Spring 5 для профессионалов

Обстоятельное руководство по каркасу Spring и его инструментальным средствам

ПЯТОЕ ИЗДАНИЕ

Юлиана Козмина
Роб Харроп
Крис Шефер
Кларенс Хо

ДИАЛЕКТИКА
www.williamspublishing.com
Apress®

Spring 5

для профессионалов

Пятое издание

Spring 5

Fifth Edition

Iuliana Cosmina, Rob Harrop, Chris Schaefer,
Clarence Ho

Apress®

Spring 5

для профессионалов

Пятое издание

Юлиана Козмина, Роб Харроп, Крис Шефер,
Кларенс Хо

Москва • Санкт-Петербург
2019

ББК 32.973.26-018.2.75
К59
УДК 681.3.07

ООО "Диалектика"

Зав. редакцией *С.Н. Тригуб*
Перевод с английского и редакция *И.В. Берштейна*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Козмина, Юлиана, **Харроп**, Роб, **Шефер**, Крис, **Хо**, Кларенс.

К59 Spring 5 для профессионалов. : Пер. с англ. — СПб. : ООО "Диалектика", 2019. — 1120 с. : ил. — Парал. тит. англ.

ISBN 978-5-907114-07-4 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание
Юлиана Козмина, Роб Харроп, Крис Шефер, Кларенс Хо
Spring 5 для профессионалов

Подписано в печать 06.12.2018. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 90,3. Уч.-изд. л. 53,4.
Тираж 400 экз. Заказ № 16606.

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907114-07-4 (рус.)
ISBN 978-1-4842-2807-4 (англ.)

Оглавление

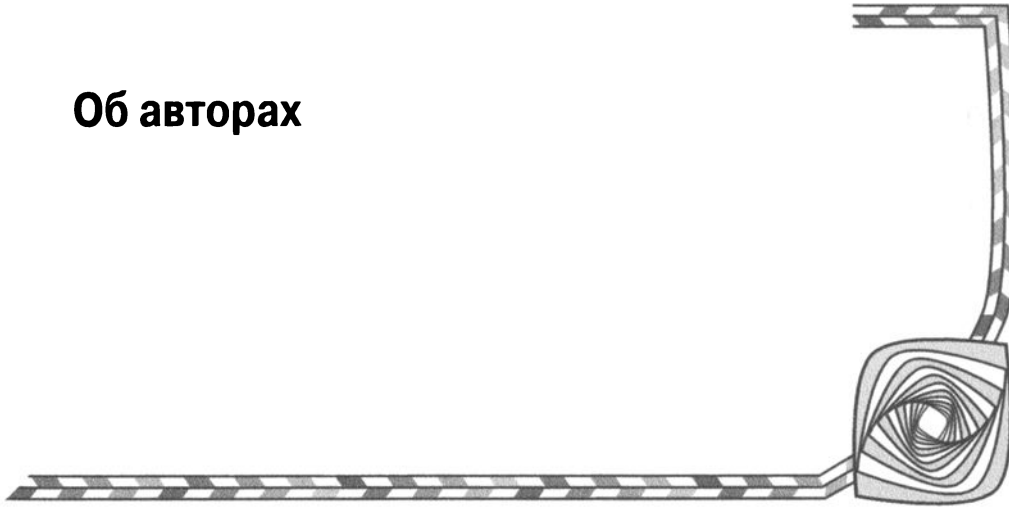
Содержание

Посвящение

Посвящаю эту книгу моим друзьям, крестнику Штефану и всем музыкантам, под музыку которых мне было легче работать над книгой.

Юлиана Козмина

Об авторах



Юлиана Козмина — разработчик веб-приложений и профессиональный пользователь Spring, аттестованный в компании Pivotal, разработавшей Spring Framework, Boot и другие инструментальные средства. Ее перу принадлежит ряд книг, вышедших в издательстве Apress и посвященных аттестации и веб-разработке в Spring Framework. Она работает архитектором программного обеспечения в компании Bearing Point Software и активно участвует в разработке программного обеспечения с открытым кодом в GitHub, обсуждении насущных вопросов программирования в Stack Overflow и прочих ресурсах.

Роб Харроп работает консультантом по программному обеспечению, специализируясь на выпуске высокопроизводительных корпоративных приложений с высокой степенью масштабируемости. У него имеется немалый опыт разработки архитектуры программного обеспечения и особая склонность уяснять и разрешать сложные вопросы проектирования. Обладая солидными знаниями платформ Java и .NET, он успешно осуществил на них немало проектов. У него имеется также немалый опыт и в других областях, включая розничную торговлю и государственную службу. Авторству Харропа принадлежит пять книг, в том числе настоящее, пятое издание этой книги, повсеместно признанной как исчерпывающий источник по Spring Framework.

Крис Шефер — главный разработчик программного обеспечения для проектов Spring в компании Pivotal, разработавшей Spring Framework, Boot и другие инструментальные средства.

Кларенс Хо работает ведущим архитектором приложений на Java в компании SkywideSoft Technology Limited, занимающейся консультациями по программному обеспечению и расположенной в Гонконге. Проработав в области информационных технологий более двадцати лет, Кларенс руководил многими проектами по разработке приложений для внутреннего потребления, а также оказывал консультационные услуги по корпоративным решениям.

О техническом рецензенте

Массимо Нардоне (Massimo Nardone) имеет более чем 23-летний опыт работы в области безопасности, разработки приложений для веб и мобильных устройств, облачных вычислений и ИТ-архитектуры. Особое пристрастие он испытывает к вопросам безопасности и платформе Android.

В настоящее время он возглавляет управление информационной безопасности (CISO) в компании Cargotec Oyj и является членом совета финского отделения международной организации ISACA, занимающейся разработкой методик и стандартов в области управления, аудита и безопасности информационных технологий. За свою долгую карьеру он занимал должности руководителя проекта, разработчика программного обеспечения, научно-технического работника, главного архитектора по безопасности, руководителя отдела информационной безопасности, аудитора безопасности систем PCI/SCADA, а также ведущего архитектора ИТ-безопасности, облачных вычислений и систем SCADA (АСУТП). Кроме того, он работал приглашенным лектором и методистом по практическим занятиям в сетевой лаборатории технического университета Хельсинки (университета Аальто).

Массимо окончил университет Салерно в Италии, получив степень магистра в области вычислительной техники, и владеет четырьмя патентами (в области инфраструктуры открытых ключей (PKI), сетевого протокола SIP, языка разметки SAML и прокси-серверов). Помимо этой книги, Массимо рецензировал более сорока книг по информационным технологиям, вышедших в разных издательствах. Он также является одним из авторов книги *Pro Android Games* (издательство Apress, 2015 г.).

Благодарности

Для меня большая честь быть основным автором пятого издания этой книги. Поверите ли, я получила такое назначение по ошибке, поскольку готовилась стать техническим рецензентом этой книги. И лишь когда я получила ее рукопись, то осознала, что мне придется стать одним из авторов пятого издания одной из самых лучших книг по Spring.

Мне довелось прочитать немало книг, вышедших в издательстве Apress, когда я училась и совершенствовала в дальнейшем свои профессиональные навыки. Это моя третья книга в издательстве Apress, и мне приятно внести свой посильный вклад образование следующего поколения разработчиков программного обеспечения.

Благодарю всех моих друзей, терпевших мои жалобы на недосыпание, слишком большой объем работы и нелегкую долю писателя. Я признательна им за то, что они оказали мне всяческую поддержку и убеждали меня, что писать книги все же интересно.

Мне бы хотелось также выразить большую признательность всем моим любимым певцам за их замечательную музыку, помогавшую мне в написании этой книги, и особенно Джону Мэйеру. Я твердо решила завершить эту книгу в срок только для того, чтобы отправиться в Соединенные Штаты на один из его концертов. Именно поэтому я изменила тематику примеров в этой книге, посвятив их певцам и их музыке как дань их искусству и таланту.

Юлиана Козмина

Введение

Охватывая версию Spring Framework 5, эта книга представляет собой наиболее исчерпывающее справочное и учебное руководство по Spring, которое позволит задействовать всю мощь этой лидирующей платформы, предназначенной для разработки корпоративных приложений на языке Java.

В настоящем издании рассматривается ядро платформы Spring и ее интеграция с другими ведущими технологиями Java, в том числе Hibernate, JPA 2, Tiles, Thymeleaf и WebSocket. Основное внимание в настоящем издании уделено применению классов конфигурирования Java, лямбда-выражениям, инструментальному средству Spring Boot и реактивному программированию. Мы, авторы этой книги, поделимся с читателями собственным опытом и реальными практическими приемами, применяемыми во время разработки корпоративных приложений, включая удаленное взаимодействие, обработку транзакций, уровни веб и представлений и многое другое.

Прочитав эту книгу, вы сможете следующее:

- применять инверсию управления (IoC) и внедрение зависимостей (DI);
- освоить новые функциональные возможности Spring Framework 5;
- строить веб-приложения на основе Spring, используя Spring MVC и WebSocket;
- создавать реактивные веб-приложения на основе Spring, используя Spring WebFlux;
- тестировать Spring-приложения, используя Junit 5;
- пользоваться лямбда-выражениями, внедренными в версии Java 8;
- моментально запускать на выполнение любые Spring-приложения, используя Spring Boot;
- пользоваться функциональными возможностями версии Java 9 в Spring-приложениях.

Вследствие того, что дата выпуска версии Java 9 была отсрочена, версия Spring 5 была выпущена на основе Java 8. Поэтому совместимость с версией Java 9 в этой книге основывается на первоначально доступной сборке.

С этой книгой связан многомодульный проект, сконфигурированный с помощью Gradle 4 и доступный в официальном хранилище издательства Apress по адресу `https://github.com/Apress/pro-spring-5`. Этот проект можно построить после копирования, следуя инструкциям, приведенным в файле `README.adoc`, а также установив локально систему автоматической сборки Gradle. В отсутствие Gradle можно загрузить и воспользоваться интегрированной средой разработки (IDE) InteliJ IDEA для построения данного проекта по сценарию Gradle Wrapper (см. `https://docs.gradle.org/current/userguide/gradle_wrapper.html`). В конце книги приведено небольшое приложение, в котором описывается структура и конфигурация данного проекта, а также дополнительные подробности, связанные с инструментальными средствами разработки, которыми можно воспользоваться для выполнения исходного кода примеров из этой книги, доступных в хранилище GitHub по указанному выше адресу.

По ходу написания этой книги появлялись новые предвыпускные версии Spring 5, новые версии IDE Intellij IDEA, Gradle и обновления других технологий, упоминаемых в этой книге. Мы обновили рукопись книги по новым версиям, чтобы предоставить самые последние сведения и согласовать ее с официальной документацией. Рукопись книги просмотрели несколько рецензентов, проверив ее на техническую точность подачи материала, но если вы обнаружите в ней какие-нибудь несоответствия, обращайтесь по адресу электронной почты `editorial@apress.com`, чтобы составить перечень ошибок и погрешностей, требующих исправления.

Исходный код примеров из этой книги можно получить, перейдя на посвященную ей веб-страницу издательства Apress по адресу `www.apress.com/9781484228074` и щелкнув на кнопке Download Source Code (Загрузить исходный код). Этот исходный код будет сопровождаться, согласовываться с новыми версиями применяемых технологий и совершенствоваться, исходя из рекомендаций разработчиков, пользующихся им для изучения Spring. Мы искренне надеемся, что вы получите такое же удовольствие от этой книги, изучая Spring, какое мы получили, писав ее.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к из данию новых книг.

Наши электронные адреса:

E-mail: `info@dialektika.com`

WWW: `http://www.dialektika.com`

ГЛАВА 1

Введение в Spring

Когда мы думаем о сообществе разработчиков приложений на языке Java, то вспоминаем полчища старателей, которые в конце 1840-х годов неистово прочесывали реки Северной Америки в поисках золотых самородков. Наши "реки" разработчиков приложений на Java изобилуют проектами с открытым кодом, но, в отличие от истории со старателями, отыскание действительно полезного проекта может оказаться долгим и трудным делом.

Многим проектам с открытым кодом на Java присуща общая особенность: они призваны просто заполнить пробел в реализации последней "модной" технологии или проектного шаблона. В связи с этим многие высококачественные и полезные проекты предназначены для решения практических потребностей в реальных приложениях, и в этой книге вы найдете ряд таких проектов. С одним из них вы ознакомитесь достаточно подробно, и это, как не трудно догадаться, каркас Spring. Первая версия Spring была выпущена в октябре 2002 года и состояла из небольшого ядра с контейнером управления инверсией (IoC), который нетрудно было настроить и применять. Со временем каркас Spring пришел на смену серверам JEE (Java Enterprise Edition — корпоративная версия Java) и дорос до уровня полноценной технологии, состоящей из целого ряда отдельных проектов с собственным назначением. Таким образом, чтобы вы ни разрабатывали, будь то микрослужбы, приложения или классические системы планирования ресурсов предприятия (ERP), в Spring найдется для этой цели подходящий проект.

На страницах этой книги вы ознакомитесь с самыми разными технологиями с открытым кодом, объединенными в каркас Spring Framework. Работая с Spring, разработчик приложений может пользоваться широким спектром инструментальных средств с открытым кодом, не прибегая к написанию больших объемов кода и не слишком тесно привязывая создаваемое приложение к какому-то конкретному инструментальному средству.

В этой главе представлено введение в каркас Spring Framework без основательных примеров или подробных пояснений. Если вы уже знакомы с каркасом Spring, сразу переходите к чтению главы 2.

Что такое Spring

Пожалуй, самой трудной частью объяснения технологии Spring является точная ее классификация. Обычно Spring описывается как легковесный каркас для построения приложений на Java, но с этой формулировкой связаны два интересных момента.

Во-первых, Spring можно применять для построения любого приложения на языке Java (например, автономных, веб-приложений или корпоративных (JEE) приложений на Java), в отличие от многих других каркасов и, в частности, от каркаса Apache Struts, предназначенного только для создания веб-приложений. Во-вторых, *легковесный* характер Spring на самом деле обозначает не количество классов или размеры дистрибутива, а главный принцип всей философии Spring — минимальное воздействие. Spring является легковесным каркасом в том смысле, что для использования всех преимуществ ядра Spring вам придется внести лишь минимальные (если вообще какие-нибудь) изменения в свой прикладной код, а если в какой-то момент вы решите не пользоваться Spring, то и это сделать будет очень просто.

Обратите внимание на то, что речь идет только о ядре Spring — многие дополнительные компоненты Spring, в том числе для доступа к данным, требуют более тесной привязки к Spring Framework. Но польза от такой привязки вполне очевидна, и на протяжении всей этой книги мы представим методики, позволяющие свести к минимуму влияние подобной привязки на разрабатываемые приложения.

Эволюция Spring Framework

Каркас Spring Framework берет свое начало из книги *Expert One-on-One: J2EE Design and Development* Рода Джонсона (Rod Johnson; издательство Wrox, 2002 г.). За прошедшее десятилетие каркас Spring Framework значительно вырос в плане основных функциональных возможностей, связанных с ним проектов и поддержки со стороны сообщества разработчиков. Теперь, когда доступен новый крупный выпуск Spring Framework, полезно взглянуть на важные функциональные средства, которые появлялись в каждом промежуточном выпуске Spring и в конечном итоге привели к выходу версии Spring Framework 5.0.

- **Spring 0.9.** Первый публичный выпуск данного каркаса, основанный на упомянутой выше книге *Expert One-on-One: J2EE Design and Development* и предоставлявший основы для конфигурирования компонентов Spring Beans, поддержку аспектно-ориентированного программирования (АОП), абстракцию каркаса JDBC, обработки транзакций и пр. В этой версии отсутствовала офи-

циальная справочная документация, хотя существующие источники и документацию можно найти на веб-сайте SourceForge[1].

- **Spring 1.x.** Первая версия, выпущенная с официальной справочной документацией. Она состоит из семи модулей, приведенных на рис. 1.1.

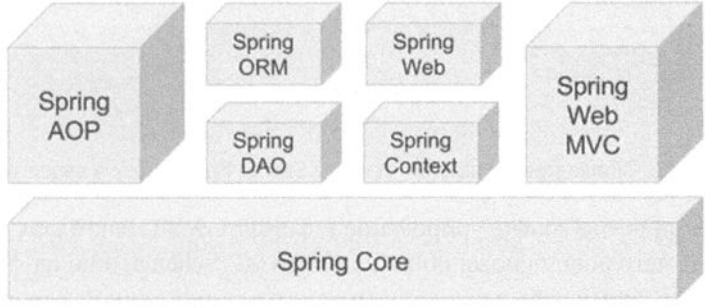


Рис. 1.1. Общее представление состава Spring Framework в версии 1.x

  - **Spring Core.** Контейнер компонентов Spring Beans и поддерживающие утилиты.
  - **Spring Context.** Интерфейс `ApplicationContext`, пользовательский интерфейс, проверка достоверности, интерфейс JNDI, компоненты Enterprise JavaBeans (EJB), удаленное взаимодействие и поддержка почты.
  - **Spring DAO.** Поддержка инфраструктуры обработки транзакций, интерфейса Java Database Connectivity (JDBC) и объектов доступа к данным (DAO).
  - **Spring ORM.** Поддержка технологий Hibernate, iBATIS и Java Data Objects (JDO).
  - **Spring AOP.** Реализация аспектно-ориентированного программирования (АОП), согласованная с набором интерфейсов AOP Alliance.
  - **Spring Web.** Базовые средства интеграции, включая составные функциональные средства, инициализацию контекста с помощью приемников сервлетов и веб-ориентированный контекст приложений.
  - **Spring Web MVC.** Каркас для построения веб-приложений по проектному шаблону "модель–представление–контроллер" (Model-View-Controller — MVC).

- **Spring 2.x.** Эта версия состоит из шести модулей, представленных на рис. 1.2. В настоящее время модуль Spring Context входит в состав ядра Spring Core, а все упоминаемые здесь веб-компоненты Spring описаны под одной рубрикой перечисляемых ниже особенностей данной версии.

[1] Прежние версии Spring, включая и версию 0.9, можно загрузить по адресу `https://sourceforge.net/projects/springframework/files/springframework/`.

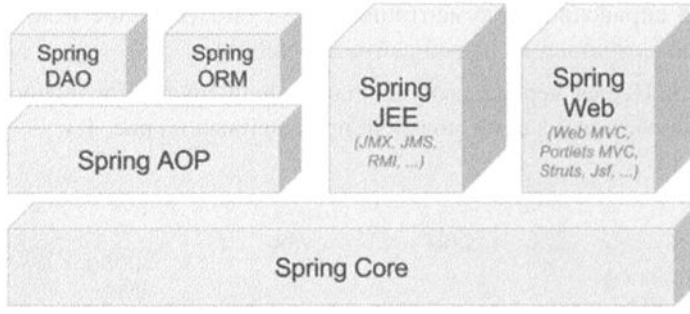


Рис. 1.2. Общее представление состава Spring Framework в версии 2.x

— Более простое конфигурирование в формате XML благодаря применению новой методики, основанной на языке XML Schema, а не на формате DTD. К числу примечательных усовершенствований данной версии относятся определения компонентов Spring Beans, АОП и декларативных транзакций.

— Новые области для применения компонентов Spring Beans в веб-приложениях и порталах (на уровне запросов, локальных и глобальных сеансов связи).

— Поддержка аннотаций `@AspectJ` для разработки средствами АОП.

— Уровень абстракции прикладного интерфейса сохраняемости Java Persistence API (JPA).

— Полная поддержка объектов POJO (Plain Old Java Object — простой старый объект Java), асинхронно управляемых системой обмена сообщениями JMS.

— Упрощение интерфейса JDBC, в том числе класса `SimpleJdbcTemplate`, когда применяется Java, начиная с версии 5.

— Поддержка именованных параметров JDBC (класс `NamedParameterJdbcTemplate`).

— Библиотека дескрипторов форм для модуля Spring MVC.

— Внедрение каркаса Portlet MVC.

— Поддержка динамических языков: компоненты Spring Beans могут быть написаны на JRuby, Groovy и BeanShell.

— Поддержка уведомлений и управляемой регистрации компонентов MBeans в JMX.

— Абстракция интерфейса `TaskExecutor`, введенная для планирования заданий.

— Поддержка аннотаций в Java 5, в том числе `@Transactional` и `@Required` в дополнение к аннотации `@AspectJ`.

■ **Spring 2.5.x.** Эта версия отличается следующими особенностями.

— Новая конфигурационная аннотация `@Autowired` и поддержка аннотаций по спецификации JSR-250 (`@Resource`, `@PostConstruct`, `@PreDestroy`).

— Новые стереотипные аннотации: `@Component`, `@Repository`, `@Service`, `@Controller`.

— Поддержка автоматического просмотра путей к классам для обнаружения и связывания классов, снабженных стереотипными аннотациями.

— Обновления АОП, включая новый элемент среза компонентов Spring Beans и привязывание аспектов во время загрузки по аннотации `@AspectJ`.

— Полная поддержка управления транзакциями на сервере приложений WebSphere.

— В дополнение к аннотации `@Controller` из модуля Spring MVC добавлены аннотации `@RequestMapping`, `@RequestParam` и `@ModelAttribute` для поддержки обработки запросов посредством конфигурирования аннотаций.

— Поддержка шаблонизатора Tiles 2.

— Поддержка каркаса JSF 1.2.

— Поддержка JAX-WS 2.0/2.1.

— Внедрение каркаса Spring TestContext Framework для поддержки управляемого аннотациями комплексного тестирования, не зависящего от применяемой среды тестирования.

— Возможность развертывать контекст приложений Spring в качестве адаптера JCA.

■ **Spring 3.0.x.** Это первая версия Spring, основанная на версии Java 5 с целью извлечь наибольшую выгоду из таких языковых средств, внедренных в версии Java 5, как обобщения, методы с аргументами переменной длины и прочими усовершенствованиями языка Java. В этой версии внедрена основанная на Java модель конфигурирования с помощью аннотаций `@Configuration`. Все модули каркаса Spring были исправлены, чтобы управлять ими по отдельности в одном дереве исходного кода, приходящемся на архивный JAR-файл каждого модуля. Общее представление состава данной версии приведено на рис. 1.3, а далее перечислены особенности данной версии.

— Поддержка языковых средств Java 5, в том числе обобщений, методов с аргументами переменной длины и прочих усовершенствований языка Java.

— Первоклассная поддержка вызываемых объектов, будущих действий, адаптеров типа `ExecutorService`, а также интеграция интерфейса `ThreadFactory`.

— Управление модулями каркаса по отдельности в одном дереве исходного кода, приходящемся на архивный JAR-файл каждого модуля.

— Внедрение языка выражений Spring (Spring Expression Language — SpEL).

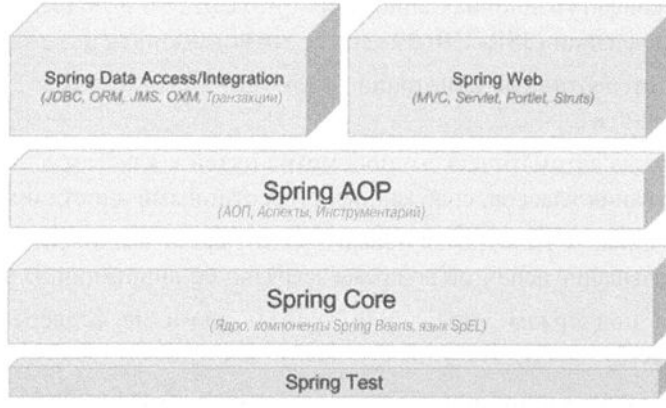


Рис. 1.3. Общее представление состава Spring Framework в версии 3.x

— Интеграция основных средств и аннотаций для конфигурирования на языке Java (сокращенно — Java Config).

— Универсальная система преобразования типов и система форматирования полей.

— Всесторонняя поддержка технологии REST.

— Новое пространство имен MVC XML и дополнительные аннотации вроде `@CookieValue` и `@RequestHeaders` для модуля Spring MVC.

— Усовершенствованная проверка достоверности и поддержка спецификации JSR-303 (Проверка достоверности компонентов Spring Beans).

— Первоначальная поддержка версии Java EE 6: аннотаций `@Async`/`@Asynchronous`, JSR-303, JSF 2.0, JPA 2.0 и т.д.

— Поддержка встроенных баз данных, в том числе HSQL, H2 и Derby.

■ **Spring 3.1.x.** Эта версия отличается следующими особенностями.

— Новая абстракция кеша.

— Профили определения компонентов Spring Beans могут быть заданы в формате XML. Поддерживается также аннотация `@Profile`.

— Абстракция среды для единообразного управления свойствами.

— Эквиваленты аннотаций для таких распространенных элементов пространства имен Spring XML, как `@ComponentScan`, `@EnableTransactionManagement`, `@EnableCaching`, `@EnableWebMvc`, `@EnableScheduling`, `@EnableAsync`, `@EnableAspectJAutoProxy`, `@EnableLoadTimeWeaving` и `@EnableSpringConfigured`.

— Поддержка каркаса Hibernate 4.

— Поддержка в каркасе Spring TestContext Framework конфигурационных классов с аннотацией `@Configuration` и профилей определения компонентов Spring Beans.

— Пространство имен `c:` для упрощенного внедрения зависимостей через конструктор.

— Поддержка конфигурирования контейнера сервлетов на основе кода Servlet 3.

— Возможность выполнять начальную загрузку интерфейса `EntityManagerFactory` из прикладного интерфейса сохраняемости JPA без файла `persistence.xml`.

— Внедрение временно сохраняемых (классы `FlashMap` и `FlashMapManager`) и переадресовываемых (класс `RedirectAttributes`) атрибутов в модуль Spring MVC, чтобы атрибуты продолжали существовать после переадресации в сеансе связи по сетевому протоколу HTTP.

— Усовершенствование переменных шаблонов URI.

— Возможность снабжать аннотацией `@Valid` аргументы методов из контроллеров с аннотациями `@RequestBody` в модуле Spring MVC.

— Возможность снабжать аннотацией `@RequestPart` аргументы методов из контроллеров в модуле Spring MVC.

- **Spring 3.2.x.** Эта версия отличается следующими особенностями.

— Поддержка асинхронной обработки запросов на основе Servlet 3.

— Новая среда тестирования в модуле Spring MVC.

— Новые аннотации `@ControllerAdvice` и `@MatrixVariable` в модуле Spring MVC.

— Поддержка обобщенных типов в классе `RestTemplate` и аргументах с аннотацией `@RequestBody`.

— Поддержка формата Jackson JSON 2.

— Поддержка шаблонизатора Tiles 3.

— Возможность указывать аргумент `Errors` после аргумента, снабженного аннотацией `@RequestBody` или `@RequestPart`, чтобы обрабатывать ошибки проверки достоверности.

— Возможность исключать шаблоны URL, используя пространство имен MVC и параметры конфигурации Java Config.

— Поддержка аннотации `@DateTimeFormat` без библиотеки Joda Time.

— Глобальное форматирование даты и времени.

— Усовершенствование параллелизма во всем каркасе, сводя к минимуму блокировки и в целом улучшая параллельное создание компонентов Spring Beans: как прототипных, так и с локальной областью видимости.

— Новая система сборки, основанная на Gradle.

— Переход на хранилище GitHub по адресу `https://github.com/SpringSource/spring-framework`.

— Усовершенствованная поддержка Java SE 7/OpenJDK 7 как в самом каркасе, так и в сторонних зависимостях. Включение библиотек CGLIB и ASM в состав Spring. Помимо версии AspectJ 1.6, поддерживается версия AspectJ 1.7.

- **Spring 4.0.x.** Основной выпуск Spring, где впервые полностью поддерживается версия Java 8. И хотя могут быть использованы прежние версии Java, минимальные требования повышены вплоть до версии Java SE6. Не рекомендованные к применению классы и методы исключены, а модульная организация Spring осталась практически без изменений, как показано на рис. 1.4. Далее перечислены особенности данной версии.

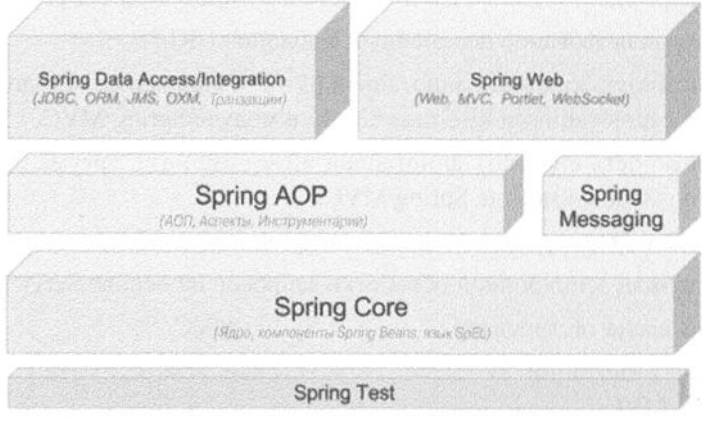


Рис. 1.4. Общее представление состава Spring Framework в версии 4.x

— Улучшенная практика начала работы с помощью целого ряда вводных руководств, доступных по адресу `www.spring.io/guides`.

— Удаление устаревших пакетов и методов, относящихся к предыдущей версии Spring 3.

— Поддержка Java 8 с повышением требований к минимальной версии до уровня Java 6 Update 18.

— Назначение версии Java EE 6 и выше в качестве базового уровня для версии Spring Framework 4.

— Предметно-ориентированный язык программирования (DSL), позволяющий конфигурировать определения компонентов Groovy Beans с помощью синтаксиса языка Groovy.

— Усовершенствование основных контейнеров, тестирования и общей инфраструктуры для разработки веб-приложений.

— Обмен сообщениями по протоколам WebSocket, SockJS и STOMP.

- **Spring 4.2.x.** Эта версия отличается следующими особенностями.
  - — Усовершенствование ядра (например, внедрение аннотации `@AliasFor` и видоизменение существующей аннотации, чтобы воспользоваться ею).
  - — Полная поддержка библиотеки Hibernate ORM 5.0.
  - — Усовершенствование средств для разработки веб-разработки и службы JMS.
  - — Усовершенствование обмена сообщениями по протоколу WebSocket.
  - — Усовершенствование средств тестирования и в особенности внедрение аннотации `@Commit` вместо аннотации `@Rollback(false)`, а также служебного класса `AopTestUtils`, обеспечивающего доступ к базовому объекту, скрывающемуся позади объекта-заместителя в Spring.
- **Spring 4.3.x.** Эта версия отличается следующими особенностями.
  - — Уточнение модели программирования.
  - — Существенное усовершенствование контейнера ядра Spring (благодаря внедрению библиотек ASM 5.1, CGLIB 3.2.4 и Objenesis 2.4 в архивный JAR-файл `spring-core.jar`) и модуля Spring MVC.
  - — Внедрение составных аннотаций.
  - — Повышение требований до версии JUnit 4.12 или выше в каркасе Spring TestContext Framework.
  - — Поддержка новых библиотек, в том числе Hibernate ORM 5.2, Hibernate Validator 5.3, Tomcat 8.5 и 9.0, Jackson 2.8 и т.д.
- **Spring 5.0.x.** Основной выпуск Spring, в котором вся кодовая база данного каркаса написана на Java 8 и полностью совместима с версией Java 9 на момент ее выпуска в июле 2016 года[2]. Эта версия отличается следующими особенностями.
  - — Прекращена поддержка Portlet, Velocity, JasperReports, XMLBeans, JDO, Guava, Tiles2 и Hibernate3.
  - — Конфигурирование пространств имен в формате XML сведено к схемам без версий. Объявления, характерные для конкретных версий, по-прежнему поддерживаются, но все же проверяются на достоверность по последней схеме XSD.
  - — Общие усовершенствования с целью извлечь наибольшую пользу из языковых средств Java 8.
  - — Абстракция интерфейса `Resource`, предоставляющая метод `isFile()`, возвращающий признак для безопасного доступа к файлу методом `getFile()`.

[2] Официально версия Java 9 была выпущена в сентябре 2017 года по планам компании Oracle (см. `http://openjdk.java.net/projects/jdk9/`).

— Полная поддержка сигнатур Servlet 3.1 в реализациях интерфейса `Filter` в каркасе Spring.

— Поддержка пакета Protobuf 3.0.

— Внедрение проекта Spring Web Flow, построенного на основе модели реактивного программирования, в качестве альтернативы модулю Spring MVC. Это означает, что он полностью асинхронный, неблокирующий и предназначен для применения в модели выполнения цикла ожидания событий вместо традиционно крупного пула потоков с моделью их исполнения по каждому запросу, встроенной в библиотеку Project Reactor[3].

— Адаптация модулей веб и ядра к модели реактивного программирования[4].

— Многочисленные усовершенствования тестового модуля Spring. Поддержка платформы модульного тестирования JUnit 5 и внедрение новых аннотаций, в том числе `@SpringJUnitConfig`, `@SpringJUnitWebConfig`, `@EnabledIf`, `@DisabledIf`, предназначенных для поддержки модели программирования и расширения Jupiter.

— Поддержка параллельного выполнения тестов в каркасе Spring TestContext Framework.

Инверсия управления или внедрение зависимостей?

Ядро каркаса Spring Framework основано на принципе *инверсии управления* (Inversion of Control — IoC), когда создание зависимостей между компонентами и управление ими осуществляется внешним образом. Рассмотрим в качестве примера класс `Foo`, который зависит от экземпляра класса `Bar` для выполнения определенного вида обработки. Традиционно экземпляр класса `Bar` получается в классе `Foo` с помощью операции `new` или какой-нибудь разновидности фабричного класса. В соответствии с принципом инверсии управления экземпляр класса `Bar` (или его подкласса) предоставляется классу `Foo` во время выполнения некоторым внешним процессом. Такое поведение во время выполнения привело к тому, что понятие инверсии управления было переименовано Мартином Фаулером в более описательное понятие *внедрения зависимостей* (Dependency Injection — DI). Конкретный характер зависимостей, управляемых посредством их внедрения, обсуждается в главе 3.

---

[3] В библиотеке Project Reactor реализована спецификация на прикладной интерфейс Reactive Streams API (см. `https://projectreactor.io/`).

[4] Реактивным называется стиль программирования микроархитектуры, включающей интеллектуальную маршрутизацию и употребление событий. Это должно привести к созданию неблокирующих приложений, действующих асинхронно, управляемых событиями и требующих небольшого количества потоков исполнения для масштабирования по вертикали в пределах виртуальной машины Java, а не по горизонтали через кластеризацию.

На заметку Как будет показано в главе 3, употребление термина *внедрение зависимостей* для обозначения инверсии управления всегда корректно. В контексте Spring эти термины можно употреблять равноценно, что совершенно не теряет их смысл.

Реализация внедрения зависимостей в Spring основана на двух ключевых понятиях Java: компонентах JavaBeans и интерфейсах. Используя Spring в качестве поставщика внедрения зависимостей, вы достигаете гибкости в определении конфигурации зависимостей в своих приложениях самыми разными средствами (например, с помощью XML-файлов, конфигурационных классов Java, аннотаций в прикладном коде или нового способа определения компонентов Groovy Beans). Компоненты JavaBeans (т.е. объекты POJO) предоставляют стандартный механизм для создания ресурсов Java, которые можно конфигурировать самыми разными средствами вроде конструкторов и методов установки. Как поясняется в главе 3, для формирования ядра своей модели конфигурации внедрения зависимостей в Spring применяется спецификация компонентов JavaBeans. В действительности любой ресурс, действующий под управлением каркаса Spring, считается *компонентом Spring Bean*. На тот случай, если вы еще не знакомы с компонентами JavaBeans, в начале главы 3 будет приведен краткий их пример.

Интерфейсы и внедрение зависимостей взаимовыгодны. Очевидно, что проектирование и написание прикладного кода по интерфейсам способствует повышению гибкости приложений, но сложность связывания вместе отдельных частей приложения, разработанного с помощью интерфейсов, весьма велика и требует от разработчиков дополнительных усилий по программированию. Применяя внедрение зависимостей, можно сократить объем кода, который требуется для разработки приложения на основе интерфейсов, практически до нуля. А с помощью интерфейсов можно получить максимальную отдачу от внедрения зависимостей, потому что в компонентах Spring Beans можно использовать любую реализацию интерфейса для удовлетворения их зависимости. Кроме того, применение интерфейсов позволяет задействовать в Spring динамические объекты-заместители из комплекта JDK по проектному шаблону "Заместитель" (Proxy), чтобы предоставить эффективные средства вроде АОП для сквозной ответственности, иначе называемой сквозной функциональностью.

В контексте внедрения зависимостей Spring действует в большей степени подобно контейнеру, чем каркасу, предоставляя экземпляры классов приложения со всеми необходимыми зависимостями, но делает это гораздо менее навязчиво. Применение каркаса Spring для внедрения зависимостей опирается всего лишь на соблюдение в его классах соглашений об именовании, принятых для компонентов JavaBeans. Но ни специальных классов для наследования, ни собственных схем именования для соблюдения в Spring не предусмотрено. Во всяком случае единственным изменением, которое делается в приложении, где применяется внедрение зависимостей, является открытие доступа к дополнительным свойствам компонентов JavaBeans, что дает возможность внедрять больше зависимостей во время выполнения.

Эволюция внедрения зависимостей

Благодаря распространению Spring и других каркасов внедрения зависимостей за последние несколько лет внедрение зависимостей завоевало широкое признание в сообществе разработчиков приложений на Java. В то же время разработчики убедились, что применение внедрения зависимостей оказалось нормой наилучшей практики в разработке приложений, а преимущества от внедрения зависимостей были хорошо усвоены.

Распространенность внедрения зависимостей была подтверждена и официальным принятием в 2009 году документа JSR-330 (Dependency Injection for Java — Внедрение зависимостей для Java) в организации JCP (Java Community Process — Процесс сообщества Java). Документ JSR-330 стал формальным запросом спецификации Java (Java Specification Request — JSR), а одним из ее ведущих авторов был Род Джонсон (Rod Johnson), основатель Spring Framework.

В версии JEE 6 документ JSR-330 стал одной из спецификаций, входящих в полный набор технологий. Между тем архитектура EJB (начиная с версии 3.0) была также модернизирована в значительной степени. В ней была принята модель внедрения зависимостей, чтобы упростить разработку разнообразных приложений для компонентов Enterprise JavaBeans.

Несмотря на то что подробное обсуждение внедрения зависимостей откладывается до главы 3, стоит все же перечислить преимущества, которые оно дает по сравнению с более традиционным подходом.

- **Сокращение объема связующего кода.** К числу самых главных достоинств внедрения зависимостей относится возможность значительного сокращения объема кода, который должен быть написан для связывания вместе компонентов приложения. Зачастую этот код тривиален, и поэтому создание зависимости сводится лишь к получению нового экземпляра объекта. Но связующий код может заметно усложниться, когда потребуется найти зависимости в хранилище JNDI или если зависимости нельзя вызвать напрямую, как это происходит с удаленными ресурсами. В подобных случаях внедрение зависимостей может действительно упростить связующий код, автоматически обеспечивая поиск в хранилище JNDI и замещение удаленных ресурсов.
- **Упрощенное конфигурирование приложений.** Внедряя зависимости, можно в значительной степени упростить процесс конфигурирования приложений. Для конфигурирования классов, внедренных в другие классы, можно выбрать разные варианты. Таким же способом можно сформулировать потребности в зависимостях для внедряющего объекта, когда внедряется соответствующий экземпляр или свойство компонента Spring Bean. Кроме того, внедрение зависимостей намного упрощает замену одной реализации зависимости на другую. Рассмотрим, например, случай, когда имеется объект DAO, выполняющий операции над данными в базе данных PostgreSQL, и требуется перейти на базу данных Oracle. Применяя внедрение зависимостей, можно без особого труда

переконфигурировать соответствующую зависимость объектов предметной области, чтобы воспользоваться реализацией базы данных Oracle вместо PostgreSQL.

- **Возможность управлять общими зависимостями в одном хранилище.** При традиционном подходе к управлению зависимостями общих служб (например, для подключения к источнику данных, обработки транзакций и удаленных вызовов) экземпляры зависимостей получаются (непосредственно или из некоторых фабричных классов) там, где они требуются (обычно в зависимом классе). Это приводит к распространению зависимостей среди многих классов в приложении, что может затруднить их изменение. А если применяется внедрение зависимостей, то вся информация об общих зависимостях находится в единственном хранилище, значительно упрощая управление ими и уменьшая подверженность ошибкам.
- **Улучшенная тестируемость.** Когда классы проектируются для внедрения зависимостей, становится возможной простая замена зависимостей. Это особенно удобно для тестирования приложений. Рассмотрим в качестве примера объект предметной области, выполняющий сложную обработку, используя при этом объект DAO для доступа к информации, хранящейся в реляционной базе данных. При тестировании нас не интересует проверка объекта DAO, поскольку требуется протестировать объект предметной области с различными наборами данных. При традиционном подходе, когда объект предметной области самостоятельно отвечает за получение экземпляра объекта DAO, тестирование довольно затруднительно, поскольку невозможно заменить реализацию объекта DAO имитированной реализацией, которая возвращала бы наборы тестовых данных. Вместо этого придется обеспечить наличие правильной информации в тестовой базе данных и применить полную реализацию объекта DAO для выполнения тестов. А применяя внедрение зависимостей, можно сымитировать реализацию объекта DAO, возвращающую наборы тестовых данных, а затем передать их объекту предметной области для проверки. Такой подход можно расширить, чтобы тестировать приложения на любом уровне, что особенно удобно для тестирования веб-компонентов, для которых имитируются реализации объектов типа `HttpServletRequest` и `HttpServletResponse`.
- **Стимулирование качественных проектных решений для приложений.** Вообще говоря, проектирование для внедрения зависимостей означает проектирование по отношению к интерфейсам. Типичное приложение, ориентированное на внедрение зависимостей, построено таким образом, чтобы все его основные компоненты были сначала определены как интерфейсы, а затем конкретные их реализации должны быть созданы и связаны вместе с помощью контейнера внедрения зависимостей. Принять такое проектное решение в Java можно было еще до появления внедрения зависимостей и основанных на нем контейнеров, подобных Spring. Но, применяя Spring, вы получаете в свое рас-

поряжение целый ряд средств для внедрения зависимостей и можете сосредоточиться на построении логики приложения, а не на самом каркасе, поддерживающем внедрение зависимостей.

Как видите, внедрение зависимостей дает приложению немало преимуществ, хотя оно и не лишено недостатков. В частности, внедрение зависимостей способно затруднить тем, кто не особенно хорошо ориентируется в прикладном коде, возможность выяснить, какая именно реализация отдельной зависимости привязана к конкретным объектам. Как правило, подобное затруднение возникает только из-за отсутствия опыта внедрения зависимостей. Приобретя необходимый опыт и следуя нормам надлежащей практики программирования для внедрения зависимостей (например, размещая все классы, внедряемые на каждом уровне приложения в одном и том же пакете), разработчики смогут легко представить всю картину в целом. Перечисленные выше преимущества обычно перевешивают этот небольшой недостаток, хотя о нем следует помнить, планируя свое приложение.

Другие возможности, помимо внедрения зависимостей

Сам модуль Spring Core со своими развитыми возможностями внедрения зависимостей является вполне достойным инструментальным средством, но каркас Spring выгодно отличается наличием обширного набора дополнительных средств, изящно спроектированных и построенных по принципам внедрения зависимостей. В каркасе Spring предоставляются необходимые средства для всех уровней приложения: от вспомогательных прикладных интерфейсов API, предназначенных для доступа к данным, до расширенных возможностей проектного шаблона MVC. Все эти средства примечательны тем, что легко интегрируются с другими инструментальными средствами Spring, становясь полноправными членами семейства Spring, хотя в Spring нередко предоставляется собственный подход.

Поддержка Java 9

В версии Java 8 предоставляется немало привлекательных средств, поддерживаемых в Spring Framework 5, наиболее примечательными из которых являются лямбда-выражения и ссылки на методы с интерфейсами обратного вызова Spring. План выпуска версии Spring 5 был согласован с первоначальным планом выпуска комплекта JDK 9. И если конечный срок выпуска комплекта JDK 9 был отодвинут на более позднюю дату, то версия Spring 5 была выпущена по плану. Предполагается, что возможности JDK 9 будут полностью охвачены в версии Spring 5.1, тогда как в версии Spring 5 — использованы такие средства JDK 9, как компактные строки, стек протоколов уровня ALPN и новая реализация HTTP-клиента. Несмотря на то что в версии Spring Framework 4.0 поддерживается версия Java 8, в ней по-прежнему обеспечивается обратная совместимость с версией JDK 6 Update 18. При разработке новых проектов рекомендуется применять самую последнюю версию Java (например, 7 или 8).

А в версии Spring 5.0 требуется Java, начиная с версии 8, поскольку разработчики Spring обновили кодовую базу этого каркаса до языкового уровня версии Java 8. Хотя версия Spring 5 изначально построена и на основе комплекта JDK 9, чтобы обеспечить всестороннюю поддержку языковых средств, заявленных в версии Java 9.

Аспектно-ориентированное программирование средствами Spring

Аспектно-ориентированное программирование (АОП) предоставляет возможность реализовать в одном месте *сквозную логику*, т.е. такую, которая применяется во многих частях приложения и обеспечивает ее автоматическое применение повсюду в приложении. Подход, принятый в Spring к АОП, состоит в создании динамических заместителей для целевых объектов и привязывании объектов к сконфигурированному совету для выполнения сквозной логики. В силу характера динамических заместителей в JDK целевые объекты должны реализовывать интерфейс, объявляющий метод, в котором будет применен совет АОП. Еще одной популярной библиотекой АОП является проект Eclipse AspectJ[5], в котором предоставляются более эффективные средства, в том числе конструирование объектов, загрузка классов и больше возможностей для сквозной функциональности. Разработчиков, применяющих Spring и АОП, можно порадовать тем, что с версии 2.0 в Spring поддерживается более тесная интеграция с библиотекой AspectJ. Ниже перечислены некоторые особенности такой интеграции.

- Поддержка выражений со срезами в стиле AspectJ.
- Поддержка стиля аннотаций `@AspectJ`, хотя для привязывания применяются встроенные в Spring средства АОП.
- Поддержка аспектов, реализованных в AspectJ для внедрения зависимостей.
- Поддержка привязывания во время загрузки в объекте типа `Application Context` из каркаса Spring.

На заметку Начиная с версии Spring Framework 3.2, поддержка аннотации **`@AspectJ`** может быть включена в конфигурацию, выполняемую средствами Java.

Обе упомянутых выше разновидности АОП имеют свои области применения, и в большинстве случаев встроенных в Spring средств АОП должно быть достаточно для удовлетворения требований сквозной функциональности в приложении. Но при более сложных требованиях может использоваться библиотека AspectJ, причем в одном приложении Spring можно сочетать обе разновидности АОП.

АОП находит немало примеров применения. Типичное применение АОП, демонстрируемое во многих традиционных примерах, связано с выполнением некоторого вида регистрации, но возможности АОП выходят далеко за рамки тривиальных приложений для регистрации. На самом деле АОП применяется в самом каркасе Spring

---

[5] См. `www.eclipse.org/aspectj`.

Framework для многих целей и, в частности, для управления транзакциями. Встроенные в Spring средства АОП подробно рассматриваются в главе 5, где демонстрируются типичные примеры применения АОП как в самом каркасе Spring Framework, так и в приложениях, а также обсуждаются вопросы производительности АОП и те области, в которых традиционные технологии подходят лучше, чем АОП.

Язык выражений Spring (SpEL)

Язык выражений (Expression Language — EL) — это технология, позволяющая приложению манипулировать объектами Java во время выполнения. Но дело в том, что разные технологии предлагают собственные реализации и синтаксис языка EL. Например, у технологий Java Server Pages (JSP) и Java Server Faces (JSF) имеются собственные языки EL, синтаксис которых отличается друг от друга. В качестве выхода из этого затруднительного положения был создан унифицированный язык выражений (Unified Expression Language).

Каркас Spring Framework развивался настолько быстро, что возникла потребность в стандартном языке выражений, который мог бы стать общим для всех модулей Spring Framework и других проектов Spring. Поэтому в версии 3.0 в Spring появился язык выражений Spring (Spring Expression Language — SpEL), предоставляющий эффективные средства для вычисления выражений, а также для доступа к объектам Java и компонентам Spring Beans во время выполнения. Получаемые в итоге результаты могут применяться в приложении или внедряться в другие компоненты JavaBeans.

Проверка достоверности в Spring

Проверка достоверности — еще одна крупная задача в приложениях любого вида. В идеальном сценарии правила проверки достоверности для атрибутов в компонентах JavaBeans, содержащих данные, могут быть применены согласованно и независимо от того, где инициирован запрос на манипулирование данными: в пользовательском интерфейсе, пакетном задании или же удаленно (например, через веб-службы вроде REST или удаленные вызовы процедур (Remote Procedure Call — RPC)).

Для решения подобных задач в Spring предлагается встроенный прикладной интерфейс API, предназначенный для проверки достоверности через интерфейс `Validator`. Этот интерфейс предоставляет простой и лаконичный механизм, позволяющий инкапсулировать логику проверки достоверности в класс, ответственный за проверку достоверности целевого объекта. Помимо целевого объекта, метод проверки достоверности принимает объект типа `Errors`, который используется для сбора любых ошибок, возникающих при проверке достоверности. В каркасе Spring имеется также служебный класс `ValidationUtils`, предоставляющий удобные методы для вызова других средств проверки достоверности, проверки наличия распространенных ошибок вроде пустых строк и сообщения об ошибках указанному объекту типа `Errors`.

Руководствуясь необходимостью, организация JCP разработала также спецификацию JSR-303 (Bean Validation), в которой определяются правила проверки достовер-

ности компонентов JavaBeans. Например, когда к свойству компонента JavaBean применяется аннотация `@NotNull`, она указывает, что данное свойство не должно содержать пустое значение `null` перед его сохранением в базе данных.

Начиная с версии 3.0 в Spring доступна встроенная поддержка спецификации JSR-303. Чтобы воспользоваться прикладным интерфейсом API для проверки достоверности, достаточно объявить объект типа `LocalValidatorFactoryBean` и внедрить интерфейс `Validator` в любые компоненты Spring Beans, а каркас Spring сам выявит базовую реализацию. По умолчанию Spring сначала найдет Hibernate Validator (`http://hibernate.org/validator/`) — распространенную реализацию спецификации JSR-303. Многие клиентские технологии (например, JSF 2 и Google Web Toolkit), включая модуль Spring MVC, также поддерживают применение проверки достоверности по спецификации JSR-303 в пользовательском интерфейсе. Прошли те времена, когда разработчикам приходилось писать одну и ту же логику проверки достоверности как в пользовательском интерфейсе, так и на уровне взаимодействия с базой данных. Более подробно речь об этом пойдет в главе 10.

На заметку С версии Spring Framework 4.0 поддерживается версия 1.1 спецификации JSR-349: Bean Validation.

Доступ к данным в Spring

Доступ к данным и их сохраняемость являются, пожалуй, самыми горячо обсуждаемыми темами в среде Java. Каркас Spring обеспечивает превосходную интеграцию с любым инструментальным средством, выбранным для доступа к данным. Кроме того, каркас Spring предлагает в качестве жизнеспособного решения для многих проектов простой интерфейс JDBC (Java Database Connectivity — подключение к базам данных Java) с упрощенными прикладными интерфейсами API, служащими оболочками для стандартного прикладного интерфейса API. Отдельный модуль Spring для доступа к данным обеспечивает стандартную поддержку технологий JDBC, Hibernate, JDO и JPA.

На заметку Начиная с версии Spring Framework 4.0 каркас iBAT IS не поддерживается. Интеграция с Spring обеспечивается в проекте MyBatis-Spring (подробнее об этом см. по адресу **`http://www.mybatis.org/spring/`**).

Но по причине бурного роста Интернета и облачных вычислений за последние несколько лет было разработано много других баз данных специального назначения, помимо реляционных. Примерами могут служить базы данных, основанные на парах "ключ–значение" и предназначенные для обработки очень больших объемов данных (в целом они относятся к категории нереляционных баз данных типа NoSQL), а также графовые и документные базы данных. Чтобы помочь разработчикам в поддержке таких баз данных и не усложнять модель доступа к данным Spring, был создан от-

дельный проект Spring Data[6], который в дальнейшем был разделен на разные категории для поддержки более конкретных требований доступа к данным.

На заметку Поддержка в Spring нереляционных баз данных в этой книге не рассматривается. Для тех, кто интересуется этой темой, упомянутый выше проект Spring Data послужит неплохой отправной точкой. На веб-странице данного проекта, доступной по адресу, указанному в приведенной выше сноске, перечислены нереляционные базы данных, которые он поддерживает, а также приведены ссылки на начальные страницы со сведениями об этих базах данных.

Поддержка интерфейса JDBC в Spring делает вполне реальным построение приложения на основе JDBC даже в особенно сложных случаях. А поддержка технологий Hibernate, JDO и JPA еще больше упрощает и без того простые прикладные интерфейсы API, облегчая бремя разработчиков приложений. Используя прикладные интерфейсы API для доступа к данным через любое инструментальное средство, можно извлечь выгоду из превосходной поддержки транзакций в Spring. Подробнее об этом речь пойдет в главе 9.

Одной из самых примечательных особенностей Spring является возможность простого сочетания технологий доступа к данным в приложении. Например, приложение может работать с базой данных Oracle, но использовать технологию Hibernate для большей части логики доступа к данным. Но если требуются какие-то особые средства Oracle, то соответствующую часть уровня доступа к данным очень просто реализовать с помощью прикладных интерфейсов JDBC API, поддерживаемых в Spring.

Поддержка механизма OXM в Spring

Многие приложения должны интегрироваться с другими приложениями или предоставлять для них соответствующие службы. К числу общих требований относится обмен данными с другими системами на регулярной основе или в реальном времени. Наиболее распространенным форматом данных считается XML. В итоге возникает потребность во взаимном преобразовании объектов Java и данных формата XML.

В каркасе Spring поддерживается немало общепринятых каркасов для взаимного преобразования объектов Java и данных формата XML и, как обычно, исключается потребность в непосредственной привязке к любой конкретной реализации. В частности, для маршализации (преобразования компонентов JavaBeans в данные формата XML), демаршализации (преобразования данных формата XML в объекты Java) и внедрения зависимостей в любые компоненты Spring Beans в каркасе Spring предоставляются общие интерфейсы, а также поддерживаются такие распространенные библиотеки, как Java Architecture for XML Binding (JAXB), Castor, XStream, JiBX и XMLBeans. При рассмотрении в главе 12 удаленного доступа из приложения Spring к данным формата XML из предметной области будет показано, как использовать в своих приложениях поддержку механизма OXM (Object to XML Mapping — преобразование объектов в данные формата XML), предоставляемую в Spring.

---

[6] См. `http://projects.spring.io/spring-data`.

Управление транзакциями

В каркасе Spring обеспечивается превосходный уровень абстракции для управления транзакциями, на котором можно производить программный и декларативный контроль над транзакциями. Благодаря такому уровню абстракции упрощается смена базового протокола транзакций и диспетчеров ресурсов. Начать можно с простых, локальных, характерных для конкретного ресурса транзакций, а затем перейти к глобальным, многоресурсным транзакциям, не меняя прикладной код. Транзакции подробно рассматриваются в главе 9.

