установка Java EE 7 SDK завершена. Теперь вы можете запустить свой сервер Java EE. Далее мы займемся исследованием консоли администратора и вы узнаете, как запустить и остановить сервер.

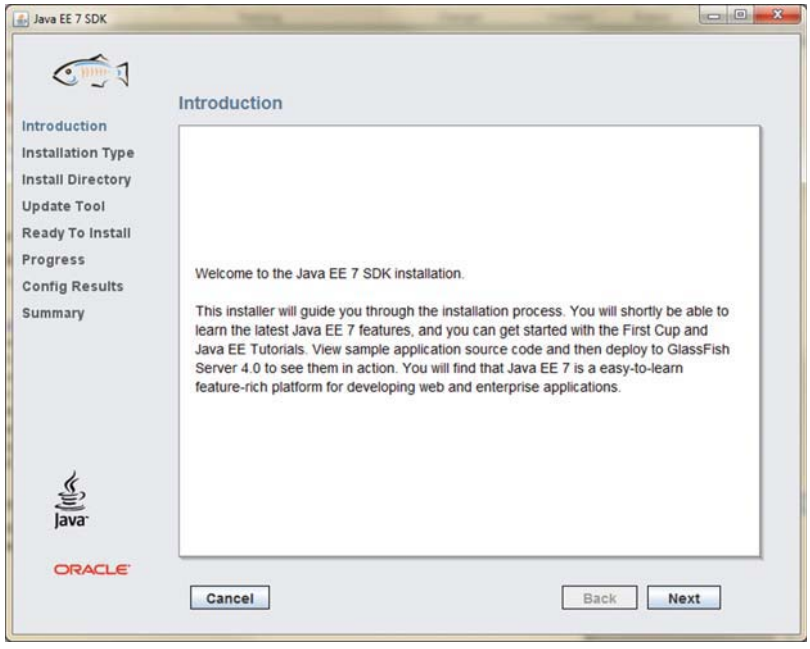

Рис. B.1. Первый экран мастера установки Java EE 7 SDK

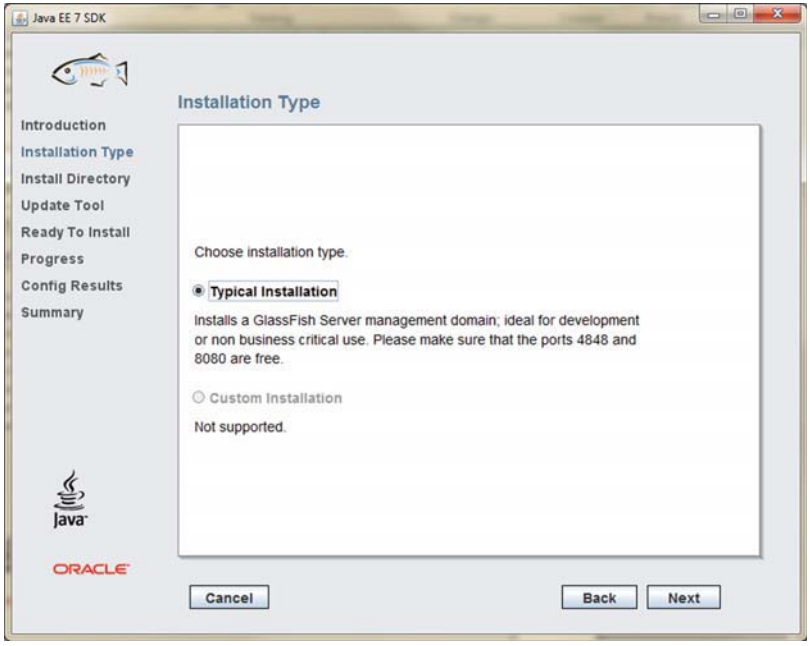

Рис. B.2. Выбор типа установки Java EE 7 SDK

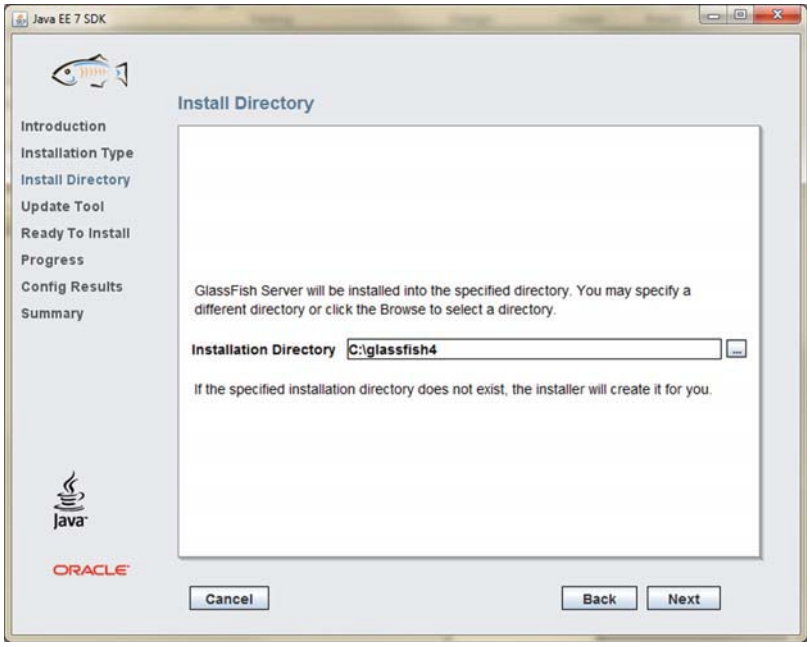

Рис. В.3. Выбор каталога установки Java EE 7 SDK

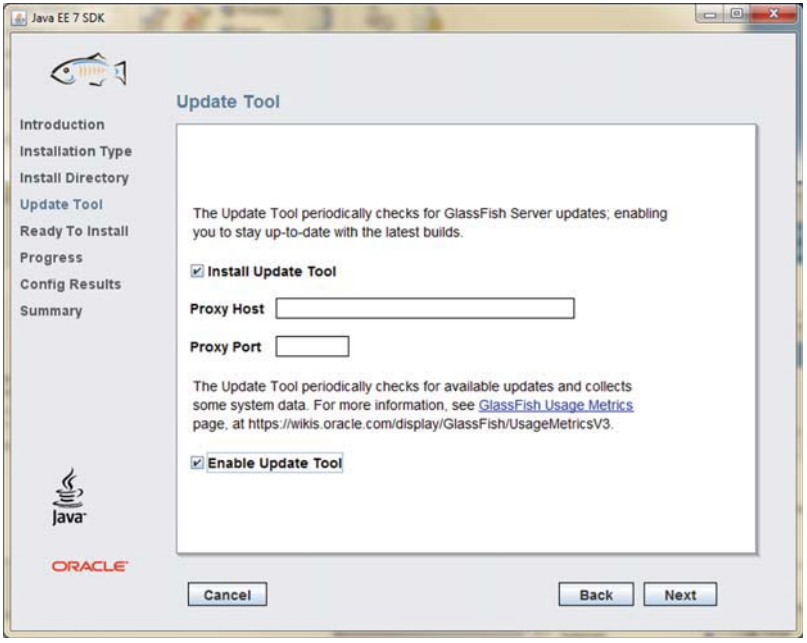

Рис. В.4. Установка и параметры инструмента обновления Java EE 7 SDK

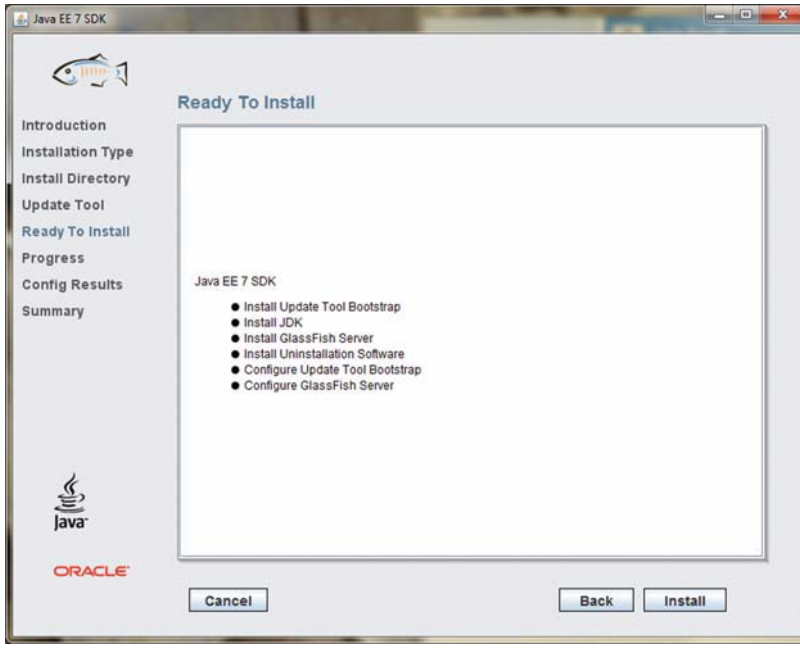

Рис. B.5. Подтверждение параметров установки Java EE 7 SDK

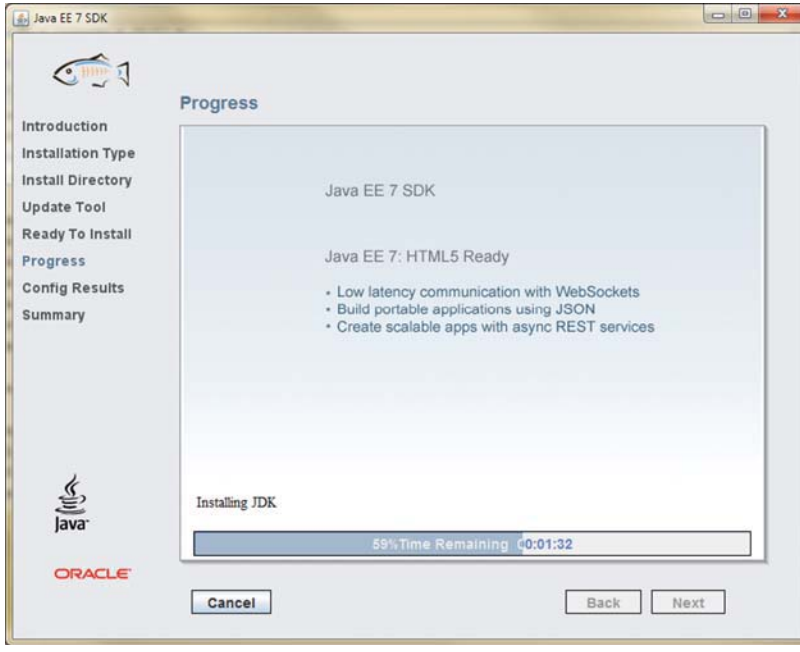

Рис. B.6. Процесс установки Java EE 7 SDK

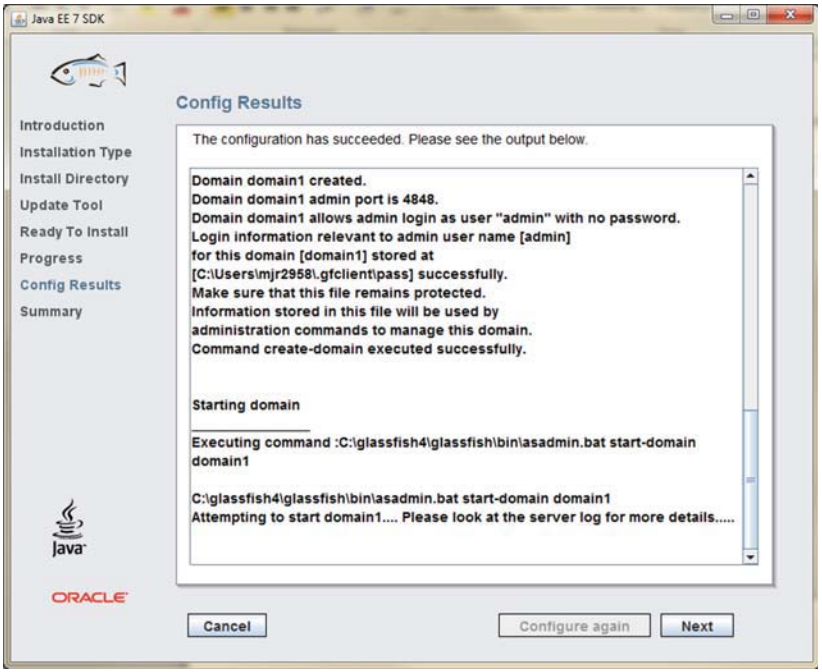

Рис. B.7. Отчет с результатами установки Java EE 7 SDK

B.2. GlassFish Administration Console

После установки Java EE 7 SDK с параметрами по умолчанию консоль администратора GlassFish Administration Console доступна на порту с номером 4848, с именем пользователя «admin» и без пароля. Выполните следующие шаги, чтобы открыть консоль администратора и изменить пароль:

1. Запустите Internet Explorer и откройте страницу http://localhost:4848. Так как сразу после установки пароль для пользователя «admin» не определен, вход в консоль будет выполнен автоматически и вы должны увидеть страницу, как показано на рис. B.8.
2. Выберите пункт **Configurations** (Настройки) → **server-config** (Настройки сервера) → **Security** (Безопасность) → **Realms** (Области) → **admin-realm** в панели слева (рис. B.9).
3. Теперь вы можете отредактировать настройки области `admin-realm` и выполнить какие-либо операции по администрированию пользователей в ней. Щелкните на кнопке **Manage Users** (Управление пользователями) (рис. B.10); затем щелкните на ссылке `admin`, чтобы отредактировать параметры учетной записи пользователя «admin» (рис. B.11).
4. Измените пароль пользователя «admin», набрав, например, строку `adminadmin` (рис. B.12). Щелкните на кнопке **Save** (Сохранить).

5. Теперь щелкните на кнопке **Logout** (Выйти). В окне браузера появится форма входав консоль (рис. B.13).

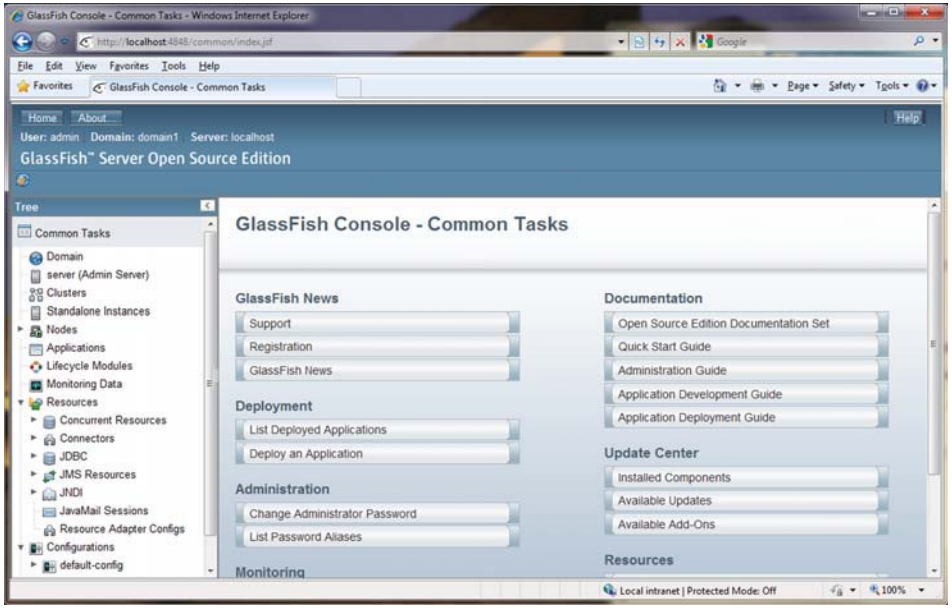

Рис. B.8. Приложение GlassFish Administration Console

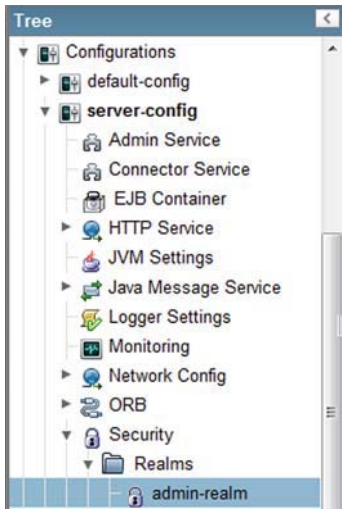

Рис. B.9. Выбор области admin-realm для перехода к настройкам административного доступа к консоли администратора