Упрощение и интеграция с платформой JEE

Благодаря растущему принятию на вооружение таких каркасов для внедрения зависимостей, как Spring, многие разработчики решили строить свои приложения, используя подобные каркасы для поддержки EJB на платформе JEE. В итоге сообщество JCP осознало также сложность архитектуры EJB. В версии 3.0 спецификации EJB доступный прикладной интерфейс API был упрощен и теперь включает в себя многие концепции, заимствованные из внедрения зависимостей.

Но для приложений, которые построены на основе EJB или должны развертывать приложения Spring в контейнере JEE и пользоваться корпоративными службами сервера приложений (например, диспетчером транзакций JTA (Java Transaction API — прикладной интерфейс транзакций Java), пулом подключений к источникам данных и фабриками подключений через службу JMS), в каркасе Spring обеспечивается также упрощенная поддержка подобных технологий. С одной стороны, для поддержки EJB в каркасе Spring предоставляется простое объявление, позволяющее выполнять поиск через интерфейс JNDI и внедрять компоненты EJB в компоненты Spring Beans. А с другой с стороны, в Spring предоставляется простая аннотация для внедрения компонентов Spring Beans в компоненты EJB.

Что же касается любых ресурсов, сохраняемых в местах, доступных через интерфейс JNDI, то каркас Spring позволяет избавиться от сложного поискового кода и внедрять ресурсы, управляемые через интерфейс JNDI, как зависимости в другие объекты во время выполнения. В качестве побочного эффекта приложения перестают быть привязанными к интерфейсу JNDI, расширяя тем самым возможности для повторного использования кода в перспективе.

MVC на веб-уровне

Несмотря на то что каркас Spring можно применять практически в любых средах: от настольной системы до веб, он предлагает широкий спектр классов, поддерживающих создание веб-приложений. Каркас Spring обеспечивает максимальную гибкость при выборе способа реализации пользовательского интерфейса для веб-приложения. Чаще всего при разработке веб-приложений применяется проектный шаблон MVC. В последних версиях Spring постепенно развился от простого каркаса веб-приложений до полноценной реализации проектного шаблона MVC.

Прежде всего следует отметить обширную поддержку представлений в модуле Spring MVC. Помимо стандартной поддержки JSP и JSTL (Java Standard Tag Library — стандартная библиотека дескрипторов Java), которая значительно подкреплена библиотеками дескрипторов Spring, в распоряжении разработчиков имеется полностью интегрированная поддержка Apache Velocity, FreeMarker, Apache Tiles, Thymeleaf и XSLT. Кроме того, доступен набор базовых классов представлений, которые упрощают ввод в приложения данных в форматах Microsoft Excel, PDF и JasperReports.

Как правило, поддержки проектного шаблона MVC в Spring оказывается достаточно для удовлетворения потребностей разрабатываемых веб-приложений. Но каркас Spring может интегрироваться и с другими распространенными каркасами веб-приложений, в том числе Struts, JSF, Atmosphere, Google Web Toolkit (GWT) и т.д.

Технология каркасов веб-приложений стремительно развивалась в последние годы. Пользователи требовали более быстрого отклика и высокой интерактивности, и это привело к появлению Ajax — широко распространенной технологии для разработки насыщенных (т.е. полнофункциональных) интернет-приложений (RIA). С другой стороны, пользователи также хотят иметь доступ к своим приложениям на любом устройстве, включая смартфоны и планшеты. Это порождает потребность в каркасах веб-приложений, поддерживающих HTML5, JavaScript и CSS3. Разработка веб-приложений с помощью модуля Spring MVC будет рассмотрена в главе 16.

Поддержка протокола WebSocket

В версии Spring Framework 4.0 стала доступной поддержка прикладного интерфейса Java API для протокола WebSocket (спецификация JSR-356). В протоколе WebSocket определен прикладной интерфейс API для установления постоянного соединения клиента с сервером, обычно реализуемого в веб-браузерах и на серверах. Разработка в стиле WebSocket открывает простор для эффективной двухсторонней связи, обеспечивающей обмен сообщениями в реальном времени для быстро реагирующих приложений. О поддержке протокола WebSocket подробнее речь пойдет в главе 17.

Поддержка удаленного взаимодействия

Доступ к удаленным компонентам или их раскрытие в Java никогда не было простой задачей. Используя каркас Spring, можно получить в свое распоряжение обширную поддержку обширного ряда методик удаленного взаимодействия для быстрого раскрытия и доступа к удаленным службам.

Каркас Spring поддерживает разнообразные механизмы удаленного доступа, включая RMI (Java Remote Method Invocation — удаленный вызов методов Java), JAX-WS, протоколы Hessian и Burlap от компании Caucho Technology, JMS, AMQP (Advanced Message Queuing Protocol — расширенный протокол организации очереди сообщений) и REST. Помимо этих протоколов удаленного взаимодействия, в Spring предоставляется собственный вызывающий компонент, основанный на сетевом протоколе HTTP и стандартной сериализации в Java. Используя возможности динамичес-

кого замещения в Spring, можно внедрить заместитель удаленного ресурса в качестве зависимости в свои классы. Благодаря этому исключается необходимость привязывать приложение к конкретной реализации удаленного взаимодействия и сокращается объем прикладного кода. Поддержка технологий удаленного взаимодействия в Spring рассматривается в главе 12.

Поддержка электронной почты

Отправка электронной почты является типичным требованием ко многим видам приложений и полноценно поддерживается в Spring Framework. В каркасе Spring предоставляется упрощенный прикладной интерфейс API, предназначенный для отправки сообщений электронной почты и хорошо согласующийся с возможностями внедрения зависимостей в Spring. Кроме того, в Spring поддерживается стандартный прикладной интерфейс JavaMail API.

Spring позволяет создать прототипное сообщение в контейнере внедрения зависимостей, чтобы использовать его как основание для всех остальных сообщений, отправляемых из приложения. Это упрощает настройку таких параметров почтового сообщения, как тема и адрес отправителя. Кроме того, для настройки тела сообщения каркас Spring интегрируется с шаблонизаторами вроде Apache Velocity, которые дают возможность вынести содержимое сообщений за пределы кода на Java.

Поддержка планирования заданий

В большинстве нетривиальных приложений требуются определенные возможности для планирования заданий. Возможность запланировать запуск прикладного кода в заранее определенный момент времени, будь то для отправки обновлений заказчикам или выполнения служебных операций, оказывается очень удобной для разработчиков.

В каркасе Spring поддерживается планирование заданий, удовлетворяющее требованиям в большинстве типичных случаев. Задание может быть запланировано на запуск через заданный промежуток времени или с помощью выражения для утилиты `cron` в Unix.

С другой стороны, для планирования и выполнения заданий Spring интегрируется с другими библиотеками, предназначенными для планирования заданий. Например, в среде сервера приложений каркас Spring может делегировать выполнение задания библиотеке CommonJ, применяемой на многих серверах приложений. Для планирования заданий в Spring поддерживается также ряд известных библиотек с открытым кодом, в том числе JDK Timer API и Quartz. Поддержка планирования заданий в Spring обсуждается в главе 11.

Поддержка динамических языков сценариев

В версии JDK 6 в Java появилась поддержка динамических языков, которая позволяет запускать сценарии, написанные на других языках, в среде виртуальной машины JVM. К числу таких языков относятся Groovy, JRuby, BeanShell и JavaScript.

В каркасе Spring также поддерживается выполнение динамических сценариев в приложениях, построенных на основе Spring. Кроме того, можно определить компонент Spring Bean, написанный на языке динамических сценариев, и внедрить его в нужные компоненты JavaBeans. Более подробно поддержка языков динамических сценариев в Spring рассматривается в главе 14.

Упрощенная обработка исключений

Областью, в которой Spring действительно помогает сократить объем повторяющегося рутинного кода, является обработка исключений. Основой философии Spring в этом отношении служит тот факт, что проверяемые исключения используются в Java чрезмерно, и каркас не должен принуждать к перехвату любого исключения, после которого вряд ли будет возможность произвести восстановление, и мы полностью разделяем такую точку зрения.

В действительности многие каркасы спроектированы таким образом, чтобы сократить потребность в написании кода для обработки проверяемых исключений. Но в большинстве таких каркасов принято придерживаться проверяемых исключений и в то же время искусственно снижать степень детализации в иерархии классов исключений. В этом отношении каркас Spring примечателен тем, что иерархия классов исключений в нем оказывается весьма детализированной из-за тех удобств, которые разработчикам доставляют непроверяемые исключения. В этой книге приведено немало примеров, в которых демонстрируются механизмы обработки исключений в Spring, способные сократить объем необходимого кода и в то же время расширить возможности выявления, классификации и диагностики ошибок в приложениях.

Проект Spring

Одной из самых привлекательных особенностей проекта Spring является уровень активности, наблюдаемой в сообществе разработчиков, и степень взаимодействия Spring с другими проектами, в том числе CGLIB, Apache Geronimo и AspectJ. Наиболее расхваливаемое преимущество открытого кода связано с тем, что даже если проект будет неожиданно свернут, у вас все равно останется исходный код. Но, откровенно говоря, вы вряд ли захотите остаться наедине с кодовой базой масштаба Spring, чтобы поддерживать и совершенствовать ее. В связи с этим отрадно признать, что проект Spring вполне упрочился, а сообщество разработчиков сохраняет высокую активность.

Происхождение Spring

Как уже отмечалось в этой главе, истоки Spring уходят корнями в книгу *Expert One-to-One J2EE Design and Development*, в которой Род Джонсон представил каркас под названием Interface 21 Framework, разработанный им для применения в своих приложениях. После выпуска в свет открытого кода этот каркас послужил основанием для известного нам теперь каркаса Spring Framework. Каркас Spring быстро

прошел через стадии бета-тестирования и предвыпускной версии, а первый его официальный выпуск 1.0 появился 24 марта 2004 года. С тех пор каркас Spring заметно разросся, и на момент написания этой книги имелась его версия Spring Framework 5.0.

Сообщество разработчиков Spring

Сообщество разработчиков Spring — одно из лучших среди всех проектов с открытым кодом, с которыми нам приходилось иметь дело. Списки рассылки и форумы всегда активны, а скорость внедрения новых средств, как правило, высока. Команда разработчиков действительно сосредоточена на том, чтобы сделать Spring самым успешным каркасом для построения приложений на Java, и об этом можно судить по качеству производимого ими кода. Как уже упоминалось, Spring также получает преимущества от тесных взаимоотношений с другими проектами с открытым кодом, и этот факт чрезвычайно важен, если полагаться в значительной степени на полный дистрибутив Spring.

Самыми выдающимися особенностями Spring с точки зрения пользователя, пожалуй, являются превосходная документация и тестовый набор, входящие в дистрибутив. В документации описаны практически все возможности каркаса Spring, и это упрощает его освоение новыми пользователями. Тестовый набор Spring всеобъемлющ, поскольку команда разработчиков пишет тесты для любого аспекта. Когда они обнаруживают ошибку, то исправляют ее, сначала написав тест, который выявляет ошибку, а затем обеспечив его прохождение.

Исправление ошибок и создание новых функциональных средств не ограничивается одной лишь командой разработчиков! Вы можете внести свой посильный вклад в написание исходного кода, отреагировав на запросы в любом перечне проектов Spring через официальные хранилища GitHub (`http://github.com/spring-projects`). Кроме того, можно поставить насущные вопросы и отслеживать их решение с помощью системы JIRA, официально существующей для проекта Spring (`https://jira.springsource.org/secure/Dashboard.jspa`). Что же все это означает для вас? Попросту говоря, это означает, что вы можете быть уверены в высоком качестве исходного кода Spring Framework, а также в том, что в обозримом будущем команда разработчиков Spring продолжит совершенствовать и без того замечательный каркас.

Комплект Spring Tool Suite

С целью упростить процесс разработки основанных на Spring приложений в IDE Eclipse в рамках главного проекта Spring был создан специальный проект Spring IDE. Вскоре после этого компания SpringSource, которую Род Джонсон основал в рамках проекта Spring, создала интегрированное инструментальное средство Spring Tool Suite (STS), доступное для загрузки по адресу `https://spring.io/tools`. И хотя это средство раньше было платным продуктом, теперь оно доступно бесплатно.

Инструментальное средство STS интегрирует IDE Eclipse, Spring IDE, Mylyn (задачно-ориентированную среду разработки в Eclipse), Maven for Eclipse, AspectJ Development Tools и много других полезных подключаемых модулей Eclipse в единый пакет. В каждой новой версии появляются дополнительные функциональные средства, в том числе поддержка языка сценариев Groovy, графический редактор конфигурирования в Spring, инструментальные средства визуальной разработки для проектов вроде Spring Batch и Spring Integration, а также поддержка сервера приложений Pivotal tc Server.

На заметку Компания SpringSource была приобретена компанией VMWare и вошла в состав корпорации Pivotal Software, Inc.

Помимо комплекта на основе Java, имеется комплект Groovy/Grails Tool Suite, обладающий аналогичными возможностями, но ориентированный на разработку прикладных программ на Groovy и Grails (`https://spring.io/tools`).

Проект Spring Security

Проект Spring Security (`http://projects.spring.io/spring-security/`), ранее называвшийся Acegi Security System for Spring, является еще одним важным проектом в рамках главного проекта Spring. Проект Spring Security обеспечивает всеобъемлющую поддержку безопасности как на уровне всего веб-приложения в целом, так и на уровне отдельных методов. Он тесно интегрирован с каркасом Spring Framework и другими распространенными механизмами аутентификации, в том числе с элементарной аутентификацией по сетевому протоколу HTTP, регистрацией с помощью форм, сертификатами X.509 и продуктами с единой регистрацией (SSO), например CA SiteMinder. Проект Spring Security обеспечивает управление доступом к ресурсам приложения на основе ролей, а в приложениях с более сложными требованиями к безопасности (например, с разделением данных) поддерживается ведение списка управления доступом (ACL). Тем не менее проект Spring Security применяется главным образом в защищенных веб-приложениях, как обсуждается в главе 16.

Проект Spring Boot

Настройка приложения на нормальную работу — дело хлопотное. Для этого придется создать файлы конфигурации, установить и настроить дополнительные инструментальные средства (вроде сервера приложений). Проект Spring Boot (`http://projects.spring.io/spring-boot/`) упрощает создание самостоятельных приложений промышленного уровня на основе Spring, которые остается лишь запустить на выполнение. В состав Spring Boot входят готовые конфигурации для различных типов приложений Spring, организованные в *стартовые* пакеты. Например, пакет `web-starter` содержит предварительно сконфигурированный и легко настраивае-

мый контекст веб-приложений, а также обеспечивает стандартную поддержку встраиваемых контейнеров сервлетов Tomcat 7+, Jetty 8+ и Undertow 1.3.

Кроме того, Spring Boot внедряет все зависимости, требующиеся в приложении Spring, принимая во внимание совместимость версий. На момент написания данной книги имелась версия Spring Boot 2.0.0.RELEASE. Более подробно Spring Boot рассматривается в главе 4. Альтернативная конфигурация проекта Spring и большинства проектов, упоминаемых в последующих главах, опираются на Spring Boot, поскольку это упрощает и ускоряет процессы разработки и тестирования.

Проекты Spring Batch и Spring Integration

Нечего и говорить, что выполнение пакетных заданий и интеграция являются распространенными примерами требований к приложениям. Чтобы удовлетворить подобные требования и упростить задачу разработчикам приложений, в рамках главного проекта Spring были созданы специальные проекты Spring Batch и Spring Integration. В частности, проект Spring Batch предоставляет общую структуру и разнообразные правила для реализации пакетных заданий, значительно сокращая объем стереотипного кода. Реализуя проектные шаблоны интеграции корпоративных приложений (Enterprise Integration Patterns — EIP), проект Spring Integration помогает упростить интеграцию приложений Spring с внешними системами. Подробнее об этом проекте речь пойдет в главе 18.

Другие проекты

Выше были вкратце описаны основные модули Spring и некоторые важные проекты в рамках главного проекта Spring, но имеется немало других проектов, предназначенных для удовлетворения различных требований сообщества разработчиков. В качестве примера можно упомянуть проекты Spring XD, Spring for Android, Spring Mobile, Spring Social и Spring AMQP. За дополнительными сведениями о них обращайтесь на веб-сайт Spring by Pivotal (`https://spring.io/projects`).

Альтернативы Spring

Учитывая предыдущие замечания относительно количества проектов с открытым кодом, не должен вызывать удивление тот факт, что Spring — далеко не единственный каркас, предлагающий средства для внедрения зависимостей или полноценные комплексные решения для построения веб-приложений. На самом деле проектов настолько много, что их сложно даже упомянуть вскользь. В духе открытости мы приводим здесь краткое описание нескольких подобных каркасов, но убеждены, что ни один из них не предлагает настолько исчерпывающее решение, как это делает Spring.

JBoss Seam Framework

Seam Framework (`www.seamframework.org`) основан Гэвином Кингом (Gavin King), создателем библиотеки Hibernate ORM, и представляет собой еще один полнофункциональный каркас, ориентированный на внедрение зависимостей. Он поддерживает разработку пользовательских интерфейсов веб-приложений (JSF), уровень бизнес-логики (EJB 3) и прикладной интерфейс JPA для сохраняемости данных. Основное отличие каркаса Seam Framework от Spring состоит в том, что он построен полностью на стандартах JEE. Сервер приложений JBoss также способствовал донесению идей Seam Framework до организации JCP и появлению спецификации JSR-299 (Contexts and Dependency Injection for the Java EE Platform — Контексты и внедрение зависимостей для платформы Java EE).

Google Guice

Следующим распространенным каркасом для внедрения зависимостей является Google Guice (`http://code.google.com/p/google-guice`). Поддерживаемый поисковым гигантом Google, Guice представляет собой облегченный каркас, предназначенный для внедрения зависимостей при управлении конфигурацией приложений. Он также стал эталонной реализацией спецификации JSR-330 (Dependency Injection for Java — Внедрение зависимостей для Java).

PicoContainer

PicoContainer (`http://picocontainer.com`) — это очень компактный контейнер для внедрения зависимостей в приложения, который не вносит никаких других зависимостей, кроме PicoContainer. А поскольку PicoContainer является всего лишь контейнером для внедрения зависимостей, то по мере роста приложения придется внедрить другой каркас (например, Spring). В таком случае лучше применять Spring с самого начала. Но если требуется лишь небольшой контейнер для внедрения зависимостей, то PicoContainer окажется подходящим вариантом выбора. А поскольку Spring упаковывает контейнер для внедрения зависимостей отдельно от остальной части каркаса, этим обстоятельством можно выгодно воспользоваться, сохранив гибкость на будущее.

Контейнер JEE 7

Как упоминалось ранее, принцип внедрения зависимостей получил широкое распространение и также реализован в организации JCP. Если приложение разрабатывается для серверов приложений, совместимых с платформой JEE 7 (JSR-342)[7], то стандартные методики внедрения зависимостей можно применять на всех уровнях.

---

[7] В сентябре 2017 г. была выпущена платформа JEE 8 (JSR-366), подробнее см. `https://jcp.org/en/jsr/detail?id=366`.

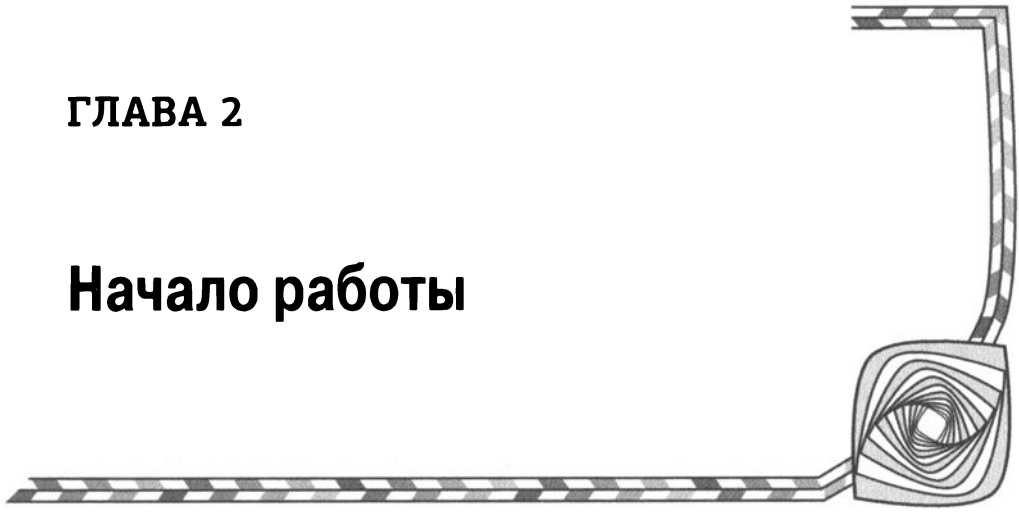
Резюме

В этой главе был сделан общий обзор каркаса Spring Framework с обсуждением всех его основных функциональных возможностей, а также указаны те главы, где эти возможности рассматриваются более подробно. Прочитав данную главу, вы должны ясно представлять, что может предложить каркас Spring, и тогда вам останется лишь выяснить, *как* это делается.

В следующей главе мы выясним, что требуется для построения и запуска простого приложения Spring. В ней будет показано, как получить Spring Framework, а также рассмотрены варианты упаковки, тестовый набор и документация. Кроме того, будет представлен пример простого прикладного кода Spring для вывода освященного временем сообщения "Hello World!", реализованный в контексте внедрения зависимостей.

ГЛАВА 2

Начало работы



Зачастую при освоении нового инструментального средства разработки сложнее всего выяснить, с чего следует начать. Как правило, положение усугубляется, если инструментальное средство предоставляет слишком много вариантов выбора, как это делает Spring. Правда, приступить к работе с Spring не так уж и трудно, если знать, где и что искать в первую очередь. В этой главе поясняется, с чего следует начать. В частности, здесь будут рассмотрены следующие вопросы.

- **Получение Spring.** Первый логический шаг состоит в получении и сборке архивных JAR-файлов Spring. Если вы хотите сделать все быстро, воспользуйтесь фрагментами кода для управления зависимостями в своей системе сборки, руководствуясь примерами, предоставленными по адресу `http://projects.spring.io/spring-framework`. Но если вы стремитесь оказаться на переднем крае разработки средствами Spring, поищите последнюю версию исходного кода Spring в хранилище GitHub[1].
- **Варианты упаковки Spring.** Упаковка Spring является модульной; разрешается выбирать те компоненты, которые должны использоваться в приложении, а при распространении готового приложения включать в его состав только эти компоненты. Каркас Spring состоит из многих модулей, но вам понадобится только их подмножество, которое зависит от конкретных потребностей приложения. У каждого модуля имеется свой скомпилированный двоичный код в архивном JAR-файле вместе с документацией, автоматически составленной утилитой Javadoc, а также исходными архивными JAR-файлами.

[1] Хранилище GitHub исходного кода Spring находится по адресу `http://github.com/spring-projects/spring-framework`.

- **Руководства по Spring.** На веб-сайте Spring имеется раздел Guides (Руководства), доступный по адресу `https://spring.io/guides`. Под руководствами понимаются краткие практические инструкции по составлению начального примера для решения любой задачи разработки средствами Spring. В этих руководствах отражены также последние выпуски проектов и технологий Spring, а следовательно, в ваше распоряжение предоставлены наиболее актуальные примеры.
- **Тестовый комплект и документация.** К предметам особой гордости членов сообщества разработчиков Spring относится всеобъемлющий тестовый набор и комплект документации. Тестированию отводится львиная доля работы в команде разработчиков. Комплект документации, входящий в стандартный дистрибутив, также составлен превосходно.
- **Пример приложения "Hello World!" в Spring.** Как бы то ни было, мы считаем, что начинать работу с любым новым инструментальным средством для программирования лучше всего с написания какого-нибудь кода. Поэтому мы представим простой пример полноценной реализации, основанной на внедрении зависимостей, хорошо известного приложения "Hello World!". Не отчаивайтесь, если вы не сразу поймете весь его исходный код, поскольку исчерпывающие пояснения будут приведены далее в книге.

Если вы уже знакомы с основами Spring Framework, можете переходить непосредственно к главе 3, где рассматривается реализация принципов инверсии управления и внедрения зависимостей в Spring. Но, даже зная основы Spring, вы наверняка найдете в этой главе интересные сведения, особенно касающиеся упаковки и зависимостей.

Получение Spring Framework

Прежде чем приступить к разработке средствами Spring, необходимо получить код самого каркаса. Для этого имеются две возможности: воспользоваться своей системой сборки для установки требующихся модулей или извлечь и построить код из хранилища Spring в GitHub. Применение инструментального средства для управления зависимостями (например, Maven или Gradle) зачастую оказывается самым простым подходом, поскольку для этого достаточно объявить зависимость в файле конфигурации и дать инструментальному средству возможность автоматически получить требующиеся библиотеки.

На заметку Если у вас имеется подключение к Интернету и вы пользуетесь инструментальным средством сборки вроде Maven или Gradle вместе с такой IDE, как Eclipse или IntelliJ IDEA, то загрузите документацию в формате Javadoc и библиотеки автоматически, чтобы иметь к ним доступ в процессе разработки. При переходе на новые версии в файлах конфигурации сборки эти библиотеки и документирующие комментарии будут также обновлены в процессе сборки проекта.

Быстрое начало

Посетите веб-страницу проекта Spring Framework[2], чтобы получить фрагмент кода управления зависимостями для системы сборки, который позволит включить в ваш проект последнюю версию RELEASE каркаса Spring. Можете также воспользоваться промежуточными моментальными снимками предстоящих выпусков или предыдущих версий данного каркаса.

Если применяется модуль Spring Boot, то указывать требующуюся версию Spring не нужно, поскольку этот модуль предоставляет файлы "стартовой" объектной модели проекта (POM) с целью упростить конфигурацию Maven и выбираемую по умолчанию стартовую конфигурацию Gradle. Следует лишь иметь в виду, что в версиях Spring Boot, предшествующих версии 2.0.0.RELEASE, используются версии Spring 4.*x*.

Извлечение Spring из хранилища GitHub

Если вы хотите иметь доступ к новым функциональным средствам Spring еще до того, как они будут отражены в моментальных снимках, можете извлечь исходный код непосредственно из хранилища GitHub в Pivotal. Для получения последней версии исходного кода Spring установите сначала систему контроля версий Git, которую можно загрузить по адресу `http://git-scm.com/`, а затем откройте окно командной строки или терминальной оболочки и введите следующую команду:

```
git clone git://github.com/spring-projects/spring-framework.git
```

В корневой папке проекта находится файл `README.md` с подробным описанием требований и процесса сборки каркаса Spring из исходного кода.

Выбор подходящего комплекта JDK

Каркас Spring Framework построен на Java, а это означает, что для его применения необходимо иметь возможность выполнять приложения Java на своем компьютере. Для этого необходимо установить Java. Когда речь заходит о разработке приложений на Java, разработчики обычно употребляют следующие термины и их сокращения.

- **Виртуальная машина Java (JVM)** — это абстрактная машина. По ее спецификации предоставляется среда выполнения, в которой исполняется байт-код Java.
- **Среда выполнения Java (Java Runtime Environment — JRE).** Предоставляет окружение для исполнения байт-кода Java и является физически существующей реализацией виртуальной машины JVM. Состоит из ряда библиотек и прочих файлов, применяемых в виртуальной машине JVM во время выполнения. Корпорация Oracle приобрела компанию Sun Microsystems в 2010 году и с тех пор активно предоставляет новые версии и обновления Java. Другие компании,

[2] См. `http://projects.spring.io/spring-framework`.

в том числе IBM, предоставляют собственные реализации виртуальной машины JVM.

- **Комплект разработки прикладных программ на Java (Java Development Kit — JDK).** Содержит среду JRE, документацию, а также инструментальные средства Java. Именно этот комплект устанавливают разработчики на своих машинах. В интегрированной среде разработки (IDE) вроде IntelliJ IDEA или Eclipse требуется указывать место установки комплекта JDK, чтобы загружать из него классы и документацию в процессе разработки.

Если вы пользуетесь инструментальным средством сборки вроде Maven или Gradle (исходный код примеров из этой книги организован в многомодульный проект Gradle), для него потребуется также виртуальная машина JVM. Оба инструментальных средства сборки Maven и Gradle сами являются проектами, построенными на Java.

На момент написания данной книги последней устойчивой версией Java считалась версия Java 8, хотя в конце сентября 2017 года была выпущена Java 9. Комплект JDK можно загрузить по адресу `https://www.oracle.com/`. По умолчанию он будет установлен в каком-нибудь стандартном месте на вашем компьютере, хотя это зависит от конкретной операционной системы. Если вы желаете пользоваться Maven или Gradle в режиме командной строки, вам придется определить переменные окружения для комплекта JDK и инструментального средства сборки Maven или Gradle, указав путь к их исполняемым файлам в системе. Инструкции, поясняющие, как это сделать, находятся на официальном веб-сайте каждого продукта, а также приведены в приложении к данной книге.

В главе 1 был приведен перечень версий Spring и требующиеся версии JDK. В этой книге рассматривается версия Spring 5.0.*x*. Исходный код примеров, представленных в книге, написан с использованием синтаксиса Java 8, поэтому вам понадобится версия JDK 8, чтобы скомпилировать и выполнить исходный код этих примеров.

Упаковка Spring

Модули Spring просто являются архивными JAR-файлами, в которых упакован код, требующийся для конкретного модуля. Уяснив назначение каждого модуля, вы сможете выбрать модули для вашего проекта и включить их в свой код.

В версии 5.0.0.RELEASE каркас Spring состоит из 21 модуля, упакованного в 21 архивный JAR-файл. Эти JAR-файлы и соответствующие им модули перечислены в табл. 2.1. В действительности имена архивных JAR-файлов представлены в форме, подобной следующей: `springaop-5.0.0.RELEASE.jar`, но ради простоты в табл. 2.1 приводится только та их часть, которая характерна для данного модуля (например, `aop`).

Таблица 2.1. Модули Spring

Модуль	Описание
`aop`	Содержит все классы, требующиеся для применения в приложении средств АОП из Spring. Этот архивный JAR-файл должен быть также включен в приложение, если планируется пользоваться другими средствами Spring, в которых применяется АОП (например, декларативным управлением транзакциями). Кроме того, в этот модуль упакованы классы, поддерживающие интеграцию с библиотекой AspectJ
`aspects`	Содержит все классы, предназначенные для расширенной интеграции с библиотекой AspectJ для АОП. Он понадобится, например, в том случае, если вы применяете классы Java для своей конфигурации Spring и нуждаетесь в управлении транзакциями с помощью аннотаций в стиле AspectJ
`beans`	Содержит все классы, поддерживающие манипулирование компонентами Spring Beans. Большинство классов из этого модуля поддерживают реализацию фабрики компонентов Spring Beans. Так, в этот модуль упакованы классы, требующиеся для обработки XML-файла конфигурации Spring и аннотаций Java
`beans-groovy`	Содержит классы Groovy, поддерживающие манипулирование компонентами Spring Beans
`context`	Содержит классы, которые предоставляют многие расширения для ядра Spring. Все классы должны использовать интерфейс `ApplicationContext` из Spring, описанный в главе 5, а также классы для интеграции с EJB, Java Naming and Directory Interface (JNDI) и Java Management Extensions (JMX). В этом модуле содержатся также классы Spring для удаленного взаимодействия, классы для интеграции с языками динамических сценариев (например, JRuby, Groovy, BeanShell), классы из прикладного интерфейса API по спецификации JSR-303 (Beans Validation), классы для планирования и выполнения заданий и т.д.
`context-indexer`	Содержит реализацию индексатора, предоставляющую доступ к подходящим компонентам, определенным в файле `META-INF/spring.components`. Базовый класс `CandidateComponentsIndex` не предназначен для внешнего применения
`context-support`	Содержит дополнительные расширения для модуля `spring-context`. На стороне пользовательского интерфейса имеются классы для поддержки электронной почты и интеграции с такими шаблонизаторами, как Velocity, FreeMarker и JasperReports. В этом модуле также упакованы классы для интеграции с различными библиотеками выполнения и планирования заданий, в том числе CommonJ и Quartz

Продолжение табл. 2.1

Модуль	Описание
`core`	Основной модуль, требующийся для каждого приложения Spring. В его архивном JAR-файле находятся классы, общие для всех остальных модулей Spring (например, классы для доступа к файлам конфигурации). Здесь можно также найти ряд исключительно полезных служебных классов, которые применяются во всей кодовой базе Spring и которые можно употреблять в своих приложениях
`expression`	Содержит все классы для поддержки SpEL (Spring Expression Language — язык выражений Spring)
`instrument`	В этот модуль входит агент инструментального оснащения Spring для начальной загрузки виртуальной машины JVM. Его архивный JAR-файл непременно потребуется в приложении Spring для привязывания во время загрузки с помощью библиотеки AspectJ
`jdbc`	В этот модуль входят все классы, предназначенные для поддержки JDBC. Он необходим для всех приложений, которым требуется доступ к базам данных. В этот модуль упакованы классы для поддержки источников данных, типов данных JDBC, шаблонов JDBC, платформенно-ориентированных подключений JDBC и т.д.
`jms`	В этот модуль входят все классы, предназначенные для поддержки системы JMS
`messaging`	Содержит ключевые абстракции, заимствованные из проекта Spring Integration и служащие основанием для приложений, ориентированных обмен сообщениями. Он внедряет поддержку сообщений по протоколу STOMP
`orm`	Расширяет стандартный набор функциональных средств JDBC в Spring, поддерживая распространенные инструментальные средства ORM, в том числе Hibernate, JDO, JPA, а также преобразователь данных iBATIS. Многие классы из архивного JAR-файла этого модуля зависят от классов, содержащихся в архивном JAR-файле модуля **`spring-jdbc`**, поэтому его следует включать в свои приложения
`oxm`	Обеспечивает поддержку OXM (Object/XML Mapping — взаимное преобразование объектов и данных формата XML). В этот модуль упакованы классы, предназначенные для абстрагирования маршализации и демаршализации данных формата XML, а также для поддержки таких распространенных инструментальных средств, как Castor, JAXB, XMLBeans и XStream
`test`	Как упоминалось ранее, в Spring предоставляется ряд имитирующих классов, оказывающих помощь в тестировании приложений. Многие из этих классов используются в тестовом наборе Spring, а следовательно, они хорошо проверены и значительно упрощают тестирование разрабатываемых приложений. С одной стороны, в модульных

Окончание табл. 2.1

Модуль	Описание
	тестах веб-приложений интенсивно применяются имитирующие классы **`HttpServletRequest`** и **`HttpServletResponse`**. А с другой стороны, Spring обеспечивает тесную интеграцию со средой модульного тестирования JUnit, и в этом модуле предоставляются многие классы, поддерживающие разработку тестовых сценариев JUnit; например, класс **`SpringJUnit4ClassRunner`** предоставляет простой способ начальной загрузки контекста типа **`ApplicationContext`** в среду модульного тестирования
`tx`	Предоставляет все классы, предназначенные для поддержки инфраструктуры транзакций в Spring. Здесь можно найти классы из уровня абстракции транзакций, поддерживающие прикладной интерфейс Java Transaction API (JTA), а также интеграцию с серверами приложений от ведущих производителей
`web`	Содержит основные классы для применения Spring в веб-приложениях, в том числе классы для автоматической загрузки контекста типа **`ApplicationContext`**, классы для поддержки выгрузки файлов и ряд полезных классов для выполнения таких повторяющихся заданий, как извлечение целочисленных значений из строки запроса
`web-reactive`	Содержит базовые интерфейсы и классы для модели реактивного веб-программирования в Spring
`web-mvc`	Содержит все классы для собственного каркаса по проектному шаблону MVC в Spring. А если применяется отдельный каркас по шаблону MVC, то классы из архивного JAR-файла этого модуля не требуются. Более подробно модуль Spring MVC рассматривается в главе 16
`websocket`	Обеспечивает поддержку прикладного интерфейса Java API для протокола WebSocket (JSR-356)

Выбор модулей для приложения

Без инструментального средства управления зависимостями, подобного Maven или Gradle, выбор модулей для применения в разрабатываемом приложении может оказаться затруднительным. Так, если требуется лишь фабрика компонентов Spring Beans и поддержка внедрения зависимостей, то все равно потребуются такие модули, как `spring-core`, `spring-beans`, `spring-context` и `spring-aop`. Если же требуется поддержка веб-приложений Spring, то придется добавить модуль `spring-web` и т.д. Благодаря таким функциональным возможностям инструментальных средств сборки, как поддержка транзитивных зависимостей в Maven, все обязательные сторонние библиотеки будут включены в разрабатываемое приложение автоматически.

Доступ к модулям Spring в хранилище Maven

Проект Maven[3], основанный организацией Apache Software Foundation, стал одним из самых распространенных инструментальных средств управления зависимостями для приложений на Java, которые охватывают как среды с открытым кодом, так и корпоративные среды. Это весьма эффективное средство для сборки, упаковки и управления зависимостями приложений, поддерживающее полный цикл сборки приложения, начиная с обработки ресурсов и компиляции и кончая тестированием и упаковкой. Кроме того, существует большое разнообразие модулей, подключаемых к Maven для решения разных задач, включая обновление баз данных и развертывание упакованного приложения на конкретном сервере (например, Tomcat, Jboss или WebSphere). На момент написания этой книги текущей была версия Maven 3.3.9.

Практически во всех проектах с открытым кодом поддерживается распространение их библиотек через хранилище Maven. Наиболее распространено хранилище Maven Central, размещаемое на сервере Apache Software Foundation. А на веб-сайте Maven Central[4] можно осуществлять поиск артефактов и получать сведения о них. После загрузки и установки Maven в среде разработки хранилище Maven Central становится доступным автоматически. Ряд других сообществ разработчиков открытого кода (например, JBoss и Spring от компании Pivotal) также предоставляют своим пользователям доступ к своим хранилищам Maven. Но для доступа к таким хранилищам их придется добавить в файл настроек Maven или файл POM (Project Object Model — объектная модель проекта) своего проекта.

Подробное обсуждение Maven выходит за рамки данной книги, но вы можете всегда обратиться к оперативно доступной документации или соответствующей литературе, чтобы получить дополнительную информацию. Тем не менее здесь стоит хотя бы вкратце упомянуть структуру упаковки разрабатываемого проекта в хранилище Maven по причине столь широкого распространения Maven.

С каждым артефактом Maven связаны идентификатор группы, идентификатор артефакта, тип упаковки и версия. Например, для артефакта log4j идентификатором группы является `log4j`, идентификатором артефакта — `log4j`, а типом упаковки — `jar`. Далее следует номер версии. Так, для версии 1.2.17 файл артефакта будет иметь имя `log4j-1.2.17.jar` и располагаться в папке для конкретного идентификатора группы, идентификатора артефакта и версии. Файлы конфигурации Maven составлены в формате XML и должны соблюдать стандартный синтаксис, определенный в схеме, доступной по адресу `http://maven.apache.org/xsd/maven-4.0.0.xsd`. По умолчанию файлу конфигурации Maven для разрабатываемого проекта присваивается имя `om.xml`, а ниже приведено его примерное содержимое.

```
<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
```

[3] См. `http://maven.apache.org`.

[4] См. `http://search.maven.org`.

```
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.apress.prospring5.ch02</groupId>
  <artifactId>hello-world</artifactId>
  <packaging>jar</packaging>
  <version>5.0-SNAPSHOT</version>
  <name>hello-world</name>
  <properties>
    <project.build.sourceEncoding>UTF-8
    </project.build.sourceEncoding>
    <spring.version>5.0.0.RELEASE</spring.version>
  </properties>
  <dependencies>
    <!-- https://mvnrepository.com/artifact/log4j/log4j -->
    <dependency>
       <groupId>log4j</groupId>
       <artifactId>log4j</artifactId>
       <version>1.2.17</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
      ...
      </plugin>
    </plugins>
  </build>
</project>
```

Кроме того, в Maven определяется стандартная структура типичного проекта, как показано на рис. 2.1.

Рис. 2.1. Структура типичного проекта в Maven

В каталоге `main/java` находятся основные классы, а в каталоге `main/resources` — файлы конфигурации приложения. В каталоге `test/java` находятся тестовые классы, а в каталоге `test/resources` — файлы конфигурации для тестирования конкретного приложения из каталога `main`.

Доступ к модулям Spring из Gradle

Стандартная структура проекта в Maven, а также разделение и организация артефактов по категориям очень важны, поскольку в Gradle соблюдаются те же самые правила и даже используется центральное хранилище Maven для извлечения артефактов. Gradle является весьма эффективным инструментальным средством сборки, для конфигурации которого ради простоты и гибкости вместо громоздкой XML-разметки применяется синтаксис языка Groovy. На момент написания данной книги текущей была версия Gradle 4.0[5]. Начиная с версии Spring 4.*x*, разработчики перешли на Gradle для конфигурирования каждого продукта Spring. Именно поэтому исходный код примеров из этой книги может быть построен и выполнен и средствами Gradle. По умолчанию файлу конфигурации Gradle для разрабатываемого проекта присваивается имя `pom.xml`, а ниже приведено его примерное содержимое.

```
group 'com.apress.prospring5.ch02'
version '5.0-SNAPSHOT'

apply plugin: 'java'

repositories {
  mavenCentral()
}

ext{
  springVersion = '5.0.0.RELEASE'
}

tasks.withType(JavaCompile) {
  options.encoding = "UTF-8"
}

dependencies {
  compile group: 'log4j', name: 'log4j', version: '1.2.17'
  ...
}
```

Как видите, содержимое такого файла конфигурации оказывается более удобочитаемым. Артефакты определяются в нем с помощью идентификаторов группы, артефакта и номера версии, как это было принято ранее в Maven, хотя имена свойств отличаются. Но поскольку рассмотрение Gradle выходит за рамки данной книги, на этом придется его завершить.

---

[5] Подробные сведения о загрузке, установке и конфигурировании Gradle для целей разработки можно найти на официальном веб-сайте данного проекта по адресу `https://gradle.org/install`.

Пользование документацией на Spring

Одна из отличительных черт Spring, которая делает этот каркас столь удобным для разработчиков, строящих реальные приложения, состоит в обилии грамотно и аккуратно написанной документации. В каждом выпуске Spring Framework команда, отвечающая за составление документации, старается довести всю документацию до состояния полной готовности, после чего она уточняется командой разработки. Это означает, что каждое функциональное средство Spring не только полностью документировано с помощью утилиты Javadoc, но и описано в справочном руководстве, включаемом в каждый выпуск. Если вы еще не знакомы с документацией в формате Javadoc и справочным руководством по Spring, то теперь самое время это сделать. Ведь эта книга не в состоянии заменить любой из упомянутых выше ресурсов и служит лишь дополняющим справочным пособием, где демонстрируются особенности построения приложений Spring с самого начала.

Внедрение Spring в приложение "Hello World!"

Надеемся, что к этому моменту вы уже осознали, что Spring является серьезным, хорошо поддерживаемым проектом, обладающим всем необходимым для того, чтобы стать отличным инструментальным средством для разработки приложений. Но мы пока еще не привели ни одного примера исходного кода. И вы, вероятно, с нетерпением ждете того момента, когда каркас Spring будет продемонстрирован в действии, а поскольку это невозможно сделать, не написав исходный код, то займемся этим вплотную. Не отчаивайтесь, если вы не сразу поймете приведенный далее исходный код; по ходу изложения материала будут предоставлены дополнительные пояснения.

Построение примера приложения "Hello World!"

Вы определенно должны быть знакомы с традиционным для программирования примером "Hello World!" (Здравствуй, мир), но на тот случай, если вы не в курсе дела, ниже приведен его исходный код на Java во всей своей красе.

```
package com.apress.prospring5.ch2;

public class HelloWorld {
  public static void main(String... args) {
    System.out.println("Hello World!");
  }
}
```

Этот пример очень прост: он делает свое дело, но совсем не пригоден для расширения. Что, если требуется изменить сообщение, выводимое на консоль? А что, если сообщение требуется выводить разными способами, например, в стандартный поток вывода ошибок вместо стандартного потока вывода данных или заключить его в дескрипторы HTML-разметки, а не представлять простым текстом?

Итак, переопределим требования к данному примеру приложения, указав, что в нем должен поддерживаться простой и гибкий механизм изменения выводимого сообщения, а также возможность легко изменить режим воспроизведения. В первоначальном примере приложения "Hello World!" оба изменения можно сделать быстро и легко, внеся соответствующие поправки в исходный код. Но более крупное приложение потребует больше времени на перекомпиляцию и повторное тестирование. Поэтому более удачное решение предусматривает вынесение содержимого сообщения наружу и его чтение во время выполнения — возможно, из аргументов командной строки, как демонстрируется в следующем фрагменте кода:

```
package com.apress.prospring5.ch2;

public class HelloWorldWithCommandLine {
  public static void main(String... args) {
    if (args.length > 0) {
      System.out.println(args[0]);
    } else {
      System.out.println("Hello World!");
    }
  }
}
```

В данном примере мы добились то, чего хотели: теперь можно изменять сообщение, не меняя исходный код. Но в рассматриваемом здесь приложении по-прежнему остается следующее затруднение: компонент, отвечающий за воспроизведение сообщения, отвечает также и за его получение. Изменение порядка получения выводимого сообщения, по существу, означает изменение кода воспроизведения. К этому следует добавить еще и тот факт, что мы по-прежнему не можем так просто сменить средство воспроизведения, поскольку для этого придется внести коррективы в класс, запускающий данное приложение на выполнение.

В порядке дальнейшего совершенствования рассматриваемого здесь приложения следует отметить, что лучшее решение предполагает реорганизацию кода с целью вынести логику воспроизведения и получения сообщений в отдельные компоненты. А для того чтобы сделать данное приложение действительно гибким, в этих компонентах должны быть реализованы интерфейсы, с помощью которых определяются взаимозависимости между компонентами и классом, запускающим данное приложение на выполнение. Реорганизовав код, реализующий логику получения сообщений, можно определить простой интерфейс `MessageProvider` с единственным методом `getMessage()`:

```
package com.apress.prospring5.ch2.decoupled;

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

Интерфейс `MessageRenderer` реализуется во всех компонентах, способных воспроизводить сообщения. И один из таких компонентов приведен в следующем фрагменте кода:

```
package com.apress.prospring5.ch2.decoupled;

public interface MessageRenderer {
  void render();
  void setMessageProvider(MessageProvider provider);
  MessageProvider getMessageProvider();
}
```

Как видите, в интерфейсе `MessageRenderer` определен метод `render()`, а также метод `setMessageProvider()` в стиле компонентов JavaBeans. Любые реализации интерфейса `MessageRenderer` отделены от получения сообщений и поручают эту обязанность интерфейсу `MessageProvider`, с которым они поставляются. Здесь интерфейс `MessageProvider` представляет собой зависимость от интерфейса `MessageRenderer`. Создать простые реализации этих интерфейсов совсем не трудно, как показано в следующем фрагменте кода:

```
package com.apress.prospring5.ch2.decoupled;

public class HelloWorldMessageProvider
       implements MessageProvider {
  @Override
  public String getMessage() {
    return "Hello World!";
  }
}
```

Как видите, мы создали простую реализацию интерфейса `MessageProvider`, всегда возвращающую в качестве сообщения символьную строку `"Hello World!"`. Создать класс `StandardOutMessageRenderer` для воспроизведения этой строки так же просто, как показано ниже.

```
package com.apress.prospring5.ch2.decoupled;

public class StandardOutMessageRenderer
       implements MessageRenderer {
  private MessageProvider messageProvider;

  @Override
  public void render() {
    if (messageProvider == null) {
      throw new RuntimeException("You must set the "
            + "property messageProvider of class:"
            + StandardOutMessageRenderer.class.getName());
            // Установите свойство messageProvider
            // в данном классе
```

```
    }
    System.out.println(messageProvider.getMessage());
  }

  @Override
  public void setMessageProvider(MessageProvider provider) {
    this.messageProvider = provider;
  }

  @Override
  public MessageProvider getMessageProvider() {
    return this.messageProvider;
  }
}
```

Теперь осталось лишь переписать метод `main()` в главном классе данного приложения:

```
package com.apress.prospring5.ch2.decoupled;

public class HelloWorldDecoupled {
  public static void main(String... args) {
    MessageRenderer mr = new StandardOutMessageRenderer();
    MessageProvider mp = new HelloWorldMessageProvider();
    mr.setMessageProvider(mp);
    mr.render();
  }
}
```

Абстрактная схема построенного до сих пор приложения приведена на рис. 2.2.

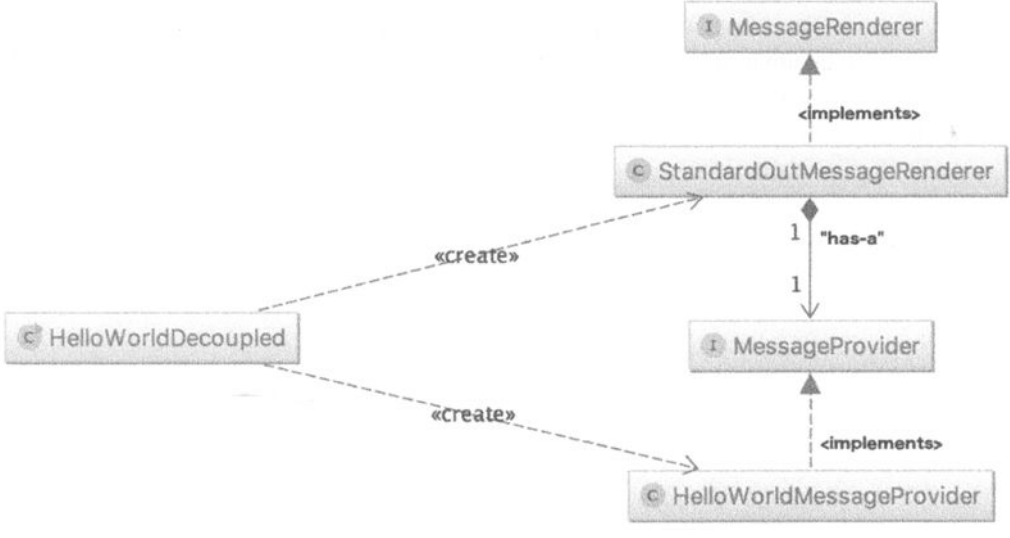


Рис. 2.2. Несколько более развязанное приложение "Hello World!"

Приведенный выше код довольно прост. Сначала в нем получаются экземпляры типа `HelloWorldMessageProvider` и `StandardOutMessageRenderer`, хотя объявленными оказываются типы `MessageProvider` и `MessageRenderer` соответственно. Дело в том, что взаимодействовать в этом коде требуется только с методами, предоставляемыми этими интерфейсами, а в классах `HelloWorldMessageProvider` и `StandardOutMessageRenderer` эти интерфейсы уже реализованы. Затем объект класса, реализующего интерфейс `MessageProvider`, передается экземпляру типа `MessageRenderer` и далее вызывается метод `MessageRenderer.render()`. Скомпилировав и запустив данное приложение на выполнение, мы получим вполне предсказуемый результат: вывод символьной строки `"Hello World!"` на консоль.

Теперь рассматриваемый здесь пример приложения больше похож на то, к чему мы стремимся, но осталось одно небольшое затруднение. Изменение реализации любого из интерфейсов `MessageRenderer` или `MessageProvider` означает изменение в исходном коде. В качестве выхода из этого затруднительного положения можно создать простой фабричный класс, в котором имена классов реализации читаются из файла свойств, а их экземпляры получаются от имени данного приложения:

```
package com.apress.prospring5.ch2.decoupled;
import java.util.Properties;

public class MessageSupportFactory {
  private static MessageSupportFactory instance;

  private Properties props;
  private MessageRenderer renderer;
  private MessageProvider provider;

  private MessageSupportFactory() {
    props = new Properties();

    try {
      props.load(this.getClass().getResourceAsStream(
                                  "/msf.properties"));
      String rendererClass = props.getProperty(
                                  "renderer.class");
      String providerClass = props.getProperty(
                                  "provider.class");

      renderer = (MessageRenderer)
               Class.forName(rendererClass).newInstance();
      provider = (MessageProvider)
               Class.forName(providerClass).newInstance();
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
```

```
  static {
    instance = new MessageSupportFactory();
  }

  public static MessageSupportFactory getInstance() {
    return instance;
  }

  public MessageRenderer getMessageRenderer() {
    return renderer;
  }

  public MessageProvider getMessageProvider() {
    return provider;
  }
}
```

Приведенная выше реализация элементарна и несколько наивна, обработка ошибок упрощена, а имя файла конфигурации жестко закодировано, но мы уже имеем значительный объем кода. Файл конфигурации этого фабричного класса очень прост:

```
renderer.class= com.apress.prospring5.ch2.decoupled
                .StandardOutMessageRenderer
provider.class= com.apress.prospring5.ch2.decoupled
                .HelloWorldMessageProvider
```

Чтобы воспользоваться приведенной выше реализацией, необходимо внести снова коррективы в метод `main()`, как показано ниже.

```
package com.apress.prospring5.ch2.decoupled;

public class HelloWorldDecoupledWithFactory {
  public static void main(String... args) {
    MessageRenderer mr = MessageSupportFactory
                .getInstance().getMessageRenderer();
    MessageProvider mp = MessageSupportFactory
                .getInstance().getMessageProvider();
    mr.setMessageProvider(mp);
    mr.render();
  }
}
```

Прежде чем перейти к внедрению Spring в данное приложение, напомним, что было сделано раньше. Начав с простого приложения "Hello World!", мы определили два дополнительных требования, которым должно удовлетворять данное приложение. Первое требование: изменение сообщения должно осуществляться просто, а второе требование: изменение механизма воспроизведения должно быть столь же простым. Чтобы удовлетворить этим требованиям, мы определили два интерфейса: `MessageProvider` и `MessageRenderer`. А для того чтобы получить сообщение для воспро-

изведения, интерфейс `MessageRenderer` полагается на реализацию интерфейса `MessageProvider`. И, наконец, мы определили простой фабричный класс для извлечения имен классов реализации и получения их экземпляров по мере необходимости.

Реорганизация кода средствами Spring

Продемонстрированный выше окончательный пример соответствует целям, намеченным для рассматриваемого здесь приложения, но и он не лишен недостатков. Первый состоит в том, что приходится писать немало связующего кода для соединения всех частей в единое приложение, в то же время сохраняя компоненты слабо связанными. Второй недостаток заключается в том, что мы все еще должны вручную предоставлять реализацию интерфейса `MessageRenderer` вместе с экземпляром реализации интерфейса `MessageProvider`. Оба эти недостатка можно устранить, применяя Spring.

Чтобы устранить недостаток, связанный со слишком большим объемом связующего кода, достаточно полностью удалить фабричный класс `MessageSupportFactory` из данного приложения и заменить его интерфейсом `ApplicationContext` из Spring, как показано ниже. В отношении этого интерфейса пока что достаточно знать, что он применяется в Spring для сохранения всей информации о среде, относящейся к приложению, которым управляет каркас Spring. Этот интерфейс расширяет другой интерфейс `ListableBeanFactory`, действующий в качестве поставщика для любого экземпляра компонентов Spring Beans.

```
package com.apress.prospring5.ch2;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support
          .ClassPathXmlApplicationContext;

public class HelloWorldSpringDI {
  public static void main(String args) {
    ApplicationContext ctx = new
               ClassPathXmlApplicationContext
               ("spring/app-context.xml");
    MessageRenderer mr =
           ctx.getBean("renderer",MessageRenderer.class);
    mr.render();
  }
}
```

Как следует из приведенного выше фрагмента кода, в теле метода `main()` сначала получается экземпляр класса `ClassPathXmlApplicationContext` (сведения о конфигурации данного приложения загружаются из файла `spring/app-context.xml`, находящегося по пути к классам текущего проекта), типизированный как `ApplicationContext`, а затем из этого экземпляра получаются экземпляры реализации интерфейса `MessageRenderer` с помощью метода `ApplicationContext`.

getBean(). Не обращайте пока что особого внимания на метод getBean(); достаточно знать, что он читает конфигурацию приложения (в данном случае — из XML-файла app-context.xml), инициализирует среду интерфейса ApplicationContext (по существу, контекст приложения Spring), а затем возвращает экземпляр сконфигурированного компонента Spring Bean[6]. Этот XML-файл служит тем же целям, что и аналогичный файл для фабричного класса MessageSupportFactory, как показано ниже.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation=
        "http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans
               /spring-beans.xsd">

  <bean id="provider"
      class="com.apress.prospring5.ch2.decoupled
                .HelloWorldMessageProvider"/>
  <bean id="renderer"
      class="com.apress.prospring5.ch2.decoupled
                .StandardOutMessageRenderer"
      p:messageProvider-ref="provider"/>
</beans>
```

Выше приведена типичная конфигурация контекста типа ApplicationContext в Spring. Сначала в ней объявляется пространство имен Spring, а им является стандартное пространство имен beans, которое служит для объявления управляемых компонентов Spring Beans, а также их требований к зависимостям. В данном примере свойство messageProvider компонента, реализующего средство воспроизведения (renderer), ссылается на компонент, реализующий поставщика (provider). Все эти зависимости должны быть разрешены и внедрены средствами Spring.

Далее объявляется компонент Spring Bean с идентификатором provider и соответствующий класс реализации. Обнаружив такое определение компонента Spring Bean во время инициализации интерфейса ApplicationContext, каркас Spring получает экземпляр заданного класса и сохраняет его с указанным идентификатором.

После этого объявляется компонент Spring Bean с идентификатором renderer и соответствующим классом реализации. Напомним, что при получении сообщения для воспроизведения этот компонент полагается на интерфейс MessageProvider. Чтобы известить Spring о таком требовании к внедрению зависимостей, в данном случае используется атрибут p разметки пространства имен. В частности, атрибут дескриптора p:messageProvider-ref="provider" извещает Spring, что свойство messageProvider одного компонента Spring Bean должно быть внедрено с по-

[6] См. http://search.maven.org.

мощью другого компонента. Внедряемый в это свойство компонент должен иметь идентификатор `provider`. Обнаружив это определение компонента Spring Bean, каркас Spring получает экземпляр заданного класса, находит в данном компоненте свойство `messageProvider` и внедряет его, используя экземпляр компонента с идентификатором `provider`.

Теперь, как видите, после инициализации контекста типа `ApplicationContext` в методе `main()` просто получается компонент Spring Bean типа `MessageRenderer`. С этой целью сначала вызывается типизированный метод `getBean()`, которому передается идентификатор и ожидаемый возвращаемый тип (в данном случае — интерфейса `MessageRenderer`), а затем метод `render()`. А каркас Spring создает реализацию интерфейса `MessageProvider` и внедряет ее в реализацию интерфейса `MessageRenderer`. Обратите внимание на то, что в данном случае никаких изменений не было внесено в классы, связанные вместе с помощью Spring. На самом деле эти классы не имеют никакого отношения к каркасу Spring и находятся в полном неведении относительно его существования. Хотя это не всегда так. Ваши классы могут реализовывать интерфейсы Spring, чтобы взаимодействовать с контейнером внедрения зависимостей самыми разными способами.

А теперь необходимо выяснить, каким образом действуют новая конфигурация Spring и модифицированный метод `main()`. С этой целью воспользуйтесь Gradle и введите приведенную ниже команду из командной строки, чтобы собрать проект и корневой каталог исходного кода.

```
gradle clean build copyDependencies
```

Единственным обязательным модулем Spring, который должен быть объявлен в файле конфигурации, является модуль `spring-context`. Gradle автоматически внедрит любые транзитивные зависимости, требующиеся для данного модуля. Транзитивные зависимости модуля `spring-context` приведены на рис. 2.3.

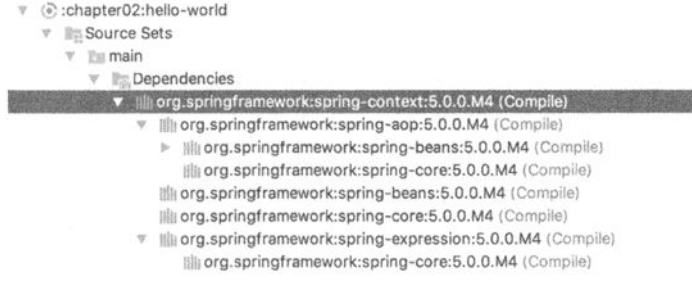


Рис. 2.3. Транзитивные зависимости модуля **`spring-context`**, отображаемые в IDE IntelliJ IDEA

По приведенной выше команде проект собирается заново. При этом удаляются сформированные ранее файлы, а все требующиеся зависимости копируются в то же место, где находится результирующий артефакт, т.е. по пути `build/libs`. Этот же

путь присоединяется в качестве префикса к именам библиотечных файлов, введенных в файл манифеста `MANIFEST.MF` при построении архивного JAR-файла. Если вы незнакомы с конфигурированием и процессом построения архивного JAR-файла средствами Gradle, обращайтесь за справкой к файлу `hello-world/build.gradle` свойств Gradle, находящемуся в папке `chapter02` исходного кода примеров, доступного для загрузки на веб-сайте издательства Apress по адресу, указанному в конце введения.

И, наконец, чтобы запустить пример реализации внедрения зависимостей в Spring, введите команды

```
cd build/libs; java -jar hello-world-5.0-SNAPSHOT.jar
```

В итоге вы должны увидеть несколько протокольных сообщений, формируемых в процессе начального запуска контейнера Spring, а после них — ожидаемый вывод сообщения `"Hello World!"`.

Конфигурирование Spring с помощью аннотаций

Начиная с версии Spring 3.0, XML-файлы конфигурации больше не требуются для разработки приложений в Spring. Их можно заменить аннотациями и конфигурационными классами. Последние являются классами Java, снабженными аннотацией `@Configuration` и содержащими определения компонентов Spring Beans, где методы снабжены аннотацией `@Bean`. Они могут быть сами сконфигурированы для обозначения определений компонентов Spring Beans в приложении с помощью аннотации `@ComponentScanning`. Ниже приведена конфигурация, равнозначная содержимому XML-файла конфигурации `app-context.xml`, представленному ранее в этой главе.

```
package com.apress.prospring5.ch2.annotated;

import com.apress.prospring5.ch2.decoupled
          .HelloWorldMessageProvider;
import com.apress.prospring5.ch2.decoupled.MessageProvider;
import com.apress.prospring5.ch2.decoupled.MessageRenderer;
import com.apress.prospring5.ch2.decoupled
          .StandardOutMessageRenderer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class HelloWorldConfiguration {

  // равнозначно разметке <bean id="provider" class=".."/>
  @Bean
  public MessageProvider provider() {
    return new HelloWorldMessageProvider();
  }
```

```
  // равнозначно разметке <bean id="renderer" class=".."/>
  @Bean
  public MessageRenderer renderer() {
    MessageRenderer renderer = new
                    StandardOutMessageRenderer();
    renderer.setMessageProvider(provider());
    return renderer;
  }
}
```

В таком случае в метод `main()` необходимо внести коррективы, заменив класс `ClassPathXmlApplicationContext` другим классом, реализующим интерфейс `ApplicationContext` и способным читать определения компонентов Spring Beans из конфигурационных классов. И таким заменителем является класс `AnnotationConfigApplicationContext`:

```
package com.apress.prospring5.ch2.annotated;

import com.apress.prospring5.ch2.decoupled.MessageRenderer;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;

public class HelloWorldSpringAnnotated {
  public static void main(String... args) {
    ApplicationContext ctx = new
              AnnotationConfigApplicationContext
             (HelloWorldConfiguration.class);
    MessageRenderer mr = ctx.getBean(
                    "renderer", MessageRenderer.class);
    mr.render();
  }
}
```

Это лишь один из вариантов конфигурирования с помощью аннотаций и конфигурационных классов. В отсутствие XML-разметки конфигурирование Spring становится довольно гибким процессом. Подробнее об этом речь пойдет далее, где основное внимание будет уделено конфигурированию на языке Java и с помощью аннотаций.

На заметку Некоторые интерфейсы и классы, определенные в примере приложения "Hello World", могут использоваться в последующих главах. И хотя в данном примере исходный код был приведен полностью, в других главах могут быть показаны менее полные версии исходного кода, чтобы соблюсти краткость особенно в тех случаях, когда в код вносятся постепенные коррективы. Исходный код примеров из этой книги был сделан более организованным. В частности, все классы, применяемые в последующих примерах, размещены в пакетах **`com.apress.prospring5.ch2.decoupled`** и **`com.apress.prospring5.ch2.annotated`**. Однако исходный код реального приложения должен быть организован соответствующим образом.

Резюме

В этой главе были представлены основные сведения, необходимые для подготовки и приведения в действие каркаса Spring. В ней было показано, как приступить к работе с каркасом Spring, используя системы управления зависимостями, и получить текущую разрабатываемую версию непосредственно из хранилища GitHub. Затем было описано, каким образом упакован каркас Spring, а также перечислены зависимости, требующиеся для его функциональных средств. Располагая этими сведениями, можно принимать обоснованные решения относительно того, какие архивные JAR-файлы Spring требуются для приложения и какие зависимости должны распространяться вместе с ним. Документация на Spring, руководства и тестовый набор служат пользователям Spring идеальным основанием, чтобы приступить к разработке приложений, поэтому часть этой главы была посвящена исследованию тех возможностей, которые становятся доступными благодаря Spring. И, наконец, в этой главе был рассмотрен пример внедрения зависимостей в Spring, где традиционное приложение "Hello World!" было превращено в слабо связанное и расширяемое приложение для воспроизведения сообщений.

Важно понимать, что в данной главе мы лишь слегка коснулись особенностей внедрения зависимостей в частности и каркаса Spring в целом. А в следующей главе мы подробно рассмотрим принципы инверсии управления и внедрения зависимостей в Spring.

ГЛАВА 3

Инверсия управления и внедрение зависимостей в Spring

В главе 2 были изложены основные принципы инверсии управления (IoC). С практической точки зрения внедрение зависимостей (DI) является особой формой инверсии управления, хотя нередко можно обнаружить, что оба эти понятия используются равноценно. В этой главе мы более подробно рассмотрим инверсию управления и внедрение зависимостей, формализовав отношения между этими двумя понятиями и продемонстрировав, каким образом Spring вписывается в общую картину.

Дав определение инверсии управления и внедрению зависимостей и их взаимоотношениям с Spring, мы рассмотрим понятия, существенные для реализации внедрения зависимостей в Spring. В этой главе излагаются лишь основы реализации внедрения зависимостей в Spring, а более развитые возможности внедрения зависимостей будут рассмотрены в главе 4. В частности, здесь будут рассмотрены следующие вопросы.

- **Принципы инверсии управления.** В этой части главы описываются разновидности инверсии управления, включая пассивную инверсию зависимостей, или внедрение зависимостей, и активную инверсию зависимостей, или поиск зависимостей. Здесь будут пояснены отличия в разных подходах к инверсии управления и представлены доводы за и против каждого из них.
- **Инверсия управления в Spring.** В этой части рассматриваются возможности инверсии управления, доступные в Spring, и показано, каким образом они реализованы. В частности, здесь будут описаны услуги по внедрению зависимостей, предоставляемые в Spring, включая внедрение через метод установки, конструктор и метод класса.
- **Внедрение зависимостей в Spring.** В этой части излагается реализация контейнера инверсии управления в Spring. Для определения компонентов Spring Beans и требований к внедрению зависимостей приложению предстоит взаи-

модействовать главным образом с интерфейсом `BeanFactory`. Но, за исключением нескольких начальных примеров, во всех остальных примерах исходного кода в этой главе основное внимание будет уделено применению интерфейса `ApplicationContext`, который расширяет интерфейс `BeanFactory` и обеспечивает намного больше возможностей. Отличия интерфейсов `BeanFactory` и `ApplicationContext` будут объяснены в последующих разделах настоящей главы.

- **Конфигурирование контекста приложения Spring.** Завершающая часть этой главы посвящена применению двух подходов к конфигурированию интерфейса `ApplicationContext`: с помощью XML-разметки и аннотаций. Конфигурирование средствами Groovy и Java более подробно обсуждается в главе 4. Сначала в этой части будет рассмотрено конфигурирование внедрения зависимостей, а затем описаны дополнительные услуги, предоставляемые интерфейсом `BeanFactory`, в том числе наследование, управление жизненным циклом и автосвязывание.

Инверсия управления и внедрение зависимостей

По существу, инверсия управления, а следовательно, и внедрение зависимостей направлены на то, чтобы предоставить простой механизм для снабжения компонента зависимостями (часто называемыми *взаимодействующими объектами*) и управления ими на протяжении всего их жизненного цикла. Компонент, которому требуются определенные зависимости, зачастую называется *зависимым объектом*, а в случае инверсии управления — *целевым объектом*. Вообще говоря, инверсия управления может быть разделена на два подтипа: внедрение зависимостей и поиск зависимостей. Эти подтипы подразделяются далее на конкретные реализации служб инверсии управления. Из этого определения ясно видно, что когда речь идет о внедрении зависимостей, всегда имеется в виду инверсия управления. Но когда речь идет об инверсии управления, то не всегда имеется в виду внедрение зависимостей. Например, поиск зависимостей — это также форма инверсии управления.

Типы инверсии управления

У вас может невольно возникнуть вопрос: почему существуют два типа инверсии управления, которые дополнительно разделены на разные реализации? По-видимому, ясный ответ на этот вопрос отсутствует. Безусловно, разные типы обеспечивают определенный уровень гибкости, но, на наш взгляд, инверсия управления скорее сочетает в себе старые и новые идеи, что отражают два ее основных типа.

Первый тип — поиск зависимостей — является намного более традиционным подходом и на первый взгляд выглядит более знакомым тем, кто программирует на

Java. Второй тип — внедрения зависимостей — в действительности обеспечивает более высокую гибкость и удобство применения по сравнению с поиском зависимостей, хотя поначалу он кажется нелогичным.

Если инверсия управления реализуется как поиск зависимостей, то компонент должен получить ссылку на зависимость, тогда как при внедрении зависимостей последние внедряются в компонент контейнером инверсии управления. У поиска зависимостей имеются две разновидности: извлечение зависимостей и контекстный поиск зависимостей (CDL). И у внедрения зависимостей имеются две разновидности: через конструктор и через метод установки.

На заметку При обсуждении механизмов инверсии управления в этом разделе нас не интересует, каким образом вымышленный контейнер инверсии управления узнает о самых разных зависимостях. Важно лишь то, что в определенный момент он выполняет действия, описанные для каждого механизма.

Извлечение зависимостей

Для разработчика приложений на Java извлечение зависимостей является самым узнаваемым типом инверсии управления. В этом случае зависимости извлекаются из реестра по мере необходимости. Всякий, кому приходилось писать код для доступа к каркасу EJB (вплоть до версии 2.1), пользовался извлечением зависимостей (через прикладной интерфейс JNDI API для поиска компонента EJB). На рис. 3.1 приведен характерный пример извлечения зависимостей через механизм поиска.

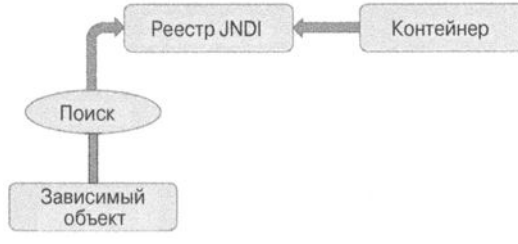


Рис. 3.1. Извлечение зависимостей через поиск в реестре JNDI

Извлечение зависимостей предоставляется и в каркасе Spring как механизм для извлечения компонентов, которыми он управляет, что и было продемонстрировано на практике в главе 2. В следующем фрагменте кода приведен типичный пример поиска с извлечением зависимостей в приложении, основанном на Spring:

```
package com.apress.prospring5.ch3;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support
          .ClassPathXmlApplicationContext;
```

```
public class DependencyPull {
  public static void main(String... args) {
    ApplicationContext ctx = new
              ClassPathXmlApplicationContext
              ("spring/app-context.xml");
    MessageRenderer mr = ctx.getBean(
                  "renderer", MessageRenderer.class);
    mr.render();
  }
}
```

Этот тип инверсии управления не только преобладает в приложениях на платформе JEE, где применяется каркас EJB вплоть до версии 2.1 и широко употребляются операции поиска в службе JNDI для получения зависимостей из реестра, но и играет ключевую роль в работе с Spring во многих средах.

Контекстный поиск зависимостей

В известной мере контекстный поиск зависимостей (CDL) подобен извлечению зависимостей, но в этом случае поиск осуществляется в контейнере, управляющем ресурсом, а не только в каком-то центральном реестре. Как правило, контекстный поиск зависимостей производится в установленной точке. Механизм контекстного поиска зависимостей наглядно показан на рис. 3.2.

Рис. 3.2. Контекстный поиск зависимостей

Механизм контекстного поиска приводится в действие через реализацию в компоненте интерфейса, аналогично приведенному в следующем фрагменте кода:

```
package com.apress.prospring5.ch3;

public interface ManagedComponent {
  void performLookup(Container container);
}
```

Реализуя этот интерфейс, компонент извещает контейнер, что ему требуется получить зависимость. Контейнер обычно предоставляется базовым сервером приложений (например, Tomcat или JBoss) или каркасом (в частности, Spring). В следующем

фрагменте кода приведен простой интерфейс `Container`, предоставляющий услуги поиска зависимостей:

```
package com.apress.prospring5.ch3;

public interface Container {
  Object getDependency(String key);
}
```

Как только контейнер будет готов передать зависимости компоненту, он вызовет метод `performLookup()` по очереди для каждого компонента. И тогда компонент сможет искать свои зависимости, используя интерфейс `Container`, как показано в следующем фрагменте кода:

```
package com.apress.prospring5.ch3;

public class ContextualizedDependencyLookup
       implements ManagedComponent {
  private Dependency dependency;

  @Override
  public void performLookup(Container container) {
    this.dependency = (Dependency)
         container.getDependency("myDependency");
  }

  @Override
  public String toString() {
    return dependency.toString();
   }
}
```

Внедрение зависимостей через конструктор

Внедрение зависимостей через конструктор происходит в том случае, когда зависимости предоставляются компоненту в его конструкторе (или нескольких конструкторах). С этой целью в компоненте объявляется один или ряд конструкторов, получающих в качестве аргументов его зависимости, а контейнер инверсии управления передает зависимости компоненту при получении его экземпляра, как показано в приведенном ниже фрагменте кода. Очевидно, что вследствие внедрения зависимостей через конструктор объект не может быть создан без зависимостей, а следовательно, они обязательны.

```
package com.apress.prospring5.ch3;

public class ConstructorInjection {
  private Dependency dependency;
  public ConstructorInjection(Dependency dependency) {
```

```
    this.dependency = dependency;
  }
  @Override
  public String toString() {
    return dependency.toString();
  }
}
```

Внедрение зависимостей через метод установки

При внедрении зависимостей через метод установки контейнер инверсии управления внедряет зависимости компонента через методы установки в стиле компонентов JavaBeans. Методы установки компонента отражают зависимости, которыми может управлять контейнер инверсии управления. В приведенном ниже фрагменте кода показан типичный компонент, основанный на внедрении зависимостей через метод установки. Очевидно, что вследствие внедрения зависимостей через метод установки объект может быть создан без зависимостей, которые могут быть предоставлены в дальнейшем через вызов метода установки.

```
package com.apress.prospring5.ch3;

public class SetterInjection {
  private Dependency dependency;

  public void setDependency(Dependency dependency) {
    this.dependency = dependency;
  }

  @Override
  public String toString() {
    return dependency.toString();
  }
}
```

Требование зависимости, предоставляемое через метод `setDependency()`, обозначается в контейнере по имени в стиле компонентов JavaBeans (в данном случае — `dependency`). На практике внедрение зависимостей через метод установки оказывается наиболее широко применяемым механизмом инверсии управления, который проще всего реализовать.

На заметку В каркасе Spring поддерживается еще один тип внедрения зависимостей, называемый *внедрением зависимостей через поле*, но о нем речь пойдет далее в этой главе, когда будет рассматриваться автосвязывание с помощью аннотации **`@Autowired`**.

Выбор между внедрением и поиском зависимостей

Выбрать конкретный тип инверсии управления для применения — внедрение или поиск зависимостей — обычно не трудно. Как правило, выбор типа инверсии управления зависит от применяемого контейнера. Так, если применяется каркас EJB (вплоть до версии 2.1), то следует выбрать инверсию управления в стиле поиска (через службу JNDI), чтобы получить доступ к EJB из контейнера JEE. А в Spring компоненты и их зависимости всегда связываются вместе с помощью инверсии управления в стиле внедрения, за исключением первоначальных поисков компонентов Spring Beans.

На заметку Применяя Spring, можно получить доступ к ресурсам EJB, и не прибегая к явному поиску. Каркас Spring может действовать в роли адаптера между системами инверсии управления в стиле поиска и внедрения, позволяя таким образом управлять всеми ресурсами с помощью внедрения.

Вопрос на самом деле заключается в следующем: когда есть выбор, то какой тип инверсии управления должен использоваться: внедрение или поиск зависимостей? Очевидно, что внедрение. Если обратиться к приведенным выше примерам исходного кода, то можно заметить, что внедрение зависимостей не оказывает никакого влияния на исходный код компонентов. С другой стороны, код, реализующий извлечение зависимостей, должен активно получать ссылку на реестр и взаимодействовать с ним при получении зависимостей, а для применения контекстного поиска зависимостей требуется, чтобы в классах был реализован конкретный интерфейс и поиск зависимостей вручную. Если же выбрать внедрение зависимостей, то в классах достаточно разрешить их внедрение через конструкторы или же через методы установки.

Выбрав внедрение зависимостей, можно пользоваться своими классами полностью отдельно от контейнера инверсии управления, который поставляет зависимые объекты и взаимодействующие с ними объекты вручную, тогда как при поиске зависимостей классы всегда будут зависеть от классов и интерфейсов, определяемых в контейнере инверсии управления. Еще один недостаток поиска зависимостей состоит в том, что он сильно затрудняет тестирование классов отдельно от контейнера. А при внедрении зависимостей тестирование компонентов не составит особого труда, поскольку для этого достаточно предоставить зависимости с помощью подходящего конструктора или метода установки.

На заметку Более подробно тестирование с помощью Spring и внедрения зависимостей обсуждается в главе 13.

Решения на основе поиска зависимостей неизбежно оказываются более сложными, чем решения, основанные на внедрении зависимостей. И хотя такой сложности не следует бояться, мы ставим под сомнение обоснованность излишнего усложнения

процесса в целом, поскольку считаем этот аспект не менее важным, чем управление зависимостями в приложении.

Если оставить все эти второстепенные причины в стороне, то главной причиной для выбора внедрения, а не поиска зависимостей является значительное упрощение задачи разработки. Если выбрать внедрение зависимостей, то придется написать намного меньше кода, который окажется несложным и, как правило, генерируемым автоматически в хорошей IDE. Обратите внимание на то, что весь код в примерах внедрения зависимостей является пассивным в том смысле, что он не пытается активно выполнить какую-то задачу. Но самое примечательное в таком коде состоит в том, что получаемые объекты хранятся только в полях, а для извлечения зависимости из любого реестра или контейнера никакого другого кода не требуется. Таким образом, код оказывается намного более простым и менее подверженным ошибкам. Пассивный код легче сопровождать, чем активный, поскольку в нем мало что может пойти не так. Рассмотрим следующий фрагмент кода, взятый из приведенного выше примера контекстного поиска зависимостей:

```
public void performLookup(Container container) {
  this.dependency = (Dependency)
       container.getDependency("myDependency");
}
```

В этом коде многое может пойти не так, как было задумано: ключ зависимости может измениться, экземпляр контейнера может оказаться пустым (`null`), а возвращаемая зависимость — относиться к неподходящему типу. Мы привели этот код потому, что он содержит большое количество подвижных частей, и в нем может быть многое нарушено. Внедрение зависимостей способствует развязке компонентов в приложении, но в то же время усложняет дополнительный код, который требуется для того, чтобы связать эти компоненты вместе и решить любые полезные задачи.

Выбор между внедрением зависимостей через конструктор и метод установки

Теперь, когда мы прояснили, какой механизм инверсии управления следует предпочесть, осталось выбрать применяемую разновидность внедрения зависимостей: через конструктор или метод установки. Внедрение зависимостей через конструктор особенно удобно в том случае, если экземпляр класса зависимости должен существовать перед применением компонента. Многие контейнеры, включая и Spring, предоставляют механизм, позволяющий убедиться, все ли зависимости определены, когда они внедряются через метод установки. Но если зависимости внедряются через конструктор, то их требование задается независимо от контейнера. Кроме того, внедрение зависимостей через конструктор помогает в применении неизменяемых объектов.

Внедрение зависимостей через метод установки полезно в самых разных случаях. Если компонент предоставляет свои зависимости контейнеру, но готов обеспечить для них свои стандартные настройки, то достичь этого лучше всего, внедрив зависи-

мости через метод установки. Еще одно достоинство внедрения зависимостей через метод установки состоит в том, что оно позволяет объявлять зависимости в интерфейсе, хотя это и не так удобно, как может показаться на первый взгляд. Допустим, имеется типичный бизнес-интерфейс с одним методом `defineMeaningOfLife()`. Если помимо этого метода определить метод установки `setEncyclopedia()` для внедрения зависимостей, то все реализации данного интерфейса будут обязаны использовать или хотя бы учитывать наличие зависимости от энциклопедии. Но определять метод `setEncyclopedia()` в интерфейсе предметной области совсем не обязательно. Вместо этого данный метод можно определить в классах, реализующих интерфейс предметной области. При таком подходе к программированию все современные контейнеры инверсии управления, в том числе и Spring, могут взаимодействовать с компонентом через интерфейс предметной области, но по-прежнему предоставлять зависимости реализованного класса. Несколько прояснить это положение поможет конкретный пример. Рассмотрим интерфейс предметной области, приведенный в следующем фрагменте кода:

```
package com.apress.prospring5.ch3;

public interface Oracle {
  String defineMeaningOfLife();
}
```

Обратите внимание, что в этом интерфейсе предметной области не определено никаких методов установки для внедрения зависимостей. Реализовать этот интерфейс можно так, как показано ниже.

```
package com.apress.prospring5.ch3;

public class BookwormOracle implements Oracle {
  private Encyclopedia encyclopedia;

  public void setEncyclopedia(Encyclopedia encyclopedia) {
    this.encyclopedia = encyclopedia;
  }

  @Override
  public String defineMeaningOfLife() {
    return "Encyclopedias are a waste of money - "
         + "go see the world instead";
  }
}
```

Как видите, в классе `BookwormOracle` не только реализуется интерфейс `Oracle`, но и определяется метод установки для внедрения зависимостей. Каркас Spring очень удобен для работы со структурой подобного рода, и нет никакой необходимости определять зависимости от интерфейса предметной области. Возможность использовать интерфейсы для определения зависимостей является часто пропагандируемым

преимуществом внедрения зависимостей через метод установки, но на самом деле следует стараться применять методы установки исключительно для внедрения из своих интерфейсов. Если нет абсолютной уверенности, что все реализации отдельного интерфейса предметной области требуют конкретной зависимости, то в каждом классе реализации лучше определить его собственные зависимости, а в интерфейсе предметной области оставить одни только методы предметной области.

Несмотря на то что методы установки для внедрения зависимостей совсем не обязательно размещать в интерфейсе предметной области, определение в нем методов установки и получения параметров конфигурации считается удачным решением, которое повышает ценность внедрения зависимостей через метод установки как инструментального средства программирования. Мы рассматриваем параметры конфигурации как особый случай зависимостей. Безусловно, компоненты зависят от параметров конфигурации, но параметры конфигурации (или конфигурационные данные) заметно отличаются от рассмотренных до сих пор типов зависимостей. Эти отличия мы обсудим далее, а до тех пор рассмотрим приведенный ниже интерфейс предметной области.

```
package com.apress.prospring5.ch3;

public interface NewsletterSender {
  void setSmtpServer(String smtpServer);
  String getSmtpServer();
  void setFromAddress(String fromAddress);
  String getFromAddress();
  void send();
}
```

Интерфейс `NewsletterSender` реализуется в тех классах, где по электронной почте отправляются информационные бюллетени. В этом интерфейсе определен единственный метод `send()` предметной области, но обратите внимание на определение в нем двух свойств компонентов JavaBeans. Зачем это было сделано? Ведь только что утверждалось, что не следует определять зависимости в интерфейсе предметной области. Дело в том, что эти свойства, определяющие адрес SMTP-сервера и адрес электронной почты, по которому отправляются сообщения, не являются зависимостями в практическом смысле. Они представляют собой конфигурационные данные, оказывающие влияние на функционирование всех реализаций интерфейса `NewsletterSender`. И в связи с этим возникает вопрос: чем параметр конфигурации отличается от любого другого вида зависимости? Как правило, выяснить, следует ли отнести зависимость к разряду параметров конфигурации, совсем не трудно. Но если вы не уверены, то обратите внимание на следующие характеристики, явно указывающие на параметр конфигурации.

- **Параметры конфигурации пассивны.** В приведенном выше интерфейсе `NewsletterSender` параметр настройки SMTP-сервера может служить примером пассивной зависимости. Пассивные зависимости не используются на-

прямую для выполнения конкретного действия. Напротив, они применяются внутренним образом или другими зависимостями для выполнения своих действий. В примере интерфейса `MessageRenderer` из главы 2 зависимость от интерфейса `MessageProvider` не была пассивной, поскольку она выполняла функцию, которая требовалась интерфейсу `MessageRenderer` для завершения своей задачи.

- **Параметры конфигурации, как правило, относятся к данным, а не к компонентам.** Это означает, что параметр конфигурации представляет собой определенную порцию информации, которая необходима компоненту для завершения своей работы. Очевидно, что параметр конфигурации SMTP-сервера — это порция информации, которая требуется интерфейсу `NewsletterSender`, тогда как интерфейс `MessageProvider` — это на самом деле другой компонент, который необходим для правильного функционирования интерфейса `MessageRenderer`.
- **Параметры конфигурации обычно представляют простые значения или их коллекции.** В действительности эта характеристика вытекает из первых двух характеристик, но параметры конфигурации обычно представляют простые значения. В языке Java это означает, что они относятся к примитивному типу данных (или соответствующего класса-оболочки), к типу String или к коллекции подобных значений. Простые значения, как правило, пассивны. Иными словами, со значениями типа `String` мало что можно сделать, кроме манипулирования данными, которые они представляют. Практически всегда такие значения используются в информационных целях. Например, значение типа `int` может представлять номер порта для приема запросов через сокет сетевого соединения, а значение типа `String` — SMTP-сервер, через который будут отправлять почтовые сообщения.

Принимая решение, следует ли определять параметры конфигурации в интерфейсе предметной области, необходимо также выяснить, распространяется ли конкретный параметр конфигурации на все реализации этого интерфейса предметной области или же только на одну из них. Так, в примере с интерфейсом `NewsletterSender` очевидно, что во всех его реализациях должно быть известно, какой именно SMTP-сервер следует использовать для отправки почтовых сообщений. Но такой параметр конфигурации, как признак защищенной отправки сообщений, не имеет смысла включать в интерфейс предметной области, потому что он поддерживается не во всех прикладных интерфейсах API для электронной почты. Поэтому справедливо предположить, что многие реализации вообще не будут принимать во внимание такой параметр.

На заметку Напомним, что в главе 2 было решено определять зависимости на уровне бизнес-логики. Это было сделано лишь в целях демонстрации и ни в коем случае не должно рассматриваться как рекомендуемая норма практики.

Внедрение зависимостей через метод установки позволяет также оперативно сменять зависимости для различных реализаций, не создавая нового экземпляра родительского компонента. Это становится возможным благодаря поддержке технологии JMX в Spring. Пожалуй, самое значительное преимущество внедрения зависимостей через метод установки заключается в том, что оно является наименее навязчивым из всех существующих механизмов внедрения зависимостей.

Вообще говоря, выбирать тип внедрения зависимостей следует исходя из своего варианта использования. Внедрение зависимостей через метод установки позволяет менять местами зависимости, не создавая новые объекты, а также разрешает классу выбирать подходящие стандартные настройки, не прибегая к явному внедрению объекта. А внедрение зависимостей через конструктор окажется удачным выбором в том случае, когда требуется гарантировать передачу зависимостей компоненту и когда проектирование выполняется для неизменяемых объектов. Следует, однако, иметь в виду, что если внедрение зависимостей через конструктор гарантирует предоставление компоненту всех зависимостей, то аналогичные возможности обеспечивает и соответствующий механизм, предоставляемый в большинсте контейнеров, хотя и за счет дополнительных издержек на привязку прикладного кода к каркасу.

Инверсия управления в Spring

Как упоминалось ранее, к инверсии управления относится бо́льшая часть того, что делается в Spring. Ядро реализации каркаса Spring основано на внедрения зависимостей, хотя в нем обеспечиваются также возможности для поиска зависимостей. Каркас Spring автоматически предоставляет взаимодействующие объекты зависимому объекту, используя внедрение зависимостей. В приложении, основанном на Spring, всегда предпочтительнее применять внедрение зависимостей для передачи взаимодействующих объектов зависимым объектам, а не заставлять зависимые объекты получать взаимодействующие объекты через поиск зависимостей. Механизм внедрения зависимостей в Spring наглядно показан на рис. 3.3.

Рис. 3.3. Механизм внедрения зависимостей в Spring

Несмотря на то что внедрение зависимостей является предпочтительным механизмом связывания вместе взаимодействующих и зависимых объектов, для доступа к зависимым объектам понадобится и поиск зависимостей. Во многих средах Spring не может автоматически связать все компоненты приложения с помощью внедрения зависимостей, поэтому для доступа к первоначальному набору компонентов придется прибегнуть к поиску зависимостей. Например, в автономных приложениях на Java необходимо выполнить начальную загрузку контейнера Spring в методе `main()` и получить зависимости (через интерфейс `ApplicationContext`) для программной

обработки. Но при построении веб-приложений с поддержкой проектного шаблона MVC в Spring этого можно избежать благодаря автоматическому связыванию всего приложения. Пользоваться внедрением зависимостей вместе с платформой Spring следует при всякой возможности. А в крайнем случае можно обратиться к возможностям поиска зависимостей. Далее в этой главе будут представлены примеры обоих механизмов в действии, и мы укажем на них при первом же упоминании.

Контейнер инверсии управления в Spring примечателен тем, что он может выполнять функцию адаптера между его собственным контейнером внедрения зависимостей и внешними контейнерами поиска зависимостей. Данная функция рассматривается далее в этой главе. В каркасе Spring поддерживается внедрение зависимостей как через конструктор, так и через метод установки, подкрепляя стандартный набор средств инверсии управления рядом полезных дополнений и упрощая разработку в целом.

В оставшейся части главы представлены основы контейнера внедрения зависимостей в Spring, дополняемые многими наглядными примерами.

Внедрение зависимостей в Spring

Поддержка внедрения зависимостей в Spring является всеобъемлющей и, как будет показано в главе 4, выходит за рамки обсуждавшегося до сих пор стандартного набора средств инверсии управления. В оставшейся части этой главы рассматриваются основы контейнера внедрения зависимостей в Spring, в том числе внедрение зависимостей через конструктор, метод установки и метод класса, наряду с подробным описанием порядка конфигурирования внедрения зависимостей в Spring.

Компоненты Spring Beans и их фабрики

Ядром контейнера внедрения зависимостей в Spring служит интерфейс `BeanFactory`, который отвечает за управление компонентами Spring Beans, в том числе их зависимостями и жизненными циклами. Термин *компонент Spring Bean* употребляется в Spring для обозначения любого компонента, управляемого контейнером. Как правило, компоненты Spring Beans придерживаются на определенном уровне спецификации компонентов JavaBeans, хотя это совсем не обязательно, особенно если для связывания компонентов Spring Beans друг с другом предполагается применять внедрение зависимостей через конструктор.

Если в приложении требуется лишь поддержка внедрения зависимостей, то с контейнером внедрения зависимостей в Spring можно взаимодействовать через интерфейс `BeanFactory`. В этом случае в приложении необходимо создать экземпляр класса, реализующего интерфейс `BeanFactory`, и сконфигурировать его на основании сведений о компонентах Spring Beans и зависимостях. Как только это будет сделано, компоненты Spring Beans могут быть доступны в приложении через интерфейс `BeanFactory` для последующей обработки.

В ряде случаев вся подобного рода настройка производится автоматически (например, в веб-приложении экземпляр типа `ApplicationContext` будет загружаться веб-контейнером во время начальной загрузки приложения с помощью класса `ContextLoaderListener`, предоставляемого в Spring и объявленного в дескрипторном файле `web.xml`). Но зачастую программировать настройку приходится вручную. Во всех примерах, приведенных далее в этой главе, требуется ручная настройка реализации интерфейса `BeanFactory`.

Несмотря на то что интерфейс `BeanFactory` можно сконфигурировать программно, более распространено внешнее конфигурирование с помощью определенного рода файла конфигурации. Внутренне конфигурация компонентов Spring Beans представлена экземплярами классов, реализующих интерфейс `BeanDefinition`. В конфигурации компонента Spring Bean хранятся сведения не только о самом компоненте, но и о тех компонентах Spring Beans, от которых он зависит. Для любых классов реализации интерфейса `BeanFactory`, в которых также реализуется интерфейс `BeanDefinitionReader`, данные типа `BeanDefinition` можно прочитать из файла конфигурации, используя классы `PropertiesBeanDefinitionReader` или `XmlBeanDefinitionReader`. В частности, класс `PropertiesBeanDefinitionReader` читает определение компонента Spring Bean из файла свойств, а класс `XmlBeanDefinitionReader` — из XML-файла.

Итак, компоненты Spring Beans можно идентифицировать в интерфейсе `BeanFactory`, и каждому компоненту Spring Bean может быть назначен идентификатор, имя или то и другое. Экземпляр компонента Spring Bean можно получить и без идентификатора или имени (это так называемый *анонимный* компонент Spring Bean) или как один компонент внутри другого компонента Spring Bean. У каждого компонента Spring Bean имеется по крайней мере одно имя, но их может быть сколько угодно, причем дополнительные имена разделяются запятыми. Все имена после первого считаются псевдонимами того же самого компонента Spring Bean. Чтобы извлечь компоненты Spring Beans из интерфейса `BeanFactory` и установить отношения зависимости, когда, например, компонент X зависит от компонента Y, можно воспользоваться их идентификаторами или именами.

Реализации интерфейса `BeanFactory`

Описание интерфейса `BeanFactory` может показаться на первый взгляд слишком сложным, но на самом деле это не так. Обратимся к простому примеру. Допустим, имеется следующая реализация данного интерфейса, в которой имитируется оракул, вещающий о смысле жизни:

```
// интерфейс
package com.apress.prospring5.ch3;

public interface Oracle {
  String defineMeaningOfLife();
}
```

```
// реализация
package com.apress.prospring5.ch3;

public class BookwormOracle implements Oracle {
  private Encyclopedia encyclopedia;

  public void setEncyclopedia(Encyclopedia encyclopedia) {
    this.encyclopedia = encyclopedia;
  }

  @Override
  public String defineMeaningOfLife() {
    return "Encyclopedias are a waste of money - go "
         + "see the world instead";
  }
}
```

А теперь посмотрим, как в автономной программе на Java можно инициализировать интерфейс `BeanFactory` и получить компонент `oracle` для обработки. Ниже приведен соответствующий исходный код.

```
package com.apress.prospring5.ch3;

import org.springframework.beans.factory.support
          .DefaultListableBeanFactory;
import org.springframework.beans.factory.xml
          .XmlBeanDefinitionReader;
import org.springframework.core.io.ClassPathResource;

public class XmlConfigWithBeanFactory {
  public static void main(String... args) {
    DefaultListableBeanFactory factory =
           new DefaultListableBeanFactory();
    XmlBeanDefinitionReader rdr =
           new XmlBeanDefinitionReader(factory);
    rdr.loadBeanDefinitions(new ClassPathResource(
           "spring/xml-bean-factory-config.xml"));
    Oracle oracle = (Oracle) factory.getBean("oracle");
    System.out.println(oracle.defineMeaningOfLife());
  }
}
```

В приведенном выше примере кода используется класс `DefaultListableBeanFactory` в качестве одной из двух основных реализаций интерфейса `BeanFactory`, предоставляемых в Spring, а данные типа `BeanDefinition` читаются из XML-файла средствами класса `XmlBeanDefinitionReader`. Как только реализация интерфейса `BeanFactory` будет создана и сконфигурирована, компонент Spring Bean извлекается по его имени `oracle`, указанному в XML-файле конфигурации:

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
      "http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans
            /spring-beans.xsd">

<bean id="oracle"
  name="wiseworm"
  class="com.apress.prospring5.ch3.BookwormOracle"/>
</beans>
```

На заметку Не включайте номер версии в объявления конкретного местоположения XSD-файла в Spring. Каркас Spring уже поддерживает его распознавание как XSD-файла с конкретной версией, сконфигурированного по указателю в файле **`spring.schemas`**. Этот файл находится в модуле **`spring-beans`**, определенном как зависимость в вашем проекте. Вам также не придется модифицировать все файлы компонентов Spring Beans при переходе на новую версию Spring.

В приведенном выше файле объявлен компонент Spring Bean с назначенным идентификатором `oracle` и именем `wiseworm`, а также указан базовый класс реализации `com.apress.prospring4.ch3.BookwormOracle`. Не отчаивайтесь, если вам не все понятно в этой конфигурации, — мы обсудим ее подробно в последующих разделах. Определив такую конфигурацию, запустите приложение из предыдущего примера кода, чтобы увидеть на консоли текст, возвращаемый методом `defineMeaningOfLife()`.

Помимо класса `XmlBeanDefinitionReader`, в Spring предоставляется класс `PropertiesBeanDefinitionReader`, который позволяет управлять конфигурацией компонентов Spring Beans с помощью свойств, а не XML-разметки. И хотя свойства идеально подходят для мелких и простых приложений, они могут быстро стать неудобными, когда придется иметь дело с большим количеством компонентов Spring Beans. Именно поэтому лучше применять XML-формат для конфигурации всех приложений, кроме самых простых.

Разумеется, вы вольны объявлять собственные реализации интерфейса `BeanFactory`, но помните, что для этого придется немало потрудиться. Чтобы добиться такого же уровня функционирования, как и в готовых реализациях интерфейса `BeanFactory`, вам придется реализовать намного больше интерфейсов, чем только `BeanFactory`. Если же требуется лишь определить новый механизм конфигурации, создайте собственное средство чтения определений, разработав класс, расширяющий класс `DefaultListableBeanFactory` и реализующий интерфейс `BeanFactory`.

Интерфейс ApplicationContext

Интерфейс ApplicationContext служит расширением интерфейса BeanFactory в Spring. Помимо услуг по внедрению зависимостей, интерфейс ApplicationContext предоставляет такие услуги, как транзакции и АОП, источник сообщений для интернационализации (i18n), обработка событий в приложениях и пр.

При разработке приложений, основанных на Spring, рекомендуется взаимодействовать с Spring через интерфейс ApplicationContext. Начальная загрузка интерфейса ApplicationContext поддерживается в Spring посредством ручного программирования (получения экземпляра вручную и загрузки подходящей конфигурации) или в среде веб-контейнера через класс ContextLoaderListener. Во всех приведенных далее примерах исходного кода будет использоваться интерфейс ApplicationContext.

Конфигурирование интерфейса ApplicationContext

Итак, рассмотрев основные принципы инверсии управления и внедрения зависимостей, а также представив простой пример применения интерфейса BeanFactory в Spring, перейдем непосредственно к вопросам конфигурирования приложений Spring. В частности, уделим основное внимание интерфейсу ApplicationContext, который предоставляет намного больше возможностей для конфигурирования, чем традиционный интерфейс BeanFactory.

Способы конфигурирования приложений Spring

Прежде чем перейти непосредственно к конфигурированию интерфейса ApplicationContext, выясним, какие имеются способы для определения конфигурации приложений в Spring. Первоначально определение компонентов Spring Beans поддерживалось через свойства или через XML-файл. После выпуска версии JDK 5 и появления в Spring, начиная с версии 2.5, поддержки аннотаций Java последние можно также применять при конфигурировании интерфейса ApplicationContext.

Так что же лучше — формат XML или аннотации? На эту тему велась горячая полемика, часть которой можно найти в Интернете[1]. Определенного ответа на этот вопрос не существует, поскольку каждому из этих способов присущи свои достоинства и недостатки. Применение XML-файла позволяет вынести всю конфигурацию за пределы кода Java, тогда как аннотации дают разработчику возможность определять и видеть настройку внедрения зависимостей в самом коде. В каркасе Spring допускается также сочетать оба эти способа в одном интерфейсе ApplicationContext. Зачастую инфраструктура приложения (например, источника данных, диспетчера

[1] Для примера почитайте дискуссии на форумах сообщества разработчиков Spring по адресу http://forum.spring.io.

транзакций, фабрики подключений службы JMS или JMX) определяется в XML-файле, а конфигурация внедрения зависимостей (внедряемые компоненты Spring Beans и их зависимости) — в аннотациях. Но какой бы способ ни был выбран, его следует строго придерживаться, доведя это правило до сведения всех членов команды разработчиков. Следование избранному способу конфигурирования и согласованное его применение во всем приложении Spring намного упростит последующие действия по его разработке и сопровождению.

Чтобы облегчить понимание обоих способов конфигурирования приложений Spring, мы постараемся привести примеры кода с XML-разметкой и аннотациями рядом. Но в данной книге основной акцент делается на аннотациях и конфигурировании на языке Java, поскольку способ XML-разметки уже рассматривался в предыдущих ее изданиях.

Краткое описание простой конфигурации

Для конфигурирования с помощью разметки в формате XML необходимо объявить обязательное для приложения базовое пространство имен, предоставляемое в Spring. Ниже приведен простой пример XML-файла, в котором объявлено только пространство имен `beans` для определения компонентов Spring Beans. В последующих примерах делается ссылка `app-context-xml.xml` на этот XML-файл для конфигурирования с помощью XML-разметки:

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:c="http://www.springframework.org/schema/c"
       xsi:schemaLocation=
          "http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans
            /spring-beans.xsd">
</beans>
```

Кроме пространства имен `beans` в Spring предоставляется немало других пространств имен, предназначенных для разнообразных целей. К их числу относится пространство имен `context` для конфигурирования интерфейса `Application Context`, пространство имен `aop` для поддержки АОП и пространство имен `tx` для поддержки транзакций. Эти пространства имен описываются в соответствующих главах.

Чтобы воспользоваться поддержкой аннотаций в Spring при разработке приложения, необходимо объявить соответствующие дескрипторы в XML-файле конфигурации, как демонстрируется в приведенном ниже примере. В последующих примерах делается ссылка `app-context-xml.xml` на этот XML-файл для конфигурирования с помощью XML-разметки и с поддержкой аннотаций.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org
                           /schema/context"
       xmlns:c="http://www.springframework.org/schema/c"
       xsi:schemaLocation="http://www.springframework.org
                                /schema/beans
          http://www.springframework.org/schema/beans
               /spring-beans.xsd
          http://www.springframework.org/schema/context
          http://www.springframework.org/schema/context
               /spring-context.xsd">

  <context:component-scan
      base-package="com.apress.prospring5.ch3.annotation"/>
</beans>
```

Дескриптор `<context:component-scan>` сообщает Spring о необходимости просмотра исходного кода на предмет внедряемых компонентов Spring Beans, снабженных аннотациями `@Component`, `@Controller`, `@Repository` и `@Service`, а также поддерживающих аннотации `@Autowired` и `@Inject` в указанном пакете (и всех его подчиненных пакетах). В дескрипторе `<context:component-scan>` можно определить целый ряд пакетов, используя в качестве разделителя запятую, точку с запятой или пробел. А для более точного управления в этом дескрипторе поддерживается включение и исключение средств просмотра компонентов. Рассмотрим в качестве примера следующую конфигурацию:

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org
                           /schema/context"
       xmlns:c="http://www.springframework.org/schema/c"
       xsi:schemaLocation="http://www.springframework.org
                                /schema/beans
       http://www.springframework.org/schema/beans
            /spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context
            /spring-context.xsd">

  <context:component-scan
      base-package="com.apress.prospring5.ch3.annotation">
  <context:exclude-filter type="assignable"
      expression="com.example.NotAService"/>
```

```
  </context:component-scan>
</beans>
```

Приведенный выше дескриптор сообщает Spring о необходимости просмотра указанного пакета, но с пропуском классов, для которых был назначен тип, заданный в выражении (им может быть класс или интерфейс). Помимо исключающего фильтра, можно применять и включающий фильтр. В качестве критерия фильтрации в атрибуте `type` можно указать `annotation`, `regex`, `assignable`, `aspectj` или `custom` (с собственным классом фильтра, реализующим интерфейс `org.springframework.core.type.filter.TypeFilter`). Формат выражения в атрибуте `expression` зависит от заданного типа.

Объявление компонентов Spring

Разработав класс какой-нибудь службы, чтобы использовать его в приложении, основанном на Spring, необходимо сообщить каркасу Spring, что компоненты Spring Beans пригодны для внедрения в другие компоненты, и позволить ему управлять ими. Обратимся снова к примеру из главы 2, в котором интерфейс `MessageRender` служил для вывода сообщения, опираясь на интерфейс `MessageProvider` для получения воспроизводимого сообщения. В следующем примере кода приведены оба эти интерфейса и реализация их услуг по воспроизведению сообщений:

```
package com.apress.prospring5.ch2.decoupled;

// интерфейс средства воспроизведения
public interface MessageRenderer {
  void render();
  void setMessageProvider(MessageProvider provider);
  MessageProvider getMessageProvider();
}

// реализация средства воспроизведения
public class StandardOutMessageRenderer
       implements MessageRenderer {
  private MessageProvider messageProvider;

  @Override
  public void render() {
    if (messageProvider == null) {
      throw new RuntimeException("You must set the "
             + "property messageProvider of class:"
             + StandardOutMessageRenderer.class.getName());
    }
    System.out.println(messageProvider.getMessage());
  }

  @Override
  public void setMessageProvider(MessageProvider provider) {
```

```
    this.messageProvider = provider;
  }

  @Override
  public MessageProvider getMessageProvider() {
    return this.messageProvider;
  }
}

// интерфейс поставщика услуг
public interface MessageProvider {
  String getMessage();
}

// реализация поставщика услуг
public class HelloWorldMessageProvider
       implements MessageProvider {
  @Override
  public String getMessage() {
    return "Hello World!";
  }
}
```

На заметку Приведенные выше классы входят в состав пакета **com.apress.prospring5.ch2.decoupled**. Они применяются и в конкретном проекте из этой главы, поскольку в реально эксплуатируемых приложениях разработчики стараются использовать код повторно, а не дублировать его. Именно поэтому проект из главы 2 определяется как зависимость от некоторых проектов из главы 3, как можно обнаружить в исходном коде примеров из этой книги.

Чтобы объявить определения компонентов Spring Beans в XML-файле, следует воспользоваться дескриптором `<bean ../>`, как показано в приведенном ниже результирующем файле конфигурации `app-contextxml.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org
                           /schema/beans
          http://www.springframework.org/schema/beans
               /spring-beans.xsd">

<bean id="provider"
  class="com.apress.prospring5.ch2.decoupled
            .HelloWorldMessageProvider"/>

<bean id="renderer"
```

```
  class="com.apress.prospring5.ch2.decoupled
           .StandardOutMessageRenderer"
  p:messageProvider-ref="provider"/>
</beans>
```

В приведенных выше дескрипторах объявляются два компонента Spring Beans: один — с идентификатором `provider` и реализацией в классе `HelloWorldMessage Provider`, а другой — с идентификатором `renderer` и реализацией в классе `StandardOutMessageRenderer`. Начиная с данного примера, пространства имен больше не приводятся в последующих примерах конфигурирования, если только в них не вводятся новые пространства имен. Это сделано ради большей наглядности определений компонентов Spring Beans.

Чтобы определить компоненты Spring Beans с помощью аннотаций, классы этих компонентов необходимо снабдить соответствующими стереотипными аннотациями[2], а методы и конструкторы — аннотацией `@Autowired`. В последнем случае контейнер инверсии управления уведомляется, где именно следует искать компонент Spring Bean конкретного типа, чтобы задать его в качестве аргументов при вызове данного метода. В приведенном ниже фрагменте кода аннотации, применяемые для определения компонента Spring Bean, не определены. В качестве параметра стереотипных аннотаций можно задать имя результирующего компонента Spring Bean.

```
package com.apress.prospring5.ch3.annotation;

import com.apress.prospring5.ch2.decoupled.MessageProvider;
import org.springframework.stereotype.Component;

// простой компонент Spring Bean
@Component("provider")
public class HelloWorldMessageProvider
       implements MessageProvider {
  @Override
  public String getMessage() {
    return "Hello World!";
  }
}

import com.apress.prospring5.ch2.decoupled.MessageProvider;
import com.apress.prospring5.ch2.decoupled.MessageRenderer;
```

[2] Такие аннотации называются *стереотипными* потому, что они входят в состав пакета `org.springframework.stereotype`. В этом пакете собраны все аннотации, применяемые для определения компонентов Spring Beans. Эти аннотации соответствуют также назначению определяемых компонентов Spring Beans. Например, аннотация `@Service` служит для определения служебных компонентов Spring Beans, которые выполняют более сложные функции, предоставляя требующиеся услуги другим компонентам, тогда как аннотация `@Repository` — для определения компонентов Spring Beans, предназначенных для сохранения и извлечения информации из базы данных.

```
import org.springframework.stereotype.Service;
import org.springframework.beans.factory
          .annotation.Autowired;

// сложный служебный компонент Spring Bean
@Service("renderer")
public class StandardOutMessageRenderer
       implements MessageRenderer {
  private MessageProvider messageProvider;

  @Override
  public void render() {
    if (messageProvider == null) {
      throw new RuntimeException("You must set the "
             + "property messageProvider of class:"
             + StandardOutMessageRenderer.class.getName());
    }
    System.out.println(messageProvider.getMessage());
  }

  @Override
  @Autowired
  public void setMessageProvider(MessageProvider provider) {
    this.messageProvider = provider;
  }

  @Override
  public MessageProvider getMessageProvider() {
    return this.messageProvider;
  }
}
```

При начальной загрузке контекста типа `ApplicationContext` в соответствии с конфигурацией, приведенной ниже из XML-файла `app-context-annotation.xml`, каркас Spring обнаружит определенные в ней компоненты и получит их экземпляры с указанными именами.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org
                          /schema/context"
       xsi:schemaLocation="http://www.springframework.org
                              /schema/beans
          http://www.springframework.org/schema/beans
               /spring-beans.xsd
          http://www.springframework.org/schema/context
          http://www.springframework.org/schema/context
```

```
                /spring-context.xsd">
<context:component-scan
  base-package="com.apress.prospring5.ch3.annotation"/>
</beans>
```

Такой способ конфигурирования не оказывает никакого влияния на порядок получения компонентов Spring Beans из контекста типа `ApplicationContext`, как показано ниже.

```
package com.apress.prospring5.ch3;

import com.apress.prospring5.ch2.decoupled.MessageRenderer;
import org.springframework.context.support
          .GenericXmlApplicationContext;

public class DeclareSpringComponents {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();
    MessageRenderer messageRenderer =
           ctx.getBean("renderer", MessageRenderer.class);
    messageRenderer.render();
    ctx.close();
  }
}
```

Вместо экземпляра класса `DefaultListableBeanFactory` в данном случае получается экземпляр класса `GenericXmlApplicationContext`. Класс `GenericXmlApplicationContext` реализует интерфейс `ApplicationContext` и способен выполнить его начальную загрузку из конфигураций, определенных в XML-файлах.