Рис. B.10. Щелкните на кнопке, чтобы перейти к настройкам пользователей в выбранной области

Рис. В.11. Щелкните на ссылке admin, чтобы перейти к настройкам учетной записи admin

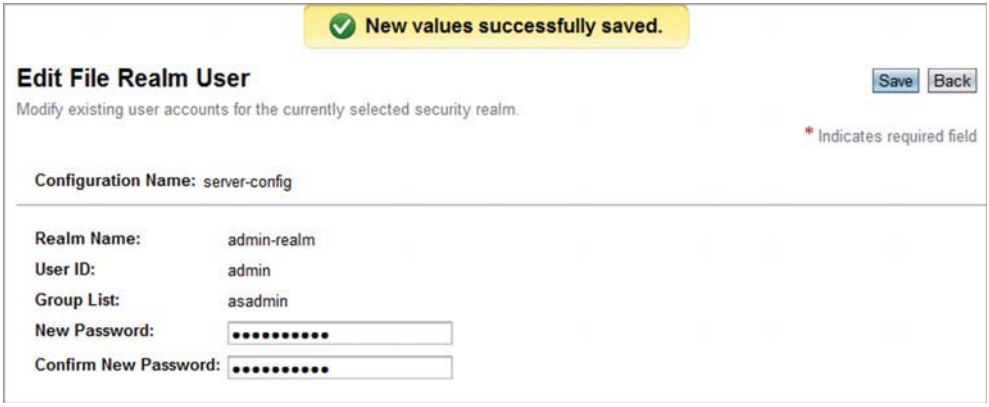

Рис. В.12. Изменение пароля пользователя admin

Рис. В.13. Страница входа в GlassFish Administration Console после изменения пароля

Вы получили лишь весьма приблизительное представление о консоли администратора. GlassFish – это полнофункциональный сервер Java EE 7 и описание его консоли администратора легко может занять целую книгу, поэтому мы не будем углубляться дальше.

Заключительная тема, которую мы рассмотрим – порядок запуска и остановки сервера приложений GlassFish.

B.3. Запуск и остановка GlassFish

Вообще говоря, существует два способа запустить и остановить сервер приложений GlassFish. Первый – посредством меню **Start** (Пуск). Другой – посредством консоли DOS. Давайте сначала посмотрим, как осуществить запуск через меню **Start** (Пуск).

В типичной установке Java EE 7 SDK, мастер установки добавит в меню **Start** (Пуск) пункты запуска и остановки GlassFish (рис. B.14).

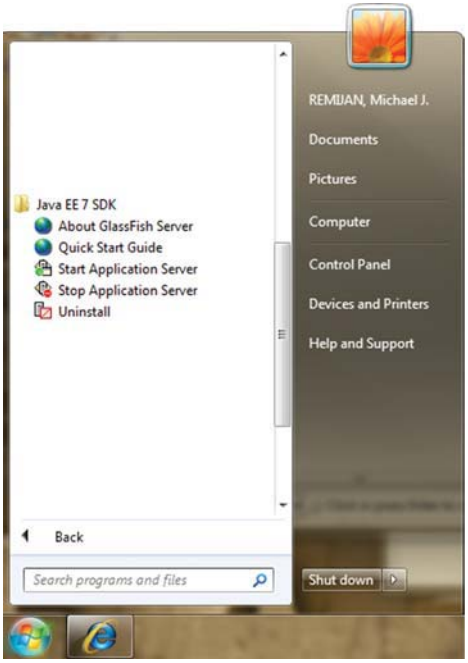

Рис. B.14. Пункты для запуска и остановки GlassFish в меню **Start** (Пуск)

Найдите пункт **Stop Application Server** (Остановить сервер приложений) и выберите его. После этого попробуйте вновь открыть консоль администратора – вы не сможете сделать этого и перед вами появится страница с сообщением об ошибке (рис. B.15).

Найдите пункт **Start Application Server** (Запустить сервер приложений) и выберите его. Через минуту или что-то около того попробуйте открыть консоль администратора. Перед вами должна появиться форма входа, как на рис. B.13.

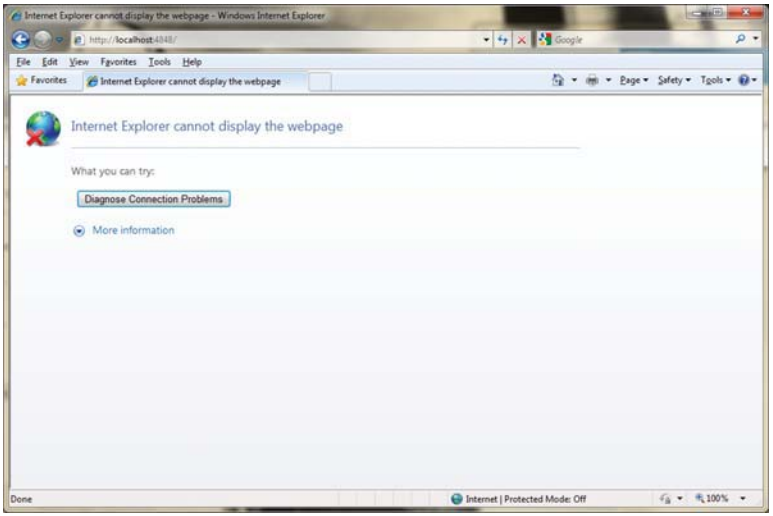

Рис. В.15. Консоль администратора недоступна, потому что сервер GlassFish был остановлен

Альтернативный способ запуска и остановки GlassFish состоит в том, чтобы воспользоваться консолью DOS. Для управления сервером GlassFish из консоли DOS используется команда `asadmin.bat`. Операция, выполняемая ею, определяется дополнительными параметрами. Чтобы остановить GlassFish, выполните следующую команду (рис. В.16):

```
> asadmin.bat stop-domain domain1
```

Аналогично, чтобы запустить GlassFish, выполните команду (рис. В.17):

```
> asadmin.bat start-domain domain1
```

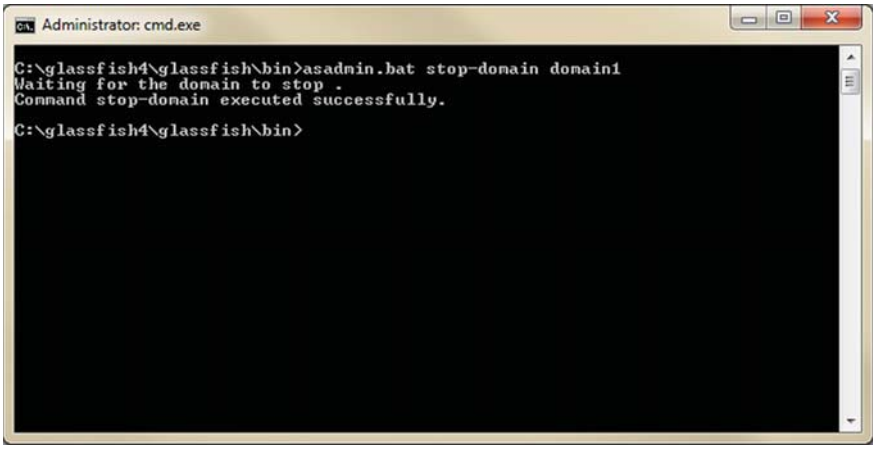

Рис. В.16. Использование команды asadmin.bat для остановки domain1

Рис. B.17. Использование команды asadmin.bat для запуска domain1

Теперь, когда вы знаете, как запускать и останавливать сервер приложений GlassFish, рассмотрим последнюю тему этого приложения – развертывание и запуск приложения.