Если поменять местами файлы конфигурации `app-context-xml.xml` и `app-context-annotation.xml` в исходном коде примеров из этой главы, то можно обнаружить, что в обоих случаях получается один и тот же результат: вывод на консоль строки `"Hello World!"`. Единственное отличие состоит в том, что после замены файла конфигурации функционировать будут те компоненты Spring Beans, которые определены с помощью аннотаций из пакета `com.apress.prospring5.ch3.annotation`.

Конфигурирование на языке Java

Как упоминалось в главе 1, файл конфигурации `app-context-xml.xml` можно заменить конфигурационным классом, не видоизменяя классы, представляющие типы создаваемых компонентов Spring Beans. Это удобно в том случае, если типы компонентов Spring Beans, требующихся в приложении, входят в состав сторонних библиотек и не подлежат изменению. В таком случае конфигурационный класс снаб-

жается аннотацией `@Configuration` и содержит методы, объявляемые с аннотацией `@Bean` и вызываемые непосредственно из контейнера инверсии управления для получения экземпляров компонентов Spring Beans. Имя компонента Spring Bean будет совпадать с именем метода, применяемого для его создания. В приведенном ниже примере кода демонстрируется конфигурационный класс, где имена методов специально подчеркнуты, чтобы стало очевиднее, каким образом именуются получающиеся в итоге компоненты Spring Beans.

```
package com.apress.prospring5.ch2.annotated;

import com.apress.prospring5.ch2.decoupled
          .HelloWorldMessageProvider;
import com.apress.prospring5.ch2.decoupled.MessageProvider;
import com.apress.prospring5.ch2.decoupled.MessageRenderer;
import com.apress.prospring5.ch2.decoupled
          .StandardOutMessageRenderer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class HelloWorldConfiguration {
  @Bean
  public MessageProvider provider() {
    return new HelloWorldMessageProvider();
  }

  @Bean
  public MessageRenderer renderer(){
    MessageRenderer renderer =
           new StandardOutMessageRenderer();
    renderer.setMessageProvider(provider());
    return renderer;
  }
}
```

Для чтения конфигурации из этого класса потребуется другая реализация интерфейса `ApplicationContext`:

```
package com.apress.prospring5.ch2.annotated;

import com.apress.prospring5.ch2.decoupled.MessageRenderer;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;

public class HelloWorldSpringAnnotated {
  public static void main(String... args) {
    ApplicationContext ctx =
              new AnnotationConfigApplicationContext
```

```
                (HelloWorldConfiguration.class);
    MessageRenderer mr = ctx.getBean("renderer",
                        MessageRenderer.class);
    mr.render();
  }
}
```

Вместо экземпляра класса `DefaultListableBeanFactory` в данном случае получается экземпляр класса `AnnotationConfigApplicationContext`. Класс `AnnotationConfigApplicationContext` реализует интерфейс `ApplicationContext` и способен выполнить его начальную загрузку из конфигураций, определенных в классе `HelloWorldConfiguration`.

Конфигурационный класс может служить и для чтения определений компонентов Spring Beans, снабженных аннотациями. В данном случае в конфигурационном классе не потребуются методы с аннотациями `@Bean`, поскольку конфигурирование определения компонента Spring Bean является составной частью его класса. Но для того чтобы найти определения компонентов Spring Beans в классах Java, придется активизировать просмотр этих компонентов. Это делается в конфигурационном классе с помощью аннотации `@ComponentScanning`, равнозначной элементу разметки `<context:component-scanning ../>`, как показано ниже. Параметры этой аннотации такие же, как и у равнозначного ей элемента XML-разметки.

```
package com.apress.prospring5.ch3.annotation;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@ComponentScan(basePackages =
                {"com.apress.prospring5.ch3.annotation"})
@Configuration
public class HelloWorldConfiguration {
}
```

Код для начальной загрузки среды Spring с помощью класса `AnnotationConfigApplicationContext` подходит и для данного класса, не требуя никаких изменений. Но в реально эксплуатируемых приложениях может присутствовать унаследованный код, разработанный с помощью прежних версий Spring, или же требоваться как XML-файл конфигурации, так и конфигурационный класс. Правда, конфигурирование в формате XML можно по-разному сочетать с конфигурированием на языке Java. Например, в конфигурационном классе можно импортировать определения компонентов Spring Beans из одного или нескольких XML-файлов конфигурации, используя аннотацию `@ImportResource`, как показано ниже. И тогда начальную загрузку можно выполнить с помощью одного и того же класса (в данном случае `AnnotationConfigApplicationContext`).

```
package com.apress.prospring5.ch3.mixed;

import org.springframework.context
          .annotation.ComponentScan;
import org.springframework.context
          .annotation.Configuration;
import org.springframework.context
          .annotation.ImportResource;
@ImportResource(locations =
                {"classpath:spring/app-context-xml.xml"})

@Configuration
  public class HelloWorldConfiguration {
}
```

Таким образом, каркас Spring дает вам возможность на деле проявить творческую инициативу, когда вы определяете компоненты Spring Beans. Подробнее об этом речь пойдет в главе 4, посвященной конфигурированию приложений Spring.

Конфигурирование внедрения зависимостей через метод установки

Для конфигурирования внедрения зависимостей через метод установки с помощью разметки в формате XML необходимо ввести дескрипторы `<property>` в дескриптор `<bean>` для каждого свойства, в котором должна быть внедрена зависимость. Например, чтобы присвоить компонент, реализующий поставщика сообщений (`provider`), свойству `messageProvider` компонента, реализующего средство воспроизведения (`renderer`), достаточно внести следующие изменения в дескриптор `<bean>` разметки этого компонента:

```
<beans ...>
  <bean id="renderer"
    class="com.apress.prospring5.ch2.decoupled
              .StandardOutMessageRenderer">
    <property name="messageProvider" ref="provider"/>
  </bean>

  <bean id="provider"
    class="com.apress.prospring5.ch2.decoupled
              .HelloWorldMessageProvider"/>
</beans>
```

В приведенном выше коде разметки компонент `provider` присваивается свойству `messageProvider`. С помощью атрибута `ref` свойству можно присвоить ссылку на компонент Spring Bean, как поясняется далее.

Если каркас Spring применяется с версии 2.5, а в XML-файле конфигурации объявлено пространство имен `p`, то объявить внедрение зависимостей можно так:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org
                                /schema/beans
          http://www.springframework.org/schema/beans
               /spring-beans.xsd">

    <bean id="renderer"
      class="com.apress.prospring5.ch2.decoupled
                .StandardOutMessageRenderer"
      p:messageProvider-ref="provider"/>

    <bean id="provider"
      class="com.apress.prospring5.ch2.decoupled
                .HelloWorldMessageProvider"/>
</beans>
```

На заметку Пространство имен **p** не определено в XSD-файле и существует только в ядре Spring. Поэтому XSD-файлы и не объявляются в атрибуте **schemaLocation**.

С аннотациями дело обстоит еще проще. В объявление метода установки достаточно ввести аннотацию `@Autowired`, как показано в следующем фрагменте кода:

```
package com.apress.prospring5.ch3.annotation;
...
import org.springframework.beans.factory
          .annotation.Autowired;
@Service("renderer")
public class StandardOutMessageRenderer
       implements MessageRenderer {
  ...
  @Override
  @Autowired
  public void setMessageProvider(MessageProvider provider) {
    this.messageProvider = provider;
  }
}
```

При инициализации контекста типа `ApplicationContext` каркас Spring обнаружит подобные аннотации `@Autowired` и внедрит зависимость по мере необходимости, поскольку в XML-файле конфигурации объявлен дескриптор `<context:component-scan>`.

На заметку Чтобы добиться того же результата, вместо аннотации **@Autowired** можно также воспользоваться аннотацией **@Resource(name="messageProvider")**. Аннотация **@Resource** входит в общий набор аннотаций Java, определяемых в стандарте JSR-250 для

применения на платформах JSE и JEE. В отличие от аннотации **@Autowired**, в аннотации **@Resource** поддерживается параметр **name** для указания более точных требований к внедрению зависимостей. Кроме того, в Spring поддерживается аннотация **@Inject**, внедренная в спецификацию JSR-299 (Contexts and Dependency Injection for the Java EE Platform — Контексты и внедрение зависимостей для платформы Java EE). Аннотация **@Inject** равнозначна по своему поведению аннотации **@Autowired** в Spring.

Чтобы проверить полученный результат, можно воспользоваться упоминавшимся ранее классом `DeclareSpringComponents`. Как и прежде, файлы конфигурации `app-context-xml.xml` и `app-context-annotation.xml` можно поменять местами в исходном коде примеров из этой главы. И в этом случае, как и в предыдущем, получается один и тот же результат: вывод строки `"Hello World!"` на консоль.

Конфигурирование внедрения зависимостей через конструктор

В предыдущем примере реализации интерфейса `MessageProvider` в классе `HelloWorldMessageProvider` возвращается одно и то же жестко закодированное сообщение после каждого вызова метода `getMessage()`. В файле конфигурации Spring можно легко создать конфигурируемую реализацию интерфейса `MessageProvider`, которая позволит определять сообщение внешним образом, как демонстрируется в следующем фрагменте кода:

```
package com.apress.prospring5.ch3.xml;

import com.apress.prospring5.ch2.decoupled.MessageProvider;

public class ConfigurableMessageProvider
        implements MessageProvider {
  private String message;

  public ConfigurableMessageProvider(String message) {
    this.message = message;
  }

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

Как видите, получить экземпляр класса `ConfigurableMessageProvider`, не предоставив строковое значение для сообщения, невозможно (если только не указать пустое значение `null`). Но ведь именно это и требуется, и данный класс идеально подходит для конфигурирования внедрения зависимостей через конструктор. В следующем фрагменте кода показано, как переопределить компонент `provider`, чтобы получить экземпляр класса `ConfigurableMessageProvider` и внедрить сообщение как зависимость через конструктор:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation=
          "http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans
                /spring-beans.xsd">

  <bean id="messageProvider"
    class="com.apress.prospring5.ch3.xml
              .ConfigurableMessageProvider">
    <constructor-arg value=
        "I hope that someone gets my message in a bottle"/>
  </bean>
</beans>
```

В приведенном выше коде вместо дескриптора `<property>` применяется дескриптор `<constructor-arg>`. Но на этот раз конструктору передается не другой компонент Spring Bean, а только строковый литерал, и поэтому для указания конкретного значения аргумента конструктора используется атрибут `value` вместо атрибута `ref`.

Если же у конструктора имеется несколько аргументов, а у класса — несколько конструкторов, каждый дескриптор `<constructor-arg>` придется снабдить атрибутом `index`, чтобы указать в сигнатуре конструктора индекс аргумента, начиная с `0`. Имея дело с конструкторами, принимающими несколько аргументов, лучше применять атрибут `index`, чтобы избежать путаницы в параметрах и гарантировать выбор в Spring нужного конструктора.

Помимо пространства имен `p`, начиная с версии Spring 3.1, можно также использовать пространство имен `c`:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:c="http://www.springframework.org/schema/c"
       xsi:schemaLocation="http://www.springframework.org
                              /schema/beans
          http://www.springframework.org/schema/beans
              /spring-beans.xsd">

       <bean id="provider"
            class="com.apress.prospring5.ch3.xml
                      .ConfigurableMessageProvider"
            c:message="I hope that someone gets my
                       message in a bottle"/>
</beans>
```

На заметку Пространство имен **p** не определено в XSD-файле и существует только в ядре Spring. Поэтому XSD-файлы и не объявляются в атрибуте **schemaLocation**.

Для внедрения зависимостей через конструктор в объявлении метода-конструктора целевого компонента Spring Bean применяется также аннотация `@Autowired`. Это другой вариант по сравнению с внедрением зависимостей через метод установки, как демонстрируется в следующем фрагменте кода:

```
package com.apress.prospring5.ch3.annotated;

import com.apress.prospring5.ch2.decoupled.MessageProvider;
import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.stereotype.Service;

@Service("provider")
public class ConfigurableMessageProvider
       implements MessageProvider {
  private String message;

  @Autowired
  public ConfigurableMessageProvider(
         (@Value("Configurable message") String message) {
    this.message = message;
  }

  @Override
  public String getMessage() {
    return this.message;
  }
}
```

В приведенном выше фрагменте кода применяется также аннотация `@Value`, с помощью которой определяется значение, внедряемое в конструктор. Подобным способом значения внедряются в компоненты Spring Beans. Для динамического внедрения значений, помимо простых символьных строк, можно воспользоваться эффективным языком SpEL, как поясняется далее.

Но жесткое кодирование значений — не самая удачная идея, поскольку их изменение влечет за собой перекомпиляцию всей прикладной программы. И даже если выбрать конфигурирование внедрения зависимостей с помощью аннотаций, то значения, предназначенные для внедрения, рекомендуется все же выносить за пределы прикладного кода. Чтобы вынести сообщение за пределы прикладного кода, определим его как компонент Spring Bean в файле конфигурации с аннотациями, как демонстрируется в следующем фрагменте кода конфигурации:

```
<beans ...>
  <context:component-scan
    base-package="com.apress.prospring5.ch3.annotated"/>
  <bean id="message" class="java.lang.String"
    c:_0="I hope that someone gets my
```

```
          message in a bottle"/>

  <bean id="message2" class="java.lang.String"
    c:_0="I know I won't be injected :("/>

</beans>
```

В этом фрагменте кода определяются два компонента Spring Bean с идентификаторами `message` и `message2` и одинаковым типом `java.lang.String`, чтобы избежать путаницы при их внедрении как зависимостей через конструктор. Обратите внимание на то, что для установки строкового значения здесь применяется также пространство имен `c`, выделяемое для внедрения зависимостей через конструктор, а обозначение `_0` указывает на индекс аргумента в конструкторе.

Итак, объявив компонент Spring Bean, можно избавиться от аннотации `@Value` в целевом компоненте, как показано в следующем фрагменте кода:

```
package com.apress.prospring5.ch3.annotated;

import com.apress.prospring5.ch2.decoupled.MessageProvider;
import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.stereotype.Service;

@Service("provider")
public class ConfigurableMessageProvider
       implements MessageProvider {
  private String message;

  @Autowired
  public ConfigurableMessageProvider(String message) {
    this.message = message;
  }

  @Override
  public String getMessage() {
    return this.message;
  }
}
```

Объявленный компонент `message` и его идентификатор совпадают с именем аргумента в конструкторе, и поэтому Spring обнаружит аннотацию и внедрит заданное значение в метод-конструктор. Если выполнить тест, используя приведенный ниже код для проверки конфигурации с помощью разметки в формате XML (файл `app-context-xml.xml`) и аннотаций (файл `app-context-annotation.xml`), то в обоих случаях будет воспроизведено сконфигурированное сообщение.

```
package com.apress.prospring5.ch3;

import com.apress.prospring5.ch2.decoupled.MessageProvider;
```

```
import org.springframework.context.support
          .GenericXmlApplicationContext;

public class DeclareSpringComponents {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-annotation.xml");
    ctx.refresh();
    MessageProvider messageProvider =
           ctx.getBean("provider", MessageProvider.class);
    System.out.println(messageProvider.getMessage());
  }
}
```

Иногда у Spring нет возможности выяснить, какой именно конструктор требуется использовать для внедрения зависимостей. Как правило, это происходит в том случае, если имеются два конструктора с одним и тем же количеством аргументов и одинаково обозначенными их типами. Рассмотрим в качестве примера следующий фрагмент кода:

```
package com.apress.prospring5.ch3.xml;

import org.springframework.context.support
          .GenericXmlApplicationContext;

public class ConstructorConfusion {
  private String someValue;

  public ConstructorConfusion(String someValue) {
    System.out.println(
               "ConstructorConfusion(String) called");
    this.someValue = someValue;
  }

  public ConstructorConfusion(int someValue) {
    System.out.println("ConstructorConfusion(int) called");
    this.someValue = "Number: "
                   + Integer.toString(someValue);
  }

  public String toString() {
  return someValue;
  }

  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
```

```
    ctx.refresh();
    ConstructorConfusion cc = (ConstructorConfusion)
              ctx.getBean("constructorConfusion");
    System.out.println(cc);
    ctx.close
  }
}
```

В данном примере кода компонент Spring Bean типа `ConstructorConfusion` просто извлекается из интерфейса `ApplicationContext` и выводит заданное значение на консоль. А теперь рассмотрим следующий фрагмент кода конфигурации:

```
<beans ...>
  <bean id="provider"
    class="com.apress.prospring5.ch3.xml
              .ConfigurableMessageProvider"
    c:message="I hope that someone gets my
               message in a bottle"/>
  <bean id="constructorConfusion"
    class="com.apress.prospring5.ch3.xml
              .ConstructorConfusion">
    <constructor-arg>
      <value>90</value>
    </constructor-arg>
  </bean>
</beans>
```

Какой конструктор будет вызван в этом случае? Выполнение кода из данного примера дает следующий результат:

```
ConstructorConfusion(String) called
```

Как показывает приведенный выше результат, был вызван конструктор с аргументом типа `String`. Но ведь это совсем не тот результат, который ожидался, если учесть, что любое целочисленное значение, передаваемое с помощью внедрения зависимостей через конструктор, предваряется префиксом **`Number:`**, как следует из кода конструктора с аргументом типа `int`. Чтобы выйти из этого затруднительного положения, придется внести незначительное изменение в конфигурацию:

```
<beans ...>
  <bean id="provider"
    class="com.apress.prospring5.ch3.xml
              .ConfigurableMessageProvider"
    c:message="I hope that someone gets my
               message in a bottle"/>

  <bean id="constructorConfusion"
    class="com.apress.prospring5.ch3.xml
              .ConstructorConfusion">
    <constructor-arg type="int">
```

```
        <value>90</value>
    </constructor-arg>
  </bean>

</beans>
```

Обратите внимание на то, что в дескрипторе `<constructor-arg>` появился дополнительный атрибут `type`, обозначающий тип аргумента, который должен обнаруживать каркас Spring. Выполнение кода из данного примера дает теперь следующий результат:

```
ConstructorConfusion(int) called
Number: 90
```

При конфигурировании внедрения зависимостей через конструктор с помощью аннотаций такой путаницы можно избежать, применяя аннотацию непосредственно к целевому методу-конструктору, как показано в следующем фрагменте кода:

```
package com.apress.prospring5.ch3.annotated;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.support.GenericXmlApplicationContext;
import org.springframework.stereotype.Service;

@Service("constructorConfusion")
public class ConstructorConfusion {
  private String someValue;

  public ConstructorConfusion(String someValue) {
    System.out.println("ConstructorConfusion(String) called");
    this.someValue = someValue;
  }

  @Autowired
  public ConstructorConfusion(@Value("90") int someValue) {
    System.out.println("ConstructorConfusion(int) called");
    this.someValue = "Number: " + Integer.toString(someValue);
  }

  public String toString() {
    return someValue;
  }

  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-annotation.xml");
    ctx.refresh();

    ConstructorConfusion cc = (ConstructorConfusion)
```

```
                ctx.getBean("constructorConfusion");
    System.out.println(cc);
    ctx.close();
  }
}
```

Благодаря применению аннотации `@Autowired` в целевом методе-конструкторе именно он и будет использоваться в Spring для получения экземпляра компонента Spring Bean и внедрения указанного значения. Как и прежде, это значение должно быть вынесено во внешнюю конфигурацию.

На заметку Аннотацию **`@Autowired`** можно применять только в одном из методов-конструкторов. Если же она применяется в нескольких методах-конструкторах, каркас Spring выдаст соответствующее предупреждение во время начальной загрузки контекста типа **`ApplicationContext`**.

Конфигурирование внедрения зависимостей через поле

Третья разновидность внедрения зависимостей, поддерживаемая в Spring, называется *внедрением зависимостей через поле*. Как подразумевает само название, зависимость внедряется непосредственно в поле, и для этой цели не требуется ни конструктор, ни метод установки, достаточно лишь снабдить аннотацией `@Autowired` соответствующее поле, являющееся членом класса. Такое внедрение зависимостей оказывается вполне практичным, поскольку разработчик избавляется от необходимости писать код, который может не понадобиться после первоначального создания компонента Spring Bean, если зависимость от объекта больше не нужна за его пределами. В следующем фрагменте кода демонстрируется компонент Spring Bean типа `Singer` с полем типа `Inspiration`:

```
package com.apress.prospring5.ch3.annotated;

import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.stereotype.Service;

@Service("singer")
public class Singer {

  @Autowired
  private Inspiration inspirationBean;

  public void sing() {
    System.out.println("... " + inspirationBean.getLyric());
  }
}
```

Поле `inspirationBean` объявлено закрытым, хотя это не имеет никакого значения для контейнера инверсии управления, поскольку для заполнения обязательной зависимости в нем применяется рефлексия. Ниже приведен исходный код класса `Inspiration`, представляющего простой компонент Spring Bean с полем типа `String`.

```
package com.apress.prospring5.ch3.annotated;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class Inspiration {

  private String lyric = "I can keep the door "
              + "cracked open, to let light through";
  public Inspiration(
    @Value("For all my running, I can understand")
    String lyric) {
      this.lyric = lyric;
    }

  public String getLyric() {
    return lyric;
  }

   public void setLyric(String lyric) {
     this.lyric = lyric;
  }
}
```

В следующей конфигурации задается просмотр компонентов Spring Beans для обнаружения их определений, которые будут созданы в контейнере инверсии управления:

```
<beans ...>
  <context:component-scan
    base-package="com.apress.prospring5.ch3.annotated"/>
</beans>
```

Обнаружив один компонент Spring Bean типа `Inspiration`, контейнер инверсии управления внедрит его в поле `inspirationBean` компонента `singer`. Именно поэтому при выполнении исходного кода из приведенного ниже примера на консоль выводится текст `"For all my running, I can understand"` (Несмотря на всю мою беготню, я в состоянии понять).

```
package com.apress.prospring5.ch3.annotated;

import org.springframework.context.support
          .GenericXmlApplicationContext;
```

```
public class FieldInjection {

  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context.xml");
    ctx.refresh();

    Singer singerBean = ctx.getBean(Singer.class);
    singerBean.sing();
    ctx.close();
  }
}
```

Тем не менее внедрению зависимостей через поле присущи перечисленные ниже недостатки, поэтому его обычно принято избегать.

- Несмотря на всю простоту такого способа внедрения зависимостей, необходимо соблюдать особую осторожность, чтобы не нарушить принцип единственной ответственности. Чем больше зависимостей, тем больше обязанностей у класса, а это может затруднить разделение обязанностей во время реорганизации кода. Ситуацию, когда класс становится слишком громоздким, легче выявить, когда зависимости внедряются через конструкторы или методы установки, но довольно трудно обнаружить, когда они внедряются через поля.
- В каркасе Spring обязанности по внедрению зависимостей передаются контейнеру инверсии управления, но класс должен явно сообщить типы требующихся зависимостей через открытый интерфейс, методы или конструкторы. Если же зависимости внедряются через поля, то может быть неясно, какого зависимость именно типа требуется и является ли она обязательной или нет.
- Внедрение зависимостей через поле вносит зависимость от контейнера инверсии управления, поскольку аннотация `@Autowired` является компонентом Spring. Следовательно, компонент Spring Bean больше не является простым объектом POJO, и его экземпляр нельзя получить независимым путем.
- Внедрение зависимостей через поле неприменимо к оконечным полям. Такие поля могут быть инициализированы только с помощью внедрения зависимостей через конструктор.
- Внедрение зависимостей через поле затрудняет написание тестов, поскольку зависимости приходится внедрять вручную.

Применение параметров внедрения зависимостей

В трех предыдущих примерах было показано, как внедрять другие компоненты и значения в компонент Spring Bean через конструктор и метод установки. В каркасе Spring поддерживается огромное количество вариантов для применения параметров внедрения, что дает возможность внедрять не только другие компоненты и простые

значения, но и коллекции Java, внешне определенные свойства и даже компоненты Spring Beans из другой фабрики. Все эти типы параметров могут применяться для внедрения зависимостей как через метод установки, так и через конструктор с помощью соответствующего дескриптора разметки, вводимого в дескрипторы `<property>` и `<constructor-args>`.

Внедрение простых значений

Внедрять простые значения в компоненты Spring Beans совсем не трудно: достаточно указать в конфигурации значение, заключенное в дескриптор разметки `<value>`. По умолчанию дескриптор `<value>` допускает не только чтение значений типа `String`, но и их преобразование в любой примитивный тип данных или класс-оболочку примитивного типа. В следующем фрагменте кода приведен простой компонент Spring Bean с различными свойствами, доступными для внедрения:

```
package com.apress.prospring5.ch3.xml;

import org.springframework.context.support
          .GenericXmlApplicationContext;

public class InjectSimple {

  private String name;
  private int age;
  private float height;
  private boolean programmer;
  private Long ageInSeconds;

  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();
    InjectSimple simple = (InjectSimple)
                          ctx.getBean("injectSimple");
    System.out.println(simple);
    ctx.close();
  }

  public void setAgeInSeconds(Long ageInSeconds) {
    this.ageInSeconds = ageInSeconds;
  }

  public void setProgrammer(boolean programmer) {
    this.programmer = programmer;
  }

  public void setAge(int age) {
```

```
    this.age = age;
  }

  public void setHeight(float height) {
    this.height = height;
  }

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

  public String toString() {
    return "Name: " + name + "\n"
         + "Age: " + age + "\n"
         + "Age in Seconds: " + ageInSeconds + "\n"
         + "Height: " + height + "\n"
         + "Is Programmer?: " + programmer;
  }
}
```

Помимо свойств, в классе `InjectSimple` определяется метод `main()`, где сначала создается объект класса, реализующего интерфейс `ApplicationContext`, а затем извлекается компонент Spring Bean типа `InjectSimple`. После этого значения свойств данного компонента Spring Bean направляются для вывода на консоль. В следующем фрагменте кода представлена конфигурация данного компонента Spring Bean из файла `app-context-xml.xml`:

```
<beans ...>
    <bean id="injectSimpleConfig"
        class="com.apress.prospring5.ch3.xml
                  .InjectSimpleConfig"/>
    <bean id="injectSimpleSpel"
        class="com.apress.prospring5.ch3.xml
                  .InjectSimpleSpel"
        p:name="John Mayer"
        p:age="39"
        p:height="1.92"
        p:programmer="false"
        p:ageInSeconds="1241401112"/>
</beans>
```

Как следует из обоих приведенных выше фрагментов кода, вполне возможно определить сначала свойства компонента Spring Bean, принимающие значения типа `String`, примитивных типов данных или их классов-оболочек, а затем внедрить значения этих свойств с помощью дескриптора `<value>`. Ниже приведен результат, выводимый на консоль при выполнении исходного кода из данного примера.

```
Name: John Mayer
Age: 39
```

```
Age in Seconds: 1241401112
Height: 1.92
Is Programmer?: false
```

Что же касается внедрения простых значений с помощью аннотаций, то в свойствах компонента Spring Bean можно применить аннотацию `@Value`. Но на этот раз вместо метода установки применим эту аннотацию в операторе объявления свойства, как показано в приведенном ниже фрагменте кода. (В каркасе Spring аннотации поддерживаются в методе установки или же в свойствах.) В конечном итоге получается такой же результат, как и при конфигурировании в формате XML:

```
package com.apress.prospring5.ch3.annotated;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.support
          .GenericXmlApplicationContext;
import org.springframework.stereotype.Service;

@Service("injectSimple")
public class InjectSimple {

  @Value("John Mayer")
  private String name;
  @Value("39")
  private int age;
  @Value("1.92")
  private float height;
  @Value("false")
  private boolean programmer;
  @Value("1241401112")
  private Long ageInSeconds;

  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-annotation.xml");
    ctx.refresh();

    InjectSimple simple = (InjectSimple)
                          ctx.getBean("injectSimple");
    System.out.println(simple);
    ctx.close();
  }

  public String toString() {
    return "Name: "
         + name + "\n"
         + "Age: " + age + "\n"
         + "Age in Seconds: " + ageInSeconds + "\n"
```

```
            + "Height: " + height + "\n"
            + "Is Programmer?: " + programmer;
    }
}
```

Внедрение значений средствами SpEL

В версии Spring 3 появилось эффективное функциональное средство — язык выражений Spring (Spring Expression Language — SpEL), позволяющий вычислять выражения динамически и затем применять их в интерфейсе `ApplicationContext`. Результат вычисления заданного выражения можно использовать для внедрения в компоненты Spring Beans. В этом разделе на примере из предыдущего раздела будет показано, как применять язык SpEL для внедрения свойств из других компонентов Spring Beans.

Допустим, теперь требуется вынести в конфигурационный класс значения, предназначенные для внедрения в компонент Spring Bean, как показано в следующем фрагменте кода:

```
package com.apress.prospring5.ch3.annotated;

import org.springframework.stereotype.Component;

@Component("injectSimpleConfig")
public class InjectSimpleConfig {

  private String name = "John Mayer";
  private int age = 39;
  private float height = 1.92f;
  private boolean programmer = false;
  private Long ageInSeconds = 1_241_401_112L;

  public String getName() {
    return name;
  }

  public int getAge() {
    return age;
  }

  public float getHeight() {
    return height;
  }

  public boolean isProgrammer() {
    return programmer;
  }

  public Long getAgeInSeconds() {
```

```
    return ageInSeconds;
  }
}
```

Затем компонент Spring Bean можно определить в XML-файле конфигурации и внедрить средствами SpEL его свойства в зависимый компонент Spring Bean, как показано ниже.

```
<beans ...>
    <bean id="injectSimpleConfig"
        class="com.apress.prospring5.ch3.xml
                  .InjectSimpleConfig"/>
    <bean id="injectSimpleSpel"
         class="com.apress.prospring5.ch3.xml
                   .InjectSimpleSpel"
         p:name="#{injectSimpleConfig.name}"
         p:age="#{injectSimpleConfig.age + 1}"
         p:height="#{injectSimpleConfig.height}"
         p:programmer="#{injectSimpleConfig.programmer}"
         p:ageInSeconds=
                "#{injectSimpleConfig.ageInSeconds}"/>
</beans>
```

Обратите внимание на применение выражения `#{injectSimple Config.name}` языка SpEL в ссылке на свойство другого компонента Spring Bean. Для возраста (`age`) к значению из компонента Spring Bean прибавляется `1`, чтобы продемонстрировать возможность применения SpEL для манипулирования свойством и его внедрения в зависимый компонент Spring Bean. А теперь проверим приведенную выше конфигурацию с помощью следующего фрагмента кода:

```
package com.apress.prospring5.ch3.xml;

import org.springframework.context.support
          .GenericXmlApplicationContext;

public class InjectSimpleSpel {

  private String name;
  private int age;
  private float height;
  private boolean programmer;
  private Long ageInSeconds;

  public String getName() {
    return this.name;
  }

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

```
}

public int getAge() {
  return this.age;
}

public void setAge(int age) {
  this.age = age;
}

public float getHeight() {
  return this.height;
}

public void setHeight(float height) {
  this.height = height;
}

public boolean isProgrammer() {
  return this.programmer;
}

public void setProgrammer(boolean programmer) {
  this.programmer = programmer;
}

public Long getAgeInSeconds() {
  return this.ageInSeconds;
}

public void setAgeInSeconds(Long ageInSeconds) {
  this.ageInSeconds = ageInSeconds;
}

public String toString() {
  return "Name: "
       + name + "\n"
       + "Age: " + age + "\n"
       + "Age in Seconds: " + ageInSeconds + "\n"
       + "Height: " + height + "\n"
       + "Is Programmer?: " + programmer;
}

public static void main(String... args) {
  GenericXmlApplicationContext ctx =
            new GenericXmlApplicationContext();
  ctx.load("classpath:spring/app-context-xml.xml");
  ctx.refresh();
```

```
    InjectSimpleSpel simple = (InjectSimpleSpel)
                      ctx.getBean("injectSimpleSpel");
    System.out.println(simple);

    ctx.close();
  }
}
```

Результат выполнения приведенного выше кода выглядит следующим образом:

```
Name: John Mayer
Age: 40
Age in Seconds: 1241401112
Height: 1.92
Is Programmer?: false
```

Для внедрения значений с помощью аннотаций потребуется лишь добавить аннотации `@Value` с выражениями SpEL, как демонстрируется в следующем примере кода:

```
package com.apress.prospring5.ch3.annotated;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.support
          .GenericXmlApplicationContext;
import org.springframework.stereotype.Service;

@Service("injectSimpleSpel")
public class InjectSimpleSpel {

  @Value("#{injectSimpleConfig.name}")
  private String name;

  @Value("#{injectSimpleConfig.age + 1}")
  private int age;

  @Value("#{injectSimpleConfig.height}")
  private float height;

  @Value("#{injectSimpleConfig.programmer}")
  private boolean programmer;

  @Value("#{injectSimpleConfig.ageInSeconds}")
  private Long ageInSeconds;

  public String toString() {
    return "Name: "
         + name + "\n"
         + "Age: " + age + "\n"
         + "Age in Seconds: " + ageInSeconds + "\n"
```

```
        + "Height: " + height + "\n"
        + "Is Programmer?: " + programmer;
  }

  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-annotation.xml");
    ctx.refresh();

    InjectSimpleSpel simple = (InjectSimpleSpel)
                    ctx.getBean("injectSimpleSpel");
    System.out.println(simple);

    ctx.close();
  }
}
```

Ниже приведена версия класса `InjectSimpleConfig`, в которой применяются аннотации.

```
package com.apress.prospring5.ch3.annotated;

import org.springframework.stereotype.Component;

@Component("injectSimpleConfig")
public class InjectSimpleConfig {

  private String name = "John Mayer";
  private int age = 39;
  private float height = 1.92f;
  private boolean programmer = false;
  private Long ageInSeconds = 1_241_401_112L;

// здесь определяются методы установки свойств ...
```

В приведенном выше фрагменте кода вместо аннотации `@Service` применяется аннотация `@Component`. По существу, применение аннотации `@Component` дает тот же самый эффект, что и аннотация `@Service`. Обе аннотации указывают каркасу Spring на то, что снабженный ими класс является кандидатом на автоматическое обнаружение с помощью основанной на аннотациях конфигурации и просмотра путей к классам. Но поскольку класс `InjectSimpleConfig` сохраняет конфигурацию приложения, а не предоставляет бизнес-услуги, то применение аннотации `@Component` имеет больший смысл. На практике аннотация `@Service` является частным случаем аннотации `@Component`, отражая тот факт, что снабженный ею класс предоставляет услуги бизнес-логики другим уровням в приложении.

Тестирование данной прикладной программы даст тот же самый результат. Используя язык SpEL, можно получать доступ к любым управляемым компонентам

Spring Beans и их свойствам, а также манипулировать ими в приложении благодаря поддержке в Spring развитых языковых средств и синтаксиса.

Внедрение компонентов Spring Beans в одном и том же блоке XML-разметки

Как было показано ранее, один компонент Spring Bean можно внедрять в другой с помощью дескриптора разметки `<ref>`. В следующем фрагменте кода приведен класс, в котором предоставляется метод установки с целью разрешить внедрение компонент Spring Bean:

```
package com.apress.prospring5.ch3.xml;

import org.springframework.context.support
          .GenericXmlApplicationContext;
import com.apress.prospring5.ch3.Oracle;

public class InjectRef {
  private Oracle oracle;

  public void setOracle(Oracle oracle) {
    this.oracle = oracle;
  }

  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();

    InjectRef injectRef = (InjectRef)
                          ctx.getBean("injectRef");
    System.out.println(injectRef);

    ctx.close();
  }

  public String toString() {
    return oracle.defineMeaningOfLife();
  }
}
```

Чтобы настроить Spring на внедрение одного компонента Spring Bean в другой, сначала придется сконфигурировать два компонента: один — в качестве внедряемого, а другой — в качестве цели внедрения. И как только это будет сделано, останется лишь сконфигурировать внедрение зависимостей с помощью дескриптора разметки `<ref>` для целевого компонента. Пример такой конфигурации в XML-файле `app-context-xml.xml` приведен ниже.

```
<beans ...>
    <bean id="oracle" name="wiseworm"
        class="com.apress.prospring5.ch3.BookwormOracle"/>
    <bean id="injectRef"
        class="com.apress.prospring5.ch3.xml.InjectRef">
        <property name="oracle">
            <ref bean="oracle"/>
        </property>
    </bean>
</beans>
```

В результате выполнения исходного кода из класса `InjectRef` на консоль будет выведен следующий текст:

```
Encyclopedias are a waste of money -
go see the world instead[3]
```

Здесь важно отметить следующее обстоятельство: внедряемый тип не должен в точности совпадать с тем типом, который определен в качестве целевого; оба типа должны быть просто совместимыми. Совместимость означает, что если тип, объявленный в качестве целевого, является интерфейсом, то внедряемый тип должен реализовывать этот интерфейс. Если же целевой тип является классом, то внедряемый должен относиться к тому же самому типу или его подтипу. В данном примере в классе `InjectRef` определяется метод `setOracle()`, предназначенный для получения экземпляра типа `Oracle`, который относится к интерфейсу, а внедряемый тип относится к классу `BookwormOracle`, реализующему интерфейс `Oracle`. Данное обстоятельство может привести в замешательство некоторых разработчиков, но на самом деле здесь нет ничего сложного. Внедрение подчиняется тем же правилам типизации, что и любой код Java, поэтому, если вы знаете, каким образом механизм типизации действует в Java, понять этот механизм при внедрении зависимостей вам будет нетрудно.

В предыдущем примере идентификатор компонента Spring Bean, предназначенного для внедрения, был указан с атрибутом `local` в дескрипторе разметки `<ref>`. Как будет показано далее в разделе "Именование компонентов Spring Beans", компоненту Spring Bean можно присвоить не одно, а несколько имен и ссылаться на него с помощью самых разных псевдонимов. Применение атрибута `local` означает, что в дескрипторе разметки `<ref>` всегда просматривается только идентификатор компонента Spring Bean и вообще не принимаются во внимание его псевдонимы. Более того, определение компонента Spring Bean должно существовать в том же самом XML-файле конфигурации. Чтобы внедрить компонент Spring Bean по любому имени или импортировать его из другого XML-файла конфигурации, вместо атрибута `local` в дескрипторе разметки `<ref>` необходимо указать атрибут `bean`. В следующем фраг-

[3] Энциклопедии — напрасная трата средств, идите в мир и смотрите сами.

менте кода приведена альтернативная конфигурация для предыдущего примера, в которой используется другое имя внедряемого компонента Spring Bean:

```
<beans ...>
    <bean id="oracle" name="wiseworm"
        class="com.apress.prospring5.ch3.BookwormOracle"/>
    <bean id="injectRef"
        class="com.apress.prospring5.ch3.xml.InjectRef">
        <property name="oracle">
            <ref bean="wiseworm"/>
        </property>
    </bean>
</beans>
```

В этом примере компоненту `oracle` с помощью атрибута `name` присваивается псевдоним, после чего этот компонент Spring Bean внедряется в компонент `injectRef` с указанием псевдонима в атрибуте `bean` дескриптора `<ref>`. Не особенно отчаивайтесь, если вам пока еще непонятна семантика именования компонентов Spring Beans, поскольку мы подробно обсудим ее далее в этой главе. В результате выполнения исходного кода из класса `InjectRef` на консоль будет выведен такой же результат, как и в предыдущем примере.

Внедрение и вложение контекстов приложений

До сих пор внедряемые компоненты Spring Beans располагались в том же контексте приложения типа `ApplicationContext`, а следовательно, в той же самой фабрике типа `BeanFactory`, что и те компоненты Spring Beans, в которые они были внедрены. Тем не менее в Spring поддерживается иерархическая структура для интерфейса `ApplicationContext`, когда один контекст (и связанная с ним фабрика типа `BeanFactory`) является родительским для другого контекста. Разрешая вложение контекста типа `ApplicationContext`, каркас Spring делает возможным разнесение конфигурации по отдельным файлам, а это настоящая находка для крупных проектов, содержащих немало компонентов Spring Beans.

При вложении контекста типа `ApplicationContext` каркас Spring позволяет компонентам Spring Beans из порожденного контекста ссылаться на компоненты в родительском контексте. Вложение контекста типа `ApplicationContext` с помощью класса `GenericXmlApplicationContext` осуществляется очень просто. Чтобы вложить один контекст типа `GenericXmlApplicationContext` в другой, достаточно вызвать метод `setParent()` в порожденном контексте типа `ApplicationContext`, как показано в следующем примере кода:

```
package com.apress.prospring5.ch3;

import org.springframework.context.support
          .GenericXmlApplicationContext;

public class HierarchicalAppContextUsage {
```

```
  public static void main(String... args) {
    GenericXmlApplicationContext parent =
              new GenericXmlApplicationContext();
    parent.load("classpath:spring/parent-context.xml");
    parent.refresh();

    GenericXmlApplicationContext child =
              new GenericXmlApplicationContext();
    child.load("classpath:spring/child-context.xml");
    child.setParent(parent);
    child.refresh();

    Song song1 = (Song) child.getBean("song1");
    Song song2 = (Song) child.getBean("song2");
    Song song3 = (Song) child.getBean("song3");

    System.out.println("from parent ctx: "
                    + song1.getTitle());
    System.out.println("from child ctx: "
                    + song2.getTitle());
    System.out.println("from parent ctx: "
                    + song3.getTitle());
    child.close();
    parent.close();
  }
}
```

Класс `Song` довольно прост:

```
package com.apress.prospring5.ch3;

public class Song {
  private String title;

  public void setTitle(String title) {
    this.title = title;
  }

  public String getTitle() {
    return title;
  }
}
```

В файле конфигурации порожденного контекста типа `ApplicationContext` ссылка на какой-нибудь компонент Spring Bean из родительского контекста типа `ApplicationContext` действует таким же образом, как и ссылка на компонент Spring Bean из порожденного контекста типа `ApplicationContext`, если только в порожденном контексте данного типа не присутствует компонент Spring Bean с тем же самым именем. В таком случае необходимо просто заменить атрибут `bean` в дескрипторе `<ref>` атрибутом `parent`.

В следующем фрагменте кода конфигурации приведено содержимое файла `parent-context.xml` для конфигурирования родительского контекста типа `BeanFactory`:

```
<beans ...>
    <bean id="childTitle"
        class="java.lang.String" c:_0="Daughters"/>

    <bean id="parentTitle"
        class="java.lang.String" c:_0="Gravity"/>
</beans>
```

Как видите, в данном фрагменте кода конфигурации просто определяются следующие два компонента Spring Beans: `childTitle` и `parentTitle`, и оба они являются объектами типа `String` со строковыми значениями `Daughters` и `Gravity`. В приведенном ниже фрагменте кода приведена конфигурация порожденного контекста типа `ApplicationContext` из файла `child-context.xml`.

```
<beans ...>
    <bean id="song1" class="com.apress.prospring5.ch3.Song"
        p:title-ref="parentTitle"/>

    <bean id="song2" class="com.apress.prospring5.ch3.Song"
        p:title-ref="childTitle"/>

    <bean id="song3" class="com.apress.prospring5.ch3.Song">
        <property name="title">
            <ref parent="childTitle"/>
        </property>
    </bean>
    <bean id="childTitle" class="java.lang.String"
          c:_0="No Such Thing"/>
</beans>
```

Обратите внимание на то, что здесь определены четыре компонента Spring Beans. Компонент `childTitle` в этом фрагменте кода конфигурации похож на аналогичный компонент `childTitle` в родительском контексте, за исключением отличия в строковом значении типа `String`, указывающим на его размещение в порожденном контексте типа `ApplicationContext`.

В компоненте `song1` атрибут `ref` из дескриптора `<bean>` служит для ссылки на компонент `parentTitle`. А поскольку этот компонент существует только в родительской фабрике типа `BeanFactory`, то компонент `song1` получает ссылку на него. Здесь любопытно отметить два обстоятельства. Во-первых, атрибут `bean` можно применять для ссылки на компоненты Spring Beans как в порожденном, так и в родительском контексте типа `ApplicationContext`. Это позволяет легко и прозрачно ссылаться на компоненты Spring Beans, чтобы перемещать их между файлами конфигурации по мере того, как приложение разрастается. И во-вторых, атрибут `local`

нельзя применять для ссылки на компоненты Spring Beans в родительском контексте типа `ApplicationContext`. Синтаксический анализатор проверяет, существует ли в том же самом XML-файле конфигурации допустимый элемент разметки, соответствующий значению, указанному в атрибуте `local`, и тем самым предотвращается его употребление для ссылки на компоненты Spring Beans в родительском контексте.

В компоненте `song2` атрибут `ref` из дескриптора `<bean>` служит для ссылки на компонент `childTitle`. А поскольку этот компонент определен в обоих контекстах типа `ApplicationContext`, то компонент `song2` получает ссылку на компонент `childTitle` в собственном контексте типа `ApplicationContext`.

В компоненте `song3` дескриптор разметки `<ref>` служит для ссылки на компонент `childTitle` непосредственно из родительского контекста типа `ApplicationContext`. Но поскольку в этом компоненте применяется атрибут `parent` из дескриптора разметки `<ref>`, то экземпляр компонента `childTitle`, объявленный в порожденном контексте типа `ApplicationContext`, полностью игнорируется.

На заметку Вы, возможно, обратили внимание на то, что, в отличие от компонентов **`song1`** и **`song2`**, в компоненте **`song3`** не используется пространство имен **`p`**. Несмотря на то что пространство имен **`p`** доставляет удобные сокращения, оно все же не предоставляет все возможности, доступные в том случае, если применяются дескрипторы **`<property>`**, включая ссылку на родительский компонент Spring Bean. И хотя мы привели данное пространство имен в качестве примера, для определения компонентов Spring Beans лучше выбрать что-нибудь одно: пространство имен **`p`** или дескрипторы разметки **`<property>`**, а не оба способа конфигурирования вместе, кроме крайней необходимости в этом.

Ниже приведен результат, выводимый на консоль при выполнении исходного кода из класса `HierarchicalAppContextUsage`. Как и предполагалось, компоненты `song1` и `song3` получают ссылку на компоненты из родительского контекста типа `ApplicationContext`, тогда как компонент `song2` — ссылку на компонент из порожденного контекста типа `ApplicationContext`.

```
from parent ctx: Gravity
from child ctx: No Such Thing
from parent ctx: Daughters
```

Внедрение коллекций

Нередко компонентам Spring Beans требуется доступ к коллекциям объектов, а не только к отдельным компонентам или значениям. Поэтому нет ничего удивительного в том, что коллекции объектов можно внедрять в компоненты Spring Beans. Применить коллекцию совсем не трудно: достаточно выбрать сначала дескриптор разметки `<list>`, `<map>`, `<set>` или `<props>` для представления экземпляра коллекции типа `List`, `Map`, `Set` или `Properties`, а затем передать отдельные ее элементы, как это делается при любом другом внедрении зависимостей. Дескриптор разметки `<props>` позволяет передавать только строковые значения, потому что в свойствах класса

Properties допускаются только значения типа String. А в дескрипторах <list>, <map> и <set> можно использовать любой дескриптор разметки при внедрении зависимостей в свойство и даже дескриптор разметки другой коллекции. Это дает возможность передавать список типа List, состоящий из отображений типа Map, отображение типа Map, состоящее из множеств типа Set, и даже список типа List, состоящий из элементов коллекции типа Map, каждый из которых состоит из элементов коллекции типа Set, а те, в свою очередь, — из элементов коллекции типа List! В следующем фрагменте кода приведен класс, в который можно внедрить все четыре типа коллекций:

```
package com.apress.prospring5.ch3.xml;

import org.springframework.context.support
          .GenericXmlApplicationContext;

import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.Set;

public class CollectionInjection {

  private Map<String, Object> map;
  private Properties props;
  private Set set;
  private List list;

  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();
    CollectionInjection instance = (CollectionInjection)
                        ctx.getBean("injectCollection");
    instance.displayInfo();
    ctx.close();
  }

  public void displayInfo() {
    System.out.println("Map contents:\n");
    map.entrySet().stream()
       .forEach(e -> System.out.println("Key: "
             + e.getKey() + " - Value: " + e.getValue()));

    System.out.println("\nProperties contents:\n");
    props.entrySet().stream()
         .forEach(e -> System.out.println("Key: "
             + e.getKey() + " - Value: " + e.getValue()));
```

```
    System.out.println("\nSet contents:\n");
    set.forEach(obj ->
                System.out.println("Value: " + obj));

    System.out.println("\nList contents:\n");
    list.forEach(obj ->
                 System.out.println("Value: " + obj));
  }

  public void setList(List list) {
    this.list = list;
  }

  public void setSet(Set set) {
    this.set = set;
  }

  public void setMap(Map<String, Object> map) {
    this.map = map;
  }

  public void setProps(Properties props) {
    this.props = props;
  }
}
```

И хотя в приведенном выше фрагменте немало кода, он, по существу, мало что делает. Так, в методе `main()` сначала из Spring извлекается компонент типа `CollectionInjection`, а затем вызывается метод `displayInfo()`. Этот метод просто выводит содержимое экземпляров коллекций типа `Map`, `Properties`, `Set` и `List`, которые предстоит внедрить из Spring.

Ниже приведена конфигурация, требующаяся для внедрения значений из коллекций в каждое из свойств класса `CollectionInjection` и сохраняемая в файле `app-context-xml.xml`. Обратите внимание на объявление свойства `Map<String, Object>`. Для комплекта JDK, начиная с версии 5, в Spring поддерживается также строго типизированное объявление интерфейса `Collection`, и будет выполняться преобразование конфигурации формата XML в указанный соответственно тип.

```
<beans ...>
  <bean id="lyricHolder"
    lass="com.apress.prospring5.ch3.xml.LyricHolder"/>

  <bean id="injectCollection"
    class="com.apress.prospring5.ch3.xml
              .CollectionInjection">

    <property name="map">
      <map>
        <entry key="someValue">
```

```
        <value>It's a Friday, we finally made it</value>
      </entry>
      <entry key="someBean">
        <ref bean="lyricHolder"/>
      </entry>
    </map>
  </property>
  <property name="props">
    <props>
      <prop key="firstName">John</prop>
      <prop key="secondName">Mayer</prop>
    </props>
  </property>
  <property name="set">
    <set>
      <value>I can't believe I get to see
             your face</value>
      <ref bean="lyricHolder"/>
    </set>
  </property>
    <property name="list">
      <list>
        <value>You've been working and I've
               been waiting</value>
        <ref bean="lyricHolder"/>
      </list>
  </property>
 </bean>
</beans>
```

В приведенном выше коде конфигурации значения из коллекций внедряются во все четыре метода установки, доступные в классе `CollectionInjection`. Так, в свойство `map` внедряется экземпляр коллекции типа `Map` с помощью дескриптора `<map>`. Каждый элемент данной коллекции указывается с помощью дескриптора разметки `<entry>`, и у каждого ее элемента имеется ключ типа `String` и вводимое значение, которое может быть любым отдельно внедряемым в данное свойство. В данном примере демонстрируется применение дескрипторов разметки `<value>` и `<ref>` для ввода в отображение типа `Map` значения типа `String` и ссылки на компонент Spring Bean.

Ниже приведен класс `LyricHolder`, представляющий компонент Spring Bean типа `lyricHolder`, внедряемый в отображение согласно приведенной выше конфигурации.

```
package com.apress.prospring5.ch3.xml;

import com.apress.prospring5.ch3.ContentHolder;

public class LyricHolder implements ContentHolder {
```

```
  private String value =
          "'You be the DJ, I'll be the driver'";

  @Override public String toString() {
    return "LyricHolder: { " + value + "}";
  }
}
```

Для конфигурирования свойства `props` в данном случае применяется дескриптор `<props>` с целью получить экземпляр типа `java.util.Properties` и заполнить его с помощью дескрипторов `<prop>`. Обратите внимание на то, что для каждого свойства экземпляра типа `Properties` можно указать только строковое значение типа `String`, несмотря на то, что дескрипторы разметки `<prop>` снабжаются ключами подобно дескрипторам `<entry>`.

Кроме того, элемент разметки `<map>` служит для альтернативного и более компактного конфигурирования с помощью атрибутов `value` и `value-ref` вместо элементов разметки `<value>` и `<ref>`. Объявляемое ниже свойство `map` равнозначно аналогичному свойству из предыдущей конфигурации.

```
<property name="map">
  <map>
    <entry key="someValue"
           value="It's a Friday, we finally made it"/>

    <entry key="someBean"
           value-ref="lyricHolder"/>
  </map>
</property>
```

Дескрипторы `<list>` и `<set>` действуют совершенно одинаково: каждый элемент коллекции указывается в них с помощью любых дескрипторов значений, в том числе `<value>` и `<ref>`, которые служат для внедрения единственного значения в свойство. В приведенной выше конфигурации значение типа `String` было введено вместе со ссылкой на компонент Spring Bean в экземпляры коллекций типа `List` и `Set`.

Ниже приведен результат, выводимый методом `main()` из класса `CollectionInjection`. Как и следовало ожидать, на консоль просто выводится список элементов, введенных в коллекции, указанные в файле конфигурации.