B.4. Запуск приложения «Hello World»

В загружаемых примерах кода для главы 1 имеется простое приложение «Hello World», которым мы воспользуемся здесь, чтобы описать процедуру сборки и развертывания приложения на сервере GlassFish. Далее мы будем предполагать, что вы уже загрузили примеры для этой книги со страницы http://code.google.com/p/action-bazaar/ и открыли проект с примером для главы 1 в NetBeans (рис. B.18).

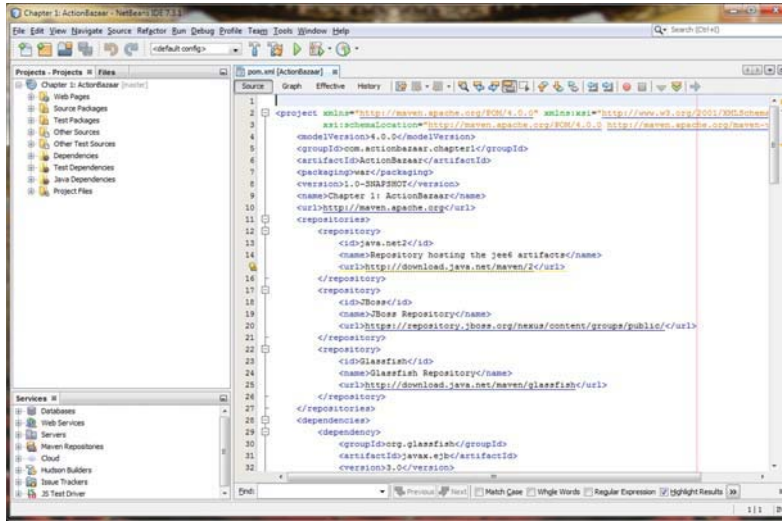

Рис. B.18. Проект с примером для главы 1, открытый в NetBeans

Чтобы собрать и развернуть приложение «Hello World», выполните следующие шаги:

1. Остановите сервер приложений GlassFish, следуя инструкциям в разделе B.3.
2. Щелкните правой кнопкой на проекте и выберите пункт **Clean and Build** (Очистить и собрать). Убедитесь, что сборка прошла без ошибок – в этом случае должно появиться сообщение `BUILD SUCCESS` в выводе инструмента Maven.
3. Запустите сервер приложений GlassFish, следуя инструкциям в разделе B.3.
4. Перейдите в раздел **Applications** (Приложения) в консоли администратора и щелкните на ссылке **Deploy** (Развернуть) (рис. B.19).
5. Щелкните на кнопке **Browse** (Обзор), найдите каталог *action-bazaar/chapter1/target*, выберите файл *ActionBazaar.war* и и щелкните на кнопке **Open** (Открыть) (рис. B.20).
6. Убедитесь, что в поле **Context Root** (Корневой контекст) и в поле **Application Name** (Имя приложения) появилось значение `ActionBazaar`. Щелкните на кнопке **OK**, чтобы развернуть приложение на серевере GlassFish (рис. B.21).
7. Если все прошло успешно, вы увидите что приложение ActionBazaar было благополучно развернуто (рис. B.22).
8. Теперь откройте новое окно браузера и введите адрес http://localhost:8080/ActionBazaar. Вы увидите приложение «Hello World» в действии (рис. B.23)!