```
Map contents:[4]

Key:[5] someValue - Value:[6] It's a Friday, we finally made it
Key: someBean - Value: LyricHolder:
     { 'You be the DJ, I'll be the driver'}
```

[4] Содержимое коллекции типа Map:

[5] Ключ:

[6] Значение:

```
Properties contents:[7]

Key: secondName - Value: Mayer
Key: firstName - Value: John

Set contents:[8]

Value: I can't believe I get to see your face
Value: LyricHolder: { 'You be the DJ, I'll be the driver'}

List contents:[9]

Value: You've been working and I've been waiting
Value: LyricHolder: { 'You be the DJ, I'll be the driver'}
```

Напомним, что в элементах разметки <list>, <map> и <set> можно задействовать любой из дескрипторов, применяемых при установке значений свойств, не относящихся непосредственно к коллекциям, чтобы указать значение одного из элементов в коллекции. Это весьма эффективный принцип, поскольку он не ограничивает ваши возможности одним лишь внедрением коллекций, состоящих из значений примитивных типов данных. Вы можете также внедрять коллекции компонентов Spring Beans или другие коллекции.

Такие функциональные возможности позволяют намного проще разбить разрабатываемое приложение на отдельные модули, чтобы предоставлять различные выбираемые пользователем реализации основных составляющих логики приложения. Рассмотрим в качестве примера систему, которая позволяет сотрудникам организации создавать, корректировать и заказывать канцелярские принадлежности для индивидуальных нужд в оперативном режиме. В такой системе подготовленное графическое изображение каждого заказа отправляется на готовый к работе принтер. Единственная сложность связана с тем, что одни принтеры ожидают получения графических изображений по электронной почте, другие — через соединение по протоколу FTP, а третьи — по протоколу защищенного копирования (Secure Copy Protocol — SCP). Применяя механизм внедрения коллекций в Spring, можно создать стандартный интерфейс для подобных функциональных возможностей:

```
package com.apress.prospring5.ch3;

public interface ArtworkSender {
  void sendArtwork(String artworkPath, Recipient recipient);
  String getFriendlyName();
  String getShortName();
}
```

---

[7] Содержимое коллекции типа Properties:

[8] Содержимое коллекции типа Set:

[9] Содержимое коллекции типа List:

В приведенном выше фрагменте кода класс `Recipient` пустой. Для интерфейса `ArtworkSender` можно создать целый ряд реализаций, каждая из которых способна описать себя самостоятельно, как показано в следующем примере:

```
package com.apress.prospring5.ch3;

public class FtpArtworkSender
       implements ArtworkSender {

  @Override
  public void sendArtwork(String artworkPath,
                          Recipient recipient) {
    // здесь следует логика обмена данными
    // по сетевому протоколу FTP...
  }

  @Override
  public String getFriendlyName() {
    return "File Transfer Protocol";
  }

  @Override
  public String getShortName() {
    return "ftp";
  }
}
```

Допустим, что далее разрабатывается класс `ArtworkManager`, в котором поддерживаются все доступные реализации интерфейса `ArtworkSender`. Имея такие реализации, достаточно передать коллекцию типа `List` классу `ArtworkManager`. А сконфигурировав каждый шаблон заказа канцелярских принадлежностей, можно вызвать `getFriendlyName()`, чтобы отобразить для системного администратора список вариантов доставки. Кроме того, рассматриваемая здесь система заказов может оставаться полностью отвязанной от отдельных реализаций, если требуется лишь написать код для интерфейса `ArtworkSender`. Реализацию класса `ArtworkManager` мы оставляем вам в качестве упражнения.

Помимо конфигурации в формате XML, для внедрения коллекций можно пользоваться аннотациями. Но ради простоты сопровождения имеет смысл также вынести значения из коллекций в файл конфигурации. В следующем фрагменте кода демонстрируется конфигурация четырех разных компонентов Spring Beans, имитирующих те же свойства коллекций, что и в предыдущем примере (эта конфигурация находится в файле `app-context-annotation.xml`).

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org
```

```
                        /schema/context"
      xmlns:util="http://www.springframework.org
                        /schema/util"
      xsi:schemaLocation="http://www.springframework.org
                              /schema/beans
         http://www.springframework.org/schema/beans
              /spring-beans.xsd
         http://www.springframework.org/schema/context
         http://www.springframework.org/schema/context
              /spring-context.xsd
         http://www.springframework.org/schema/util
         http://www.springframework.org/schema/util
              /spring-util.xsd">

    <context:component-scan
      base-package="com.apress.prospring5.ch3.annotated"/>

    <util:map id="map" map-class="java.util.HashMap">
      <entry key="someValue"
             value="It's a Friday, we finally made it"/>
      <entry key="someBean"
             value-ref="lyricHolder"/>
    </util:map>

    <util:properties id="props">
      <prop key="firstName">John</prop>
      <prop key="secondName">Mayer</prop>
    </util:properties>

    <util:set id="set" set-class="java.util.HashSet">
      <value>I can't believe I get to see your face</value>
      <ref bean="lyricHolder"/>
    </util:set>

    <util:list id="list" list-class="java.util.ArrayList">
      <value>You've been working and I've
             been waiting</value>
      <ref bean="lyricHolder"/>
    </util:list>
</beans>
```

А теперь разработаем версию класса `LyricHolder` с аннотациями, исходный код которого приведен ниже.

```
package com.apress.prospring5.ch3.annotated;

import com.apress.prospring5.ch3.ContentHolder;
import org.springframework.stereotype.Service;
```

```
@Service("lyricHolder")
public class LyricHolder implements ContentHolder {
  private String value =
          "'You be the DJ, I'll be the driver'";

  @Override public String toString() {
    return "LyricHolder: { " + value + "}";
  }
}
```

В приведенной ранее конфигурации пространство имен `util`, предоставляемое в Spring, применяется для объявления компонентов Spring Beans, предназначенных для хранения свойств коллекций. По сравнению с предыдущими версиями Spring это позволяет значительно упростить конфигурацию. В классе, предназначенном для проверки конфигурации, внедряются упомянутые ранее компоненты Spring Beans, а в аннотациях `@Resource`, применяемых по спецификации JSR-250, имена этих компонента указываются в качестве аргументов для правильного их распознавания. И, наконец, метод `displayInfo()` объявляется, как и прежде, поэтому он не показан ниже. Если выполнить программу проверки конфигурации, то результат получится такой же, как и в примере с конфигурацией в формате XML.

```
@Service("injectCollection")
public class CollectionInjection {
  @Resource(name="map")
  private Map<String, Object> map;

  @Resource(name="props")
  private Properties props;

  @Resource(name="set")
  private Set set;

  @Resource(name="list")
  private List list;

  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-annotation.xml");
    ctx.refresh();

    CollectionInjection instance = (CollectionInjection)
                        ctx.getBean("injectCollection");
    instance.displayInfo();

    ctx.close();
  }
  ...
}
```

На заметку У вас может невольно возникнуть вопрос: почему вместо аннотации **@Autowired** в данном случае применяется аннотация **@Resource**? Дело в том, что аннотация **@Autowired** семантически определена таким образом, чтобы всегда интерпретировать массивы, коллекции и отображения как наборы соответствующих компонентов Spring Beans с целевым типом, производным от объявленного типа значений в коллекции. Так, если в классе имеется свойство типа **List<ContentHolder>** и определена аннотация **@Autowired**, то каркас Spring попытается внедрить все компоненты Spring Beans типа **ContentHolder** из текущего контекста типа **ApplicationContext** в это свойство (вместо свойства **<util:list>**, объявленного в файле конфигурации), что приведет к внедрению неожиданных зависимостей или к генерированию в Spring исключения, если компоненты Spring Beans типа **ContentHolder** не определены. Таким образом, для внедрения типа коллекции придется явно сообщить каркасу Spring о необходимости произвести внедрение, указав имя компонента Spring Bean, что и поддерживается в аннотации **@Resource**.

Для тех же самых целей можно применить аннотации **@Autowired** и **@Qualifier** в определенном сочетании, но, как правило, их лучше употреблять по одной, а не по две. В приведенном ниже фрагменте кода демонстрируется равнозначная конфигурация для внедрения коллекции по имени ее компонента Spring Bean и с помощью аннотаций **@Autowired** и **@Qualifier**.

```
package com.apress.prospring5.ch3.annotated;

import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.beans.factory
          .annotation.Qualifier;

@Service("injectCollection")
public class CollectionInjection {

  @Autowired
  @Qualifier("map")
  private Map<String, Object> map;
  ...
}
```

Внедрение зависимостей через метод класса

Помимо внедрения зависимостей через конструктор и метод установки, в Spring предоставляется еще одна менее часто применяемая разновидность внедрения зависимостей, называемая *внедрением зависимостей через метод класса*. Возможности такого внедрения зависимостей доступны в двух слабо связанных формах: через метод поиска и замену метода. В форме внедрения зависимостей через метод поиска поддерживается еще один механизм, с помощью которого компонент Spring Bean может получить одну из своих зависимостей. А форма замены метода позволяет произвольно заменять реализацию любого метода из компонента Spring Bean, не изменяя первоначально написанный исходный код. Для поддержки этих двух форм внедрения

зависимостей через метод в Spring используются возможности библиотеки CGLIB[10] для динамического расширения байт-кода.

Внедрение зависимостей через метод поиска

Внедрение зависимостей через метод поиска было введено в версии Spring 1.1 для преодоления затруднений, возникающих в том случае, если один компонент Spring Bean зависит от другого компонента с отличающимся жизненным циклом, и, в частности, когда одиночный объект, иначе называемый "одиночкой", зависит от неодиночного объекта. В подобных случаях внедрение зависимостей через конструктор и метод приводит к тому, что одиночный компонент Spring Bean поддерживает единственный экземпляр того, что должно быть неодиночным компонентом. Но иногда требуется, чтобы одиночный компонент Spring Bean получал при необходимости новый экземпляр неодиночного компонента.

Рассмотрим случай, когда класс `LockOpener` предоставляет услуги для открытия любого шкафчика. С этой целью класс `LockOpener` полагается на внедренный в него класс `KeyHelper`. Но структура класса `KeyHelper` предусматривает наличие ряда внутренних состояний, что делает этот класс непригодным для повторного использования. Всякий раз, когда вызывается метод `openLock()`, требуется новый экземпляр типа `KeyHelper`. В этом случае объект типа `LockOpener` оказывается одиночным. Но если внедрить класс `KeyHelper` с помощью обычного механизма, то повторно будет использоваться тот же самый его экземпляр, который был получен, когда каркас Spring внедрял данный класс в первый раз. Чтобы обеспечить передачу нового экземпляра класса `KeyHelper` методу `openLock()` при каждом его вызове, необходимо внедрить данный класс через метод поиска.

Как правило, этого можно добиться, реализовав в одиночном компоненте Spring Bean интерфейс `ApplicationContextAware`, который рассматривается в следующей главе. После этого одиночный компонент Spring Bean сможет искать новый экземпляр неодиночной зависимости всякий раз, когда он потребуется, используя для этой цели экземпляр типа `ApplicationContext`. Внедрение зависимостей через метод поиска позволяет объявить в одиночном компоненте Spring Bean, что ему требуется неодиночная зависимость и что он будет получать новый экземпляр неодиночного компонента Spring Bean всякий раз, когда ему приходится взаимодействовать с ним, не прибегая к реализации любого характерного для Spring интерфейса.

Внедрение зависимостей через метод поиска действует по принципу объявления в одиночном компоненте Spring Bean метода поиска, возвращающего экземпляр неодиночного компонента Spring Bean. Когда в приложении получается ссылка на одиночный компонент Spring Bean, на самом деле эта ссылка на динамически созданный подкласс, в котором реализован метод поиска средствами Spring. Типичная реализа-

[10] CGLIB (или `cglib`) — это эффективная, высокопроизводительная библиотека, предназначенная для генерирования высококачественного кода. Она способна динамически расширять классы Java и реализовывать интерфейсы во время выполнения. Ее исходный код открыт для свободного доступа в официальном хранилище по адресу `https://github.com/cglib`.

ция включает определение метода поиска, а следовательно, и класса компонента, как абстрактного. Благодаря этому предотвращается возникновение странных ошибок в том случае, если забыть сконфигурировать внедрение зависимостей через метод класса и пользоваться непосредственно классом компонента Spring Bean с пустой реализацией метода, а не с расширенным в Spring подклассом. Это довольно сложный вопрос, который лучше разъяснить на конкретном примере.

В этом примере создается неодиночный компонент Spring Bean и два одиночных компонента, реализующих тот же самый интерфейс. Первый одиночный компонент Spring Bean получает экземпляр неодиночного компонента, применяя традиционный подход к внедрению зависимостей через метод установки, а второй одиночный компонент — через метод класса. Ниже приведен класс `Singer`, который в данном примере относится к типу неодиночного компонента Spring Bean.

```
package com.apress.prospring5.ch3;

public class Singer {
  private String lyric = "I played a quick game of chess "
                       + "with the salt and pepper shaker";

  public void sing() {
    // закомментировано, поскольку оскверняет
    // вывод на консоль
    // System.out.println(lyric);
  }
}
```

Этот класс не особенно впечатляет, но он прекрасно подходит для рассматриваемого здесь примера. Ниже приведен интерфейс `DemoBean`, который реализуется в классах обоих одиночных компонентов Spring Beans.

```
package com.apress.prospring5.ch3;

public interface DemoBean {
  Singer getMySinger();
  void doSomething();
}
```

В интерфейсе `DemoBean` объявлены два метода: `getMySinger()` и `doSomethi ng()`. В рассматриваемом здесь примере приложения метод `getMySinger()` применяется для получения ссылки на экземпляр типа `Singer`, а если это компонент Spring Bean с методом поиска, то для выполнения конкретного поиска. Что же касается метода `doSomething()`, то действие этого простого метода зависит от класса `Singer`. В следующем фрагменте кода приведен класс `StandardLookupDemoBean`, в котором внедрение зависимостей через метод установки служит для получения экземпляра класса `Singer`:

```
package com.apress.prospring5.ch3;

public class StandardLookupDemoBean
       implements DemoBean {
  private Singer mySinger;

  public void setMySinger(Singer mySinger) {
    this.mySinger = mySinger;
  }

  @Override
  public Singer getMySinger() {
    return this.mySinger;
  }

  @Override
  public void doSomething() {
    mySinger.sing();
  }
}
```

Исходный код данного класса должен быть вам уже знаком, но обратите внимание на то, что для выполнения необходимых действий в методе `doSomething()` применяется сохраненный экземпляр класса `Singer`. Ниже приведен класс `AbstractLookupDemoBean`, в котором внедрение зависимостей через метод класса служит для получения экземпляра класса `Singer`.

```
package com.apress.prospring5.ch3;

public abstract class AbstractLookupDemoBean
       implements DemoBean {

  public abstract Singer getMySinger();

  @Override
  public void doSomething() {
    getMySinger().sing();
  }
}
```

Обратите внимание на то, что метод `getMySinger()` объявлен как абстрактный и вызывается в теле метода `doSomething()` для получения экземпляра класса `Singer`. Ниже приведена конфигурация в формате XML, требующаяся для данного примера (она находится в файле `app-context-xml.xml`).

```
<beans ...>
  <bean id="singer"
    class="com.apress.prospring5.ch3.Singer"
    scope="prototype"/>
```

```
  <bean id="abstractLookupBean"
    class="com.apress.prospring5.ch3
              .AbstractLookupDemoBean">
    <lookup-method name="getMySinger" bean="singer"/>
  </bean>

  <bean id="standardLookupBean"
    class="com.apress.prospring5.ch3
              .StandardLookupDemoBean">
    <property name="mySinger" ref="singer"/>
  </bean>
</beans>
```

Конфигурация для компонентов `singer` и `standardLookupBean` должна быть вам уже знакомой. Для компонента `abstractLookupBean` необходимо сконфигурировать метод поиска в дескрипторе `<lookup-method>`. Атрибут `name` в дескрипторе `<lookup-method>` сообщает каркасу Spring имя метода поиска для компонента Spring Bean, чтобы переопределить его. Этот метод не должен принимать любые аргументы, а возвращать — тип компонента Spring Bean. В данном случае этот метод должен возвращать объект класса `Singer` или какого-нибудь из его подклассов. Атрибут `bean` указывает Spring, какой именно компонент Spring Bean должен возвращать метод поиска.

Ниже приведен завершающий фрагмент кода из рассматриваемого здесь примера. В нем определяется главный класс с методом `main()` для выполнения исходного кода из данного примера.

```
package com.apress.prospring5.ch3;

import org.springframework.context.support
          .GenericXmlApplicationContext;
import org.springframework.util.StopWatch;

public class LookupDemo {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();

    DemoBean abstractBean =
        ctx.getBean("abstractLookupBean", DemoBean.class);
     DemoBean standardBean =
         ctx.getBean("standardLookupBean", DemoBean.class);

    displayInfo("abstractLookupBean", abstractBean);
    displayInfo("standardLookupBean", standardBean);

    ctx.close();
```

```
  }

  public static void displayInfo(String beanName,
                                  DemoBean bean) {
    Singer singer1 = bean.getMySinger();
    Singer singer2 = bean.getMySinger();
    System.out.println("" + beanName + ": "
                    + "Singer Instances the Same? "
                    + (singer1 == singer2));
    StopWatch stopWatch = new StopWatch();
    stopWatch.start("lookupDemo");
    for (int x = 0; x < 100000; x++) {
      Singer singer = bean.getMySinger();
      singer.sing();
    }
    stopWatch.stop();
    System.out.println("100000 gets took "
              + stopWatch.getTotalTimeMillis() + " ms");
  }
}
```

Как следует из приведенного выше фрагмента кода, компоненты `abstractLookupBean` и `standardLookupBean` извлекаются из контекста типа `GenericXmlApplicationContext`, а ссылки на них передаются методу `displayInfo()`. Получение экземпляра абстрактного класса поддерживается лишь в том случае, если зависимости внедряются через метод поиска, и тогда в Spring применяется библиотека CGLIB для формирования подкласса, производного от класса `AbstractLookupDemoBean`, где метод поиска переопределяется динамически. В первой части метода `displayInfo()` создаются две локальные переменные типа `Singer` и каждой из них присваивается значение с помощью метода `getMySinger()`, вызываемого для переданного компонента Spring Bean. С помощью этих двух переменных на консоль выводится сообщение, извещающее, указывают ли две ссылки на один и тот же объект.

Что касается компонента `abstractLookupBean`, то новый экземпляр типа `Singer` должен извлекаться при каждом вызове метода `getMySinger()`, и поэтому ссылки не должны быть одинаковыми. А что касается компонента `standardLookupBean`, то одиночный экземпляр типа `Singer` передается этому компоненту путем внедрения зависимостей через метод установки. Этот экземпляр сохраняется и возвращается при каждом вызове метода `getMySinger()`, и поэтому обе ссылки должны совпадать.

На заметку Класс **`StopWatch`**, применяемый в предыдущем примере, является служебным классом, доступным в Spring. Этот класс очень удобен в тех случаях, когда требуется оценить производительность и протестировать разрабатываемые приложения.

В завершающей части метода `getMySinger()` выполняется простой тест производительности, чтобы выяснить, какой из компонентов Spring Beans действует быстрее. Очевидно, что компонент `standardLookupBean` должен действовать быстрее, потому что всякий раз он возвращает один и тот же экземпляр, но интересно увидеть эту разницу в быстродействии.

А теперь можно выполнить исходный код класса `LookupDemo` для целей тестирования. Ниже приведен результат, получаемый из рассматриваемого здесь примера.

```
[abstractLookupBean]: Singer Instances the Same? false
100000 gets took 431 ms
[standardLookupBean]: Singer Instances the Same? true
100000 gets took 1 ms[11]
```

Нетрудно заметить, что экземпляры типа `Singer`, как и ожидалось, оказываются одинаковыми, если применяется компонент `standardLookupBean`, но разными, если применяется компонент `abstractLookupBean`. Отличия в производительности разительны, хотя и они вполне ожидаемы.

Безусловно, имеется равнозначный способ сконфигурировать упомянутые выше компоненты Spring Beans с помощью аннотаций. Так, в компонент `singer` необходимо ввести дополнительную аннотацию, чтобы обозначить область видимости `prototype`, как показано ниже.

```
package com.apress.prospring5.ch3.annotated;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component("singer)
@Scope("prototype")
public class Singer {
  private String lyric = "I played a quick game of chess "
                       + "with the salt and pepper shaker";

  public void sing() {
    // закомментировано, поскольку оскверняет
    // вывод на консоль
    //System.out.println(lyric);
  }
}
```

Класс `AbstractLookupDemoBean` теперь не является абстрактным, а метод `getMySinger()` имеет пустое тело и снабжается аннотацией `@Lookup`, принимающей

[11] Компонент `[abstractLookupBean]`: экземпляры типа Singer одинаковы? Неверно.
На 100000 попыток получения ушла 431 мс.
Компонент `[standardLookupBean]`: экземпляры типа Singer одинаковы? Верно.
На 100000 попыток получения ушла 1 мс.

в качестве аргумента имя компонента `singer`, как показано ниже. Тело данного метода будет переопределено в динамически формируемом подклассе.

```
package com.apress.prospring5.ch3.annotated;

import org.springframework.beans.factory.annotation.Lookup;
import org.springframework.stereotype.Component;

@Component("abstractLookupBean")
public class AbstractLookupDemoBean implements DemoBean {
  @Lookup("singer")
  public Singer getMySinger() {
    return null; // переопределяется автоматически
  }

  @Override
  public void doSomething() {
    getMySinger().sing();
  }
}
```

Класс `StandardLookupDemoBean` должен быть снабжен лишь аннотацией `@Component`, а метод `setMySinger()` — аннотациями `@Autowired` и `@Qualifier` для внедрения компонента `singer`:

```
package com.apress.prospring5.ch3.annotated;

import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.beans.factory
          .annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component("standardLookupBean")
public class StandardLookupDemoBean implements DemoBean {

  private Singer mySinger;

  @Autowired
  @Qualifier("singer")
  public void setMySinger(Singer mySinger) {
    this.mySinger = mySinger;
  }

  @Override
  public Singer getMySinger() {
    return this.mySinger;
  }
```

```
  @Override
  public void doSomething() {
    mySinger.sing();
  }
}
```

В файле конфигурации `app-context-annotated.xml` потребуется лишь активизировать режим просмотра компонентов из пакета, содержащего классы, снабженные аннотациями:

```
<beans ...>
  <context:component-scan
    base-package="com.apress.prospring5.ch3.annotated"/>
</beans>
```

Для выполнения исходного кода из данного примера по-прежнему служит класс `LookupDemo`. Единственное отличие от предыдущего примера состоит в том, что XML-файл конфигурации указывается в качестве аргумента при создании объекта типа `GenericXmlApplicationContext`.

Если же требуется полностью избавиться от XML-файлов, это можно сделать с помощью конфигурационного класса, в котором следует активизировать режим просмотра компонентов из пакета `com.apress.prospring5.ch3.annotated`. И этот класс можно объявить именно там, где он требуется, а в данном случае — в классе, выполняемом для проверки компонентов Spring Beans, как показано ниже. Подробнее об альтернативных способах конфигурирования с помощью аннотаций и классов Java речь пойдет в главе 4.

```
package com.apress.prospring5.ch3.config;

import com.apress.prospring5.ch3.annotated.DemoBean;
import com.apress.prospring5.ch3.annotated.Singer;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.context.support
          .GenericApplicationContext;
import org.springframework.util.StopWatch;
import java.util.Arrays;

public class LookupConfigDemo {

  @Configuration
  @ComponentScan(basePackages =
                 {"com.apress.prospring5.ch3.annotated"})
  public static class LookupConfig {}
```

```
  public static void main(String... args) {
    GenericApplicationContext ctx =
          new AnnotationConfigApplicationContext(
              LookupConfig.class);

    DemoBean abstractBean =
        ctx.getBean("abstractLookupBean", DemoBean.class);
    DemoBean standardBean =
        ctx.getBean("standardLookupBean", DemoBean.class);

    displayInfo("abstractLookupBean", abstractBean);
    displayInfo("standardLookupBean", standardBean);

    ctx.close();
  }

  public static void displayInfo(String beanName,
                                 DemoBean bean) {
    // такая же самая реализация, как и прежде
    ...
  }
}
```

Соображения по поводу внедрения зависимостей через метод поиска

Внедрение зависимостей через метод поиска предназначено для тех случаев, когда требуется работать с двумя компонентами Spring Beans, имеющими разные жизненные циклы. Не поддавайтесь искушению применять внедрение зависимостей через метод поиска, если компоненты Spring Beans разделяют один и тот же жизненный цикл, особенно когда они являются одиночными. В результатах выполнения примера из предыдущего подраздела наглядно показаны заметные отличия в производительности между внедрением через метод поиска для получения новых экземпляров зависимости и стандартным внедрением для получения одиночного экземпляра зависимости. Кроме того, внедрением зависимостей через метод поиска не следует пользоваться без особой нужды, даже если имеются компоненты Spring Beans с разными жизненными циклами.

Допустим, имеются три одиночных компонента Spring Beans, разделяющих общую зависимость. Каждый одиночный компонент Spring Bean должен иметь собственный экземпляр зависимости, поэтому зависимость создается как неодиночный объект, но требуется, чтобы каждый одиночный компонент Spring Bean использовал один и тот же экземпляр взаимодействующего объекта на протяжении всего срока своего действия. В таком случае идеальным решением будет внедрение зависимостей через метод установки, а внедрение зависимостей через метод поиска лишь повлечет за собой нежелательные издержки.

Применяя внедрение зависимостей через метод поиска при построении классов, необходимо принимать во внимание ряд принципов проектирования. В представленных ранее примерах метод поиска объявлялся в интерфейсе. И это было сделано лишь для того, чтобы не дублировать метод `displayInfo()` в двух разных типах компонентов Spring Beans. Как упоминалось ранее, в общем случае не следует засорять бизнес-интерфейс ненужными определениями, предназначенными исключительно для инверсии управления. Необходимо также иметь в виду следующее обстоятельство: несмотря на то, что объявлять метод поиска абстрактным совсем не обязательно, это все же поможет не забыть сконфигурировать его, чтобы случайно не воспользоваться пустой его реализацией. Это, конечно, относится лишь к конфигурированию в формате XML. А конфигурирование с помощью аннотаций само принуждает к пустой реализации метода поиска, иначе компонент Spring Bean просто не будет создан.

Замена метода

Несмотря на то что в документации на Spring замена метода классифицируется как форма внедрения зависимостей, она заметно отличается от того, что было рассмотрено до сих пор, когда внедрение зависимостей применялось исключительно для предоставления компонентов Spring Beans и взаимодействующих с ними объектов. Применяя замену метода, можно произвольно заменить реализацию любого метода в каком угодно компоненте Spring Bean, не внося коррективы в исходный код этого компонента. Допустим, что в приложении применяется сторонняя библиотека и требуется изменить логику действия определенного метода. Но изменить исходный код нельзя, поскольку он предоставлен сторонней организацией, и поэтому единственным решением будет замена метода, позволяющая внедрить собственную реализацию вместо логики действия данного метода.

Внутренне это достигается путем динамического создания подкласса, производного от класса компонента Spring Bean. Применяя библиотеку CGLIB, обращения к заменяемому методу следует переадресовать другому компоненту Spring Bean, в котором реализуется интерфейс `MethodReplacer`. В следующем примере кода демонстрируется простой компонент Spring Bean, в котором объявлены две перегружаемые версии метода `formatMessage()`:

```
package com.apress.prospring5.ch3;

public class ReplacementTarget {
  public String formatMessage(String msg) {
    return "<h1>" + msg + "</h1>";
  }

  public String formatMessage(Object msg) {
    return "<h1>" + msg + "</h1>";
  }
}
```

Используя функциональные средства замены метода в Spring, можно заменить любой метод из класса `ReplacementTarget`. В рассматриваемом здесь примере демонстрируется, как заменить метод `formatMessage(String)`, а также сравнивается производительность замененного и исходного методов.

Чтобы заменить метод, необходимо реализовать сначала интерфейс `MethodReplacer`, как демонстрируется в следующем примере кода:

```
package com.apress.prospring5.ch3;

import org.springframework.beans.factory.support
          .MethodReplacer;
import java.lang.reflect.Method;

public class FormatMessageReplacer
       implements MethodReplacer {

  @Override
  public Object reimplement(Object arg0, Method method,
                Object... args) throws Throwable {
    if (isFormatMessageMethod(method)) {
      String msg = (String) args0;
      return "<h2>" + msg + "</h2>";
    } else {
      throw new IllegalArgumentException(
            "Unable to reimplement method "
          + method.getName());
    }
  }

  private boolean isFormatMessageMethod(Method method) {
    if (method.getParameterTypes().length != 1) {
      return false;
    }

    if (!("formatMessage".equals(method.getName()))) {
      return false;
    }

    if (method.getReturnType() != String.class) {
      return false;
    }

    if (method.getParameterTypes()[0] != String.class) {
      return false;
    }
    return true;
  }
}
```

У интерфейса `MethodReplacer` имеется единственный метод `reimplement()`, который требуется реализовать. Этому методу передаются три аргумента: компонент Spring Bean, для которого вызывается исходный метод, экземпляр типа `Method`, представляющий переопределяемый метод, а также массив аргументов, передаваемых данному методу. Метод `reimplement()` должен возвращать результат выполнения замененной логики, а тип возвращаемого значения, очевидно, должен быть совместим с возвращаемым типом заменяемого метода.

В классе `FormatMessageReplacer` из приведенного выше примера кода сначала проверяется, переопределяется ли метод `formatMessage(String)`. Если это так, то вызывается заменяющая логика (в данном случае сообщение заключается в дескрипторы разметки `<h2>` и `</h2>`), и отформатированное сообщение возвращается в вызывающий код. И хотя проверять правильность сообщения совсем не обязательно, это все же полезно сделать, если применяется несколько заменяющих методов с похожими аргументами. Благодаря проверке можно избежать ситуации, когда случайно используется другой заменяющий метод с совместимыми аргументами и возвращаемыми типами.

В приведенном ниже примере конфигурации из файла `app-context-xml.xml` показано, что в экземпляре типа `ApplicationContext` определяются два компонента Spring Beans типа `ReplacementTarget`: один — с заменой метода `formatMessage(String)`, а другой — без такой замены.

```
<beans ...>
  <bean id="methodReplacer"
    class="com.apress.prospring5.ch3
              .FormatMessageReplacer"/>

  <bean id="replacementTarget"
    class="com.apress.prospring5.ch3
              .ReplacementTarget">
    <replaced-method name="formatMessage"
                replacer="methodReplacer">
        <arg-type>String</arg-type>
    </replaced-method>
  </bean>

  <bean id="standardTarget"
    class="com.apress.prospring5.ch3
              .ReplacementTarget"/>
</beans>
```

Как видите, реализация интерфейса `MethodReplacer` объявлена как компонент Spring Bean в контексте типа `ApplicationContext`. Затем в дескрипторе `<replaced-method>` производится замена метода `formatMessage(String)` из компонента `replacementTargetBean`. С этой целью в атрибуте `name` дескриптора `<replaced-method>` указывается имя заменяемого метода, а в атрибуте `replacer` —

имя компонента `methodReplacer`, который требуется для замены реализации метода. При наличии перегружаемых методов, как, например, в классе `ReplacementTarget`, в дескрипторе `<arg-type>` можно указать соответствующую сигнатуру метода. Дескриптор `<arg-type>` поддерживает сопоставление с шаблоном, поэтому тип `String` будет соответствовать типу `java.lang.String`, а также типу `java.lang.StringBuffer`.

В следующем фрагменте кода приведено простое демонстрационное приложение, которое извлекает компоненты `standardTarget` и `replacementTarget` из контекста типа `ApplicationContext`, вызывает их методы `formatMessage(String)`, а затем проводит несложный тест производительности, чтобы выяснить, какой из этих методов действует быстрее.

```
package com.apress.prospring5.ch3;

import org.springframework.context.support
          .GenericXmlApplicationContext;
import org.springframework.util.StopWatch;

public class MethodReplacementDemo {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();

    ReplacementTarget replacementTarget =
      (ReplacementTarget) ctx.getBean("replacementTarget");
    ReplacementTarget standardTarget =
      (ReplacementTarget) ctx.getBean("standardTarget");

    displayInfo(replacementTarget);
    displayInfo(standardTarget);

    ctx.close();
  }

  private static void displayInfo(
                ReplacementTarget target) {
    System.out.println(target.formatMessage(
              "Thanks for playing, try again!"));

    StopWatch stopWatch = new StopWatch();
    stopWatch.start("perfTest");
    for (int x = 0; x < 1000000; x++) {
      String out =
             target.formatMessage("No filter in my head");
      // закомментировано, поскольку оскверняет вывод на консоль
```

```
            // System.out.println(out);
        }

        stopWatch.stop();

        System.out.println("1000000 invocations took: "
                + stopWatch.getTotalTimeMillis() + " ms");
    }
}
```

Этот код должен быть вам уже знаком, поэтому мы воздержимся от дальнейших пояснений. На нашей машине выполнение исходного кода из данного примера дало следующий результат:

```
<h2>Thanks for playing, try again!</h2>
1000000 invocations took: 188 ms[12]

<h1>Thanks for playing, try again!</h1>
1000000 invocations took: 24 ms
```

Как и следовало ожидать, результат, выводимый из компонента `replacementTarget`, отражает выполнение переопределенной реализации, предоставленной компонентом `methodReplacer`. Но любопытно, что динамически заменяемый метод действует намного медленнее, чем метод, определяемый статически. Исключение проверки правильности метода из компонента `methodReplacer` приводит лишь к незначительным отличиям в результатах при многих попытках выполнения исходного кода из данного примера. Из этого можно сделать вывод, что основные издержки приходятся на подкласс, производный от класса из библиотеки CGLIB.

Когда целесообразно применять замену метода

Замена метода может оказаться весьма полезной в самых разных обстоятельствах, особенно когда требуется переопределить только отдельный метод в единственном, а не во всех компонентах Spring Beans одного и того же типа. Тем не менее для переопределения методов предпочтительнее пользоваться стандартными механизмами Java, а не полагаться на динамическое расширение байт-кода.

Если вы собираетесь применять замену метода как составную часть своего приложения, то рекомендуется использовать одну реализацию интерфейса `MethodReplacer` на каждый метод или группу перегружаемых методов. Не поддавайтесь искушению применять одну реализацию интерфейса `MethodReplacer` для замены многих несвязанных вместе методов, поскольку это приведет к выполнению дополнительных ненужных сравнений типов `String`, когда в прикладном коде требуется выяснить, какой именно метод должен получить новую реализацию. Мы обнаружили, что простые проверки правильности заменяемого метода в реализации интерфейса `MethodReplacer` весьма полезны и не влекут за собой значительные издержки.

[12] На 1000000 вызовов ушло __ мс

Если вас действительно заботит вопрос производительности, можете предусмотреть в реализации интерфейса `MethodReplacer` свойство логического типа, которое позволит включать и отключать такие проверки с помощью внедрения зависимостей.

Именование компонентов Spring Beans

В каркасе Spring поддерживается довольно сложная структура именования компонентов Spring Beans, допускающая большую гибкость в разрешении многих ситуаций. Каждый компонент Spring Bean должен иметь по крайней мере одно однозначное имя в содержащем его контексте типа `ApplicationContext`. Для определения имени компонента каркас Spring следует простому процессу разрешения имен. Так, если в дескрипторе разметки `<bean>` предусмотрен атрибут `id`, его значение служит в качестве имени компонента Spring Bean. Если же атрибут `id` не указан, Spring ищет атрибут `name`, и если он определен, то используется первое имя, заданное в атрибуте `name`. (Именно *первое имя*, потому что в атрибуте `name` допускается определять целый ряд имен, как поясняется далее.) Если не указан ни атрибут `id`, ни атрибут `name`, то в качестве имени компонента в Spring выбирается имя его класса, разумеется, при условии, что оно не используется для обозначения других компонентов Spring Beans. А если целый ряд компонентов Spring Beans без указанных атрибутов `id` и `name` имеет одно и то же имя класса, то Spring сгенерирует исключение (типа `org.springframework.beans.factory.NoSuchBeanDefinitionException`) при внедрении зависимостей во время инициализации контекста типа **`ApplicationContext`**. В следующем примере кода из файла конфигурации `app-context-01.xml` применяются все три способа именования компонентов Spring Beans:

```
<beans ...>
    <bean id="string1" class="java.lang.String"/>
    <bean name="string2" class="java.lang.String"/>
    <bean class="java.lang.String"/>
</beans>
```

Формально все эти способы именования одинаково допустимы, но какой из них лучше выбрать для приложения? Прежде всего, старайтесь не пользоваться именами, автоматически формируемыми исходя из поведения классов. Это существенно снижает гибкость при определении многих однотипных компонентов Spring Beans, и потоэму лучше присваивать им имена вручную. Таким образом, если стандартное поведение Spring в будущем изменится, разработанное приложение будет и далее функционировать нормально. Чтобы посмотреть, каким образом именуются компоненты Spring Beans, выполните исходный код из следующего примера, используя приведенную выше конфигурацию:

```
package com.apress.prospring5.ch3.xml;

import org.springframework.context.support
          .GenericXmlApplicationContext;
```

```
public class BeanNamingTest {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-01.xml");
    ctx.refresh();

    Map<String,String> beans =
        ctx.getBeansOfType(String.class);
    beans.entrySet().stream()
         .forEach(b -> System.out.println(b.getKey()));
    ctx.close();
  }
}
```

Вызов `ctx.getBeansOfType(String.class)` в приведенном выше коде делается с целью получить отображение со всеми компонентами Spring Beans типа `String` и их идентификаторами, существующими в контексте типа `ApplicationContext`. Ключами к этому отображению служат идентификаторы компонентов Spring Beans, которые выводятся на консоль с помощью лямбда-выражения. С учетом упомянутой выше конфигурации выполнение этого кода даст следующий результат:

```
string1
string2
java.lang.String#0
```

В последней строке приведенного выше результата находится идентификатор, присвоенный компоненту Spring Bean типа `String`, который не получил явного имени в представленной ранее конфигурации. Если изменить конфигурацию с целью ввести еще один безымянный компонент Spring Bean типа `String`, в таком случае она будет выглядеть следующим образом:

```
<beans ...>
  <bean id="string1" class="java.lang.String"/>
  <bean name="string2" class="java.lang.String"/>
  <bean class="java.lang.String"/>
  <bean class="java.lang.String"/>
</beans>
```

А выводимый в итоге результат изменится таким образом:

```
string1
string2
java.lang.String#0
java.lang.String#1
```

До версии Spring 3.1 атрибут `id` был равнозначен XML-идентификации (т.е. `xsd:ID`), что накладывало определенное ограничение на символы, употреблявшиеся в именах компонентов Spring Beans. Но начиная с версии Spring 3.1 для атрибута `id`

применяется идентификация xsd:String, а следовательно, прежнее ограничение на символы снято. Тем не менее в Spring по-прежнему проверяется однозначность идентификаторов в атрибутах id во всем контексте типа ApplicationContext. В связи с этим рекомендуется присвоить сначала имя компоненту Spring Bean с помощью атрибута id, а затем связать этот компонент с именами других компонентов, назначив соответствующие псевдонимы, как поясняется в следующем подразделе.

Назначение псевдонимов для имен компонентов Spring Beans

Компонентам Spring Beans Spring разрешается иметь не одно, а несколько имен. Для этого в атрибуте name дескриптора разметки <bean> необходимо указать список имен, разделяемых пробелами, запятыми или точками с запятой. Это можно сделать как вместо атрибута id, так и вместе с ним. Чтобы назначить псевдонимы для имен компонентов Spring Beans, можно также воспользоваться дескриптором разметки <alias>, помимо атрибута name. В следующем фрагменте кода из файла конфигурации app-context-02.xml приведено простое определение компонента Spring Bean с несколькими именами:

```
<beans ...>
  <bean id="john" name="jon johnny,jonathan;jim"
     class="java.lang.String"/>
  <alias name="john" alias="ion"/>
</beans>
```

Как видите, для компонента Spring Bean назначено шесть имен. Первое имя указано в атрибуте id, а еще четыре имени заданы в атрибуте name списком, в котором применяются все разрешенные разделители (это сделано только в целях демонстрации, но в реальных приложениях поступать так не рекомендуется). При разработке реальных приложений следует стандартизировать разделитель для имен компонентов Spring Beans в объявлениях. Еще один псевдоним определен с помощью дескриптора <alias>. Ниже приведен пример программы на Java, шесть раз извлекающей один и тот же компонент Spring Bean из контекста типа ApplicationContext, используя разные имена и проверяя их на соответствие одному и тому же компоненту. В этой программе вызывается также упоминавшийся ранее метод ctx.getBeansOf Type(..) с целью обеспечить наличие лишь одного компонента Spring Bean типа String в заданном контексте.

```
package com.apress.prospring5.ch3.xml;

import org.springframework.context.support
          .GenericXmlApplicationContext;
import java.util.Map;

public class BeanNameAliasing {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
```

```
    ctx.load("classpath:spring/app-context-02.xml");
    ctx.refresh();

    String s1 = (String) ctx.getBean("john");
    String s2 = (String) ctx.getBean("jon");
    String s3 = (String) ctx.getBean("johnny");
    String s4 = (String) ctx.getBean("jonathan");
    String s5 = (String) ctx.getBean("jim");
    String s6 = (String) ctx.getBean("ion");

    System.out.println((s1 == s2));
    System.out.println((s2 == s3));
    System.out.println((s3 == s4));
    System.out.println((s4 == s5));
    System.out.println((s5 == s6));

    Map<String,String> beans =
       ctx.getBeansOfType(String.class);

    if(beans.size() == 1) {
      System.out.println("There is only one String bean.");
    }

    ctx.close();
  }
}
```

В результате выполнения приведенного выше кода на консоль пять раз подряд выводится строка `"true"` и текст `"There is only one String bean"` (Имеется лишь один компонент Spring Bean типа `String`). Это означает, что один и тот же компонент Spring Bean доступен под разными именами.

Чтобы извлечь список псевдонимов компонента Spring Bean, достаточно вызвать метод `ApplicationContext.getAliases(String)`, передав ему любое имя или идентификатор данного компонента. Этот метод возвращает в массиве типа `String` список псевдонимов, содержащий все псевдонимы, кроме указанного при вызове.

Как упоминалось ранее, до версии Spring 3.1 атрибут `id` был равнозначен XML-идентификации (т.е. `xsd:ID`). Это означает, что в названиях идентификаторов нельзя было употреблять такие специальные символы, в том числе пробелы, запятые или точки с запятыми. Но, начиная с версии Spring 3.1, для атрибута `id` применяется идентификация `xsd:String`, а следовательно, прежнее ограничение на символы снято. Но это совсем не означает, что можно употребить следующую разметку для определения компонента Spring Bean:

```
<bean name="jon johnny,jonathan;jim"
     class="java.lang.String"/>
```

вместо такой разметки:

```
<bean id="jon johnny,jonathan;jim" class="java.lang.String"/>
```

Значения атрибутов `name` и `id` по-разному интерпретируются при инверсии управления в Spring. Чтобы извлечь список псевдонимов компонента Spring Bean, достаточно вызвать метод `ApplicationContext.getAliases(String)`, передав ему любое из имен или идентификаторов данного компонента. В итоге список псевдонимов будет возвращен в массиве типа `String`, кроме псевдонима, указанного при вызове данного метода. Это означает, что в первом из приведенных выше случаев определения компонента Spring Bean имя `jon` станет идентификатором, а остальные имена будут псевдонимами данного компонента.

Во втором случае вся символьная строка, указанная в качестве значения атрибута `id`, станет однозначным идентификатором компонента Spring Bean. Это нетрудно проверить с помощью следующего фрагмента кода из файла конфигурации `app-context-03.xml`:

```
<beans ...>
  <bean name="jon johnny,jonathan;jim"
       class="java.lang.String"/>
  <bean id="jon johnny,jonathan;jim"
     class="java.lang.String"/>
</beans>
```

а также главного класса, приведенного ниже.

```
package com.apress.prospring5.ch3.xml;

import org.springframework.context.support
          .GenericXmlApplicationContext;
import java.util.Arrays;
import java.util.Map;

public class BeanCrazyNaming {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-03.xml");
    ctx.refresh();

    Map<String,String> beans =
        ctx.getBeansOfType(String.class);

    beans.entrySet().stream().forEach(b ->
    {
      System.out.println("id: "
             + b.getKey()
             + "\n aliases: "
             + Arrays.toString(ctx.getAliases(b.getKey()))
             + "\n");
    });
```

```
    ctx.close();
  }
}
```

Выполнение приведенного выше кода даст следующий результат:

```
id: jon
 aliases: jonathan, jim, johnny

id: jon johnny,jonathan;jim
 aliases:
```

Как видите, отображение содержит два компонента Spring Beans типа `String`: один — с однозначным идентификатором `jon` и тремя псевдонимами, а другой — с однозначным идентификатором `jon johnny,jonathan;jim`, но без псевдонимов.

Назначение псевдонимов для имен компонентов Spring Beans — особое средство, которым совсем не обязательно пользоваться при разработке приложений. Если в приложении заранее предусмотрено много одних компонентов Spring Beans, внедряемых в другой компонент, то для доступа к целевому компоненту во внедряемых компонентах можно употреблять одно и то же имя. Но на стадии эксплуатации и сопровождения готового приложения могут вноситься модификации, и вот тут может пригодиться механизм назначения псевдонимов.

Рассмотрим такой случай. Допустим, имеется приложение, в котором всем 50 сконфигурированным компонентам Spring Beans требуется реализация интерфейса `Foo`. При этом в одних 25 компонентах используется его реализация `StandardFoo` под именем компонента `standardFoo`, а в других 25 компонентах — реализация `SuperFoo` под именем компонента `superFoo`. Через полгода после передачи приложения в эксплуатацию принимается решение о переходе первых 25 компонентов Spring Beans на реализацию `SuperFoo`. И сделать это можно тремя способами.

- Во-первых, заменить реализацию класса компонента `standardFoo` на `SuperFoo`. Недостаток такого способа заключается в том, что появляются два экземпляра класса `SuperFoo`, когда на самом деле требуется лишь один. А когда изменится конфигурация, придется вносить изменения в два компонента, а не в один.
- Во-вторых, обновить конфигурацию внедрения зависимостей для 25 изменяемых компонентов Spring Beans, сменив имена компонентов с `standardFoo` на `superFoo`. Это не самый изящный способ, поскольку можно воспользоваться поиском и заменой. Но последующий откат изменений в том случае, если решение окажется неприемлемым, означает, что придется извлечь прежнюю версию исходного кода из системы контроля версий.
- В-третьих, удалить (или закомментировать) определения компонента `standardFoo` и назначить `standardFoo` в качестве псевдонима для компонента `superFoo`. Такое изменение требует минимальных усилий и упрощает восстановление системы в ее прежней конфигурации.

Именование компонентов Spring Beans с помощью аннотаций в конфигурациях

Если определения компонентов Spring Beans объявляются с помощью аннотаций, то именование компонентов несколько отличается от принятого при конфигурировании в формате XML, а кроме того, оно открывает дополнительные возможности. Начнем, однако, с простого объявления определений компонентов Spring Beans с помощью стереотипной аннотации `@Component` и всех ее специальных разновидностей вроде `@Service`, `@Repository` и `@Controller`. Для этого рассмотрим следующий вариант класса `Singer`:

```
package com.apress.prospring5.ch3.annotated;

import org.springframework.stereotype.Component;

@Component
public class Singer {
  private String lyric = "We found a message in a bottle "
                       + "we were drinking";
  public void sing() {
    System.out.println(lyric);
  }
}
```

В этом классе содержится объявление одиночного компонента Spring Bean типа `Singer`, записанное с помощью аннотации `@Component`. У этой аннотации отсутствуют аргументы, и поэтому контейнер инверсии управления в Spring выберет однозначный идентификатор для данного компонента. В данном случае соблюдается следующее соглашение об именовании: компонент Spring Bean именуется так же, как и сам класс, но первая буква в его имени становится строчной. Это означает, что рассматриваемый здесь компонент получает имя `singer`. Такое соглашение об именовании соблюдается и во всех остальных стереотипных аннотациях. Чтобы убедиться в этом, можно воспользоваться следующим классом:

```
package com.apress.prospring5.ch3.annotated;

import org.springframework.context.support
         .GenericXmlApplicationContext;
import java.util.Arrays;
import java.util.Map;

public class AnnotatedBeanNaming {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-annotated.xml");
    ctx.refresh();
```

```
    Map<String,Singer> beans =
        ctx.getBeansOfType(Singer.class);

    beans.entrySet().stream().forEach(b ->
          System.out.println("id: " + b.getKey()));
    ctx.close();
  }
}
```

В файле конфигурации `app-context-annotated.xml` содержится только объявление режима просмотра компонентов в пакете `com.apress.prospring5.ch3.annotated`, и поэтому нет смысла приводить его снова. Выполнение исходного кода из приведенного выше класса приведет к выводу на консоль следующего результата:

```
id: singer
```

Таким образом, применение аннотации `@Component("singer")` равнозначно снабжению класса `Singer` аннотацией `@Component`. Если же требуется присвоить имя компоненту Spring Bean по-другому, это имя следует указать в качестве аргумента аннотации `@Component`, как показано ниже.

```
package com.apress.prospring5.ch3.annotated;

import org.springframework.stereotype.Component;

@Component("johnMayer")

public class Singer {
  private String lyric =
          "Down there below us, under the clouds";

  public void sing() {
    System.out.println(lyric);
  }
}
```

Как и следовало ожидать, выполнение исходного кода из класса `AnnotatedBeanNaming` дает приведенный ниже результат.

```
id: johnMayer
```

А как насчет псевдонимов? Если задать в качестве аргумента аннотации `@Component` однозначный идентификатор компонента Spring Bean, то назначение псевдонимов окажется невозможным при таком способе объявления компонента. Именно здесь и приходит на помощь конфигурирование на языке Java. Рассмотрим в качестве примера приведенный ниже класс, в котором определен статический конфигурационный класс. Такая возможность допускается в Spring, и мы не преминули воспользоваться ею в данном примере, сохранив всю логику в одном и том же исходном файле.

```
package com.apress.prospring5.ch3.config;

import com.apress.prospring5.ch3.annotated.Singer;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.support
          .GenericApplicationContext;
import org.springframework.context.support
          .GenericXmlApplicationContext;

import java.util.Arrays;
import java.util.Map;

public class AliasConfigDemo {

  @Configuration
  public static class AliasBeanConfig {
    @Bean
    public Singer singer() {
      return new Singer();
    }
  }

  public static void main(String... args) {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               AliasBeanConfig.class);

    Map<String,Singer> beans =
        ctx.getBeansOfType(Singer.class);
    beans.entrySet().stream().forEach(b ->
          System.out.println("id: "
              + b.getKey()
              + "\n aliases: "
              + Arrays.toString(ctx.getAliases(b.getKey()))
              + "\n")
    );

    ctx.close();
  }
}
```

В этом классе содержится определение компонента Spring Bean типа `Singer`, объявленного с помощью аннотации `@Bean`, которой снабжается метод `singer()`. Если эта аннотация указана без аргументов, то однозначным идентификатором определяемого компонента Spring Bean становится имя данного метода. Так, если выполнить

исходный код приведенного выше класса, то на консоль будет выведен следующий результат:

```
id: singer
 aliases:
```

Чтобы объявить псевдонимы, можно воспользоваться атрибутом `name` аннотации `@Bean`, который используется в данной аннотации по умолчанию. И это означает, что результат будет один и тот же, объявить ли компонент Spring Bean в методе `singer()` с помощью аннотации `@Bean("singer")` или `@Bean(name="singer")`. Но в любом случае контейнер инверсии управления в Spring создаст компонент Spring Bean типа `Singer` с идентификатором `singer`.

Если присвоить атрибуту `name` символьную строку, содержащую характерный для псевдонимов разделитель (пробел, запятую или точку с запятой), эта строка станет идентификатором определяемого компонента Spring Bean. Но если присвоить данному атрибуту массив символьных строк, то первая из них станет идентификатором определяемого компонента, а остальные будут его псевдонимами. Ниже приведена конфигурация компонента Spring Bean, видоизмененная с целью назначить для него псевдонимы.

```
@Configuration
public static class AliasBeanConfig {
  @Bean(name={"johnMayer","john","jonathan","johnny"})
  public Singer singer(){
    return new Singer();
  }
}
```

Если выполнить исходный код класса `AliasConfigDemo`, то выводимый на консоль результат изменится по сравнению с прежним:

```
id: johnMayer
 aliases: jonathan, johnny, john
```

Для поддержки псевдонимов в версии Spring 4.2 была внедрена аннотация `@AliasFor`, позволяющая объявлять псевдонимы для атрибутов большинства других аннотаций, доступных в Spring. Например, у аннотации `@Bean` имеются два атрибута, `name` и `value`, неявно объявляемых как псевдонимы друг для друга, а с помощью аннотации `@AliasFor` они явно становятся псевдонимами. Приведенный ниже фрагмент кода взят из определения аннотации `@Bean` в официальном хранилище GitHub для Spring. В нем опущен код и документирующие комментарии, не относящиеся к рассматриваемому здесь вопросу[13].

[13] Полная реализация аннотации `@AliasFor` доступна по адресу `https://github.com/spring-projects/spring-framework/blob/master/spring-core/src/main/java/org/springframework/core/annotation/AliasFor.java`.

```
package org.springframework.context.annotation;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.springframework.core.annotation.AliasFor;
...
@Target({ElementType.METHOD, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Bean {
  @AliasFor("name")
  String value() default {};

  @AliasFor("value")
  String name() default {};

  ...
}
```

Обратимся к конкретному примеру, в котором объявляется аннотация `@Award`, которая может, конечно, использоваться для экземпляров класса `Singer`:

```
package com.apress.prospring5.ch3.annotated;

import org.springframework.core.annotation.AliasFor;

public @interface Award {

  @AliasFor("prize")
  String value() default {};

  @AliasFor("value")
  String prize() default {};
}
```

Используя эту аннотацию, можно видоизменить класс `Singer` приведенным ниже способом. Приведенная ниже аннотация равнозначна аннотациям `@Award(value={"grammy", "platinum disk"})` и `@Award({"grammy", "platinum disk"})`.

```
package com.apress.prospring5.ch3.annotated;

import org.springframework.beans.factory
          .annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component("johnMayer")
```

```
@Award(prize = {"grammy", "platinum disk"})
public class Singer {

  private String lyric = "We found a message in "
                       + "a bottle we were drinking";

  public void sing() {
    System.out.println(lyric);
  }
}
```

Но с помощью аннотации `@AliasFor` можно сделать нечто более интересное и, в частности, объявить псевдонимы мета-аннотаций. В приведенном ниже фрагменте кода объявляется особая форма аннотации `@Award`, объявляющая атрибут `name` в качестве псевдонима для атрибута `value` из аннотации `@Award`. И это делается для того, чтобы сделать очевидным, что в аргументе данной аннотации задан однозначный идентификатор компонента Spring Bean.

```
package com.apress.prospring5.ch3.annotated;

import org.springframework.core.annotation.AliasFor;

@Award
public @interface Trophy {

  @AliasFor(annotation = Award.class, attribute = "value")
  String name() default {};
}
```

Таким образом, вместо того, чтобы написать класс `Singer` следующим образом:

```
package com.apress.prospring5.ch3.annotated;

import org.springframework.stereotype.Component;

@Component("johnMayer")
@Award(value={"grammy", "platinum disk"})
public class Singer {

  private String lyric = "We found a message in a "
                       + "bottle we were drinking";

  public void sing() {
    System.out.println(lyric);
  }
}
```

его можно написать так:

```
package com.apress.prospring5.ch3.annotated;

@Component("johnMayer")
@Trophy(name={"grammy", "platinum disk"})
public class Singer {

  private String lyric = "We found a message in a "
                       + "bottle we were drinking";
  public void sing() {
    System.out.println(lyric);
  }
}
```

На заметку На создание псевдонимов для атрибутов аннотаций с помощью еще одной аннотации **@AliasFor** не накладывается никаких ограничений. Аннотацию **@AliasFor** нельзя применять в любых стереотипных аннотациях (например, аннотации **@Component** и ее специальных форм). Дело в том, что специальная обработка значений атрибутов **value** выполнилась задолго до внедрения аннотации **@AliasFor**. Следовательно, из соображений обратной совместимости воспользоваться аннотацией **@AliasFor** с такими атрибутами **value** просто невозможно. Так, если написать код с целью назначить псевдонимы для атрибутов **value** в стереотипных аннотациях, то никаких ошибок компиляции не появится и такой код можно даже выполнить, но любой аргумент, предоставленный для псевдонима, все равно будет проигнорирован. Это же относится и к аннотации **@Qualifier**.

Режим получения экземпляров компонентов Spring Beans

По умолчанию все компоненты Spring Beans являются одиночными. Это означает, что Spring обслуживает одиночный экземпляр компонента, во всех зависимых объектах используется один и тот же экземпляр, а в результате всех вызовов метода `ApplicationContext.getBean()` возвращается один тот же экземпляр. Это положение было наглядно продемонстрировано в примере из предыдущего подраздела, где для проверки идентичности компонентов Spring Beans можно было воспользоваться операцией сравнения ==, а не методом `equals()`.