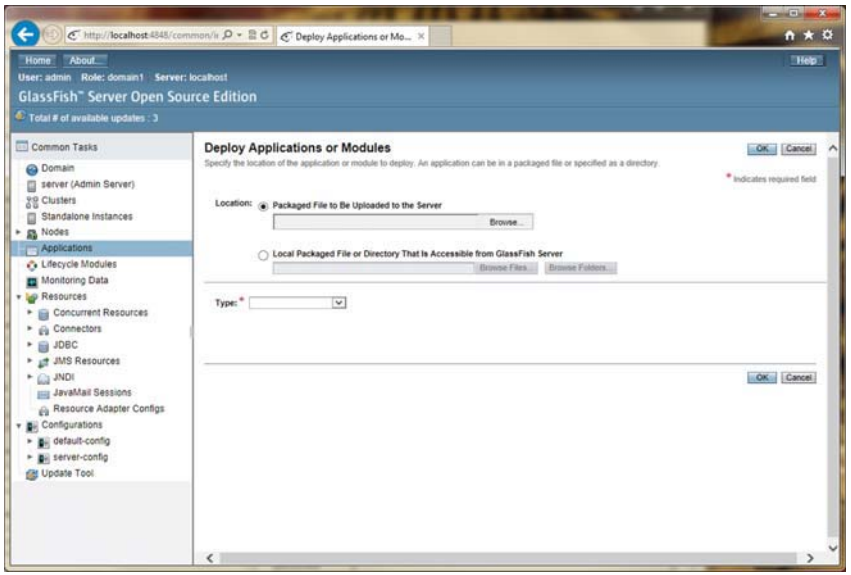

Рис. B.19. Развертывание приложения или модуля на сервере GlassFish

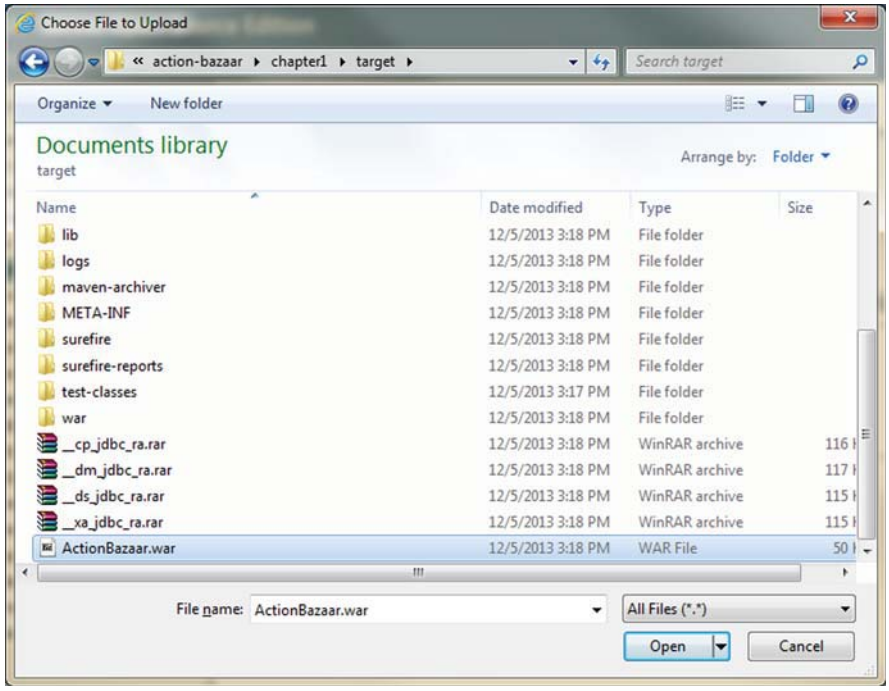

Рис. B.20. Выберите файл ActionBazaar.war для выгрузке

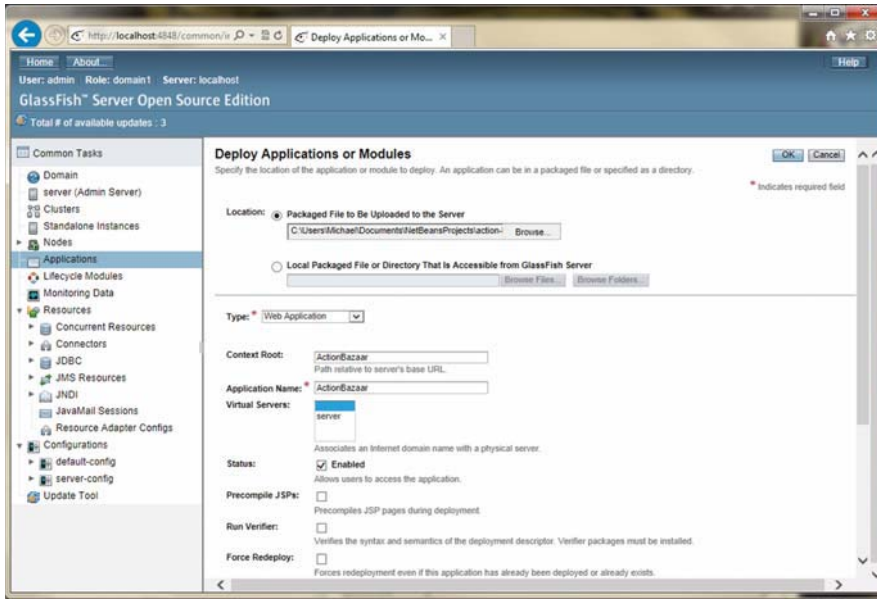

Рис. B.21. Проверьте параметры настройки перед развертыванием

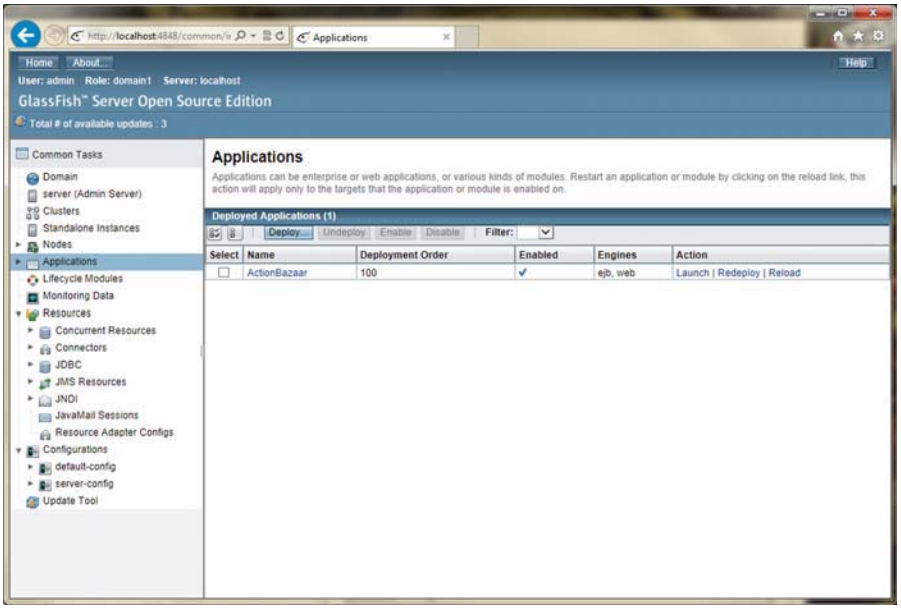

Рис. В.22. Приложение ActionBazaar успешно развернуто

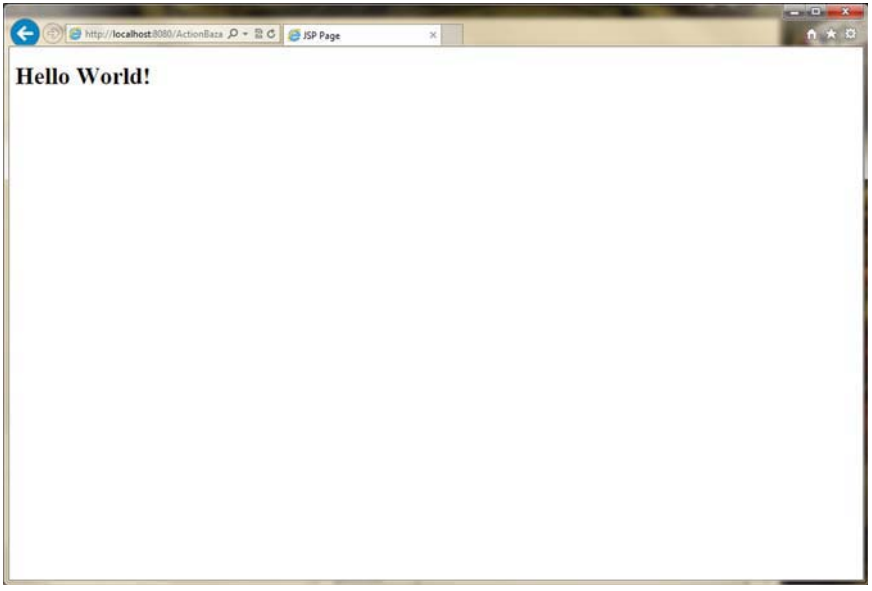

Рис. В.23. Приложение «Hello World» в действии

ПРИЛОЖЕНИЕ С. Сертификационные экзамены разработчика для EJB 3

В этом приложении приводится общая информация о процессе сертификации Oracle Java с уклоном в сторону EJB. На странице сертификации, на сайте Oracle приводится следующая информация:

> *Соискатель квалификации Sun Certified EJB Developer for the Java EE 6 Platform должен обладать знаниями, необходимыми для создания надежных серверных приложений с использованием технологии Enterprise JavaBeans (EJB) версии 3.1. Чтобы успешно сдать сертификационные испытания, соискатель должен иметь практический опыт использования технологии EJB для создания сеансовых компонентов и компонентов, управляемых сообщениями. Соискатель должен также знать архитектуру EJB, эффективные приемы программирования, приемы управления транзакциями, владеть основами обмена сообщениями и управления безопасностью.* (Выдержка взята 22.12.2013 по дресу: http://education.oracle.com/pls/web_prod-plq-dad/db_pages.getpage?page_id=458&get_params=p_track_id:JEE6JPE.)

Сертификация – отличный способ для профессиональных программистов на Java расширить свои навыки. Обучение в процессе подготовки к сертификации даст вам возможность изучить новые технологии, приемы и процедуры разработки, что едва ли возможно в повседневной практике. После изучения материалов вы сможете пройти сертификационные испытания и продемонстрировать свою компетенцию. Наконец, вы можете стать пропагандистом в своем рабочем коллективе и, как мы надеемся, помочь продвижению технологий Java EE в проекты и инфраструктуру в своей компании.

Эта книга, посвященная EJB 3, является отличным источником информации для тех, кто готовится к сертификационным испытаниям как разработчик EJB 3. Ниже приводится общее описание процесса сертификации Oracle Java с возможной целью участия в сертификационных испытаниях. Далее мы рассмотрим следующие вопросы:

- как начать процесс сертификации;
- порядок прохождения сертификационных испытаний для разработчиков EJB 3;
- какие знания необходимы для прохождения испытаний;
- как готовиться к сертификационным испытаниям;
- что из себя представляют сами испытания.

Итак, давайте посмотрим, что из себя представляет процесс сертификации в Oracle.

C.1. Начало процесса сертификации

Процесс сертификации в Oracle – это последовательность сертификационных испытаний, которые могут пройти разработчики на Java, чтобы продемонстрировать свои знания стандартов и технологии Enterprise Java. Процесс сертификации начинается с проверки знаний основ Java Standard Edition (Java SE) и затем постепенно усложняется и разветвляется по разным технологиям Java Enterprise Edition (Java EE).