Термины *одиночный* и *"одиночка"* равноценно употребляются в Java для обозначения двух разных понятий: объекта, имеющего единственный экземпляр в пределах приложения, а также проектного шаблона "Одиночка" (Singleton). Первое понятие мы будем далее называть одиночным объектом, а второе — проектным шаблоном "Одиночка". Этот проектный шаблон нашел широкое распространение благодаря книге *Design Patterns: Elements of Reusable Object Oriented Software* Эриха Гаммы и др. (Erich Gamma el al.; издательство Addison-Wesley, 1995 г.)[14]. Затруднение возникает в том случае, когда начинают путать потребность в одиночных экземплярах и не-

---

[14] В русском переводе эта книга вышла под названием *Приемы объектно-ориентированного проектирования. Паттерны проектирования*, Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес; издательство "Питер", СпБ. 2007 г.

обходимость в применении проектного шаблона "Одиночка". Ниже приведена типичная реализация этого шаблона на языке Java.

```
package com.apress.prospring5.ch3;

public class Singleton {
  private static Singleton instance;

  static {
    instance = new Singleton();
  }

  public static Singleton getInstance() {
    return instance;
  }

  private Singleton() {
    // этот конструктор без аргументов требуется для того,
    // чтобы разработчики не смогли получить экземпляр
    // данного класса непосредственно
  }
}
```

Этот проектный шаблон достигает своей цели, позволяя поддерживать и оперировать единственным экземпляром класса во всем приложении, хотя и за счет увеличения степени связанности. Так, в прикладном коде должно быть всегда точно известно о классе `Singleton`, чтобы получить его экземпляр, но это полностью исключает возможность вынести прикладной код в интерфейсы.

В действительности шаблон "Одиночка" представляет собой два проектных шаблона в одном. Первый (и желательный) шаблон включает в себя сопровождение единственного экземпляра объекта, а второй (менее желательный) шаблон касается поиска объекта, хотя это полностью исключает возможность пользоваться интерфейсами. Применение проектного шаблона "Одиночка" также существенно затрудняет произвольную замену реализаций, поскольку большинство объектов, которым требуется экземпляр класса `Singleton`, получают непосредственный доступ к объекту этого класса. А это может привести к различным осложнениям при попытке модульного тестирования приложения, поскольку объект типа `Singleton` не удастся заменить имитирующим объектом в целях тестирования.

Правда, в Spring можно воспользоваться моделью получения одиночного экземпляра, не прибегая непосредственно к проектному шаблону "Одиночка". Все компоненты Spring Beans по умолчанию создаются как одиночные экземпляры, и поэтому для всех запросов к конкретному компоненту Spring Bean применяется один и тот же экземпляр. Каркас Spring, конечно, не ограничивается применением только одиночного экземпляра; для удовлетворения каждой зависимости и каждого обращения к методу `getBean()` можно по-прежнему получить новый экземпляр компонента Spring Bean. Все это делается без всякого влияния на прикладной код, и поэтому мож-

но вполне обоснованно назвать каркас Spring *независимым от режима получения экземпляров*. На самом деле это весьма эффективный принцип. Если в начале разработки вы считаете, что какой-то объект является одиночным, но впоследствии обнаруживаете, что он не подходит для многопоточного доступа, то можете спокойно заменить этот объект на неодиночный (прототип), не оказав никакого влияния на прикладной код.

На заметку Несмотря на то что изменение режима получения экземпляров не оказывает никакого влияния на прикладной код, оно может все же привести к некоторым осложнениям, если полагаться на интерфейсы жизненного цикла Spring. Подробнее об этом речь пойдет в главе 4.

Сменить режим получения экземпляров с одиночного на неодиночный совсем не трудно. В приведенных ниже фрагментах кода конфигурации наглядно показано, как это делается в формате XML и с помощью аннотаций.

```
<!-- файл конфигурации app-context-xml.xml -->
<beans ...>
  <bean id="nonSingleton"
     class="com.apress.prospring5.ch3.annotated.Singer"
     scope="prototype" c:_0="John Mayer"/>
</beans>

// исходный файл Singer.java
package com.apress.prospring5.ch3.annotated;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component("nonSingleton")
@Scope("prototype")
public class Singer {
  private String name = "unknown";

  public Singer(@Value("John Mayer") String name) {
    this.name = name;
  }
}
```

В конфигурации, составляемой в формате XML, класс `Singer` может служить типом для определяемого компонента Spring Bean. И если режим просмотра компонентов не активизирован, то аннотации в этом классе будут просто проигнорированы.

Как видите, такое объявление компонента Spring Bean отличается от любых рассмотренных до сих пор объявлений внедрением атрибута `scope` (область видимости) и установкой в нем значения `prototype` (прототип). По умолчанию каркас Spring устанавливает в атрибуте `scope` значение `singleton` (одиночный). Область видимос-

ти на уровне прототипа вынуждает Spring получать новый экземпляр компонента Spring Bean всякий раз, когда он запрашивается в приложении. В следующем фрагменте кода демонстрируется влияние такой установки на приложение:

```
package com.apress.prospring5.ch3;

import com.apress.prospring5.ch3.annotated.Singer;
import org.springframework.context.support
          .GenericXmlApplicationContext;

public class NonSingletonDemo {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();

    Singer singer1 =
           ctx.getBean("nonSingleton", Singer.class);
    Singer singer2 =
           ctx.getBean("nonSingleton", Singer.class);

    System.out.println("Identity Equal?: "
                    + (singer1 ==singer2));
    System.out.println("Value Equal:? "
                    + singer1.equals(singer2));
    System.out.println(singer1);
    System.out.println(singer2);

    ctx.close();
  }
}
```

Выполнение исходного кода из данного примера дает следующий результат:

```
Identity Equal?: false
Value Equal:? false
John Mayer
John Mayer
```

Как показывает приведенный выше результат, оба объекта типа `String` не идентичны, несмотря на то, что их значения одинаковы, а их экземпляры были получены по одному и тому же имени компонента Spring Bean.

Выбор режима получения экземпляров

В большинстве случаев очень просто выяснить, какой именно режим получения экземпляров является для них подходящим. Как правило, режим получения одиночных экземпляров считается стандартным для компонентов Spring Beans. А в общем, одиночные экземпляры должны применяться в следующих случаях.

- **Общий объект без состояния.** Имеется объект, не поддерживающий состояние, и множество зависимых от него объектов. В связи с тем что состояние не поддерживается, синхронизация не нужна, а следовательно, не придется получать новый экземпляр компонента Spring Bean всякий раз, когда он требуется зависимому объекту для какой-нибудь обработки.
- **Общий объект с состоянием только для чтения.** Этот случай похож на предыдущий, но отличается поддержанием состояния только для чтения. Но и в этом случае синхронизация не нужна, и получение экземпляра по каждому запросу компонента Spring Bean только влечет дополнительные издержки.
- **Общий объект с разделяемым состоянием.** Если требуется компонент Spring Bean, состояние которого должно разделяться с другими объектами, то одиночный экземпляр будет идеальным выбором. Но в этом случае придется обеспечить как можно более точную синхронизацию при записи состояния.
- **Высокопроизводительные объекты с записываемым состоянием.** Если имеется компонент Spring Bean, который интенсивно применяется в приложении, то может оказаться, что сохранение его одиночным и синхронизация всего доступа к нему для записи позволит добиться более высокой производительности, чем постоянное получение сотен экземпляров этого объекта. Применяя такой подход, старайтесь обеспечить как можно более точную синхронизацию без ущерба для согласованности. Такой подход особенно удобен в тех случаях, когда в приложении получается большое количество экземпляров в течение длительного периода времени, когда у общего объекта имеется лишь небольшая доля записываемого состояния или когда получение нового экземпляра сопряжено с большим расходованием ресурсов.

Неодиночные экземпляры должны применяться в перечисленных ниже случаях.

- **Объекты с записываемым состоянием.** Если имеется компонент Spring Bean с большой долей записываемого состояния, то может оказаться, что затраты на синхронизацию превышают затраты на получение нового экземпляра для обработки каждого запроса от зависимого объекта.
- **Объекты с закрытым состоянием.** В ряде случаев зависимым объектам требуется компонент Spring Bean с закрытым состоянием, чтобы зависимые объекты могли выполнять свою обработку раздельно. В таком случае одиночный экземпляр совсем не подходит, а следовательно, требуется неодиночный экземпляр.

Главное преимущество, которое дает управление режимом получения экземпляров в Spring, связано с уменьшением потребления оперативной памяти в приложениях благодаря одиночным экземплярам, и для этого не придется прилагать больших усилий. А если окажется, что режим получения одиночных экземпляров не отвечает потребностям приложения, то его конфигурацию совсем не трудно перенастроить на режим получения неодиночных экземпляров.

Области видимости компонентов Spring Beans

Помимо областей видимости на уровне одиночного экземпляра и прототипа, при определении компонента Spring Bean для более конкретных целей доступными становятся другие области видимости. Можно также реализовать специальную область видимости и зарегистрировать ее в контексте `ApplicationContext`. Ниже приведен перечень областей видимости, которые поддерживаются в версии Spring 4.

- **Одиночный экземпляр.** Стандартная область видимости. В этом случае объекты будут создаваться лишь по одному на каждый контейнер инверсии управления в Spring.
- **Прототип.** Каркас Spring получит новый экземпляр по запросу из приложения.
- **Запрос.** Служит для применения в веб-приложениях. Если для построения веб-приложений применяется модуль Spring MVC, то компоненты Spring Beans с областью видимости на уровне запроса будут создаваться по каждому HTTP-запросу и уничтожаться по окончании его обработки.
- **Сеанс связи.** Служит для применения в веб-приложениях. Если для построения веб-приложений применяется модуль Spring MVC, то компоненты Spring Beans с областью видимости на уровне сеанса связи будут создаваться для каждого HTTP-сеанса и уничтожаться по его завершении.
- **Глобальный сеанс связи.** Служит для применения в веб-приложениях, основанных на портлетах. Компоненты Spring Beans с областью видимости на уровне глобального сеанса связи могут совместно применяться всеми портлетами в портальном приложении, приводимом в действие модулем Spring MVC.
- **Поток исполнения.** Каркас Spring получит новый экземпляр компонента Spring Bean по запросу из нового потока исполнения. А по запросу из одного и того же потока исполнения возвратится тот же самый экземпляр компонента Spring Bean. Однако эта область видимости не регистрируется по умолчанию.
- **Специальная.** Это специальная область видимости компонента Spring Bean, которую можно создать, реализовав интерфейс `org.springframework.beans.factory.config.Scope` и зарегистрировав ее в конфигурации Spring (для конфигурирования в формате XML следует применять класс `org.springframework.beans.factory.config.CustomScopeConfigurer`).

Разрешение зависимостей

При нормальном функционировании каркас Spring способен разрешать зависимости, просто просматривая файл конфигурации или аннотации в классах. Подобным образом каркас Spring может гарантировать, что все его компоненты сконфигурированы в правильном порядке и каждый компонент имеет корректно настроенные зависимости. Если каркас Spring не сделает этого, а просто создаст компоненты и скон-

фигурирует их в произвольном порядке, то вполне возможно, что какой-нибудь его компонент будет создан и настроен прежде своих зависимостей. Очевидно, что при этом в приложении могут возникнуть самые разные осложнения.

К сожалению, Spring ничего не известно о тех зависимостях между его компонентами в прикладном коде, которые отсутствуют в конфигурации. Допустим, что один компонент `johnMayer` типа `Singer` получает экземпляр другого компонента `gopher` типа `Guitar` через вызов метода `ctx.getBean()` и затем использует его при вызове метода `johnMayer.sing()`. В этом методе экземпляр типа `Guitar` получается через вызов `ctx.getBean("gopher")` без запроса Spring на автоматическое внедрение зависимостей. И в этом случае каркас Spring остается в неведении относительно того, что один его компонент `johnMayer` зависит от другого его компонента `gopher`, и в конечном итоге он может получить экземпляр своего компонента `johnMayer` прежде экземпляра компонента `gopher`. Дополнительные сведения о зависимостях между компонентами Spring Beans в прикладном коде можно предоставить с помощью атрибута `depends-on` в дескрипторе разметки `<bean>`. В следующем фрагменте кода из файла конфигурации `app-context-01.xml` демонстрируется надлежащий порядок конфигурирования компонентов `johnMayer` и `gopher` в рассматриваемом здесь примере:

```
<beans ...">
  <bean id="johnMayer"
     class="com.apress.prospring5.ch3.xml.Singer"
     depends-on="gopher"/>

<bean id="gopher"
   class="com.apress.prospring5.ch3.xml.Guitar"/>
</beans>
```

В приведенной выше конфигурации утверждается, что компонент `johnMayer` зависит от компонента `gopher`. Это утверждение должно быть принято каркасом Spring к сведению, чтобы экземпляр компонента `gopher` был получен прежде экземпляра компонента `johnMayer`. Но для этого компоненту `johnMayer` требуется доступ к контексту типа `ApplicationContext`. Следовательно, необходимо также предписать каркасу Spring внедрить эту ссылку, чтобы воспользоваться ею при вызове метода `johnMayer.sing()` для предоставления компонента `gopher`. С этой целью в классе `Singer` реализуется интерфейс `ApplicationContextAware`, обеспечивающий в Spring реализацию метода установки специально для объекта типа `ApplicationContext`. Он автоматически обнаруживается контейнером инверсии управления в Spring для внедрения контекста типа `ApplicationContext`, в котором создается компонент Spring Bean. А поскольку это делается после вызова конструктора компонента Spring Bean, то совершенно очевидно, что применение контекста типа `ApplicationContext` в конструкторе компонента Spring Bean приведет к исключению типа `NullPointerException`. Ниже приведен исходный код класса

Singer, в который внесены коррективы для обеспечения надлежащего порядка получения экземпляров обоих рассматриваемых здесь компонентов Spring Beans.

```
package com.apress.prospring5.ch3.xml;

import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;

public class Singer implements ApplicationContextAware {

  ApplicationContext ctx;

  @Override
  public void setApplicationContext(
              ApplicationContext applicationContext)
              throws BeansException {
    this.ctx = applicationContext;
  }

  private Guitar guitar;

  public Singer() {
  }

  public void sing() {
    guitar = ctx.getBean("gopher", Guitar.class);
    guitar.sing();
  }
}
```

Класс Guitar довольно прост. Он содержит лишь метод sing(), как показано ниже.

```
package com.apress.prospring5.ch3.xml;

public class Guitar {
  public void sing(){
    System.out.println("Cm Eb Fm Ab Bb");
  }
}
```

Чтобы проверить правильность разрешения зависимостей в данном примере, можно воспользоваться следующим классом:

```
package com.apress.prospring5.ch3.xml;

import org.springframework.context.support
          .GenericXmlApplicationContext;
```

```
public class DependsOnDemo {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-01.xml");
    ctx.refresh();

    Singer johnMayer =
           ctx.getBean("johnMayer", Singer.class);
    johnMayer.sing();

    ctx.close();
  }
}
```

Разумеется, для представленной выше конфигурации в формате XML имеется равнозначная аннотация. Классы `Singer` и `Guitar` должны быть объявлены как компоненты Spring Beans с помощью одной из стереотипных аннотаций (в данном случае — аннотации `@Component`). Новшество здесь состоит в том, что в классе `Singer` размещается аннотация `@DependsOn`, равнозначная атрибуту `depends-on` из конфигурации в формате XML.

```
package com.apress.prospring5.ch3.annotated;

import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.context.annotation.DependsOn;
import org.springframework.stereotype.Component;

@Component("johnMayer")
@DependsOn("gopher")
public class Singer implements ApplicationContextAware {

  ApplicationContext applicationContext;

  @Override
  public void setApplicationContext(
             ApplicationContext applicationContext)
             throws BeansException {
    this.applicationContext = applicationContext;
  }

  private Guitar guitar;
  public Singer() {
 }

  public void sing() {
    guitar = applicationContext.getBean("gopher", Guitar.class);
```

```
    guitar.sing();
  }
}
```

Осталось лишь активизировать режим просмотра компонентов, как показано ниже, а затем воспользоваться файлом конфигурации `application-context-02.xml` в классе `DependsOnDemo`, чтобы создать контекст типа `ApplicationContext`. В результате выполнения исходного кода из данного примера на консоль будет выведена строка `"Cm Eb Fm Ab Bb"`.

```
<!-- файл конфигурации application-context-02.xml -->
<beans...>
  <context:component-scan
    base-package="com.apress.prospring5.ch3.annotated"/>
</beans>
```

В процессе разработки приложений такого способа разрешения зависимостей рекомендуется все же избегать и определять зависимости с помощью контрактов на внедрение зависимостей через метод установки и конструктор. Но если каркас Spring интегрируется с унаследованным кодом, то вполне возможно, что определенные в прикладном коде зависимости потребуют предоставления дополнительных сведений для Spring Framework.

Автосвязывание компонентов Spring Beans

В каркасе поддерживается пять режимов автосвязывания. Ниже приведено их краткое описание.

- **Режим `byName`.** Если применяется этот режим, каркас Spring пытается связать каждое свойство с одноименным компонентом Spring Bean. Так, если у целевого компонента Spring Bean имеется свойство `foo` и в контексте типа `ApplicationContext` определен компонент `foo`, то этот компонент присваивается свойству `foo` целевого компонента Spring Bean.
- **Режим `byType`.** Если применяется этот режим, каркас Spring пытается связать каждое свойство целевого компонента Spring Bean с компонентом того же самого типа, автоматически выбираемым из контекста типа `ApplicationContext`.
- **Режим `constructor`.** Этот режим действует подобно режиму `byType`, за исключением того, что зависимости внедряются в нем через конструкторы, а не методы установки. Каркас Spring пытается обнаружить совпадение с как можно бо́льшим числом аргументов в конструкторе. Так, если у компонента Spring Bean имеются два конструктора, причем первый из них принимает аргумент типа `String`, а второй — аргументы типа `String` и `Integer`, и в контексте типа `ApplicationContext` определены компоненты Spring Beans типа `String` и `Integer`, то в Spring будет использован конструктор с двумя аргументами.

- **Режим `default`.** В этом режиме каркас Spring автоматически делает выбор между режимами `constructor` и `byType`. Если у компонента Spring Bean имеется конструктор по умолчанию (т.е. без аргументов), то в Spring выбирается режим автосвязывания `byType`, а иначе — режим `constructor`.
- **Режим `no`.** Выбирается по умолчанию.

Так, если у целевого компонента Spring Bean имеется свойство типа `String`, а в контексте типа `ApplicationContext` определен компонент Spring Bean типа `String`, то каркас Spring автоматически свяжет этот компонент со свойством типа `String` целевого компонента. Если же в одном и том же экземпляре контекста типа `ApplicationContext` присутствует не один, а несколько компонентов Spring Beans одного и того же типа (в данном случае — `String`), то каркас Spring не сможет выбрать подходящий режим автосвязывания и сгенерирует исключение типа `org.springframework.beans.factory.NoSuchBeanDefinitionException`.

В следующем примере демонстрируется простая конфигурация из файла `app-context-03.xml`, устанавливающая автосвязывание трех однотипных компонентов Spring Beans в первых трех из описанных выше режимов:

```
<beans ...>

  <bean id="fooOne"
     class="com.apress.prospring5.ch3.xml.Foo"/>
  <bean id="barOne"
     class="com.apress.prospring5.ch3.xml.Bar"/>

  <bean id="targetByName" autowire="byName"
    class="com.apress.prospring5.ch3.xml.Target"
    lazy-init="true"/>

  <bean id="targetByType" autowire="byType"
    class="com.apress.prospring5.ch3.xml.Target"
    lazy-init="true"/>

  <bean id="targetConstructor" autowire="constructor"
    class="com.apress.prospring5.ch3.xml.Target"
    lazy-init="true"/>
</beans>
```

Эта конфигурация должна быть вам уже знакома. Здесь `Foo` и `Bar` — пустые классы. Обратите внимание на отличие значений, указанных в атрибуте `autowire` каждого целевого компонента Spring Bean типа `Target`. Кроме того, в атрибуте `lazy-init` установлено логическое значение `true`, чтобы известить Spring о необходимости получить экземпляр целевого компонента только по первому его запросу, а не во время начальной загрузки. Благодаря этому результаты могут быть выведены в нужном месте тестовой программы. Ниже приведен исходный код простого приложения на Java,

в котором каждый целевой компонент Spring Bean типа `Target` извлекается из контекста типа `ApplicationContext`.

```
package com.apress.prospring5.ch3.xml;

import org.springframework.context.support
          .GenericXmlApplicationContext;

public class Target {
  private Foo fooOne;
  private Foo fooTwo;
  private Bar bar;

  public Target() {
  }

  public Target(Foo foo) {
    System.out.println("Target(Foo) called");
  }

  public Target(Foo foo, Bar bar) {
    System.out.println("Target(Foo, Bar) called");
  }

  public void setFooOne(Foo fooOne) {
    this.fooOne = fooOne;
    System.out.println("Property fooOne set");
  }

  public void setFooTwo(Foo foo) {
    this.fooTwo = foo;
    System.out.println("Property fooTwo set");
  }

  public void setBar(Bar bar) {
    this.bar = bar;
    System.out.println("Property bar set");
  }

  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-03.xml");
    ctx.refresh();

    Target t = null;

    System.out.println("Using byName:\n");
    t = (Target) ctx.getBean("targetByName");

    System.out.println("\nUsing byType:\n");
```

```
    t = (Target) ctx.getBean("targetByType");

    System.out.println("\nUsing constructor:\n");
    t = (Target) ctx.getBean("targetConstructor");

    ctx.close();
  }
}
```

В приведенном выше прикладном коде у класса `Target` имеются три конструктора: конструктор без аргументов, конструктор, принимающий экземпляр типа `Foo`, а также конструктор, принимающий экземпляры типа `Foo` и `Bar`. Помимо этих конструкторов у компонента Spring Bean типа `Target` имеются три свойства, два из которых относятся к типу `Foo`, а третье — к типу `Bar`. При обращении ко всем этим свойствам и конструкторам на консоль выводится соответствующее сообщение. А метод `main()` просто извлекает все компоненты Spring Beans типа `Target`, объявленные в контексте типа `ApplicationContext`, инициируя тем самым процесс автосвязывания. Ниже приведен результат, выводимый на консоль при выполнении исходного кода из данного примера.

```
Using byName:

Property fooOne set

Using byType:

Property bar set
Property fooOne set
Property fooTwo set

Using constructor:

Target(Foo, Bar) called
```

Как следует из приведенного выше результата, если в Spring применяется режим `byName`, то заданное значение устанавливается только в свойстве `fooOne`, поскольку лишь для него в файле конфигурации задан этот режим автосвязывания. Если же в Spring применяется режим автосвязывания `byType`, то заданные значения устанавливаются во всех трех свойствах целевого компонента. В частности, свойства `fooOne` и `fooTwo` устанавливаются компонентом `fooOne`, а свойство `bar` — компонентом `barOne`. А если применяется режим автосвязывания `constructor`, то каркас Spring выбирает конструктор с двумя аргументами, потому что он в состоянии предоставить компоненты Spring Beans для обоих его аргументов, не обращаясь к другому конструктору.

Дело несколько усложняется при автосвязывании по типу, когда типы компонентов Spring Beans оказываются родственными. Так, если имеется несколько классов, реализующих один и тот же интерфейс, тип которого указан в свойстве, которое тре-

буется связать автоматически, то в каркасе Spring генерируются соответствующие исключения, поскольку ему неизвестно, какой именно компонент Spring Bean следует внедрить. Чтобы воссоздать подобную ситуацию, преобразуем класс `Foo` в интерфейс и объявим по отдельности два типа компонентов Spring Beans, реализующих этот интерфейс. Оставим также стандартную конфигурацию без дополнительного именования компонентов Spring Beans.

```
package com.apress.prospring5.ch3.xml.complicated;

public interface Foo {
  // пустой интерфейс interface, используемый как маркерный
}

public class FooImplOne implements Foo {
}

public class FooImplTwo implements Foo {
}
```

Если бы ввести новый файл конфигурации `app-context-04.xml`, он содержал бы следующую конфигурацию:

```
<beans ...>
  <bean id="fooOne"
    class="com.apress.prospring5.ch3.xml
              .complicated.FooImplOne"/>

  <bean id="fooTwo"
    class="com.apress.prospring5.ch3.xml
             .complicated.FooImplTwo"/>

  <bean id="bar"
     class="com.apress.prospring5.ch3.xml.Bar"/>

  <bean id="targetByType" autowire="byType"
     class="com.apress.prospring5.ch3.xml
               .complicated.CTarget"
     lazy-init="true"/>
</beans>
```

Для целей рассматриваемого здесь упрощенного примера внедрим также класс `CTarget`, аналогичный представленному ранее классу `Target`, но отличающийся от него лишь методом `main()`. Ниже приведен исходный код класса `CTarget`.

```
package com.apress.prospring5.ch3.xml.complicated;

import com.apress.prospring5.ch3.xml.*;
import org.springframework.context.support
          .GenericXmlApplicationContext;

public class CTarget {
```

```
  ...
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-04.xml");
    ctx.refresh();
    System.out.println("\nUsing byType:\n");
    CTarget t = (CTarget) ctx.getBean("targetByType");
    ctx.close();
}
```

Выполнение исходного кода приведенного выше класса даст следующий результат:

```
Using byType:

Exception in thread "main"
  org.springframework.beans.factory
     .UnsatisfiedDependencyException:
  Error creating bean with name 'targetByType' defined in
    class path resource spring/app-context-04.xml:
    Unsatisfied dependency expressed through
    bean property 'foo'; nested exception is
    org.springframework.beans.factory
       .NoUniqueBeanDefinitionException:
    No qualifying bean of type
    'com.apress.prospring5.ch3.xml.complicated.Foo'
    available:
    expected single matching bean but found 2: fooOne,fooTwo
    ...
```

Выводимый на консоль результат весьма многословен, но из первых его строк можно выяснить возникшее затруднение. В данном случае каркас Spring сгенерировал исключение типа `UnsatisfiedDependencyException` с подробным описание ошибки, поскольку ему неизвестно, какой именно компонент Spring Bean необходимо привязать автоматически. Из этого описания следует, что каркас Spring обнаружил компоненты, но не в состоянии выбрать из них тот, который требуется применить. Данное затруднение можно разрешить двумя способами. Первый способ состоит в том, чтобы ввести атрибут `primary` в определение того компонента Spring Bean, который должен быть привязан автоматически первым, установив в этом атрибуте логическое значение `true`, как показано ниже.

```
<beans ...>
  <bean id="fooOne"
    class="com.apress.prospring5.ch3.xml
              .complicated.FooImpl1"
    primary="true"/>

  <bean id="fooTwo"
    class="com.apress.prospring5.ch3.xml
```

```
            .complicated.FooImpl2"/>

    <bean id="bar" class="com.apress.prospring5
                          .ch3.xml.Bar"/>

    <bean id="targetByType" autowire="byType"
      class="com.apress.prospring5.ch3.xml
                .complicated.CTarget"
      lazy-init="true"/>
</beans>
```

Если внести приведенные выше коррективы в конфигурацию, то выполнение исходного кода из данного примера даст следующий результат:

```
Using byType:

Property bar set
Property fooOne set
Property fooTwo set
```

Итак, все приведено в норму. Однако решение ввести атрибут `primary` пригодно лишь в том случае, если имеются два родственных типа компонентов Spring Beans. А если их больше, то избавиться от исключения типа `UnsatisfiedDependency Exception` с помощью атрибута `primary` все равно не удастся. В таком случае положение можно исправить вторым способом, позволяющим полностью контролировать, где именно внедряются компоненты Spring Beans. С этой целью компоненты Spring Beans именуются и конфигурируются в формате XML для внедрения в определенном месте.

В предыдущем примере продемонстрирована довольно сложная и неопрятная реализация, предложенная лишь для того, чтобы убедиться, что каждый из автоматически связываемых типов может быть сконфигурирован в формате XML. Но совсем другое дело, если перейти к конфигурированию с помощью аннотаций. Для атрибута `lazy-init` имеется равнозначная аннотация `@Lazy`, применяемая на уровне класса для объявления компонентов Spring Beans, экземпляры которых получаются при первом обращении к ним. С помощью стереотипных аннотаций можно составить лишь одну конфигурацию для компонента Spring Bean, и поэтому кажется вполне логичным, что имена компонентов Spring Beans не имеют особого значения, поскольку для каждого типа предполагается лишь один компонент. Так, если составить конфигурацию с помощью упомянутой выше аннотации, то по умолчанию будет выбран режим автосвязывания `byType`. А если имеются компоненты Spring Beans родственных типов, то целесообразно выбрать режим автосвязывания `byName`. Это можно сделать с помощью аннотации `@Qualifier` вместе с аннотацией `@Autowired`, указав имя внедряемого компонента Spring Bean в качестве аргумента.

Рассмотрим в качестве примера следующий фрагмент кода:

```
package com.apress.prospring5.ch3.sandbox;

import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.beans.factory
          .annotation.Qualifier;
import org.springframework.context.annotation.Lazy;
import org.springframework.context.support
          .GenericXmlApplicationContext;
import org.springframework.stereotype.Component;

@Component
@Lazy
public class TrickyTarget {

  Foo fooOne;
  Foo fooTwo;
  Bar bar;

  public TrickyTarget() {
    System.out.println("Target.constructor()");
  }

  public TrickyTarget(Foo fooOne) {
    System.out.println("Target(Foo) called");
  }

  public TrickyTarget(Foo fooOne, Bar bar) {
    System.out.println("Target(Foo, Bar) called");
  }

  @Autowired
  public void setFooOne(Foo fooOne) {
    this.fooOne = fooOne;
    System.out.println("Property fooOne set");
  }

  @Autowired
  public void setFooTwo(Foo foo) {
    this.fooTwo = foo;
    System.out.println("Property fooTwo set");
  }

  @Autowired
  public void setBar(Bar bar) {
    this.bar = bar;
    System.out.println("Property bar set");
  }
```

```
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-04.xml");
    ctx.refresh();

    TrickyTarget t = ctx.getBean(TrickyTarget.class);

    ctx.close();
  }
}
```

Если определить класс `Foo` следующим образом:

```
package com.apress.prospring5.ch3.sandbox;

@Component
  public class Foo {
}
```

то выполнение исходного кода из класса `TrickyTarget` даст следующий результат:

```
Property fooOne set
Property fooTwo set
Property bar set
```

Столь же просто определяется и класс `Bar`, как показано ниже.

```
package com.apress.prospring5.ch3.sandbox;

import org.springframework.stereotype.Component;

@Component
public class Bar {
}
```

Если видоизменить класс `TrickyTarget`, присвоив имя компоненту Spring Bean следующим образом:

```
@Component("gigi")
@Lazy
public class TrickyTarget {
...
}
```

то выполнение его исходного кода даст такой же самый результат, поскольку имеется лишь один компонент Spring Bean типа `Target`. И когда делается запрос контекста через вызов `ctx.getBean(TrickyTarget.class)`, то из контекста возвращается единственный компонент Spring Bean данного типа, каким бы ни было его имя. А если задать имя компонента Spring Bean типа `Bar` следующим образом:

```
package com.apress.prospring5.ch3.sandbox;

import org.springframework.stereotype.Component;

@Component("kitchen")
public class Bar {
}
```

и снова выполнить исходный код из данного примера, то на консоль будет выведен тот же самый результат. Это означает, что по умолчанию выбирается режим автосвязывания byType.

Как упоминалось ранее, дело несколько усложняется, если типы компонентов Spring Beans оказываются родственными. Чтобы убедиться в этом, преобразуем класс Foo в интерфейс и объявим по отдельности два реализующих его типа компонентов Spring Beans. Оставим также стандартную конфигурацию без дополнительного именования компонентов Spring Beans.

```
package com.apress.prospring5.ch3.sandbox;

// исходный файл Foo.java
public interface Foo {
  // пустой интерфейс interface, используемый как маркерный
}

// исходный файл FooImplOne.java
@Component
public class FooImplOne implements Foo {
}

// исходный файл FooImplTwo.java
@Component
public class FooImplTwo implements Foo{
}
```

Класс TrickyTarget остается без изменения, и если выполнить его исходный код, то выводимый на консоль результат изменится и, вероятнее всего, будет выглядеть следующим образом:

```
Property bar set
Exception in thread "main"
    org.springframework.beans.factory
       .UnsatisfiedDependencyException:
  Error creating bean with name 'gigi':
    Unsatisfied dependency expressed through method
    'setFoo' parameter 0;
  nested exception is org.springframework.beans.factory
                        .NoUniqueBeanDefinitionException:
  No qualifying bean of type
       'com.apress.prospring5.ch3.sandbox.Foo' available:
```

```
    expected single matching bean but found 2:
    fooImplOne,fooImplTwo
...
```

Выводимый на консоль результат весьма многословен, но из первых его строк можно выяснить возникшее затруднение. В данном случае каркас Spring сообщает, что ему неизвестно, какой именно компонент Spring Bean следует автоматически привязать через метод `setFoo()`. Он также сообщает, какие именно компоненты Spring Beans им выбраны. Имена компонентов Spring Beans определяются по имени класса, где первая буква становится строчной. На основании этих сведений можно внести необходимые коррективы в исходный код класса `TrickyTarget`. И сделать это можно двумя способами. Первый способ состоит в том, чтобы ввести аннотацию `@Primary`, равнозначную представленному ранее атрибуту `primary`, в класс, определяющий компонент Spring Bean. Эта аннотация предписывает каркасу Spring отдать предпочтение данному компоненту Spring Bean при автосвязывании по типу. Итак, снабдим аннотацией класс `FooImplOne`, как показано ниже.

```
package com.apress.prospring5.ch3.sandbox;

import org.springframework.context.annotation.Primary;
import org.springframework.stereotype.Component;

@Component
@Primary
public class FooImplOne implements Foo {
}
```

Аннотация `@Primary` действует как маркерный интерфейс и не имеет атрибутов. Ее наличие в конфигурации компонента Spring Bean помечает его как приоритетный, когда требуется автоматически привязать компонент Spring Bean данного типа в режиме `byType`. Если выполнить исходный код класса `TrickyTarget` снова, то на консоль будет выведен вполне ожидаемый результат:

```
Property fooOne set
Property fooTwo set
Property bar set
```

Как и атрибут `primary`, аннотация `@Primary` приносит пользу лишь в том случае, если имеются два родственных типа компонентов Spring Beans. А если их окажется больше, то больше пользы принесет аннотация `@Qualifier`. Эта аннотация размещается сразу же после аннотации `@Autowired` в объявлениях неоднозначно функционирующих методов установки (в данном случае — `setFooOne()` и `setFooTwo()`). Ниже приведен соответствующий код, а код, оставшийся без изменений, не показан ради краткости примера.

```
@Component("gigi")
@Lazy
```

```
public class TrickyTarget {
  ...
  @Autowired
  @Qualifier("fooImplOne")
  public void setFoo(Foo foo) {
    this.foo = foo;
    System.out.println("Property fooOne set");
  }

  @Autowired
  @Qualifier("fooImplTwo")
  public void setFooTwo(Foo fooTwo) {
    this.fooTwo = fooTwo;
    System.out.println("Property fooTeo set");
  }
  ...
}
```

Если теперь выполнить исходный из данного примера, то на консоль будет снова выведен вполне ожидаемый результат:

```
Property fooOne set
Property fooTwo set
Property bar set
```

Если же прибегнуть к конфигурированию на Java, то изменить придется лишь порядок определения компонентов Spring Beans. В этом случае вместо аннотации `@Component` в классах компонентов Spring Beans будет применяться аннотация `@Bean` в конфигурационном классе, где объявляются методы определения компонентов Spring Beans. Ниже приведен пример такого класса.

```
package com.apress.prospring5.ch3.config;

import com.apress.prospring5.ch3.sandbox.*;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context
          .annotation.Configuration;
import org.springframework.context.support
          .GenericApplicationContext;

public class TargetDemo {
  @Configuration
  static class TargetConfig {
    @Bean
    public Foo fooImplOne() {
      return new FooImplOne();
    }
```

```
    @Bean
     public Foo fooImplTwo() {
      return new FooImplTwo();
    }

    @Bean
    public Bar bar() {
      return new Bar();
    }

    @Bean
    public TrickyTarget trickyTarget() {
      return new TrickyTarget();
    }
  }

  public static void main(String args) {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
                                TargetConfig.class);
    TrickyTarget t = ctx.getBean(TrickyTarget.class);
    ctx.close();
  }
}
```

И в этом случае классы, входящие в пакет `com.apress.prospring5.ch3.sandbox`, используются повторно, чтобы исключить дублирование кода, поскольку режим просмотра компонентов не активизирован, а следовательно, любые определения компонентов Spring Beans с помощью стереотипных аннотаций будут проигнорированы. Если выполнить исходный код приведенного выше класса, то на консоль будет выведен такой же результат, как и прежде. Как упоминалось ранее, при определении компонентов Spring Beans с помощью аннотации `@Bean` именем компонента становится имя метода, и поэтому класс `TrickyTarget`, сконфигурированный с помощью аннотации `@Qualifier`, будет по-прежнему действовать, как и предполагалось.

Когда следует применять автосвязывание

Как правило, ответ на вопрос, следует ли применять автосвязывание, оказывается определенно отрицательным. Автосвязывание может сэкономить время в мелких приложениях, но во многих случаях оно приводит к неудачным решениям и потере гибкости в крупных приложениях. Применение режима `byName` кажется неплохой идеей, но может потребовать присваивания искусственных имен свойствам в классах, чтобы выгодно воспользоваться функциональными возможностями автосвязывания. Весь замысел Spring в том и состоит, что вы создаете свои классы как вам угодно и позволяете Spring работать с ними, а не наоборот. Искушение воспользоваться режимом автосвязывания `byType` будет возникать до тех пор, пока вы не поймете, что

в контексте типа `ApplicationContext` может существовать только один компонент Spring Bean каждого типа. Такое ограничение вызывает затруднения, когда требуется обслуживать компоненты Spring Beans одного и того же типа, но с разными конфигурациями. То же самое касается и режима автосвязывания `constructor`.

В ряде случаев автосвязывание может сберечь время, но определение связывания вручную потребует не так много дополнительных усилий, чтобы отказываться от преимуществ ясной семантики и полной свободы в отношении именования свойств и произвольного количества экземпляров одного и того же типа, которыми можно манипулировать. Разрабатывая любое нетривиальное приложение, старайтесь держаться подальше от автосвязывания.

Настройка наследования компонентов Spring Beans

Иногда может понадобиться целый ряд определений компонентов Spring Beans одного и того же типа или реализация общего для них интерфейса. Это может вызвать трудности, если компоненты Spring Beans должны совместно использовать одни общие параметры конфигурации, но не другие. Процесс поддержания общих параметров конфигурации в синхронизированном состоянии подвержен ошибкам, а в крупных проектах способен отнять немало времени. Чтобы обойти подобное препятствие, в Spring допускается предоставлять в дескрипторе разметки `<bean>` определение такого компонента Spring Bean, который наследует настройки своих свойств от другого компонента в том же контексте типа `ApplicationContext`. При необходимости значения любых свойств порожденного компонента можно переопределить, что дает полный контроль над ними, но родительский компонент может предоставлять каждому порожденному компоненту базовую конфигурацию. В следующем фрагменте кода приведена простая конфигурация из файла `app-context-xml.xml` с двумя компонентами Spring Beans, один из которых является порожденным по отношению к другому.

```
<beans ...>
  <bean id="parent"
    class="com.apress.prospring5.ch3.xml.Singer"
    p:name="John Mayer" p:age="39"/>

  <bean id="child"
    class="com.apress.prospring5.ch3.xml.Singer"
    parent="parent" p:age="0"/>
</beans>
```

В приведенном выше коде конфигурации дескриптор `<bean>` с определением компонента `child` содержит дополнительный атрибут `parent`, где указывается, что каркас Spring должен считать компонент `inheritParent` родительским для определяемого компонента. Если же определение родительского компонента не должно

быть доступным для поиска из контекста типа ApplicationContext, в дескриптор разметки <bean> с объявлением родительского компонента можно ввести атрибут abstract="true". А поскольку у компонента child имеется собственное значение для свойства age, то Spring передаст ему это значение. Но у компонента child отсутствует значение для свойства name, и поэтому Spring использует значение, заданное для компонента parent. Ниже приведен пример довольно простого компонента Spring Bean типа Singer.

```
package com.apress.prospring5.ch3.xml;

public class Singer {
  private String name;
  private int age;

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

  public void setAge(int age) {
    this.age = age;
  }

  public String toString() {
    return "\tName: " + name + "\n\t" + "Age: " + age;
  }
}
```

Для его проверки можно написать следующий простой класс:

```
package com.apress.prospring5.ch3.xml;

import org.springframework.context.support
          .GenericXmlApplicationContext;

public class InheritanceDemo {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();
    Singer parent = (Singer) ctx.getBean("parent");
    Singer child = (Singer) ctx.getBean("child");
    System.out.println("Parent:\n" + parent);
    System.out.println("Child:\n" + child);
  }
}
```

Как видите, метод main() из класса Singer извлекает компоненты child и parent из контекста типа ApplicationContext и направляет содержимое их

свойств в стандартный поток вывода stdout. Ниже приведен результат, выводимый на консоль при выполнении исходного кода из данного примера. Как и следовало ожидать, компонент child унаследовал значение своего свойства name от компонента parent, но предоставил собственное значение для свойства age.

```
Parent:
        Name: John Mayer
        Age: 39

Child:
       Name: John Mayer
       Age: 0
```

Порожденные компоненты Spring Beans наследуют аргументы конструкторов и значения свойств от родительских компонентов, поэтому при наследовании компонентов Spring Beans можно использовать оба способа внедрения зависимостей. Такой уровень гибкости превращает наследование компонентов Spring Beans в весьма эффективный инструмент для построения приложений с большим количеством определений подобных компонентов. Объявляя целый ряд компонентов Spring Beans с одинаковыми значениями в общих для них свойствах, следует не поддаваться искушению применять копирование и вставку для дублирования этих значений, а вместо этого настроить надлежащим образом иерархию наследования в своей конфигурации.

Однако не следует забывать, что наследование компонентов Spring Beans совсем не обязательно должно совпадать с иерархией наследования классов Java. Вполне допустимо использовать наследование, например, для пяти компонентов Spring Beans одного и того же типа. Наследование компонентов Spring Beans больше похоже на создание шаблонов, чем на само наследование. Но, изменяя тип порожденного компонента Spring Bean, следует иметь в виду, что этот тип должен расширять тип родительского компонента.

Резюме

В этой главе было разъяснено немало много общих понятий, связанных с ядром Spring и инверсией управления. В ней были продемонстрированы примеры различных механизмов инверсии управления и обсуждены достоинства и недостатки применения каждого ее механизма в разрабатываемых приложениях. По ходу изложения материала этой главы было выяснено, какие механизмы инверсии управления поддерживаются в Spring и когда следует применять их в приложениях. С этой целью был представлен интерфейс BeanFactory как главный компонент для реализации функциональных возможностей инверсии управления в Spring, а также интерфейс ApplicationContext, расширяющий интерфейс BeanFactory с целью предоставить дополнительные функциональные возможности. При обсуждении интерфейса ApplicationContext основное внимание было уделено классу GenericXml

`ApplicationContext`, допускающему внешнее конфигурирование Spring в формате XML. Был рассмотрен и другой способ объявления требований к интерфейсу `ApplicationContext` для внедрения зависимостей — с помощью аннотаций Java. Кроме того, был приведен ряд примеров применения конфигурационного класса `AnnotationConfigApplicationContext` на языке Java, чтобы постепенно раскрыть подобный способ конфигурирования компонентов Spring Beans.

В этой главе был представлен основной набор функциональных средств инверсии управления в Spring, в том числе внедрение зависимостей через метод установки, конструктор, метод класса, автосвязывание и наследование компонентов Spring Beans. При обсуждении вопросов конфигурирования было показано, каким образом можно настраивать свойства компонентов Spring Beans на самые разные значения, в том числе из других компонентов и конфигураций, составляемых в формате XML и с помощью аннотаций, а также класса `GenericXmlApplicationContext`.

В этой главе мы лишь слегка коснулись ядра Spring и встроенного в этот каркас контейнера инверсии управления. В следующей главе мы обсудим ряд специальных средств Spring, связанных с инверсией управления, и более подробно рассмотрим другие функциональные возможности, доступные в ядре Spring Core.

ГЛАВА 4

Конфигурирование и начальная загрузка в Spring

В предыдущей главе мы представили принцип инверсии управления (IoC) и показали, каким образом он вписывается в каркас Spring Framework. Тем не менее мы лишь слегка коснулись функциональных возможностей ядра Spring Core. В каркасе Spring предоставляется обширный ряд служб, дополняющих и расширяющих его основные возможности в отношении инверсии управления. В этой главе мы собираемся рассмотреть подобные возможности более подробно. В частности, мы остановимся на следующих вопросах.

- **Управление жизненным циклом компонентов Spring Beans.** Все представленные до сих пор компоненты Spring Beans были довольно просты и совершенно не привязаны к контейнеру инверсии управления в Spring. В этой части будут представлены стратегии, которые дают компонентам Spring Beans возможность получать уведомления из контейнера инверсии управления в Spring в разные моменты их жизненного цикла. Добиться этого можно, реализовав специальные интерфейсы, предоставляемые в Spring, указав методы, которые Spring может вызывать через рефлексию, или применив аннотации жизненного цикла компонентов JavaBeans, описанные в спецификации JSR-250.
- **Информирование компонентов об их контексте в Spring.** Иногда требуется, чтобы компонент Spring Bean имел возможность взаимодействовать со сконфигурировавшим его экземпляром типа `ApplicationContext`. Для этих целей в Spring предоставляются два интерфейса, `BeanNameAware` и `ApplicationContextAware` (этот интерфейс был представлен в конце главы 3), с помощью которых компонент Spring Bean может получить присвоенное ему имя и ссылку на его контекст типа `ApplicationContext` соответственно. В этой части рассматриваются реализации обоих интерфейсов и приводятся некоторые соображения по поводу их практического применения в приложении.

- **Применение фабрик компонентов Spring Beans.** Интерфейс `FactoryBean` предназначен, как следует из его имени, для реализации в любом компоненте Spring Bean, который служит в качестве фабрики для других компонентов. В интерфейсе `FactoryBean` предоставляется механизм, позволяющий легко интегрировать собственные фабрики с помощью интерфейса `BeanFactory` из каркаса Spring.
- **Работа с редакторами свойств компонентов JavaBeans.** В пакете `java.beans` предоставляется стандартный интерфейс `PropertyEditor`. Редакторы свойств, реализующие интерфейс `PropertyEditor`, служат для взаимного преобразования значений свойств и их строковых представлений типа `String`. Эти редакторы широко применяются в Spring, главным образом, для чтения значений, указанных в конфигурации по интерфейсу `BeanFactory`, а также для их приведения к нужному типу данных. В этой части обсуждается ряд редакторов свойств, предоставляемых в Spring, показывается, как пользоваться ими в приложениях, а также кратко поясняется, как самостоятельно реализовать специальные редакторы свойств.
- **Подробное описание интерфейса `ApplicationContext`.** Как вам должно быть уже известно, интерфейс `ApplicationContext` расширяет интерфейс `BeanFactory` для применения в полнофункциональных приложениях. В интерфейсе `ApplicationContext` предоставляется полезный ряд дополнительных функциональных возможностей, включая поддержку интернационализированных сообщений, загрузку ресурсов и публикацию событий. В этой части главы подробно рассматриваются функциональные средства, дополняющие возможности инверсии управления, предоставляемые в интерфейсе `ApplicationContext`. Забегая немного вперед, здесь также показано, каким образом интерфейс `ApplicationContext` упрощает применение Spring при построении веб-приложений.
- **Применение классов Java для конфигурирования.** До версии 3.0 каркас Spring поддерживал только базовую конфигурацию в формате XML с аннотациями для конфигурирования компонентов Spring Beans и зависимостей. Начиная с версии 3.0 в Spring разработчикам предлагается еще один способ конфигурирования интерфейса `ApplicationContext` с помощью классов Java. Этот новый способ конфигурирования приложений Spring вкратце рассматривается в данной части.
- **Применение модуля Spring Boot.** Конфигурирование приложений Spring можно сделать еще более практичным с помощью модуля Spring Boot. Этот модуль (и одноименный проект Spring) настолько упрощает создание автономных приложений промышленного уровня на основе Spring, что их остается лишь запустить на выполнение.
- **Применение функциональных средств, совершенствующих конфигурирование.** В этой части представлены функциональные средства, упрощающие

конфигурирование приложений, в том числе управление профилями, абстракция среды и источников свойств и т.д. Здесь показывается, как применять эти функциональные средства для удовлетворения конкретных потребностей в конфигурировании.

- **Применение языка Groovy для конфигурирования.** В версии Spring 4.0 появилась новая возможность для конфигурирования определений компонентов Spring Beans на языке Groovy, который можно применять в качестве альтернативы или дополнения существующих способов конфигурирования средствами XML и Java.

Влияние Spring на переносимость приложений

Большинство функциональных средств, обсуждаемых в этой главе, характерны для Spring и во многих случаях недоступны в других контейнерах инверсии управления. Несмотря на то что во многих контейнерах инверсии управления поддерживаются функциональные возможности управления жизненным циклом, это делается в них, вероятнее всего, через ряд интерфейсов, отличающихся от тех, что предлагаются в Spring. Если переносимость приложения между разными контейнерами инверсии управления действительно имеет значение, можно отказаться от применения тех средств, которые привязывают приложение к каркасу Spring.

Не следует, однако, забывать, что если наложить ограничение, обеспечивающее переносимость приложения между контейнерами инверсии управления, то можно утратить все богатство функциональных возможностей, предоставляемых в Spring. Но раз уж принято стратегическое решение применять каркас Spring, то имеет смысл извлечь из него максимум возможностей.

Остерегайтесь составлять требование о переносимости на пустом месте. Зачастую конечных пользователей вообще не волнует, что приложение может выполняться в трех разных контейнерах инверсии управления, поскольку им достаточно его запустить на выполнение. Как показывает наш опыт, разработчики нередко совершают ошибку, пытаясь построить приложение на основе наименьшего общего знаменателя функциональных средств, доступных в выбранной технологии. Это может с самого начала поставить создаваемое приложение в невыгодные условия. Но если приложение должно быть переносимым между контейнерами инверсии управления, то не следует рассматривать это требование как недостаток, поскольку это нормальное требование, которому, помимо прочих, должно удовлетворять приложение. В книге *Expert One-on-One: J2EE Development without EJB* Рода Джонсона и Юргена Хёллера (Rod Johnson, Jürgen Höller; издательство Wrox, 2004 г.) такого рода требования называются фантомными. Там же можно найти подробное обсуждение их влияния на разрабатываемые проекты.

Несмотря на то что применение рассматриваемых здесь функциональных средств способно привязать приложение к каркасу Spring Framework, на самом деле перено-

симость приложения расширяется в намного более широких пределах. Ведь в данном случае речь идет о свободно доступном каркасе с открытым кодом, который не принадлежит какому-то конкретному поставщику. Приложение, построенное с помощью контейнера инверсии управления в Spring, выполняется везде, где функционирует код Java. В отношении переносимости каркас Spring открывает новые возможности для корпоративных приложений Java. В каркасе Spring предоставляется немало из тех средств, которые доступны на платформе JEE, а также поддерживаются классы для абстрагирования и упрощения многих других аспектов JEE. Как правило, средствами Spring можно построить веб-приложение, которое выполняется в простом контейнере сервлетов, но с таким же уровнем сложности, как и у приложения, ориентированного на полнофункциональный сервер приложений на платформе JEE. Связываясь с каркасом Spring, вы тем самым повышаете степень переносимости своего приложения, заменяя многие функциональные средства, характерные для отдельного поставщика или зависящие от конкретной конфигурации этого поставщика, равнозначными функциональными средствами Spring.

Управление жизненным циклом компонентов Spring Beans

Важной особенностью любого контейнера инверсии управления, в том числе и в Spring, является возможность построения компонентов таким образом, чтобы получать уведомления в определенные моменты своего жизненного цикла. Это дает компонентам возможность выполнять надлежащую обработку в такие моменты их жизненного цикла. В общем, непосредственное отношение к компонентам имеют два события, наступающие в течение жизненного цикла: после инициализации и перед уничтожением компонентов.

В контексте Spring событие после инициализации наступает в тот момент, когда завершается установка всех значений свойств компонента Spring Bean и все проверки зависимостей, сконфигурированные для выполнения. А событие перед уничтожением наступает непосредственно перед тем, как Spring приступит к уничтожению экземпляра компонента Spring Bean. Но для компонентов Spring Beans с областью видимости на уровне прототипа событие перед уничтожением не будет инициировано в каркасе Spring. Этот каркас спроектирован таким образом, чтобы методы обратного вызова жизненного цикла, выполняющие инициализацию компонента Spring Bean, вызывались для объектов независимо от области видимости данного компонента. Хотя для компонентов Spring Beans с областью видимости на уровне прототипа методы обратного вызова, выполняющие уничтожение компонента Spring Bean, вызываться не будут. В каркасе Spring предоставляются три механизма, которые можно применять в компоненте Spring Bean для привязки к каждому из этих событий и выполнения дополнительной обработки. Эти механизмы основаны на интерфейсах, методах и аннотациях соответственно.

Если в компоненте Spring Bean применяется механизм, основанный на интерфейсах, то в нем реализуется интерфейс, характерный для того типа уведомлений, которые необходимо получать, а каркас Spring посылает уведомления данному компоненту через метод обратного вызова, определенный в этом интерфейсе. Если же применяется механизм, основанный на методах, то в Spring допускается указывать в конфигурации интерфейса `ApplicationContext` имя метода, который должен быть вызван при инициализации компонента Spring Bean, а также имя метода, который должен быть вызван при уничтожении данного компонента. А механизм, основанный на аннотациях, предполагает применение аннотаций по спецификации JSR-250 для указания методов, которые должны вызываться после построения и перед уничтожением компонентов Spring Beans.

В отношении обоих упомянутых выше событий эти механизмы достигают одной и той же цели. Механизм, основанный на интерфейсах, широко применяется непосредственно в Spring, поэтому не обязательно помнить, что следует указывать инициализацию и уничтожение всякий раз, когда применяются компоненты Spring Beans. Но в собственных компонентах, возможно, лучше применять механизм, основанный на методах или аннотациях, чтобы не реализовывать в них специальные интерфейсы из Spring. И хотя выше утверждалось, что переносимость зачастую оказывается не настолько важной, как обычно заявляется в многочисленной литературе на данную тему, это совсем не означает, что переносимостью можно пожертвовать, когда имеется вполне пригодная альтернатива. Иными словами, если вы привязываете свое приложение к каркасу Spring другими способами, то, применяя интерфейсный метод, можете определить обратный вызов лишь один раз и просто забыть о нем. А если вы определяете целый ряд однотипных компонентов Spring Beans, которым требуются уведомления в течение их жизненного цикла, то, применяя механизм, основанный на интерфейсах, вы можете избавить себя от необходимости указывать в XML-файле конфигурации методы обратного вызова жизненного цикла для каждого компонента Spring Bean. Применение аннотаций по спецификации JSR-250 — еще один приемлемый способ, поскольку они соответствуют стандарту, определенному в организации JCP, а вы не привязываетесь непосредственно к аннотациям, характерным только для Spring. Для этого достаточно убедиться, что в том контейнере инверсии управления, где выполняется приложение, поддерживается стандарт JSR-250.