Процедура сертификационных испытаний не особенно сложна. Как и в учебных заведениях, где, прежде чем перейти на следующий курс необходимо сдать экзамены за текущий, в сертификационном процессе, прежде чем пройти интересующие вас сертификационные испытания, необходимо подтвердить свои знания, получив сертификаты предыдущих уровней. Так, двигаясь снизу верх, вы будете накапливать свой портфель сертификатов Java. Чем большее число сертификатов вы будете иметь, тем убедительнее вы сможете продемонстрировать свой уровень владения технологиями Java работодателям и коллегам, серьезность отношения к Java, а так же вашу способность к обучению и стремление к самостоятельным исследованиям, что высоко ценится многими работодателями. Подробнее о порядке сертификации в области EJB мы поговорим в разделе C.2.

Сертификационные испытания обычно заключаются в заполнении тестов, состоящих из вопросов с вариантами ответов. Когда вы почувствуете, что готовы к испытаниям, заходите на сайт, наметьте время прохождения испытаний в сертификационном центре в вашем регионе, придите на экзамен и сдайте его. По окончании вам сообщат сумму баллов, чтобы вы могли сразу же узнать, прошли вы испытание или нет. Если испытания были пройдены успешно, вам в скором времени вышлют сертификат по почте.

Некоторые испытания включают демонстрацию навыков разработки. Для этого вам вышлют описание проекта для разработки и назначат срок выполнения. После этого вы должны будете дома разработать проект и, когда по вашему мнению он будет готов, представить его (разумеется, уложившись в назначенный срок). За этим обычно следует собеседование в региональном сертификационном центре, где вам будет задано несколько вопросов о решениях, принятых в проекте. В конце этого собеседования вам не сообщат набранную сумму баллов – придется подождать некоторое время, пока эксперты оценят ваш проект.

В любом случае, свое движение через процесс сертификации Oracle Java вы должны начать с посещения страницы http://java.sun.com, откуда вы попадете на главную страницу сайта компании Oracle, посвященного технологиям Java. Попав на страницу, перейдите по ссылке **Certification Paths** (Способы сертификации), как показано на рис. C.1.

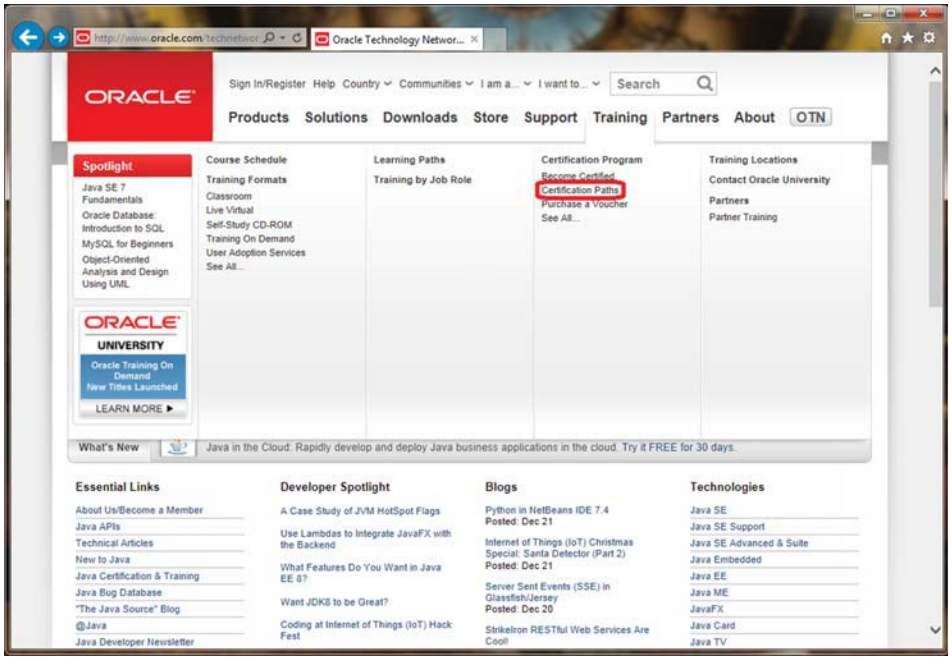

Рис. C.1. Ссылка Certification Paths (Способы сертификации) на главной странице сайта компании Oracle, посвященного технологиям Java

Оказавшись на странице Certification Paths (Способы сертификации), выберите в раскрывающихся списках интересующие вас испытания. На рис. C.2 представлен выбор для обычного разработчика Java EE:

1. Java EE Developer.
2. Java and Middleware.
3. Java.
4. Java EE.

Списки сертификационных испытаний могут изменяться с течением времени. На рис. C.2 изображены списки испытаний для Java EE 5 и Java EE 6. Но самым последним выпуском является Java EE 7 (на момент написания этих строк) и уже начались работы над Java EE 8. С появлением новых выпусков Java EE, предложения прохождения испытаний для прежних выпусков будут удаляться и замещаться новыми. Наибольший интерес для разработчиков с использованием EJB представляют сертификационные испытания по программе «Oracle Certified Expert, Java EE 6 Enterprise JavaBeans Developer».

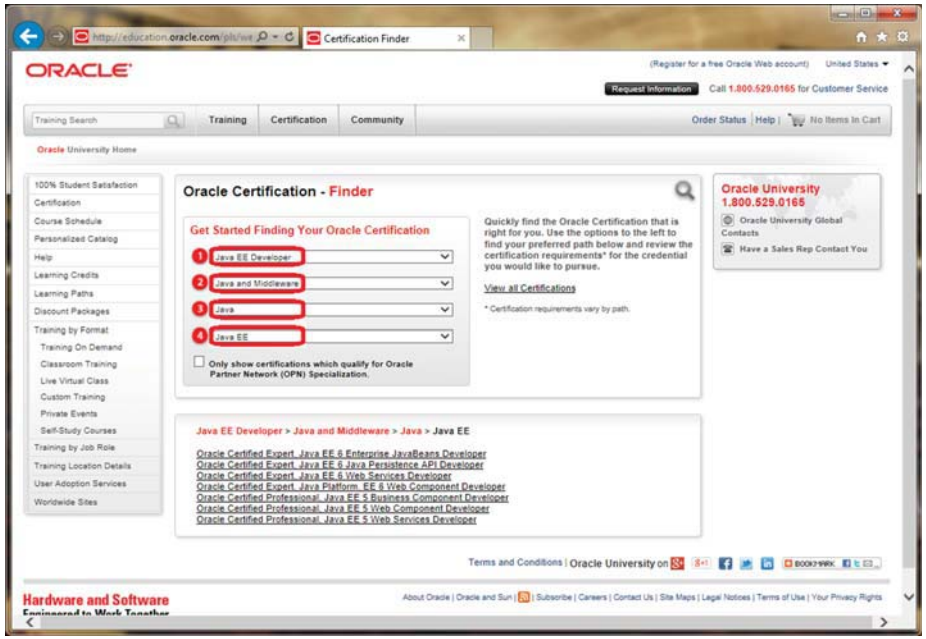

Рис. С.2. Список сертификационных испытаний для Java EE

В следующем разделе мы рассмотрим порядок прохождения сертификационных испытаний для разработчиков EJB 3.

С.2. Порядок прохождения сертификационных испытаний для разработчиков EJB 3

Программа сертификационных испытаний «Oracle Certified Expert, Java EE 6 Enterprise JavaBeans Developer» считается одной из наиболее продвинутых в технологиях Java EE. Как результат, прежде чем получить возможность пройти сертификационные испытания по этой программе, вам потребуется выполнить предварительные требования. В разделе С.1 рассказывалось, как начать процесс сертификации и определить на веб-сайте Oracle список сертификационных испытаний для Java EE. Если щелкнуть на ссылке **Oracle Certified Expert, Java EE 6 Enterprise JavaBeans Developer**, как показано на рис. С.2, вашему вниманию будет представлено описание сертификационных испытаний и перечислены предварительные требования, как показано на рис. С.3.

Программа сертификации «Oracle Certified Expert, Java EE 6 Enterprise JavaBeans Developer» имеет единственное предварительное требование: вы должны иметь сертификат «Oracle Certified Professional, Java SE (5, 6, или 7) Programmer» или более старый сертификат «Sun Certified Java Programmer» для любой версии Java.

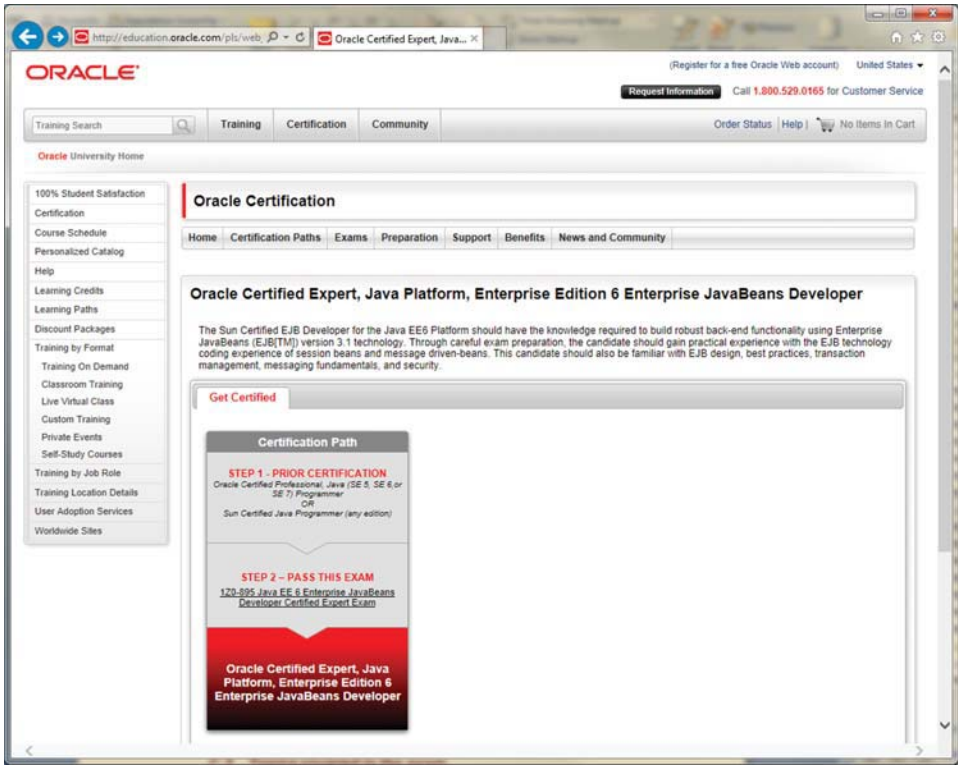

Рис. C.3. Описание процедуры сертификационных испытаний EJB

В Oracle следуют определенным правилам при выборе названий программ сертификационных испытаний. Название «Oracle Certified Professional» зарезервировано за обширными технологиями, такими как Java SE. Java SE включает широкий спектр более узкоспециализированных технологий, в том числе: JDBC, ввод/вывод, многопоточное выполнение, сети, структуры данных и так далее, но название «Oracle Certified Professional» относится к Java SE в целом, а не к какой-то конкретной технологии.

Название «Oracle Certified Expert», напротив, зарезервировано за конкретными технологиями. EJB – это конкретная технология, входящая в состав Java EE, поэтому программа сертификации по технологии EJB содержит в своем названии «Oracle Certified Expert». Так как эта книга посвящена в основном технологии EJB, именно на этой программе мы и сосредоточимся, однако вы должны знать, что для других технологий Java EE также существуют свои программы сертификации, названия которых начинаются со слов «Oracle Certified Expert». Найти их вы сможете на веб-сайте Oracle, в разделе, посвященном сертификации.

Прежде чем перейти к списку тем, знать которые необходимо для сертификационных испытаний по программе «Oracle Certified Expert, Java EE 6 Enterprise JavaBeans Developer», необходимо сделать еще одно замечание, касающееся по-

рядка прохождения сертификации. Чтобы получить свидетельство, вы должны пройти все пункты, указанные в процедуре сертификации, но порядок их прохождения не имеет значения. Вы можете зарегистрироваться и сдать экзамен по программе «Oracle Certified Expert, Java EE 6 Enterprise JavaBeans Developer» до испытаний по программе «Oracle Certified Professional, Java SE 7 Programmer». Но вы не получите сертификат, как разработчик EJB, пока не будут сданы оба экзамена. А теперь посмотрим, что необходимо знать, чтобы пройти сертификационные испытания как разработчик EJB.

C.3. Знания, необходимые для прохождения испытаний

Для прохождения сертификационных испытаний по программе «Oracle Certified Expert, Java EE 6 Enterprise JavaBeans Developer» необходимо иметь весьма обширные познания. Это объясняется большим богатством технологии EJB. Ниже приводится перечень тем, знание которых необходимо для успешного прохождения сертификационных испытаний. Если вы еще не догадались, скажем сразу, что эта книга является отличным источником информации, которая пригодится при подготовке к экзаменам.

Java EE:

- Что такое Java Platform Enterprise Edition (Java EE)?
- Каковы отличительные черты разработки приложений для Java EE?
- Из каких слоев/уровней состоит приложение для Java EE?
- Какие службы предоставляют контейнеры Java EE?
- Чем отличаются типы компонентов Java EE (EJB, MDB, Servlet и другие)?
- Сравните полную и облегченную версии контейнеров EJB.

Сеансовые компоненты:

- Что такое «сеансовый компонент»?
- Какие три типа сеансовых компонентов вы знаете?
- Какие типы сеансовых компонентов, и для каких целей вы использовали бы?
- Как бы вы организовали создание сеансовых компонентов?

Поиск сеансовых компонентов:

- Что такое «каталог JNDI» и какова его роль в приложениях Java EE?
- Как настроить JNDI для поиска компонентов EJB?
- Что такое «внедрение зависимостей» и какова роль этой технологии в приложениях Java EE?
- Как выполнить внедрение компонента EJB?

Жизненный цикл компонентов:

- Как связаны между собой контейнер EJB и компонент EJB?

- Что подразумевается под жизненным циклом для каждого из трех типов сеансовых компонентов?
- Как реализовать методы для обработки событий жизненного цикла?
- Как организовать асинхронные взаимодействия между компонентами?

Сеансовые компоненты-одиночки:

- Что такое «сеансовый компонент-одиночка»?
- Как создать сеансовый компонент-одиночку?
- Какие события возникают в жизненном цикле сеансовых компонентов-одиночек?
- Как реализовать методы для обработки событий жизненного цикла в сеансовых компонентах-одиночках?
- Как реализовать параллельный доступ к сеансовому компоненту-одиночке?

JMS:

- Что такое JMS?
- Каковы роли участников JMS?

MDB:

- Что делает сеансовые компоненты не лучшими кандидатами на роль приемников сообщений?
- Какие события возникают в жизненном цикле компонентов, управляемых сообщениями?
- Как создать компонент, управляемый сообщениями, для работы в комплексе с JMS?

Службы таймеров:

- Что такое «службы таймеров»?
- Как реализовать метод для обработки событий от таймера?

Интерцепторы:

- Что такое интерцепторы?
- Как определить класс интерцептора?
- Как применять интерцепторы к методам компонентов EJB?
- Какие события возникают в жизненном цикле интерцепторов?
- Как реализовать методы для обработки событий жизненного цикла в интерцепторах?

Транзакции:

- Что такое «транзакции, управляемые контейнером» (Container-Managed Transactions, CMT)?
- Как присоединиться к транзакции CMT?
- Что такое «транзакции, управляемые компонентом» (Bean-Managed Transactions, BMT)?