В целом выбор механизма, который будет применяться для получения уведомлений в течение жизненного цикла компонентов Spring Beans, зависит от требований к конкретному приложению. Так, если вас заботит переносимость или вы просто определяете один или два компонента Spring Beans заданного типа, где требуются обратные вызовы, выберите механизм, основанный на методах. Если же вы пользуетесь конфигурацией с аннотациями и уверены, что в целевом контейнере инверсии управления поддерживается стандарт JSR-250, выберите механизм, основанный на аннотациях. А если вы не слишком обеспокоены переносимостью или определяете целый ряд однотипных компонентов Spring Beans, которым требуются уведомления в течение их жизненного цикла, то, выбрав механизм, основанный на интерфейсах, вы обе-

спечите наилучшим образом получение компонентами ожидаемых уведомлений. И, наконец, если вы собираетесь применять какой-нибудь компонент Spring Bean в самых разных проектах, то, скорее всего, стремитесь к тому, чтобы функциональные возможности этого компонента были в наибольшей степени самодостаточными. В таком случае вам определенно понадобится механизм, основанный на интерфейсах.

На рис. 4.1 схематически показано общее представление о том, каким образом осуществляется управление жизненным циклом компонентов Spring Beans в их контейнере инверсии управления.

Создание компонентов Spring Beans

Если компонент Spring Bean информирован о том, когда он был инициализирован, такой компонент может проверить, все ли требуемые зависимости удовлетворены. Несмотря на то что каркас Spring способен автоматически проверять зависимости, он действует по принципу "все или ничего", не предоставляя никаких возможностей для применения дополнительной логики в процедуре разрешения зависимостей. Рассмотрим в качестве примера компонент Spring Bean с четырьмя зависимостями, объявленными через методы установки, причем две из них являются обязательными,

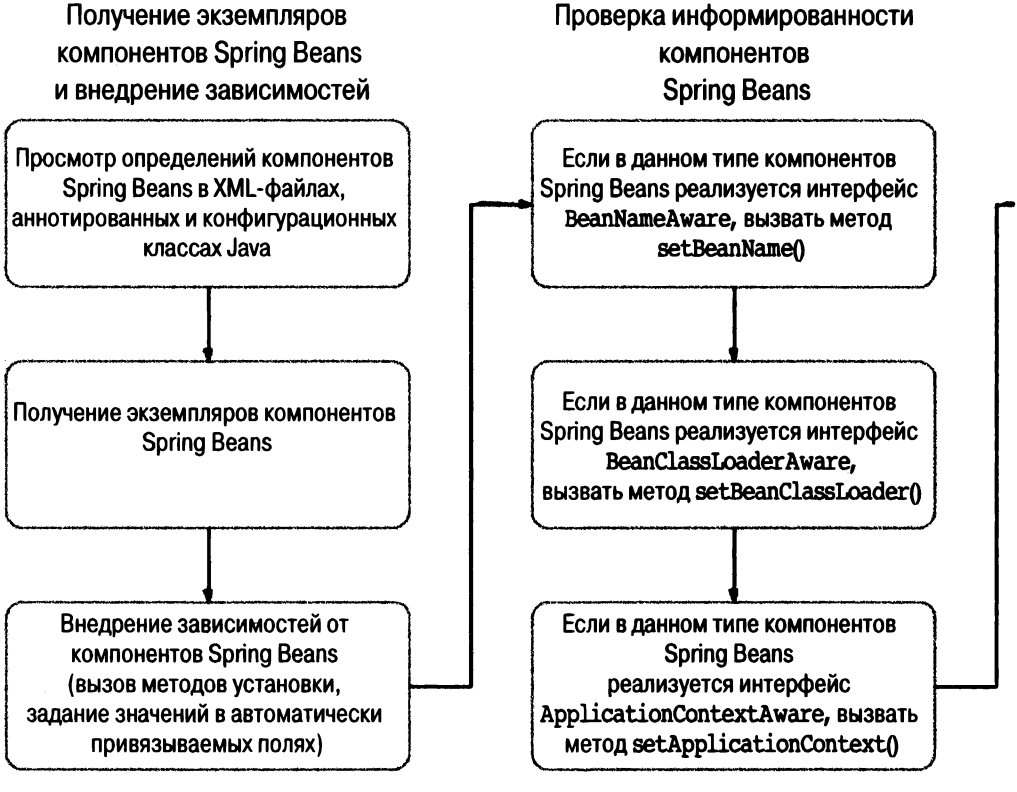


Рис. 4.1. Жизненный цикл компонентов Spring Beans

а еще одна устанавливает подходящее значение по умолчанию на тот случай, когда зависимость не предоставлена. Используя обратный вызов при инициализации, данный компонент может проверить требующиеся ему зависимости и сгенерировать, если потребуется, исключение или же предоставить значение, устанавливаемое по умолчанию.

Данный компонент Spring Bean не сможет выполнить подобные проверки в своем конструкторе, потому что до этого момента у каркаса Spring не было возможности предоставить значения для зависимостей. Обратный вызов при инициализации инициируется после того, как каркас Spring завершит предоставление зависимостей и выполнит любые запрашиваемые проверки зависимостей.

Обратный вызов при инициализации не ограничивается одной лишь проверкой зависимостей, а позволяет делать все, что угодно, хотя он приносит наибольшую пользу именно для описанной выше цели. Зачастую обратный вызов при инициализации служит также удобным местом для запуска любых действий, которые компонент Spring Bean должен выполнять автоматически, реагируя на свою конфигурацию. Так, если строится компонент Spring Bean для выполнения запланированных заданий, то обратный вызов при инициализации окажется идеальным местом для запуска

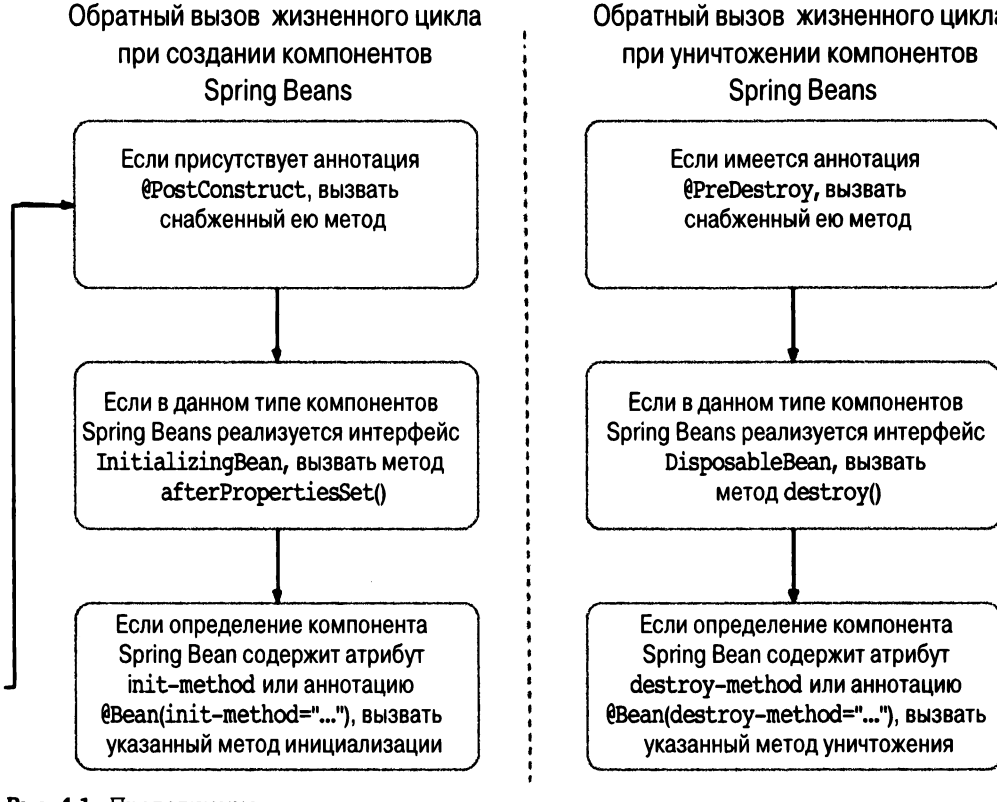


Рис. 4.1. Продолжение

планировщика заданий. Ведь данные конфигурации этого компонента все равно уже установлены.

На заметку Вам не придется самостоятельно разрабатывать компонент Spring Bean для запуска запланированных заданий, поскольку Spring может это делать автоматически с помощью встроенного средства планирования заданий или посредством интеграции с планировщиком заданий Quartz. Более подробно эти вопросы рассматриваются в главе 11.

Выполнение метода при создании компонента Spring Bean

Как упоминалось ранее, один из способов получения обратного вызова при инициализации предусматривает назначение конкретного метода из компонента Spring Bean в качестве метода инициализации и уведомление об этом каркаса Spring. Такой механизм обратного вызова оказывается удобным в тех случаях, когда имеется лишь один компонент Spring Bean определенного типа или требуется, чтобы приложение оставалось непривязанным к Spring. Еще одна причина применения этого механизма состоит в том, чтобы обеспечить взаимодействие приложения Spring с построенными ранее сторонними компонентами Spring Beans. Чтобы указать метод обратного вызова, достаточно задать его имя в атрибуте `init-method` того дескриптора разметки `<bean>`, в котором определяется компонент Spring Bean. В следующем примере кода демонстрируется простой компонент Spring Bean с двумя зависимостями:

```
package com.apress.prospring5.ch4;

import org.springframework.beans.factory
          .BeanCreationException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support
          .GenericXmlApplicationContext;

public class Singer {
  private static final String DEFAULT_NAME = "Eric Clapton";

  private String name;
  private int age = Integer.MIN_VALUE;

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

  public void setAge(int age) {
    this.age = age;
  }
```

```
  public void init() {
    System.out.println("Initializing bean");

    if (name == null) {
      System.out.println("Using default name");
      name = DEFAULT_NAME;
    }

    if (age == Integer.MIN_VALUE) {
      throw new IllegalArgumentException("You must set "
              + " the age property of any beans of type "
              + Singer.class);
    }
  }

  public String toString() {
    return "\tName: " + name + "\n\tAge: " + age;
  }

  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
           new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();

    getBean("singerOne", ctx);
    getBean("singerTwo", ctx);
    getBean("singerThree", ctx);

    ctx.close();
  }

  public static Singer getBean(String beanName,
                       ApplicationContext ctx) {
    try {
      Singer bean = (Singer) ctx.getBean(beanName);
      System.out.println(bean);
      return bean;
    } catch (BeanCreationException ex) {
      System.out.println("An error occured in bean "
               + "configuration: " + ex.getMessage());
      return null;
    }
  }
}
```

Обратите внимание на то, что в данном примере кода определяется метод `init()`, который будет обратно вызываться при инициализации. В методе `init()` проверяется, установлено ли свойство `name`, и если оно не установлено, то для его инициали-

зации используется значение по умолчанию, хранящееся в константе DEFAULT_NAME. Кроме того, в методе init() проверяется, установлено ли свойство age, и если оно не установлено, то генерируется исключение IllegalArgumentException.

В методе main() из класса Singer предпринимается попытка получить три компонента Spring Beans типа Singer из контекста типа GenericXmlApplicationContext с помощью метода getBean() из данного класса. Если компонент Spring Bean удастся получить, то из метода getBean() на консоль выводятся подробные сведения о нем. А если в методе init() возникнет исключение, как в том случае, если свойство age не установлено, то Spring заключит его в оболочку исключения типа BeanCreationException. Такие исключения перехватываются в методе getBean(), и на консоль выводится сообщение, извещающее о возникшей ошибке, а также возвращается пустое значение null.

В следующем фрагменте кода из файла app-context-xml.xml приведена конфигурация контекста типа ApplicationContext, в которой определяются компоненты Spring Beans, применяемые в предыдущем примере кода:

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:p="http://www.springframework.org/schema/p"
  xsi:schemaLocation=
     "http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans
          /spring-beans.xsd"
    default-lazy-init="true">

    <bean id="singerOne"
       class="com.apress.prospring5.ch4.Singer"
       init-method="init" p:name="John Mayer" p:age="39"/>

    <bean id="singerTwo"
       class="com.apress.prospring5.ch4.Singer"
       init-method="init" p:age="72"/>

    <bean id="singerThree"
       class="com.apress.prospring5.ch4.Singer"
       init-method="init" p:name="John Butler"/>
</beans>
```

Как видите, дескрипторы разметки <bean> в определениях всех трех компонентов Spring Beans содержат атрибут init-method, в котором каркасу Spring предписывается вызвать метод init(), как только он завершит конфигурирование соответствующего компонента Spring Bean. В определении компонента singerOne заданы значения обоих свойств, name и age, поэтому он не подвергается никаким изменениям в методе init(). А в определении компонента singerTwo не задано значение

свойства name, и поэтому данному свойству присваивается значение по умолчанию в методе init(). И, наконец, в определении компонента singerThree не задано значение свойства age. Логика, реализованная в методе init(), интерпретирует это как ошибку, и поэтому генерируется исключение типа IllegalArgumentException. Обратите также внимание на то, что в дескриптор разметки <beans> был введен атрибут default-lazy-init="true", который предписывает каркасу Spring получать экземпляры компонентов Spring Beans, определенных в файле конфигурации, только при их запросе из приложения. Если не указать этот атрибут, то каркас Spring попытается инициализировать все компоненты Spring Beans во время начальной загрузки контекста типа ApplicationContext и потерпит неудачу при инициализации компонента singerThree.

Если все три компонента Spring Beans, определенных в файле конфигурации, содержат один и тот же атрибут init-method, как показано выше, то этот файл можно упростить, установив соответствующим образом атрибут default-init-method в дескрипторе разметки <beans>. Определяемые компоненты Spring Beans могут быть разнотипными, и единственным условием для них должно быть задание имени метода инициализации в атрибуте default-init-method. Так, приведенную выше конфигурацию можно переписать следующим образом:

```
<beans ...
  default-lazy-init="true" default-init-method="init">

  <bean id="singerOne"
     class="com.apress.prospring5.ch4.Singer"
     p:name="John Mayer" p:age="39"/>

  <bean id="singerTwo"
     class="com.apress.prospring5.ch4.Singer"
     p:age="72"/>

  <bean id="singerThree"
     class="com.apress.prospring5.ch4.Singer"
     p:name="John Butler"/>
</beans>
```

Выполнение исходного кода из данного примера даст следующий результат:

```
Initializing bean
    Name: John Mayer
    Age: 39
Initializing bean
Using default name
    Name: Eric Clapton
    Age: 72
Initializing bean
An error occured in bean configuration: Error creating bean
with name 'singerThree' defined in class path
```

```
resource spring/app-context-xml.xml: Invocation of init method failed;
nested exception is java.lang.IllegalArgumentException:
You must set the age property of any beans of type class
com.apress.prospring5.ch4.Singer[1]
```

Как следует из приведенного выше результата, компонент `singerOne` был сконфигурирован верно с помощью значений, заданных в файле конфигурации. При инициализации компонента `singerTwo` для свойства `name` было использовано значение по умолчанию, поскольку в его определении никакого значения не задано. И, наконец, экземпляр компонента `singerThree` не был получен, поскольку в методе `init()` было сгенерировано исключение в связи с отсутствием значения для свойства `age`.

Как видите, применение метода инициализации — идеальный способ обеспечить правильное конфигурирование компонентов Spring Beans. Применяя такой механизм, можно извлечь наиболее полную выгоду из инверсии управления, не теряя контроля, который дает определение зависимостей вручную. Единственный недостаток метода инициализации заключается в том, что он не может принимать аргументы. Можно определить любой возвращаемый тип, хотя он игнорируется Spring, и даже воспользоваться статическим методом, но этот метод не должен принимать аргументы.

Если применяется статический метод инициализации, то преимущества данного механизма сводятся на нет, поскольку нельзя получить доступ к состоянию любого компонента Spring Bean для его проверки. Если статическое состояние используется в компоненте Spring Bean для экономии оперативной памяти и для проверки этого состояния определен статический метод инициализации, то придется рассмотреть возможность перехода статического состояния в состояние экземпляра и применения нестатического метода инициализации. Если же воспользоваться средствами Spring для управления одиночными экземплярами, то конечный результат окажется таким же, но в итоге получится компонент Spring Bean, который намного проще тестировать, а также появится дополнительная возможность получить несколько экземпляров компонента Spring Bean с собственными состояниями, если в этом возникнет потребность. Безусловно, иногда может потребоваться статическое состояние, общее для нескольких экземпляров компонента Spring Bean. И для этой цели можно всегда воспользоваться статическим методом инициализации.

---

[1] Инициализация компонента Spring Bean
В конфигурации компонента Spring Bean обнаружена ошибка:
ошибка создания компонента Spring Bean под именем
'singerThree', определенного в ресурсе по пути к классам
spring/app-context-xml.xml: не удалось вызвать init();
вложенное исключение java.lang.IllegalArgumentException:
должно быть установлено свойство age любого компонента
Spring Bean типа com.apress.prospring5.ch4.Singer

Реализация интерфейса InitializingBean

Интерфейс InitializingBean, определяемый в Spring, позволяет задать в компоненте Spring Bean код, который будет выполнен, когда компонент получит уведомление, что каркас Spring завершил его конфигурирование. Как и в методе инициализации, это дает возможность проверить конфигурацию компонента Spring Bean и попутно предоставить любые значения по умолчанию. В интерфейсе InitializingBean определен единственный метод afterPropertiesSet(), который служит той же цели, что и метод init(), представленный в предыдущем разделе. Ниже приведен переделанный вариант предыдущего примера, где вместо метода инициализации используется интерфейс InitializingBean.

```
package com.apress.prospring5.ch4;

import org.springframework.beans.factory
          .BeanCreationException;
import org.springframework.beans.factory
          .InitializingBean;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support
          .GenericXmlApplicationContext;

public class SingerWithInterface
       implements InitializingBean {
  private static final String DEFAULT_NAME = "Eric Clapton";

  private String name;
  private int age = Integer.MIN_VALUE;

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

  public void setAge(int age) {
    this.age = age;
  }

  public void afterPropertiesSet() throws Exception {
    System.out.println("Initializing bean");
    if (name == null) {
      System.out.println("Using default name");
      name = DEFAULT_NAME;
    }

    if (age == Integer.MIN_VALUE) {
      throw new IllegalArgumentException("You must set "
              + "the age property of any beans of type "
              + SingerWithInterface.class);
```

```
    }
  }

  public String toString() {
    return "\tName: " + name + "\n\tAge: " + age;
  }

  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
           new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();

    getBean("singerOne", ctx);
    getBean("singerTwo", ctx);
    getBean("singerThree", ctx);

    ctx.close();
  }

  private static SingerWithInterface getBean(
          String beanName, ApplicationContext ctx) {
    try {
      SingerWithInterface bean =
            (SingerWithInterface) ctx.getBean(beanName);
      System.out.println(bean);
      return bean;
    } catch (BeanCreationException ex) {
      System.out.println("An error occured in bean "
               + "configuration: " + ex.getMessage());
      return null;
    }
  }
}
```

Как видите, изменений в данном примере не особенно много. Кроме очевидного изменения имени класса, единственное отличие от предыдущего примера заключается в том, что этот класс реализует интерфейс `InitializingBean`, а логика инициализации перемещена в метод `afterPropertiesSet()`. Ниже приведена конфигурация для данного примера из файла `app-context-xml.xml`.

```
<beans ... default-lazy-init="true">

  <bean id="singerOne"
     class="com.apress.prospring5.ch4.SingerWithInterface"
    p:name="John Mayer" p:age="39"/>

  <bean id="singerTwo"
     class="com.apress.prospring5.ch4.SingerWithInterface"
```

```
      p:age="72"/>

  <bean id="singerThree"
     class="com.apress.prospring5.ch4.SingerWithInterface"
    p:name="John Butler"/>
</beans>
```

И в конфигурациях для данного и предыдущего примеров совсем немного отличий. Самым заметным отличием является отсутствие атрибута `init-method`. В классе `SingerWithInterface` реализуется интерфейс `InitializingBean`, и поэтому каркасу Spring известно, какой именно метод следует инициировать в качестве обратного вызова при инициализации, а следовательно, отпадает потребность в любой дополнительной конфигурации. Ниже приведен результат, выводимый на консоль при выполнении кода из данного примера.

```
Initializing bean
    Name: John Mayer
     Age: 39
Initializing bean
Using default name
    Name: Eric Clapton
    Age: 72
Initializing bean
An error occured in bean configuration:
Error creating bean with name 'singerThree'
defined in class path resource spring/app-context-xml.xml:
Invocation of init method failed;
nested exception is java.lang.IllegalArgumentException:
You must set the age property of any beans of type
class com.apress.prospring5.ch4.SingerWithInterface
```

Применение аннотации `@PostConstruct` по спецификации JSR-250

Достичь той же самой цели можно и с помощью аннотации жизненного цикла @PostConstruct, определенной в спецификации JSR-250. Начиная с версии Spring 2.5 поддерживаются аннотации по спецификации JSR-250, предназначенные для указания метода, который каркас Spring должен вызвать, если соответствующая аннотация, связанная с жизненным циклом компонента Spring Bean, существует в классе. Ниже приведен исходный код прикладной программы, в которой применяется аннотация `@PostConstruct`.

```
package com.apress.prospring5.ch4;

import javax.annotation.PostConstruct;
import org.springframework.beans.factory
          .BeanCreationException;
```

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support
          .GenericXmlApplicationContext;

public class SingerWithJSR250 {
  private static final String DEFAULT_NAME = "Eric Clapton";

  private String name;
  private int age = Integer.MIN_VALUE;

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

  public void setAge(int age) {
    this.age = age;
  }

  @PostConstruct
  public void init() throws Exception {
    System.out.println("Initializing bean");

    if (name == null) {
      System.out.println("Using default name");
      name = DEFAULT_NAME;
    }

    if (age == Integer.MIN_VALUE) {
      throw new IllegalArgumentException("You must set "
              + "the age property of any beans of type "
              + SingerWithJSR250.class);
    }
  }

  public String toString() {
    return "\tName: " + name + "\n\tAge: " + age;
  }

  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-annotation.xml");
    ctx.refresh();

    getBean("singerOne", ctx);
    getBean("singerTwo", ctx);
    getBean("singerThree", ctx);

    ctx.close();
  }
```

```
  public static SingerWithJSR250 getBean(String beanName,
                                ApplicationContext ctx) {
    try {
      SingerWithJSR250 bean =
            (SingerWithJSR250) ctx.getBean(beanName);
      System.out.println(bean);
      return bean;
    } catch (BeanCreationException ex) {
      System.out.println("An error occured in bean "
               + "configuration: " + ex.getMessage());
      return null;
    }
  }
}
```

Исходный код данной программы отличается от предыдущего примера, для конфигурирования которого применялся атрибут `init-method`, лишь наличием аннотации `@PostConstruct` в объявлении метода `init()`. Следует, однако, иметь в виду, что этому методу можно присвоить любое имя. Что же касается конфигурирования, то поскольку в данном случае применяются аннотации, то в файл конфигурации придется ввести дескриптор разметки `<context:annotation-config>` из пространства имен `context`, как показано ниже.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context=
            "http://www.springframework.org/schema/context"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation=
          "http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans
                /spring-beans.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context
                /spring-context.xsd"
       default-lazy-init="true">

    <context:annotation-config/>

    <bean id="singerOne"
      class="com.apress.prospring5.ch4.SingerWithJSR250"
      p:name="John Mayer" p:age="39"/>

    <bean id="singerTwo"
      class="com.apress.prospring5.ch4.SingerWithJSR250"
      p:age="72"/>
    <bean id="singerThree"
```

```
        class="com.apress.prospring5.ch4.SingerWithJSR250"
        p:name="John Butler"/>
</beans>
```

Выполнение данной программы приводит к такому же результату, как и прежде, когда применялись другие механизмы инициализации компонентов Spring Beans.

```
Initializing bean
    Name: John Mayer
    Age: 39
Initializing bean
Using default name
    Name: Eric Clapton
    Age: 72
Initializing bean
An error occurred in bean configuration:
Error creating bean with name 'singerThree':
Invocation of init method failed; nested exception is
java.lang.IllegalArgumentException:
You must set the age property of any beans
of type class com.apress.prospring5.ch4.SingerWithJSR250
```

Всем трем подходам к инициализации компонентов Spring Beans присущи свои достоинства и недостатки. Метод инициализации дает преимущество развязки приложения от каркаса Spring, но при этом следует сконфигурировать метод инициализации для каждого компонента Spring Bean, который в нем нуждается. А интерфейс `InitializingBean` дает преимущество, позволяющее указывать обратный вызов при инициализации один раз для всех экземпляров класса компонента Spring Bean, но для этого придется привязывать приложение к каркасу Spring. Аннотации требуется применять непосредственно в методах, а также проверять, поддерживает ли контейнер инверсии управления спецификацию JSR-250. В конечном счете решение выбрать конкретный механизм инициализации компонентов Spring Beans принимается исходя из требований к разрабатываемому приложению. Так, если особое значение имеет переносимость, применяйте метод инициализации или снабдите его аннотацией. В противном случае используйте интерфейс `InitializingBean`, чтобы сократить потребности в конфигурировании приложения и уменьшить вероятность возникновения ошибок в приложении из-за неверной конфигурации.

На заметку Конфигурирование инициализации компонентов Spring Beans с помощью атрибута **`init-method`** или аннотации **`@PostConstruct`** выгодно отличается еще и тем, что метод инициализации может быть объявлен с другим правом доступа. Методы инициализации должны вызываться из контейнера инверсии управления в Spring лишь один раз во время создания компонента Spring Bean, поскольку любые последующие вызовы приведут к неожиданным результатам или даже к отказам. Чтобы исключить лишние внешние вызовы метода инициализации, его необходимо объявить закрытым (**`private`**). Контейнер инверсии управления сможет вызвать метод инициализации через рефлексию, но любые последующие его вызовы в прикладном коде будут запрещены.

Объявление метода инициализации с помощью аннотации @Bean

Еще один способ объявить метод инициализации для компонента Spring Bean состоит в том, чтобы указать в аннотации `@Bean` атрибут `initMethod` вместе с именем данного метода. С помощью этой аннотации объявляются компоненты Spring Beans в конфигурационных классах Java. И хотя конфигурация Java рассматривается далее в этой главе, здесь обсуждается та ее часть, которая относится непосредственно к инициализации. Для целей рассматриваемого здесь примера применяется первоначальный класс `Singer`, поскольку в данном случае употребляется внешняя конфигурация, как и при инициализации с помощью атрибута `init-method`. Остается лишь написать конфигурационный класс с новым методом `main()` и протестировать его, а в определении каждого компонента Spring Bean заменить атрибут `default-lazy-init="true"` аннотацией `@Lazy`, как показано ниже.

```
package com.apress.prospring5.ch4.config;

import com.apress.prospring5.ch4.Singer;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Lazy;
import org.springframework.context.support
          .GenericApplicationContext;
import static com.apress.prospring5.ch4.Singer.getBean;

public class SingerConfigDemo {

  @Configuration
  static class SingerConfig {

    @Lazy
    @Bean(initMethod = "init")
    Singer singerOne() {
      Singer singerOne = new Singer();
      singerOne.setName("John Mayer");
      singerOne.setAge(39);
      return singerOne;
    }

    @Lazy
    @Bean(initMethod = "init")
    Singer singerTwo() {
      Singer singerTwo = new Singer();
      singerTwo.setAge(72);
```

```
      return singerTwo;
    }

    @Lazy
    @Bean(initMethod = "init")
    Singer singerThree() {
       Singer singerThree = new Singer();
       singerThree.setName("John Butler");
       return singerThree;
    }
  }

  public static void main(String args) {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               SingerConfig.class);

    getBean("singerOne", ctx);
    getBean("singerTwo", ctx);
    getBean("singerThree", ctx);

    ctx.close();
  }
}
```

Выполнение приведенного выше кода дает такой же результат, как и прежде.

```
Initializing bean
    Name: John Mayer
    Age: 39
Initializing bean
Using default name
    Name: Eric Clapton
    Age: 72
Initializing bean
An error occurred in bean configuration:
Error creating bean with name 'singerThree' defined in
com.apress.prospring5.ch4.config
   .SingerConfigDemo$SingerConfig:
Invocation of init method failed;
nested exception is
java.lang.IllegalArgumentException:
You must set the age property of any beans
of type class com.apress.prospring5.ch4.Singer
```

Описание порядка разрешения зависимостей

Все упомянутые выше механизмы инициализации могут применяться к одному и тому же экземпляру компонента Spring Bean. В данном случае Spring сначала вызывает метод, снабженный аннотацией `@PostConstruct`, затем метод `afterProperties`

Set(), а далее — метод инициализации, указанный в файле конфигурации. И на такой порядок вызовов имеются веские основания. Если обратиться снова к рис. 4.1, то можно выявить следующие стадии в процессе создания компонентов Spring Beans.

1. Сначала вызывает конструктор для создания компонента Spring Bean.
2. Затем вызываются методы установки для внедрения зависимостей.
3. Как только появятся компоненты Spring Beans с внедренными зависимостями, делается запрос компонентов типа BeanPostProcessor из инфраструктуры предварительной инициализации, чтобы выяснить, требуется ли им вызвать что-нибудь из данного компонента Spring Bean. Это компоненты специальной инфраструктуры Spring, выполняющие модификацию компонентов Spring Beans после их создания. Аннотация @PostConstruct регистрируется средствами класса CommonAnnotationBeanPostProcessor, и поэтому из данного компонента Spring Bean вызывается метод, снабженный аннотацией @PostConstruct. Этот метод выполняется сразу же после построения компонента Spring Bean, но прежде ввода данного класса в действие[2] и до фактической инициализации компонента Spring Bean (т.е. до вызова метода afterPropertiesSet() и атрибута init-method).
4. Метод afterPropertiesSet(), определяемый в интерфейсе InitializingBean, выполняется сразу же после внедрения зависимостей. Этот метод вызывается из интерфейса BeanFactory после установки всех предоставленных свойств компонента Spring Bean и удовлетворения требований интерфейсов BeanFactoryAware и ApplicationContextAware.
5. Атрибут init-method вызывается в последнюю очередь, поскольку в нем фактически указан конкретный метод инициализации компонента Spring Bean.

Ясное представление о порядке инициализации разных типов компонентов Spring Beans может принести пользу в том случае, если имеется компонент Spring Bean, выполняющий инициализацию в конкретном методе, но при этом требуется внедрить дополнительный код инициализации, когда применяется каркас Spring.

Уничтожение компонентов Spring Beans

Когда применяется реализация интерфейса ApplicationContext, заключающая в оболочку интерфейс DefaultListableBeanFactory (например, через метод getDefaultListableBeanFactory() из класса GenericXmlApplicationContext), то, вызвав метод ConfigurableBeanFactory.destroySingletons(), можно сообщить интерфейсу BeanFactory, что требуется уничтожить все одиночные экземпляры. Обычно это делается при завершении приложения, чтобы очистить любые ресурсы, которые компоненты Spring Beans могут удерживать открытыми, и тем

[2] Подробнее об этом см. официальную документацию на платформу JEE по адресу http://docs.oracle.com/javaee/7/api/javax/annotation/PostConstruct.html.

самым корректно закрыть приложение. Такой обратный вызов оказывается удобным местом для сброса любых данных из оперативной памяти в постоянное хранилище, а также для завершения долго выполняющихся процессов, которые могли быть запущены ранее.

Разрешить компонентам Spring Beans получать уведомления о вызове метода `destroySingletons()` можно тремя способами, которые похожи на механизмы для получения обратного вызова при инициализации. Обратный вызов при уничтожении часто применяется в сочетании с обратным вызовом при инициализации. Зачастую в обратном вызове при инициализации создается и конфигурируется некоторый ресурс, а в обратном вызове при уничтожении этот ресурс освобождается.

Выполнение метода при уничтожении компонента Spring Bean

Чтобы назначить метод для вызова в тот момент, когда требуется уничтожить компонент Spring Bean, достаточно указать имя этого метода в атрибуте `destroy-method` дескриптора `<bean>`, где определяется данный компонент. Каркас Spring вызывает этот метод непосредственно перед уничтожением одиночного экземпляра компонента (он не будет вызывать этот метод для компонентов Spring Beans с областью видимости на уровне прототипа). Ниже приведен пример применения обратного вызова метода при уничтожении компонента Spring Bean.

```
package com.apress.prospring5.ch4;

import java.io.File;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.context.support
          .GenericXmlApplicationContext;

public class DestructiveBean
       implements InitializingBean {
  private File file;
  private String filePath;

  public void afterPropertiesSet() throws Exception {
    System.out.println("Initializing Bean");

    if (filePath == null) {
      throw new IllegalArgumentException("You must "
              + "specify the filePath property of"
              + DestructiveBean.class);
    }

    this.file = new File(filePath);
    this.file.createNewFile();
```

```
    System.out.println("File exists: " + file.exists());
  }

  public void destroy() {
    System.out.println("Destroying Bean");

    if(!file.delete()) {
      System.err.println("ERROR: failed to delete file.");
    }

    System.out.println("File exists: " + file.exists());
  }

  public void setFilePath(String filePath) {
    this.filePath = filePath;
  }

  public static void main(String... args) throws Exception {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();

    DestructiveBean bean = (DestructiveBean)
                    ctx.getBean("destructiveBean");

    System.out.println("Calling destroy()");
    ctx.destroy();
    System.out.println("Called destroy()");
  }
}
```

В приведенном выше коде определяется метод `destroy()`, в котором удаляется созданный ранее файл. В методе `main()` сначала извлекается компонент Spring Bean типа `DestructiveBean` из контекста типа `GenericXmlApplicationContext`, а затем вызывается его метод `destroy()`, где, в свою очередь, вызывается метод `ConfigurableBeanFactory.destroySingletons()`, заключаемый в оболочку интерфейса `ApplicationContext`. Этим каркасу Spring предписывается уничтожить все управляемые им одиночные экземпляры. Обратные вызовы при инициализации и уничтожении выводят на консоль сообщение, извещающее, что они были сделаны. Ниже приведена конфигурация компонента `destructiveBean` из файла `app-context-xml.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
```

```
       xsi:schemaLocation=
         "http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans
               /spring-beans.xsd">

  <bean id="destructiveBean"
    class="com.apress.prospring5.ch4.DestructiveBean"
    destroy-method="destroy"
    p:filePath= "#{systemProperties'java.io.tmpdir'}
          #{systemProperties'file.separator'}test.txt"/>
</beans>
```

Обратите внимание на то, что метод `destroy()` указан в атрибуте `destroy-method` как метод обратного вызова при уничтожении. А значение для атрибута `filePath` составляется с помощью выражения SpEL, в котором перед именем файла `test.txt` указывается результат сцепления системных свойств `java.io.tmpdir` и `file.separator`, чтобы обеспечить совместимость платформ. Выполнение исходного кода из данного примера дает следующий результат:

```
Initializing Bean
File exists: true
Calling destroy()
Destroying Bean[3]
File exists: false
Called destroy()
```

Как видите, сначала делается обратный вызов при инициализации в каркасе Spring, а затем в экземпляре типа `DestructiveBean` получается и сохраняется экземпляр типа `File`. Далее в течение вызова метода `destroy()` каркас Spring перебирает ряд управляемых им одиночных экземпляров (в данном случае имеется лишь один такой экземпляр) и делает любые обратные вызовы при уничтожении, которые были определены. Именно здесь в экземпляре типа `DestructiveBean` удаляется созданный ранее файл и на консоль выводится сообщение о том, что он больше не существует.

Реализация интерфейса `DisposableBean`

Как и для обратных вызовов при инициализации, в Spring предоставляется интерфейс `DisposableBean`, который может быть реализован в компонентах Spring Beans как механизм для получения обратных вызовов при уничтожении. В интерфейсе `DisposableBean` определен единственный метод `destroy()`, вызываемый непосредственно перед уничтожением компонента Spring Bean. Этот интерфейс применя-

[3] Инициализация компонента Spring Bean
Файл существует: верно
Вызов метода destroy()
Уничтожение компонента Spring Bean

ется таким же образом, как и интерфейс `InitializingBean`. Ниже приведен видоизмененный вариант класса `DestructiveBean`, в котором реализуется интерфейс `DisposableBean`.

```
package com.apress.prospring5.ch4;

import java.io.File;
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.context.support
          .GenericXmlApplicationContext;

public class DestructiveBeanWithInterface
       implements InitializingBean, DisposableBean {
  private File file;
  private String filePath;

  @Override
  public void afterPropertiesSet() throws Exception {
    System.out.println("Initializing Bean");

    if (filePath == null) {
      throw new IllegalArgumentException("You must "
              + "specify the filePath property of "
              + DestructiveBeanWithInterface.class);
    }

    this.file = new File(filePath);
    this.file.createNewFile();

    System.out.println("File exists: " + file.exists());
  }

  @Override
  public void destroy() {
    System.out.println("Destroying Bean");

    if(!file.delete()) {
      System.err.println("ERROR: failed to delete file.");
    }

    System.out.println("File exists: " + file.exists());
  }

  public void setFilePath(String filePath) {
    this.filePath = filePath;
  }

  public static void main(String... args)
```

```
                    throws Exception {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();

    DestructiveBeanWithInterface bean =
               (DestructiveBeanWithInterface)
               ctx.getBean("destructiveBean");

    System.out.println("Calling destroy()");
    ctx.destroy();
    System.out.println("Called destroy()");
  }
}
```

Исходный код, в котором применяется метод обратного вызова, мало чем отличается от кода, в котором применяется интерфейс обратного вызова. В данном примере были даже употреблены одни и те же имена методов. А ниже приведена конфигурация для данного примера из файла `app-context-xml.xml`.

```
<beans ...>

  <bean id="destructiveBean"
     class="com.apress.prospring5.ch4
               .DestructiveBeanWithInterface"
     p:filePath="#{systemProperties'java.io.tmpdir'}
           #{systemProperties'file.separator'}test.txt"/>
</beans>
```

Кроме иного имени класса, эта конфигурация отличается лишь отсутствием атрибута `destroy-method`. Выполнение исходного кода из данного примера дает следующий результат:

```
Initializing Bean
File exists: true
Calling destroy()
Destroying Bean
File exists: false
Called destroy()
```

Применение аннотации `@PreDestroy` по спецификации JSR-250

Третий способ уничтожения компонентов Spring Beans предусматривает применение аннотации жизненного цикла `@PreDestroy`, которая определена в спецификации JSR-250 и полностью противоположна по своему действию `@PostConstruct`. Ниже приведена версия класса `DestructiveBean`, в которой аннотации `@Post`

Construct и @PreDestroy применяются для выполнения действий по инициализации и уничтожению компонента Spring Bean.

```
package com.apress.prospring5.ch4;

import java.io.File;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import org.springframework.context.support
          .GenericXmlApplicationContext;

public class DestructiveBeanWithJSR250 {
  private File file;
  private String filePath;

  @PostConstruct
  public void afterPropertiesSet() throws Exception {
    System.out.println("Initializing Bean");

    if (filePath == null) {
      throw new IllegalArgumentException("You must "
              + "specify the filePath property of "
              + DestructiveBeanWithJSR250.class);
    }

    this.file = new File(filePath);
    this.file.createNewFile();

    System.out.println("File exists: " + file.exists());
  }

  @PreDestroy
  public void destroy() {
    System.out.println("Destroying Bean");

    if(!file.delete()) {
      System.err.println("ERROR: failed to delete file.");
    }

    System.out.println("File exists: " + file.exists());
  }

  public void setFilePath(String filePath) {
    this.filePath = filePath;
  }

  public static void main(String... args)
                    throws Exception {
    GenericXmlApplicationContext ctx =
```

```
                new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-annotation.xml");
    ctx.refresh();
    DestructiveBeanWithJSR250 bean =
                (DestructiveBeanWithJSR250)
                ctx.getBean("destructiveBean");

    System.out.println("Calling destroy()");
    ctx.destroy();
    System.out.println("Called destroy()");
  }
}
```

Ниже приведена конфигурация для данного примера из файла `app-context-annotation.xml`, где применяется дескриптор `<context:annotation-config>`.

```
<beans ...>

  <context:annotation-config/>

  <bean id="destructiveBean"
    class="com.apress.prospring5.ch4
              .DestructiveBeanWithJSR250"
    p:filePath="#{systemProperties'java.io.tmpdir'}
          #{systemProperties'file.separator'}test.txt"/>
</beans>
```

Объявление метода уничтожения с помощью аннотации `@Bean`

Еще один способ объявить метод уничтожения для компонента Spring Bean состоит в том, чтобы указать в аннотации `@Bean` атрибут `destroyMethod` вместе с именем данного метода. С помощью этой аннотации объявляются компоненты Spring Beans в конфигурационных классах Java. И хотя конфигурирование на языке Java рассматривается далее в этой главе, здесь обсуждается та его часть, которая относится непосредственно к уничтожению. Для целей рассматриваемого здесь примера применяется первоначальный класс `DestructiveBeanWithJSR250`, поскольку в данном случае употребляется внешняя конфигурация, как и при инициализации с помощью атрибута `destroy-method`. Остается лишь написать конфигурационный класс с новым методом `main()` и протестировать его, а в определении каждого компонента Spring Bean заменить атрибут `default-lazy-init="true"` аннотацией `@Lazy`, как показано ниже.

```
package com.apress.prospring5.ch4.config;

import com.apress.prospring5.ch4
          .DestructiveBeanWithJSR250;
```

```
import org.springframework.context.annotation
         .AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Lazy;
import org.springframework.context.support
         .GenericApplicationContext;

/**
* Этот класс создан Юлианой Козминой 27.02.17.
*/
public class DestructiveBeanConfigDemo {

  @Configuration
  static class DestructiveBeanConfig {

    @Lazy
    @Bean(initMethod = "afterPropertiesSet",
                       destroyMethod = "destroy")
    DestructiveBeanWithJSR250 destructiveBean() {
      DestructiveBeanWithJSR250 destructiveBean =
                new DestructiveBeanWithJSR250();
      destructiveBean.setFilePath(
                System.getProperty("java.io.tmpdir")
              + System.getProperty("file.separator")
              + "test.txt");
      return destructiveBean;
    }
  }

  public static void main(String... args) {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               DestructiveBeanConfig.class);

    ctx.getBean(DestructiveBeanWithJSR250.class);
    System.out.println("Calling destroy()");
    ctx.destroy();
    System.out.println("Called destroy()");
  }
}
```

Аннотацию `@PostConstruct` можно применять и при конфигурировании компонентов Spring Beans. Выполнение приведенного выше кода даст такой же результат, как и прежде:

```
Initializing Bean
File exists: true
Calling destroy()
```

```
Destroying Bean
File exists: false
Called destroy()
```

Обратный вызов при уничтожении — это идеальный механизм для обеспечения аккуратного завершения приложений, когда ресурсы гарантированно не останутся в открытом или несогласованном состоянии. Но по-прежнему приходится решать, какой механизм выбрать: метод обратного вызова при уничтожении, интерфейс `DisposableBean`, аннотацию `@PreDestroy` или атрибут `destroy-method` разметки в формате XML. Как и прежде, решение следует принимать исходя из требований к разрабатываемому приложению. Так, если особое значение имеет переносимость, применяйте метод обратного вызова. В противном случае используйте интерфейс `DisposableBean` или аннотацию по спецификации JSR-250, чтобы сократить потребности в конфигурировании приложения.

Описание порядка разрешения зависимостей

Как при создании, так и при уничтожении компонентов Spring Beans все упомянутые выше механизмы можно применять к одному и тому же экземпляру компонента. С этой целью в Spring сначала вызывается метод, снабженный аннотацией `@PreDestroy`, затем метод `DisposableBean.destroy()` и метод уничтожения, сконфигурированный в определении данного компонента в формате XML.

Применение перехватчика завершения

Единственный недостаток обратных вызовов при уничтожении компонентов в Spring заключается в том, что они не запускаются автоматически, т.е. перед закрытием приложения следует вызвать метод `AbstractApplicationContext.destroy()`. Если приложение выполняется как сервлет, указанный метод `destroy()` можно вызвать в методе `destroy()` данного сервлета. Но в автономном приложении дело обстоит не так просто, особенно при наличии многих точек выхода из приложения. Правда, из этого положения есть выход. В языке Java допускается создание *перехватчика завершения* — потока, который исполняется непосредственно перед завершением приложения. Это отличный способ вызвать метод `destroy()` из контекста, расширяемого всеми конкретными реализациями интерфейса `ApplicationContext`. Задействовать этот механизм проще всего с помощью метода `registerShutdownHook()` из класса, реализующего интерфейс `AbstractApplicationContext`. Этот метод автоматически предписывает каркасу Spring зарегистрировать перехватчик завершения базовой исполняющей среды виртуальной машины JVM. Определение и конфигурация компонента Spring Bean остаются прежними, за исключением следующих изменений в методе `main()`: добавления вызова `ctx.registerShutdownHook()` и удаления вызовов `ctx.destroy()` или `ctx.close()`, как показано ниже.

```
...
public class DestructiveBeanWithHook {

  public static void main(String... args) {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
                        DestructiveBeanConfig.class);
    ctx.getBean(DestructiveBeanWithJSR250.class);
    ctx.registerShutdownHook();
  }
}
```

Выполнение приведенного выше кода даст такой же результат, как и прежде:

```
Initializing Bean
File exists: true
Destroying Bean
File exists: false
```

Как видите, метод `destroy()` был вызван, несмотря на то, что мы не написали ни единой строки кода для его явного вызова при завершении приложения.

Информирование компонентов Spring Beans об их контексте

Одно из главнейших преимуществ внедрения зависимостей над поиском зависимостей как механизма инверсии управления заключается в том, что компоненты Spring Beans совсем не обязательство должны быть информированы о подробностях реализации контейнера, который ими управляет. Для компонента Spring Bean, в котором зависимости внедряются через конструктор или метод установки, контейнер Spring ничем не отличается от контейнера, предоставляемого каркасом Google Guice, или, скажем, контейнера PicoContainer. Но при определенных обстоятельствах может потребоваться такой компонент Spring Bean, в котором внедрение зависимостей применяется для получения его зависимостей. Поэтому он может взаимодействовать с контейнером по какой-нибудь другой причине. Характерным тому примером может служить компонент Spring Bean, автоматически конфигурирующий перехватчик завершения, для чего ему требуется доступ к контексту типа `ApplicationContext`. В других случаях компоненту Spring Bean, возможно, понадобится узнать свое имя (т.е. имя, присвоенное данному компоненту в текущем контексте типа `ApplicationContext`), чтобы на основе этой информации предпринять то или иное действие.

Вообще говоря, данное средство предназначено для внутреннего употребления в Spring. Придавать имени компонента Spring Bean какой-то предметный смысл — не самая удачная идея, поскольку это может привести к осложнениям при конфигурировании, когда именами компонентов Spring Beans придется искусственно манипулировать, чтобы поддерживать их предметный смысл. Тем не менее мы считаем, что воз-

можностью компонента Spring Bean выяснить свое имя во время выполнения можно выгодно воспользоваться для целей протоколирования. Допустим, имеется целый ряд компонентов Spring Beans одного и того же типа, выполняющихся с разными конфигурациями. Имя компонента может быть включено в протокольные сообщения, что поможет отличить компоненты, сгенерировавшие исключения и ошибки, от компонентов, которые функционировали нормально, когда что-то пошло не так, как предполагалось.

Применение интерфейса BeanNameAware

В интерфейсе `BeanNameAware`, который компонент Spring Bean может реализовать, чтобы получить свое имя, определен единственный метод `setBeanName(String)`. Каркас Spring вызывает метод `setBeanName()` по окончании конфигурирования компонента Spring Bean, но перед любыми обратными вызовами жизненного цикла (при инициализации или уничтожении), как показано на рис. 4.1. Как правило, реализация метода `setBeanName()` сводится к одной строке, в которой значение, переданное контейнером, сохраняется в каком-нибудь поле для дальнейшего применения. Ниже приведен пример простого компонента Spring Bean, который получает свое имя с помощью интерфейса `BeanNameAware` и затем выводит его на консоль.

```
package com.apress.prospring5.ch4;

import org.springframework.beans.factory.BeanNameAware;

public class NamedSinger implements BeanNameAware {
private String name;

/** @Implements {@link BeanNameAware#setBeanName(String)} */
public void setBeanName(String beanName) {
  this.name = beanName;
}

  public void sing() {
    System.out.println("Singer " + name + " - sing()");
  }
}
```

Приведенная выше реализация довольно проста. Напомним, что метод `BeanNameAware.setBeanName()` вызывается перед тем, как первый экземпляр компонента Spring Bean будет возвращен приложению через вызов метода `ApplicationContext.getBean()`, поэтому в теле метода `sing()` нет необходимости проверять, доступно ли имя данного компонента. Ниже приведена простая конфигурация для данного примера из файла `app-context-xml.xml`.

```
<beans ...>
  <bean id="johnMayer"
```

```
    class="com.apress.prospring5.ch4.NamedSinger"/>
</beans>
```

Как видите, никакой специальной конфигурации для применения интерфейса `BeanNameAware` не требуется. В следующем фрагменте кода приведен пример простого приложения, которое извлекает экземпляр типа `Singer` из контекста типа `ApplicationContext` и затем вызывает метод `sing()`:

```
package com.apress.prospring5.ch4;

import org.springframework.context.support
          .GenericXmlApplicationContext;

public class NamedSingerDemo {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();

    NamedSinger bean = (NamedSinger)
                       ctx.getBean("johnMayer");
    bean.sing();

    ctx.close();
  }
}
```

В данном примере формируется приведенный ниже вывод на консоль. Обратите внимание на наличие имени компонента Spring Bean в протокольном сообщении о вызове метода `sing()`.

```
Singer johnMayer - sing()
```

Пользоваться интерфейсом `BeanNameAware` совсем не трудно, и он приносит немалую пользу в повышении качества протокольных сообщений. Не поддавайтесь, однако, искушению придавать именам компонентов Spring Beans предметный смысл просто потому, что к ним возможен доступ. Ведь поступая так, вы привязываете свои классы к Spring, а взамен получаете функциональные возможности, от которых мало пользы. Если компоненты Spring Beans испытывают внутреннюю потребность в своем имени, реализуйте в них нечто вроде интерфейса `Nameable` с методом `setName()` специально для вашего приложения, а затем присвойте каждому компоненту Spring Bean имя через внедрение зависимостей. Подобным способом можно сохранить краткими имена, употребляемые при конфигурировании, и не придется манипулировать конфигурацией вручную, чтобы придать именам компонентов Spring Beans предметный смысл.

Применение интерфейса ApplicationContextAware

Интерфейс ApplicationContextAware был представлен в главе 3 с целью показать, каким образом можно обращаться с одними компонентами Spring Beans, для нормального функционирования которых требуются другие компоненты Spring Beans, не внедренные через конструкторы или методы установки в конфигурации (см. пример применения атрибута depends-on).

Через интерфейс ApplicationContextAware компоненты Spring Beans могут получить ссылку на экземпляр контекста типа ApplicationContext, в котором они сконфигурированы. Основной причиной для создания этого интерфейса была необходимость предоставить компоненту Spring Bean доступ к контексту типа ApplicationContext в приложении (например, для получения других компонентов Spring Beans программным способом), используя метод getBean(). Но подобной нормы практики следует все же избегать, а вместо нее внедрять зависимости, чтобы предоставить свои компоненты Spring Beans, а также взаимодействующие с ними объекты. Если вы пользуетесь методом getBean() для поиска и получения зависимостей, когда можно было бы внедрить зависимости, то тем самым излишне усложняете свои компоненты Spring Beans и привязываете их к каркасу Spring Framework без веских на то оснований.

Интерфейс ApplicationContext служит, конечно, не только для поиска компонентов Spring Beans, но и для решения многих других задач. Как упоминалось ранее, к числу таких задач относится уничтожение всех одиночных экземпляров с предварительным их уведомлением по очереди. В предыдущем разделе было показано, как создать перехватчик завершения, чтобы предписать интерфейсу ApplicationContext уничтожить все одиночные экземпляры перед закрытием приложения. Применяя интерфейс ApplicationContextAware, несложно построить компонент Spring Bean, который может быть сконфигурирован в контексте типа ApplicationContext для автоматического создания и настройки перехватчика завершения. Исходный код такого компонента Spring Bean приведен ниже.

```
package com.apress.prospring5.ch4;

import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.context.support
          .GenericApplicationContext;

public class ShutdownHookBean
       implements ApplicationContextAware {
  private ApplicationContext ctx;

  /** @Implements {@link ApplicationContextAware#s
```

```
    etApplicationContext(ApplicationContext)} }*/

  public void setApplicationContext(ApplicationContext ctx)
              throws BeansException {
    if (ctx instanceof GenericApplicationContext) {
       ((GenericApplicationContext) ctx)
        .registerShutdownHook();
    }
  }
}
```

Большая часть приведенного выше кода должна быть вам уже знакома. В интерфейсе `ApplicationContextAware` определен единственный метод `setApplicationContext(ApplicationContext)`, вызываемый в Spring для передачи данному компоненту Spring Bean ссылки на его контекст типа `ApplicationContext`. В приведенном выше коде класс `ShutdownHookBean` проверяет, относится ли интерфейс `ApplicationContext` к контексту типа `GenericApplicationContext`, что означает поддержку в нем метода `registerShutdownHook()`, и если это соответствие типов подтверждается, то перехватчик завершения будет зарегистрирован в контексте типа `ApplicationContext`. В следующем фрагменте кода конфигурации из файла `app-context-annotation.xml` показано, как сконфигурировать данный компонент Spring Bean для взаимодействия с компонентом типа `DestructiveBeanWithInterface`:

```
<beans ...">

  <context:annotation-config/>

  <bean id="destructiveBean"
     class="com.apress.prospring5.ch4
               .DestructiveBeanWithInterface"
     p:filePath="#{systemProperties'java.io.tmpdir'}
           #{systemProperties'file.separator'}test.txt"/>

  <bean id="shutdownHook"
     class="com.apress.prospring5.ch4.ShutdownHookBean"/>
</beans>
```

Обратите внимание на то, что никакой специальной конфигурации в данном случае не требуется. В следующем фрагменте кода приведен простой пример приложения, в котором компонент Spring Bean типа `ShutdownHookBean` используется для управления процессом уничтожения одиночных экземпляров компонентов Spring Beans.

```
package com.apress.prospring5.ch4;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
```

```
import java.io.File;
import org.springframework.context.support
          .GenericXmlApplicationContext;

public class DestructiveBeanWithInterface {
  private File file;
  private String filePath;

  @PostConstruct
  public void afterPropertiesSet() throws Exception {
    System.out.println("Initializing Bean");

    if (filePath == null) {
      throw new IllegalArgumentException("You must "
              + "specify the filePath property of "
              + DestructiveBeanWithInterface.class);
    }

    this.file = new File(filePath);
    this.file.createNewFile();

    System.out.println("File exists: " + file.exists());
  }

  @PreDestroy
  public void destroy() {
    System.out.println("Destroying Bean");

    if(!file.delete()) {
      System.err.println("ERROR: failed to delete file.");
    }

    System.out.println("File exists: " + file.exists());
  }

  public void setFilePath(String filePath) {
    this.filePath = filePath;
  }

  public static void main(String... args)
                throws Exception {
    GenericXmlApplicationContext ctx =
           new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-annotation.xml");
    ctx.registerShutdownHook();
    ctx.refresh();
    ctx.getBean("destructiveBean",
                DestructiveBeanWithInterface.class);
  }
}
```

И этот исходный код должен быть вам уже знаком. Когда каркас Spring выполняет начальную загрузку интерфейса `ApplicationContext`, а компонент `destructive Bean` определен в конфигурации, ссылка на контекст типа `ApplicationContext` передается компоненту `shutdownHook` для регистрации перехватчика завершения. Выполнение исходного кода из данного примера дает, как и предполагалось, следующий результат:

```
Initializing Bean
File exists: true
Destroying Bean
File exists: false
```

Как видите, даже в отсутствие обращений к методу `destroy()` в главном приложении компонент Spring Bean типа `ShutdownHookBean` зарегистрирован как перехватчик завершения. Метод `destroy()` вызывается в нем непосредственно перед закрытием приложения.