- Как запустить и завершить транзакцию BMT?
- Как управлять транзакциями в компонентах EJB?
- Как управлять транзакциями в компонентах MDB?

Безопасность:

- Что такое «архитектура безопасности Java EE»?
- Что такое «декларативная авторизация»?
- Что такое «программная авторизация»?

Приемы эффективного использования EJB:

- Для каких задач лучше всего использовать компоненты EJB?
- Какие шаблоны проектирования вы могли бы порекомендовать для использования при создании веб-приложений и клиентских приложений?
- Назовите лучший способ обработки исключений?

Это вполне исчерпывающий список вопросов, тем не менее, старайтесь чаще заглядывать на сайт, чтобы не пропустить изменения или дополнительную информацию. Это особенно важно, если вы планируете пройти сертификационные испытания по программе, отличной от «Oracle Certified Expert, Java EE 6 Enterprise JavaBeans Developer». В таких ситуациях обычно предпочтительнее иметь самую свежую информацию. А теперь, после знакомства со списком тем, которые нужно знать, чтобы успешно пройти испытания, посмотрим, как можно готовиться к экзаменам!

C.4. Подготовка к испытаниям

Готовиться к сертификационным испытаниям можно по-разному:

- самостоятельно;
- самостоятельно, по руководствам и методичкам;
- на интернет-курсах;
- на очных курсах.

Самостоятельная подготовка – самый простой и самый недорогой путь. Поскольку вы читаете эти строки, вы уже ступили на этот путь! Прочитайте все темы в этой книге в наиболее комфортном для вас темпе, и вы неплохо подготовитесь к сертификационным испытаниям.

Самостоятельная подготовка по руководствам и методичкам предлагается университетом Oracle University. По сути это та же самая самостоятельная подготовка, но вы должны будете заплатить компании Oracle за учебные материалы. В этом случае вы так же можете готовиться в собственном темпе и зарегистрироваться для прохождения испытаний, только когда будете полностью готовы.

В университете Oracle University существуют также интернет-курсы. Как можно догадаться по названию, интернет-курсы предполагают обучение в домашней обстановке посредством Интернета. Но они структурированы так же, как очные курсы. Обычно они рассчитаны на три дня занятий по восемь часов в день. Это

хороший выбор для тех, кто не уверен, что сможет изучить необходимые темы самостоятельно, но не имеет возможности посещать очные курсы.

Очные курсы – это именно то, что вы могли бы себе представить. Вам придется проехать в некоторое место (возможно недалеко) и посещать класс в течение трех дней, по восемь часов в день. Этот самый лучший вариант обучения, особенно для тех, кому не хватает усидчивости, чтобы учиться самостоятельности. Кроме того, на очных курсах имеется уникальная возможность общения с преподавателем, задавать вопросы и получать ответы.

Теперь посмотрим, чего ожидать в день испытаний.

C.5. Сертификационные испытания

Прежде чем вы достигнете дня сертификационных испытаний, вам необходимо будет выполнить шаги, описанные в разделе C.1, чтобы найти сертификационный центр, где можно сдать экзамены по программе «Certified Expert, Java EE 6 Enterprise JavaBeans Developer», и зарегистрироваться. Существует множество вариантов регистрации. Вы можете зарегистрироваться в центре тестирования Pearson VUE или в центре тестирования Oracle. Экзамены могут проходить очно, то есть, под наблюдением экзаменатора, или через Интернет. Вам следует ознакомиться со всеми имеющимися вариантами и выбрать тот, что лучше подходит для вас.

Какой бы способ прохождения испытаний вы не выбрали, сами испытания во всех случаях будут одними и теми же. В табл. C.1 перечислены некоторые дополнительные сведения об экзамене. Вам будет дано 110 минут, чтобы ответить на 60 вопросов с вариантами ответов, и вы должны дать не менее 73% правильных ответов. Вот и все – все очень просто.

Таблица C.1. Дополнительные сведения об экзамене

Номер экзамена	1Z0-895
Программа сертификации	Oracle Certified Expert, Java EE 6 Enterprise JavaBeans Developer
Продукт	Java EE
Стоимость экзамена	245 долларов США
Продолжительность	110 минут
Число вопросов	60
Формат	Вопросы с вариантами ответов
Проходной балл	73%
Соответствует версии	EE 6

Хотя это и маловероятно, но, все же, есть шанс, что вы не пройдете испытания. Что делать, если это случится? Не отчаивайтесь, Oracle разрешает сделать

несколько попыток пройти испытания. Если вы сдавали экзамены очно, вам придется подождать 14 дней, прежде чем сделать повторную попытку, причем неважно, зарегистрируетесь ли вы повторно на очное прохождение испытаний или через Интернет. Если вы сдавали экзамены через Интернет, вы можете повторить попытку в любой момент. Вы не сможете повторить бета-версию экзамена. И, наконец, не пытайтесь пройти экзамены под чужими именами, зарегистрировавшись несколько раз и получив несколько идентификаторов для тестирования.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

[

_

A

Б

В

Г

Д

Е

Ж

З

И

К

Л

М

Н

О

П

P

C

Т

У

Ф

Х

Ч

Ш

Я

A

G

H

I

J

L

M

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Электронный адрес: **books@alians-kniga.ru**.

Оптовые закупки: тел. +7(499) 782-38-89.

Дебу Панда, Реза Рахман, Райан Купрак, Майкл Ремижан

EJB3 в действии

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16. Гарнитура «Петербург».
Печать офсетная. Усл. печ. л. 57,93.
Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф

EJB 3 в действии

Фреймворк EJB 3 предоставляет стандартный способ оформления прикладной логики в виде управляемых модулей, которые выполняются на стороне сервера, упрощая тем самым создание, сопровождение и расширение приложений Java EE. Версия EJB 3.2 включает большее число расширений и значений по умолчанию, и более тесно интегрируется с другими технологиями Java, такими как CDI, делая разработку еще проще.

Книга «EJB 3 в действии» — это руководство по разработке компонентов Java EE с использованием EJB 3.2, JPA и CDI. Она знакомит читателя с EJB на многочисленных примерах кода, сценариях из реальной жизни и иллюстрациях. Помимо основ в этой книге описываются некоторые особенности внутренней реализации, наиболее эффективные приемы использования, шаблоны проектирования, даются советы по оптимизации производительности и различные способы доступа, включая веб-службы, службы REST и веб-сокеты.

Читатели должны знать язык Java. Опыт работы с EJB или Java EE не требуется.

Что внутри:

- содержимое книги соответствует версии EJB 3.2;
- описывается возможность хранения POJO в базе данных с помощью JPA 2.1;
- внедрение зависимостей и управление компонентами с помощью CDI 1.1;
- разработка интерактивных приложений с применением WebSocket 1.0.

Дебу Панда, Реза Рахман, Райан Купрак, Майкл Ремижан — опытные архитекторы Java, разработчики, авторы и лидеры сообщества.

«Исчерпывающий учебник по EJB 3!»
Луис Пенья, HP

«Реза входит в состав экспертной группы EJB 3.2 Expert Group и имеет большой опыт преподавания. Здесь вы найдете все, что вам нужно.»
Джон Гриффин, Progrexion ASG

«Полно и ясно описывает, как использовать платформу JEE на полную мощь.»
Рик Вагнер, Red Hat, Inc.

«Содержит ценные сведения и для новичков, и для экспертов в EJB.»
Джит Марва, gen-E

«Если вы испытываете проблемы с EJB — эта книга поможет их решить.»
Юрген де Коммер, Imtech ICT

ISBN 978-5-97060-135-8

Интернет-магазин: www.dmkpress.com
Книга - почтой: orders@alians-kniga.ru
Оптовая продажа: "Альянс-книга"
Тел.: (499)782-3889. book@alians-kniga.ru

ДМК ПРЕСС издательство
www.дмк.рф