Применение фабрик компонентов Spring Beans

Пользуясь каркасом Spring, вы непременно столкнетесь с трудностями реализации и внедрения зависимостей, которые нельзя просто создать с помощью операции `new`. Для преодоления подобных трудностей в Spring предоставляется интерфейс `FactoryBean`, действующий в качестве адаптера для тех объектов, которые нельзя создавать и манипулировать ими с помощью стандартной семантики Spring. Как правило, фабрики применяются для создания тех компонентов Spring Beans, которые нельзя создать с помощью операции `new`. К их числу относятся компоненты Spring Beans, доступные через статические фабричные методы, хотя это возможно не всегда. Попросту говоря, интерфейс `FactoryBean` и реализующий его компонент Spring Bean выполняет функции фабрики для других компонентов Spring Beans. Такие фабрики конфигурируются в контексте типа `ApplicationContext` подобно обычным компонентам Spring Beans, но когда интерфейс `FactoryBean` применяется в Spring для удовлетворения запроса на внедрение или поиск зависимости, вместо возврата экземпляра компонента, реализующего интерфейс `FactoryBean`, вызывается метод `FactoryBean.getObject()` и возвращается результат этого вызова.

Фабрики компонентов Spring Beans эффективно применяются в Spring для решения важных задач, наиболее примечательными из которых является создание объектов-заместителей для транзакций, как поясняется в главе 9, а также автоматическое извлечение ресурсов из контекста JNDI. Но фабрики компонентов Spring Beans полезны не только для реализации внутренних функций Spring. Они очень удобны и для построения собственных приложений, поскольку позволяют манипулировать с помощью инверсии управления намного бо́льшим количеством ресурсов, чем доступно в противном случае.

Класс MessageDigestFactoryBean как пример фабрики компонентов Spring Beans

Зачастую в разрабатываемых проектах требуется определенная разновидность криптографической обработки. Как правило, это связано с формированием свертки сообщения или хешированием пользовательского пароля для его сохранения в базе данных. В языке Java имеется класс `MessageDigest`, предоставляющий функциональные средства для формирования свертки сообщения из произвольных данных. Сам класс `MessageDigest` является абстрактным, а его конкретные реализации получаются через вызов метода `MessageDigest.getInstance()`, которому передается наименование применяемого алгоритма свертки. Так, если для формирования свертки сообщения требуется применить алгоритм MD5, то получить экземпляр класса `MessageDigest` можно следующим образом:

```
MessageDigest md5 = MessageDigest.getInstance("MD5");
```

Если каркас Spring должен управлять процессом создания объекта типа `MessageDigest`, то самое лучшее, что можно сделать без интерфейса `FactoryBean`, — предусмотреть в компоненте Spring Bean определенное свойство (например, `algorithmName`) и затем сделать обратный вызов при инициализации, чтобы обратиться к методу `MessageDigest.getInstance()`. Эту логику можно инкапсулировать в самом компоненте Spring Bean с помощью интерфейса `FactoryBean`. И тогда в любых компонентах Spring Beans, где требуется экземпляр класса `MessageDigest`, достаточно объявить свойство (например, `messageDigest`) и получить нужный экземпляр с помощью интерфейса `FactoryBean`. Ниже приведен пример реализации интерфейса `FactoryBean` для выполнения этих функций.

```
package com.apress.prospring5.ch4;

import java.security.MessageDigest;
import org.springframework.beans.factory.FactoryBean;
import org.springframework.beans.factory.InitializingBean;

public class MessageDigestFactoryBean implements
       FactoryBean<MessageDigest>, InitializingBean {
  private String algorithmName = "MD5";

  private MessageDigest messageDigest = null;

  public MessageDigest getObject() throws Exception {
    return messageDigest;
  }

  public Class<MessageDigest> getObjectType() {
    return MessageDigest.class;
  }
```

```
  public boolean isSingleton() {
    return true;
  }

  public void afterPropertiesSet() throws Exception {
    messageDigest = MessageDigest
                    .getInstance(algorithmName);
  }

  public void setAlgorithmName(String algorithmName) {
    this.algorithmName = algorithmName;
  }
}
```

Метод `getObject()` вызывается в Spring с целью извлечь объект, созданный в рассматриваемом здесь компоненте Spring Bean, реализующем интерфейс `FactoryBean`. Это конкретный объект, который передается другим компонентам Spring Beans, где данный компонент применяется в качестве взаимодействующего с ними объекта. Как следует из приведенного выше фрагмента кода, класс `MessageDigestFactoryBean` передает клон сохраненного экземпляра типа `MessageDigest`, получаемого при обратном вызове метода `InitializingBean.afterPropertiesSet()`.

С помощью метода `getObjectType()` можно предписать каркасу Spring, какой именно тип объекта возвратит данный компонент, выполняющий функции фабрики компонентов Spring Beans. Если тип объекта заранее неизвестен, когда, например, фабрика компонентов Spring Beans создает объекты разных типов в зависимости от конфигурации, что удастся определить только после инициализации данной фабрики, то можно указать пустой тип `null`. Но если тип указан, то он может быть использован в Spring для автосвязывания. Возвращаемым оказывается тип `MessageDigest` (в данном случае это класс, но можно попытаться возвратить и тип интерфейса, принудив тем самым фабрику компонентов Spring Beans получить экземпляр конкретного класса, реализующего данный интерфейс, если только в этом нет особой надобности). Дело в том, что заранее неизвестно, какой именно тип будет возвращен, хотя это не так важно, поскольку компоненты Spring Beans будут все равно определять свои зависимости с помощью класса `MessageDigest`.

Метод `isSingleton()` позволяет известить Spring, что фабрика компонентов Spring Beans управляет одиночным экземпляром. Не следует, однако, забывать, что когда определяется фабрика компонентов Spring Beans и в дескрипторе разметки `<bean>` устанавливается атрибут `singleton`, то тем самым каркас Spring извещается об одиночном состоянии самой фабрики, а не возвращаемых ею объектов.

А теперь выясним, как задействовать фабрику компонентов Spring Beans в приложении. Ниже приведен пример простого компонента Spring Bean, обслуживающего два экземпляра типа `MessageDigest` и отображающего свертки сообщения, передаваемого методу `digest()`.

```
package com.apress.prospring5.ch4;

import java.security.MessageDigest;

public class MessageDigester {
  private MessageDigest digest1;
  private MessageDigest digest2;

  public void setDigest1(MessageDigest digest1) {
    this.digest1 = digest1;
  }

  public void setDigest2(MessageDigest digest2) {
    this.digest2 = digest2;
  }

  public void digest(String msg) {
    System.out.println("Using digest1");
    digest(msg, digest1);
    System.out.println("Using digest2");
    digest(msg, digest2);
  }

  private void digest(String msg, MessageDigest digest) {
    System.out.println("Using alogrithm: "
                    + digest.getAlgorithm());
    digest.reset();
    byte[] bytes = msg.getBytes();
    byte[] out = digest.digest(bytes);
    System.out.println(out);
  }
}
```

Далее приведен фрагмент из файла `app-context-xml.xml` для конфигурации двух компонентов Spring Beans типа `MessageDigestFactoryBean`, в одном из которых применяется алгоритм шифрования SHA1, а в другом — стандартный алгоритм MD5.

```
<beans ...>

  <bean id="shaDigest"
     class="com.apress.prospring5.ch4
               .MessageDigestFactoryBean"
     p:algorithmName="SHA1"/>

  <bean id="defaultDigest"
     class="com.apress.prospring5.ch4
               .MessageDigestFactoryBean"/>
```

```
  <bean id="digester"
     class="com.apress.prospring5.ch4.MessageDigester"
     p:digest1-ref="shaDigest"
     p:digest2-ref="defaultDigest"/>
</beans>
```

Как видите, выше были сконфигурированы не только два компонента Spring Beans типа `MessageDigestFactoryBean`, но и компонент типа `MessageDigester`, в свойствах `digest1` и `digest2` которого установлены ссылки на два первых компонента. А поскольку в компоненте `defaultDigest` свойство `algorithmName` не указано, то внедрения зависимостей не произойдет, и поэтому будет применен стандартный алгоритм шифрования MD5, жестко закодированный в классе `MessageDigestFactoryBean`. Ниже приведен простой пример класса, в котором извлекается компонент типа `MessageDigester` из фабрики компонентов Spring Beans и формируется свертка простого сообщения.

```
package com.apress.prospring5.ch4;

import org.springframework.context.support
          .GenericXmlApplicationContext;

public class MessageDigestDemo {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();

    MessageDigester digester =
           ctx.getBean("digester", MessageDigester.class);
    digester.digest("Hello World!");

    ctx.close();
  }
}
```

Выполнение исходного кода из данного примера дает следующий результат:

```
Using digest1
Using alogrithm: SHA1
[B@130f889
Using digest2
Using alogrithm: MD5
[B@1188e820
```

Как следует из приведенного выше результата, компонент Spring Bean типа `MessageDigest` снабжен двумя реализациями (`SHA1` и `MD5`) компонента типа `MessageDigest`, несмотря на то, что компоненты типа `MessageDigest` не были

сконфигурированы в фабрике компонентов Spring Beans, потому что именно так эта фабрика и функционирует.

Фабрики компонентов Spring Beans не являются идеальным решением для работы с классами, экземпляры которых не могут быть получены с помощью операции `new`. Если же вы работаете с объектами, созданными с помощью фабричного метода, и хотите применять эти классы в приложении Spring, создайте фабрику компонентов Spring Beans, чтобы она действовала в качестве адаптера, дающего классам возможность воспользоваться всеми преимуществами встроенных в Spring средств инверсии управления.

Фабрики компонентов Spring Beans применяются иначе, если конфигурирование осуществляется на языке Java, поскольку в этом случае компилятор накладывает ограничение на установку надлежащего типа данных в свойстве, а следовательно, метод `getObject()` придется вызывать вручную. В следующем фрагменте кода приведен пример конфигурирования тех же самых компонентов Spring Beans, что и в предыдущем примере, но на этот раз — на языке Java:

```
package com.apress.prospring5.ch4.config;

import com.apress.prospring5.ch4.MessageDigestFactoryBean;
import com.apress.prospring5.ch4.MessageDigester;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.support
          .GenericApplicationContext;

public class MessageDigesterConfigDemo {
  @Configuration
  static class MessageDigesterConfig {

    @Bean
    public MessageDigestFactoryBean shaDigest() {
      MessageDigestFactoryBean factoryOne =
             new MessageDigestFactoryBean();
      factoryOne.setAlgorithmName("SHA1");
      return factoryOne;
    }

    @Bean
    public MessageDigestFactoryBean defaultDigest() {
      return new MessageDigestFactoryBean();
    }

    @Bean
    MessageDigester digester() throws Exception {
      MessageDigester messageDigester =
```

```
                    new MessageDigester();
      messageDigester.setDigest1(shaDigest().getObject());
      messageDigester.setDigest2(defaultDigest()
                    .getObject());
      return messageDigester;
    }
  }

  public static void main(String... args) {
    GenericApplicationContext ctx =
          new AnnotationConfigApplicationContext(
                      MessageDigesterConfig.class);
    MessageDigester digester = (MessageDigester)
                              ctx.getBean("digester");
    digester.digest("Hello World!");
    ctx.close();
  }
}
```

Если выполнить исходный код из приведенного выше класса, то на консоль будет выведен тот же самый результат, что и прежде.

Непосредственный доступ к фабрике компонентов Spring Beans

В связи с тем, что каркас Spring автоматически удовлетворяет любые ссылки на фабрику компонентов Spring Beans, выдавая объекты, созданные этой фабрикой, возникает вопрос: можно ли получить доступ к такой фабрике напрямую? Да, можно. Доступ к фабрике компонентов Spring Beans осуществляется просто. Для этого имя компонента Spring Bean достаточно предварить в вызове метода `getBean()` знаком амперсанда, как показано в следующем фрагменте кода:

```
package com.apress.prospring5.ch4;

import org.springframework.context.support
         .GenericXmlApplicationContext;
import java.security.MessageDigest;

public class AccessingFactoryBeans {

  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();
    ctx.getBean("shaDigest", MessageDigest.class);
```

```
    MessageDigestFactoryBean factoryBean =
                (MessageDigestFactoryBean)
                ctx.getBean("&shaDigest");

    try {
      MessageDigest shaDigest = factoryBean.getObject();
      System.out.println(shaDigest.digest(
                        "Hello world".getBytes()));
    } catch (Exception ex) {
      ex.printStackTrace();
    }
    ctx.close();
  }
}
```

Выполнение исходного кода из приведенного выше примера дает следующий результат:

```
[B@130f889
```

Непосредственный доступ к фабрике компонентов Spring Beans применяется в нескольких местах исходного кода Spring, но в приложении не должно возникать для него каких-нибудь веских оснований. Интерфейс `FactoryBean` служит как часть поддерживающей инфраструктуры, чтобы дать возможность применять большее количество классов приложения в установке инверсии управления. Избегайте прямого доступа к интерфейсу `FactoryBean` и вызова метода `getObject()` вручную, предоставив это самому каркасу Spring. Ведь в таком случае вы делаете лишнюю работу и неизбежно привязываете свое приложение к особенностям конкретной реализации, которые могут впоследствии измениться.

Применение атрибутов `factory-bean` и `factory-method`

Иногда требуется получить экземпляры компонентов JavaBeans, предоставляемых сторонним приложением, не поддерживающим Spring. Вам, например, неизвестно, как получить их экземпляры, но в то же время известно, что стороннее приложение предоставляет класс, который можно использовать для получения экземпляра компонента JavaBean, требующегося в вашем приложении Spring. В таком случае вы можете также воспользоваться атрибутами `factory-bean` и `factory-method` дескриптора разметки `<bean>` для определения компонента Spring Bean.

Чтобы продемонстрировать этот прием на практике, ниже приведена еще одна версия класса `MessageDigestFactory`, в которой предоставляется метод для возвращения компонента Spring Bean типа `MessageDigest`.

```
package com.apress.prospring5.ch4;

import java.security.MessageDigest;

public class MessageDigestFactory {
  private String algorithmName = "MD5";

  public MessageDigest createInstance() throws Exception {
    return MessageDigest.getInstance(algorithmName);
  }
  public void setAlgorithmName(String algorithmName) {
    this.algorithmName = algorithmName;
  }
}
```

В следующем фрагменте кода из файла app-contextxml.xml показано, как сконфигурировать фабричный метод для получения соответствующего экземпляра компонента Spring Bean типа MessageDigest:

```
<beans...>

  <bean id="shaDigestFactory"
    class="com.apress.prospring5.ch4.MessageDigestFactory"
    p:algorithmName="SHA1"/>

  <bean id="defaultDigestFactory"
    class="com.apress.prospring5.ch4.MessageDigestFactory"/>

  <bean id="shaDigest"
    factory-bean="shaDigestFactory"
    factory-method="createInstance">
  </bean>

  <bean id="defaultDigest"
    factory-bean="defaultDigestFactory"
    factory-method="createInstance"/>

  <bean id="digester"
    class="com.apress.prospring5.ch4.MessageDigester"
    p:digest1-ref="shaDigest"
    p:digest2-ref="defaultDigest"/>
</beans>
```

В приведенной выше конфигурации определены компоненты Spring Beans для двух фабрик свертки сообщений, в одном из которых применяется алгоритм шифрования SHA1, а в другом — стандартный алгоритм. Затем в атрибутах factory-bean и factory-method каркасу Spring предписывается получить экземпляры компонентов shaDigest и defaultDigest с помощью соответствующей фабрики свертки

сообщений и заданного фабричного метода. Ниже приведен код класса, предназначенного для тестирования.

```
package com.apress.prospring5.ch4;

import org.springframework.context.support
          .GenericXmlApplicationContext;

public class MessageDigestFactoryDemo {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
           new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();

    MessageDigester digester = ctx.getBean("digester",
                               MessageDigester.class);
    digester.digest("Hello World!");
    ctx.close();
  }
}
```

Выполнение исходного кода из данного примера дает следующий результат:

```
Using digest1
Using alogrithm: SHA1
[B@77a57272
Using digest2
Using alogrithm: MD5
[B@7181ae3f
```

Редакторы свойств компонентов Spring Beans

Тем, кому не вполне известны принципы организации компонентов JavaBeans, напомним, что `PropertyEditor` (редактор свойств) — это интерфейс, предназначенный для взаимного преобразования типов внутреннего представления значений свойств и типа `String`. Первоначально такое преобразование задумывалось как способ ввода значений свойств в виде строковых значений типа `String` в редактор и последующего их преобразования в нужный тип. Но поскольку редакторы свойств, по существу, являются легковесными классами, то они нашли применение во многих средах, включая и Spring.

Изрядная доля значений свойств в приложении, основанном на Spring, задается в файле конфигурации по интерфейсу `BeanFactory`, и поэтому они, по существу, являются символьными строками. Тем не менее свойства, для которых указываются эти значения, могут и не относиться к типу `String`. Таким образом, вместо искусственного создания целого ряда свойств типа `String` в каркасе Spring можно определить

редакторы свойств, чтобы управлять процессом преобразования заданных строковых значений свойств типа `String` в нужные их типы. На рис. 4.2 приведены все редакторы свойств, входящие в состав пакета `spring-beans`. Полный их перечень можно просмотреть в любом логически развитом редакторе исходного кода Java.

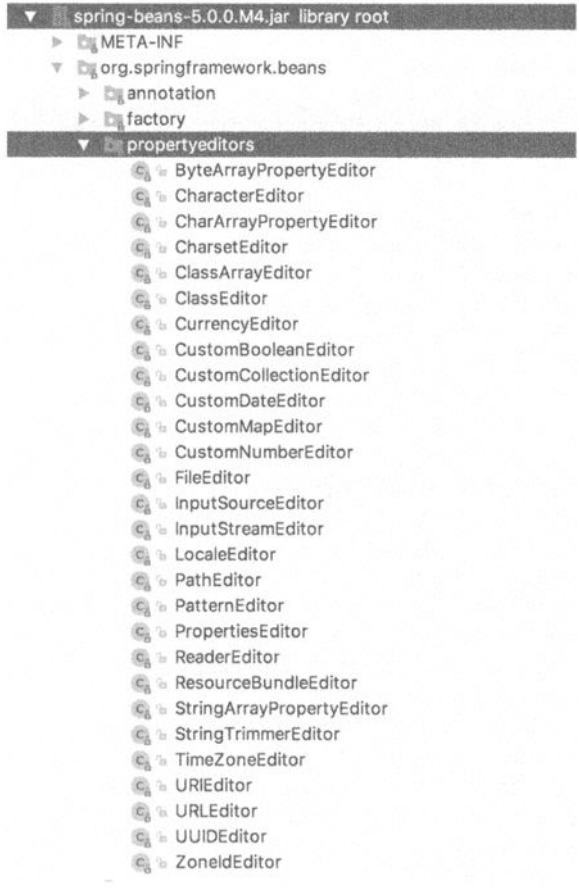


Рис. 4.2. Редакторы свойств в Spring

Все эти редакторы расширяют класс `java.beans.PropertyEditorSupport` и могут быть использованы для неявного преобразования строковых литералов типа `String` в значения свойств, внедряемых в компоненты Spring Beans. Это означает, что они предварительно зарегистрированы средствами интерфейса `BeanFactory`.

Применение встроенных редакторов строк

В приведенном ниже фрагменте кода демонстрируется простой компонент Spring Bean, в котором объявляются 14 свойств — по одному на каждый из типов, поддерживаемых в реализациях интерфейса `PropertyEditor`.

```
package com.apress.prospring5.ch4;

import java.io.File;
import java.io.InputStream;
import java.net.URL; import java.util.Date;
import java.util.List; import java.util.Locale;
import java.util.Properties;
import java.util.regex.Pattern;
import java.text.SimpleDateFormat;
import org.springframework.beans.PropertyEditorRegistrar;
import org.springframework.beans.PropertyEditorRegistry;

import org.springframework.beans
          .propertyeditors.CustomDateEditor;
import org.springframework.beans
          .propertyeditors.StringTrimmerEditor;
import org.springframework.context
          .support.GenericXmlApplicationContext;

public class PropertyEditorBean {

  private byte[] bytes; // ByteArrayPropertyEditor
  private Character character; // CharacterEditor
  private Class cls; // ClassEditor
  private Boolean trueOrFalse; // CustomBooleanEditor
  private List<String> stringList; // CustomCollectionEditor
  private Date date; // CustomDateEditor
  private Float floatValue; // CustomNumberEditor
  private File file; // FileEditor
  private InputStream stream; // InputStreamEditor
  private Locale locale; // LocaleEditor
  private Pattern pattern; // PatternEditor
  private Properties properties; // PropertiesEditor
  private String trimString; // StringTrimmerEditor
  private URL url; // URLEditor

  public void setCharacter(Character character) {
    System.out.println("Setting character: " + character);
    this.character = character;
  }

  public void setCls(Class cls) {
    System.out.println("Setting class: " + cls.getName());
    this.cls = cls;
  }

  public void setFile(File file) {
    System.out.println("Setting file: " + file.getName());
    this.file = file;
```

```
}

public void setLocale(Locale locale) {
  System.out.println("Setting locale: "
                  + locale.getDisplayName());
  this.locale = locale;
}

public void setProperties(Properties properties) {
  System.out.println("Loaded " + properties.size()
                  + " properties");
  this.properties = properties;
}

public void setUrl(URL url) {
  System.out.println("Setting URL: "
                  + url.toExternalForm());
  this.url = url;
}

public void setBytes(byte... bytes) {
  System.out.println("Setting bytes: "
                  + Arrays.toString(bytes));
  this.bytes = bytes;
}

public void setTrueOrFalse(Boolean trueOrFalse) {
  System.out.println("Setting Boolean: " + trueOrFalse);
  this.trueOrFalse = trueOrFalse;
}

public void setStringList(List<String> stringList) {
  System.out.println("Setting string list with size: "
                  + stringList.size());
  this.stringList = stringList;
  for (String string: stringList) {
    System.out.println("String member: " + string);
  }
}

public void setDate(Date date) {
  System.out.println("Setting date: " + date);
  this.date = date;
}

public void setFloatValue(Float floatValue) {
  System.out.println("Setting float value: "
                  + floatValue);
  this.floatValue = floatValue;
```

```
  }

  public void setStream(InputStream stream) {
    System.out.println("Setting stream: " + stream);
    this.stream = stream;
  }

  public void setPattern(Pattern pattern) {
    System.out.println("Setting pattern: " + pattern);
    this.pattern = pattern;
  }

  public void setTrimString(String trimString) {
    System.out.println("Setting trim string: " + trimString);
    this.trimString = trimString;
  }

  public static class CustomPropertyEditorRegistrar
        implements PropertyEditorRegistrar {
  @Override
  public void registerCustomEditors
            (PropertyEditorRegistry registry) {
    SimpleDateFormat dateFormatter =
                    new SimpleDateFormat("MM/dd/yyyy");
    registry.registerCustomEditor(Date.class,
             new CustomDateEditor(dateFormatter, true));
    registry.registerCustomEditor(String.class,
             new StringTrimmerEditor(true));
    }
  }

  public static void main(String... args) throws Exception {
    File file = File.createTempFile("test", "txt");
    file.deleteOnExit();

    GenericXmlApplicationContext ctx =
           new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-01.xml");
    ctx.refresh();

    PropertyEditorBean bean = (PropertyEditorBean)
                              ctx.getBean("builtInSample");

    ctx.close();
  }
}
```

В следующем фрагменте кода из файла `app-config-01.xml` демонстрируется конфигурация, в которой объявляется компонент Spring Bean типа `PropertyEditor Bean` со значениями, указанными для всех определенных выше свойств:

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:util="http://www.springframework.org/schema/util"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation=
       "http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans
              /spring-beans.xsd
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util
             /spring-util.xsd">

    <bean id="customEditorConfigurer"
      class="org.springframework.beans.factory.config
                .CustomEditorConfigurer"
      p:propertyEditorRegistrars-ref=
                "propertyEditorRegistrarsList"/>

    <util:list id="propertyEditorRegistrarsList">
        <bean class=
             "com.apress.prospring5.ch4.PropertyEditorBean$
              CustomPropertyEditorRegistrar"/>

    </util:list>
        <bean id="builtInSample"
           class="com.apress.prospring5. ch4
                     .PropertyEditorBean"
           p:character="A"
           p:bytes="John Mayer"
           p:cls="java.lang.String"
           p:trueOrFalse="true"
           p:stringList-ref="stringList"
           p:stream="test.txt"
           p:floatValue="123.45678"
           p:date="05/03/13"
           p:file="#{systemProperties'java.io.tmpdir'}
             #{systemProperties'file.separator'}test.txt"
           p:locale="en_US"
           p:pattern="a*b"
           p:properties="name=Chris age=32"
           p:trimString=" String need trimming "
           p:url="https://spring.io/"
        />
```

```
    <util:list id="stringList">
        <value>String member 1</value>
        <value>String member 2</value>
    </util:list>
</beans>
```

Как видите, значения для свойств указаны просто как строковые, несмотря на то, что ни одно из свойств в компоненте Spring Bean типа `PropertyEditorBean` не относится к типу `String`. Обратите также внимание на регистрацию редакторов свойств типа `CustomDateEditor` и `StringTrimmerEditor`, поскольку эти редакторы не зарегистрированы в Spring по умолчанию. Выполнение исходного кода из данного примера дает следующий результат:

```
Setting bytes: [74, 111, 104, 110, 32, 77, 97, 121, 101, 114]
Setting character: A
Setting class: java.lang.String
Setting date: Wed May 03 00:00:00 EET 13
Setting file: test.txt
Setting float value: 123.45678
Setting locale: English (United States)
Setting pattern: a*b
Loaded 1 properties
Setting stream: java.io.BufferedInputStream@42e25b0b
Setting string list with size: 2
String member: String member 1
String member: String member 2
Setting trim string: String need trimming
Setting Boolean: true
Setting URL: https://spring.io/[4]
```

[4] Заданы байты: [74, 111, 104, 110, 32, 77, 97, 121, 101, 114]
Задан символ: A
Задан класс: java.lang.String
Задана дата: Wed May 03 00:00:00 EET 13
Задан файл: test.txt
Задано числовое значение с плавающей точкой: 123.45678
Заданы региональные настройки: английский (США)
Задан шаблон: a*b
Загружено 1 свойство
Задан поток ввода: java.io.BufferedInputStream@42e25b0b
Задан список строк длиной: 2
Строковый член: строковый член 1
Строковый член: строковый член 2
Задана обрезанная строка: строка требует обрезки
Задано логическое значение: true
Задан URL: https://spring.io/

Как видите, встроенные в Spring редакторы свойств служат для преобразования строковых представлений различных свойств в нужные типы. Наиболее употребительные из встроенных в Spring редакторов свойств перечислены в табл. 4.1.

Таблица 4.1. Встроенные в Spring редакторы свойств

Редактор свойств	Описание
`ByteArrayPropertyEditor`	Преобразует строковое значение типа `String` в массив байтов
`CharacterEditor`	Заполняет свойство типа `char` или `Character` символами, взятыми из строкового значения типа `String`
`ClassEditor`	Преобразует полностью уточненное имя класса в экземпляр типа `Class`. Пользуясь этим редактором свойств, будьте внимательны, чтобы не ввести лишние пробелы до или после имени класса, когда он применяется в контексте типа `GenericXmlApplicationContext`, иначе будет сгенерировано исключение `ClassNotFoundException`
`CustomBooleanEditor`	Преобразует символьную строку в логический тип языка Java
`CustomCollectionEditor`	Преобразует исходную коллекцию, представленную в Spring, например, пространством имен `util`, в целевой тип `Collection`
`CustomDateEditor`	Преобразует строковое представление даты в значение типа `java.util.Date`. Реализацию этого редактора свойств необходимо зарегистрировать с требующимся форматом даты в контексте типа `ApplicationContext` из каркаса Spring
`FileEditor`	Преобразует строковое представление типа `String` пути к файлу в экземпляр типа `File`. Наличие файла в Spring не проверяется
`InputStreamEditor`	Преобразует строковое представление ресурса (например, файлового ресурса, задаваемого с помощью атрибута `file:D:/temp/test.txt` или `classpath:test.txt`) в свойство потока ввода
`LocaleEditor`	Преобразует строковое представление типа `String` региональных настроек (например, `en-GB`) в экземпляр типа `java.util.Locale`
`PatternEditor`	Преобразует символьную строку в объект типа `Pattern` из комплекта JDK и обратно
`PropertiesEditor`	Преобразует символьную строку формата $ключ_1$=$значение_1$ $ключ_2$=$значение_2$ $ключ_n$=$значение_n$ в экземпляр типа `java.util.Properties` с настройкой соответствующих свойств

Окончание табл. 4.1

Редактор свойств	Описание
`StringTrimmerEditor`	Выполняет усечение строковых значений перед их внедрением. Этот редактор свойств должен быть зарегистрирован вручную
`URLEditor`	Преобразует строковое представление URL в экземпляр типа `java.net.URL`

Этот набор редакторов свойств служит неплохим основанием для работы с Spring и намного упрощает конфигурирование приложения с такими типичными компонентами, как файлы и URL.

Создание специального редактора свойств

Несмотря на то что встроенные редакторы свойств охватывают большинство обычных случаев преобразования типов свойств, иногда приходится создавать собственный редактор свойств для поддержки одного класса или целого ряда классов в приложении. В каркасе Spring обеспечивается полная поддержка регистрации специальных редакторов свойств, и единственный ее недостаток заключается в том, что в интерфейсе `java.beans.PropertyEditor` определяется слишком много методов, большинство из которых не имеет никакого отношения к решаемой задаче преобразования типов свойств. Правда, начиная с версии 5, в комплекте JDK предоставляется класс `PropertyEditorSupport`, который можно расширить в специальных редакторах свойств, реализовав один только метод `setAsText()`.

Рассмотрим простой пример реализации специального редактора свойств. Допустим, имеется класс `FullName` с двумя свойствами, `firstName` и `lastName`, определяющими имя и фамилию соответственно:

```
package com.apress.prospring5.ch4.custom;

public class FullName {
  private String firstName;
  private String lastName;

  public FullName(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  public String getFirstName() {
    return firstName;
  }

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

```
  public String getLastName() {
    return lastName;
  }

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

  public String toString() {
    return "First name: " + firstName
         + " - Last name: " + lastName;
  }
}
```

Чтобы упростить конфигурацию приложения, разработаем специальный редактор свойств, преобразующий символьную строку с разделителяющим пробелом в имя и фамилию для соответствующих полей из класса `FullName`. Ниже приведен исходный код этого специального редактора свойств.

```
package com.apress.prospring5.ch4.custom;

import java.beans.PropertyEditorSupport;

public class NamePropertyEditor
       extends PropertyEditorSupport {
  @Override
  public void setAsText(String text)
              throws IllegalArgumentException {
    String[] name = text.split("\\s");
    setValue(new FullName(name[0], name[1]));
  }
}
```

Этот редактор свойств очень прост. Он расширяет класс `PropertyEditor Support` из комплекта JDK и реализует метод `setAsText()`. Сначала в этом методе исходная символьная строка разбивается на массив строк с помощью пробела, задаваемого в качестве разделителя, а затем получается экземпляр класса `FullName`. С этой целью конструктору данного класса передается первая часть исходной символьной строки с именем до пробела и вторая ее часть с фамилией после пробела. И, наконец, результат преобразования возвращается из вызова метода `setValue()`.

Чтобы воспользоваться классом `NamePropertyEditor` специального редактора свойств в приложении, его придется зарегистрировать в контексте типа `Application Context`. В следующем фрагменте кода из файла `app-context-02.xml` представлена конфигурация компонентов Spring Beans типа `CustomEditorConfigurer` и `NamePropertyEditor` в контексте типа `ApplicationContext`:

```
<beans ...>

   <bean name="customEditorConfigurer"
     class="org.springframework.beans.factory
               .config.CustomEditorConfigurer">
       <property name="customEditors">
         <map>
           <entry key="com.apress.prospring5.ch4
                           .custom.FullName"
               value="com.apress.prospring5.ch4
                         .custom.NamePropertyEditor"/>
         </map>
       </property>
   </bean>

   <bean id="exampleBean"
     class="com.apress.prospring5.ch4.custom
               .CustomEditorExample"
     p:name="John Mayer"/>
</beans>
```

В этой конфигурации следует отметить два обстоятельства. Во-первых, специальные редакторы свойств внедряются в класс `CustomEditorConfigurer` с помощью свойства `customEditors` типа `Map`. И во-вторых, каждая запись в отображении типа `Map` представляет одиночный редактор свойств, а ключом для записи служит имя того класса, для которого применяется этот редактор. В данном примере ключом для записи типа `NamePropertyEditor` служит имя класса `com.apress.prospring4.ch4.FullName`, для которого должен использоваться этот редактор.

Ниже приведен исходный код класса `CustomEditorExample`, зарегистрированного как компонент Spring Bean в упомянутой выше конфигурации.

```
package com.apress.prospring5.ch4.custom;

import org.springframework.context.support
          .GenericXmlApplicationContext;

public class CustomEditorExample {
  private FullName name;

  public FullName getName() {
    return name;
  }

  public void setName(FullName name) {
    this.name = name;
  }

  public static void main(String... args) {
```

```
    GenericXmlApplicationContext ctx =
           new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-02.xml");
    ctx.refresh();
    CustomEditorExample bean =
          (CustomEditorExample) ctx.getBean("exampleBean");
    System.out.println(bean.getName());
    ctx.close();
  }
}
```

В приведенном выше коде нет ничего особенно. Выполнение исходного кода из данного примера дает следующий результат:

```
First name: John - Last name: Mayer
```

Этот результат выводится из метода `toString()`,реализованного в классе `FullName`.Он показывает, что поля `firstName` и `lastName` объекта типа `FullName` правильно заполнены с помощью сконфигурированного в Spring редактора свойств типа `NamePropertyEditor`. Начиная с версии 3 в каркасе Spring предоставляется прикладной интерфейс Type Conversion API, а также интерфейс Field Formatting Service Provider Interface (SPI). Совместно они образуют простой и хорошо структурированный прикладной интерфейс для преобразования типов и предоставления услуг форматирования полей, что особенно удобно для разработки веб-приложений. Более подробно прикладные интерфейсы Type Conversion API и Field Formatting SPI рассматриваются в главе 10.

Еще о конфигурировании в контексте типа `ApplicationContext`

Несмотря на то что интерфейс `ApplicationContext` уже рассматривался, большая часть упомянутых ранее функциональных возможностей касалась интерфейса `BeanFactory`, заключаемого в оболочку интерфейса `ApplicationContext`. Различные реализации интерфейса `BeanFactory` в Spring отвечают за получение экземпляров компонентов Spring Beans, обеспечивая поддержку внедрения зависимостей и жизненного цикла компонентов под управлением Spring. Но, как утверждалось ранее, в интерфейсе `ApplicationContext` предоставляются и другие полезные функциональные возможности, помимо расширения интерфейса `BeanFactory`. Главная функция интерфейса `ApplicationContext` состоит в предоставлении намного более развитого каркаса, на основе которого строятся приложения. Интерфейс `ApplicationContext` осведомлен о сконфигурированных в нем компонентах Spring Beans в намного большей степени, чем интерфейс `BeanFactory`. Что же касается многих классов и интерфейсов инфраструктуры Spring (например, `BeanFactory PostProcessor`), то он взаимодействует с компонентами Spring Beans от имени раз-

работчика, сокращая объем кода, который пришлось бы написать для применения Spring.

Наибольшая выгода от применения интерфейса `ApplicationContext` состоит в том, что он позволяет конфигурировать каркас Spring и управлять им и его ресурсами полностью декларативным способом. Это означает, что по мере возможности Spring предоставляет классы, поддерживающие автоматическую загрузку контекста типа `ApplicationContext` в приложение, избавляя от необходимости писать код для доступа к этому контексту. На практике такая возможность доступна в настоящее время только при построении веб-приложений средствами Spring, позволяя инициализировать контекст типа `ApplicationContext` в дескрипторе развертывания веб-приложения. А в автономном приложении для инициализации контекста типа `ApplicationContext` потребуется написание простого кода.

Помимо предоставления модели, которая в большей степени сосредоточена на декларативном конфигурировании, в интерфейсе `ApplicationContext` поддерживается следующий ряд дополнительных возможностей.

- Интернационализация.
- Публикация событий.
- Доступ к ресурсам и управление ими.
- Дополнительные интерфейсы жизненного цикла.
- Улучшенное автоматическое конфигурирование компонентов инфраструктуры.

В последующих разделах будут описаны самые важные функциональные возможности интерфейса `ApplicationContext`, помимо внедрения зависимостей.

Интернационализация средствами интерфейса `MessageSource`

Одной из областей, в которых Spring действительно превосходит другие каркасы, является поддержка интернационализации (сокращенно i18n). Через интерфейс `MessageSource` приложение может получать доступ к строковым ресурсам типа `String`, называемым *сообщениями* и хранящимся на разных языках. Для каждого языка, который должен поддерживаться в приложении, предоставляется список сообщений с ключами соответствия сообщений на других языках. Так, если требуется отобразить английскую панграмму "The quick brown fox jumped over the lazy dog" (Шустрая бурая лиса прыгает через ленивую собаку) и на немецком языке, то для этого необходимо составить два сообщения с ключом соответствия `msg`: одно — `"The quick brown fox jumped over the lazy dog"` по-английски, а другое — `"Der schnelle braune Fuchs sprang über den faulen Hund"` по-немецки.

И хотя интерфейс `ApplicationContext` не нужен для применения интерфейса `MessageSource`, он все же расширяет интерфейс `MessageSource` и специально поддерживает загрузку сообщений, а также их доступность в рабочей среде. Если автоматическая загрузка сообщений доступна в любой среде, то автоматический до-

ступ предоставляется лишь в некоторых случаях, например, при построении веб-приложений с помощью каркаса по шаблону MVC под управлением Spring. Несмотря на то что интерфейс ApplicationContextAware можно реализовать в любом классе и таким образом получить доступ к автоматически загружаемым сообщениям, в разделе "Применение интерфейса MessageSource в автономных приложениях" далее в этой главе будет предложено лучшее решение.

Если вы еще не знакомы с поддержкой интернационализации в Java, то прежде чем продолжить чтение, ознакомьтесь с документацией в формате Javadoc, доступной по адресу http://www.oracle.com/technetwork/java/javase/tech/intl-139810.html.

Применение интерфейса MessageSource для интернационализации

Помимо интерфейса ApplicationContext, в Spring предоставляются следующие реализации интерфейса MessageSource:

- ResourceBundleMessageSource
- ReloadableResourceBundleMessageSource
- StaticMessageSource

Реализация StaticMessageSource не должна применяться в рабочем приложении, поскольку она не подлежит внешнему конфигурированию, а это, как правило, одно из главных требований для внедрения возможностей интернационализации в приложение. А реализация ResourceBundleMessageSource загружает сообщения с помощью класса ResourceBundle на языке Java. Реализация ReloadableResourceBundleMessageSource в основном такая же, хотя и поддерживает запланированную перезагрузку базовых исходных файлов.

Во всех трех реализациях интерфейса MessageSource реализуется еще один интерфейс под названием HierarchicalMessageSource, который позволяет вкладывать друг в друга экземпляры типа MessageSource. Это ключ к пониманию того, как интерфейс ApplicationContext взаимодействует со многими экземплярами типа MessageSource.

Чтобы выгодно воспользоваться поддержкой интерфейса MessageSource, предоставляемой в контексте типа ApplicationContext, в конфигурации должен быть определен компонент Spring Bean типа MessageSource под названием messageSource. Этот компонент вкладывается в контекст типа ApplicationContext, где разрешается доступ к сообщениям. Для лучшего понимания этого механизма имеет смысл обратиться к конкретному примеру. Ниже приведен исходный код простого приложения, в котором доступны английские и немецкие региональные настройки.

```
package com.apress.prospring5.ch4;

import java.util.Locale;
import org.springframework.context.support
          .GenericXmlApplicationContext;
```

```
public class MessageSourceDemo {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();

    Locale english = Locale.ENGLISH;
    Locale german = new Locale("de", "DE");

    System.out.println(ctx.getMessage("msg", null, english));
    System.out.println(ctx.getMessage("msg", null, german));
    System.out.println(ctx.getMessage("nameMsg", new Object[]
                      { "John", "Mayer" }, english));
    System.out.println(ctx.getMessage("nameMsg", new Object[]
                      { "John", "Mayer" }, german));
    ctx.close();
  }
}
```

Не обращайте пока что особого внимания на вызовы метода `getMessage()`, поскольку мы еще вернемся к этому вопросу. А до тех пор достаточно сказать, что в этих вызовах извлекаются сообщения с ключами для заданных региональных настроек. Ниже приведена конфигурация из файла `app-context-xml.xml`, применяемая в рассматриваемом здесь приложении.

```
<beans ...>

  <bean id="messageSource"
     class="org.springframework.context.support
               .ResourceBundleMessageSource"
     p:basenames-ref="basenames"/>

  <util:list id="basenames">
    <value>buttons</value>
    <value>labels</value>
  </util:list>
</beans>
```

В приведенной выше конфигурации определяется, как и требуется, компонент Spring Beans типа `ResourceBundleMessageSource` под названием `messageSource` с базовыми именами, служащими основанием для формирования набора файлов. Класс `ResourceBundle` на языке Java, применяемый в реализации `ResourceBundleMessageSource`, оперирует набором файлов свойств, распознаваемых по базовым именам. Если требуется найти сообщение для конкретных региональных настроек, в реализации `ResourceBundle` осуществляется поиск файла, имя которого состоит из базового имени и имени региональных настроек. Так, если требуется найти сообщение по базому имени `foo` и для региональных настроек `en-GB` (британский

английский), то в реализации ResourceBundle осуществляется поиск файла по имени foo_en_GB.properties.

Ниже приведено содержимое файлов свойств для английского (labels_en.properties) и немецкого (labels_de_DE.properties) языков.

```
#labels_en.properties
msg=My stupid mouth has got me in trouble
nameMsg=My name is {0} {1}
#labels_de_DE.properties
msg=Mein dummer Mund hat mich in Schwierigkeiten gebracht
nameMsg=Mein Name ist {0} {1}
```

Теперь данный пример вызывает еще больше вопросов. В частности, что означают вызовы метода getMessage()? Зачем был сделан вызов метода ApplicationContext.getMessage() вместо непосредственного доступа к компоненту Spring Bean типа ResourceBundleMessageSource? Разберем эти вопросы по очереди.

Применение метода getMessage()

В интерфейсе MessageSource определены три перегружаемые версии метода getMessage(), вкратце описанные в табл. 4.2.

Таблица 4.2. Перегружаемые версии метода MessageSource.getMessage()

Сигнатура метода	Описание
getMessage (String, Object[], Locale)	Это стандартная версия метода **getMessage()**. Первый аргумент типа **String** обозначает ключ сообщения, соответствующий ключу из файла свойств. В первом вызове метода **getMessage()** из предыдущего примера задан ключ **msg**, соответствующий следующей записи в файле свойств английских региональных настроек: **msg=The quick brown fox jumped over the lazy dog**. В качестве второго аргумента задается массив типа **Object[]**, предназначенный для хранения замен в сообщении. В третьем вызове метода **getMessage()** передается массив, состоящий из двух элементов типа **String**. В итоге по заданному ключу **nameMsg** из английских региональных настроек было выбрано сообщение **My name is {0} {1}**. Числа в фигурных скобках являются заполнителями, каждый из которых замещается соответствующим элементом из массива, заданного в качестве второго аргумента. И в качестве последнего аргумента **Locale** указывается файл, который следует искать в реализации **ResourceBundleMessageSource**. Несмотря на то что в первом и во втором вызовах **getMessage()** из предыдущего примера применялся один и тот же ключ, в итоге возвращались разные сообщения, соответствующие значению третьего аргумента **Locale** вызываемого метода **getMessage()**

Окончание табл. 4.2

Сигнатура метода	Описание
`getMessage (String, Object[], String, Locale)`	Эта перегружаемая версия действует аналогично рассмотренной выше стандартной версии метода **`getMessage()`**, но допускает указывать в качестве третьего аргумента типа **`String`** стандартное значение на тот случай, если сообщение по заданному ключу отсутствует в региональных настройках, обозначаемых в качестве последнего аргумента **`Locale`**
`getMessage (MessageSource Resolvable, Locale)`	Эта перегружаемая версия данного метода представляет собой особый случай и более подробно рассматривается далее в подразделе "Интерфейс **`MessageSourceResolvable`**"

Зачем пользоваться интерфейсом `ApplicationContext` в качестве интерфейса `MessageSource`

Чтобы ответить на этот вопрос, придется немного забежать вперед и выяснить, каким образом веб-приложения поддерживаются в Spring. Как правило, использовать интерфейс `ApplicationContext` в качестве интерфейса `MessageSource` не следует, поскольку это приводит к нежелательной привязке компонента Spring Beans к контексту типа `ApplicationContex`, как поясняется более подробно в следующем подразделе. Интерфейс `ApplicationContext` следует применять при построении веб-приложений в Spring с помощью инфраструктуры по шаблону MVC.

Базовым для инфраструктуры по шаблону MVC в Spring служит интерфейс `Controller`. В отличие от таких каркасов, как Struts, где контроллеры приходится реализовывать через наследование от конкретного класса, в Spring достаточно реализовать интерфейс `Controller` (или снабдить разрабатываемый класс контроллера аннотацией `@Controller`). Поэтому в Spring предоставляется целый ряд полезных базовых классов, которыми можно пользоваться для реализации собственных контроллеров. Все эти базовые классы сами являются (прямо или косвенно) подклассами, производными от класса `ApplicationObjectSupport`, который служит удобным суперклассом для любых объектов в приложении, которые должны быть осведомлены о контексте типа `ApplicationContext`. Не следует, однако, забывать, что в среде веб-приложения контекст типа `ApplicationContext` загружается автоматически.

Класс `ApplicationObjectSupport` обращается к данному контексту типа `ApplicationContext`, заключает его в оболочку объекта типа `MessageSource Accessor` и делает доступным для контроллера через защищенный метод `get MessageSourceAccessor()`. Такая форма автоматического внедрения очень удобна, поскольку избавляет от необходимости делать свойство компонента типа `Message Source` доступным для всех контроллеров.

Однако это не самая главная причина для применения интерфейса ApplicationContext в качестве интерфейса MessageSource в веб-приложении. Она заключается в том, что Spring делает там, где это возможно, доступным интерфейс ApplicationContext как интерфейс MessageSource на уровне представления. Это означает, что когда применяется библиотека дескрипторов JSP, поддерживаемая в Spring, в дескрипторе <spring:message> автоматически выполняется чтение сообщений из контекста типа ApplicationContext, а если применяется стандартная библиотека JSTL, то же самое происходит и в дескрипторе <fmt:message>.

Все эти преимущества означают, что при разработке веб-приложений лучше пользоваться поддержкой интерфейса MessageSource в интерфейсе ApplicationContext, чем манипулировать отдельно экземпляром компонента типа MessageSource. Это особенно справедливо, если принять во внимание, что для реализации такой возможности достаточно сконфигурировать компонент типа MessageSource под именем messageSource.

Применение интерфейса MessageSource в автономных приложениях

Если интерфейс MessageSource применяется в автономных приложениях, где каркас Spring не предоставляет никакой дополнительной поддержки, кроме автоматического вложения компонента Spring Bean типа MessageSource в контекст типа ApplicationContext, то интерфейс MessageSource лучше всего сделать доступным через внедрение зависимостей. Можно, конечно, выбрать информирование компонентов Spring Beans об их контексте через интерфейс ApplicationContextAware, но это воспрепятствует их применению в контексте типа BeanFactory. Если к этому добавить усложнение тестирования без всякой ощутимой выгоды, то необходимость придерживаться внедрения зависимостей для доступа к объектам типа MessageSource в автономных приложениях станет совершенно очевидной.

Интерфейс MessageSourceResolvable

При поиске сообщения в источнике типа MessageSource вместо ключа и целого ряда аргументов можно воспользоваться объектом, реализующим интерфейс MessageSourceResolvable. Наиболее широко этот интерфейс применяется для связывания объектов типа Error с их интернационализированными сообщениями об ошибках в библиотеках Spring, предназначенных для проверки достоверности.

События в приложениях

Еще одной особенностью интерфейса ApplicationContext, отсутствующей в интерфейсе BeanFactory, является возможность публиковать и получать события, используя интерфейс ApplicationContext в качестве посредника. Далее поясняется, как это делается.

Применение событий в приложениях

Событие представлено классом, производным от класса `ApplicationEvent`, который, в свою очередь, является производным от класса `java.util.EventObject`. Любой компонент Spring Bean может принимать события, реализовав интерфейс `ApplicationListener<T>`, а в контексте типа `ApplicationContext` автоматически регистрируется в качестве приемника событий любой сконфигурированный компонент Spring Bean, реализующий этот интерфейс. События публикуются с помощью метода `ApplicationEventPublisher.publishEvent()`, поэтому публикующий их класс должен быть осведомлен об интерфейсе `ApplicationContext`, расширяющем интерфейс `ApplicationEventPublisher`. В веб-приложении это достигается просто, поскольку многие классы являются производными от классов из каркаса Spring Framework, которые обеспечивают доступ к контексту типа `ApplicationContext` через защищенный метод. А для того чтобы публиковать события в автономном приложении, компонент Spring Bean должен реализовать интерфейс `ApplicationContextAware`.

Ниже приведен простой пример класса, представляющего событие.

```
package com.apress.prospring5.ch4;

import org.springframework.context.ApplicationEvent;

public class MessageEvent extends ApplicationEvent {
  private String msg;

  public MessageEvent(Object source, String msg) {
    super(source);
    this.msg = msg;
  }

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

В приведенном выше коде нет ничего сложного. Следует лишь заметить, что у класса `ApplicationEvent` имеется единственный конструктор, принимающий ссылку на источник события. Именно это и отражено в конструкторе класса `MessageEvent`. А ниже показано, каким образом реализуется приемник событий.

```
package com.apress.prospring5.ch4;

import org.springframework.context.ApplicationListener;

public class MessageEventListener
       implements ApplicationListener<MessageEvent> {
  @Override
```

```
  public void onApplicationEvent(MessageEvent event) {
    MessageEvent msgEvt = (MessageEvent) event;
    System.out.println("Received: " + msgEvt.getMessage());
  }
}
```

В интерфейсе `ApplicationListener` определен единственный метод `onApplicationEvent()`, который вызывается в Spring, когда наступает событие. Класс `MessageEventListener` реагирует только на события типа `MessageEvent` (или его подтипов), и с этой целью в нем реализуется строго типизированный интерфейс `ApplicationListener`. При получении события типа `MessageEvent` в стандартный поток вывода `stdout` направляется соответствующее сообщение. Публиковать события нетрудно, поскольку для этого достаточно получить экземпляр класса соответствующего события и передать его методу `ApplicationEventPublisher.publishEvent()`, как показано ниже.

```
package com.apress.prospring5.ch4;

import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.context.support
          .ClassPathXmlApplicationContext;

public class Publisher implements ApplicationContextAware {
  private ApplicationContext ctx;

  public void setApplicationContext(
              ApplicationContext applicationContext)
              throws BeansException {
    this.ctx = applicationContext;
  }

  public void publish(String message) {
    ctx.publishEvent(new MessageEvent(this, message));
  }

  public static void main(String... args) {
    ApplicationContext ctx =
                new ClassPathXmlApplicationContext(
                    "classpath:spring/app-context-xml.xml");

    Publisher pub = (Publisher) ctx.getBean("publisher");
    pub.publish("I send an SOS to the world... ");
    pub.publish("... I hope that someone gets my...");
    pub.publish("... Message in a bottle");
  }
}
```

Как видите, класс `Publisher` извлекает из контекста типа `ApplicationContext` свой экземпляр и публикует три события типа `MessageEvent` через метод `publish()` в контексте типа `ApplicationContext`. Экземпляр компонента Spring Bean типа `Publisher` получает доступ к контексту типа `ApplicationContext` благодаря реализации интерфейса `ApplicationContextAware`. Ниже приведена конфигурация из файла `app-context-xml.xml` для данного примера.

```
<beans ...>

  <bean id="publisher"
    class="com.apress.prospring5.ch4.Publisher"/>

  <bean id="messageEventListener"
    class="com.apress.prospring5.ch4.MessageEventListener"/>
</beans>
```

Обратите внимание на то, что для регистрации приемника событий типа `MessageEventListener` в контексте типа `ApplicationContext` никакой специальной конфигурации не требуется, поскольку это делается в Spring автоматически. Выполнение кода из данного примера дает следующий результат:

```
Received: I send an SOS to the world...
Received: ... I hope that someone gets my...
Received: ... Message in a bottle
```

Соображения по поводу применения событий

Как правило, в приложении требуется уведомлять определенные компоненты о тех или иных событиях. И зачастую для этого пишется код, явно уведомляющий каждый компонент, или же применяется какая-нибудь технология обмена сообщениями вроде JMS. Недостаток первого варианта, когда приходится писать специальный код для уведомления каждого компонента по очереди, заключается в том, что компоненты привязываются к публикующему классу, что во многих случаях нежелательно.

Рассмотрим ситуацию, когда подробные сведения о товаре кешируются в приложении, чтобы избежать лишних обращений к базе данных, а в другом компоненте разрешается видоизменить подробные сведения о товаре и сохранить их в базе данных. Чтобы данные в кеше не оказались недействительными, компонент обновления явно уведомляет кеш о том, что сведения о товаре изменились. В данном примере компонент обновления связан с компонентом, который на самом деле не имеет никакого отношения к его предметной ответственности. Более удачное решение может выглядеть следующим образом: компонент обновления публикует событие всякий раз, когда подробные сведения о товаре изменяются, а заинтересованные компоненты, в том числе и тот, который выполняет кеширование данных, принимают это событие. Преимущество такого решения заключается в том, что компоненты остаются несвязанными, а это упрощает удаление кеша, когда он больше не нужен, или внедре-

ние другого приемника событий, заинтересованного в получении сведений об изменении подробных сведений о товаре.

Применение технологии JMS в данном случае было бы излишним, поскольку процесс объявления недействительной записи о товаре в кеше оказывается быстрым и некритичным. А применение инфраструктуры Spring для обработки событий влечет за собой весьма незначительные издержки в приложении.

Как правило, события применяются для реализации такой логики реагирования, которая действует быстро и не является частью главной логики приложения. В приведенном выше примере объявление записи о товаре в кеше недействительной происходит в ответ на обновление подробных сведений о товаре, и это делается быстро (во всяком случае, так должно быть) и не относится к главной функции приложения. А для процессов, которые выполняются долго и образуют часть основной бизнес-логики, рекомендуется применять систему обмена сообщениями JMS или RabbitMQ. Основные преимущества применения технологии JMS заключаются в том, что она больше подходит для длительных процессов, а по мере роста системы обработку сообщений, содержащих бизнес-информацию, под управлением JMS можно, если потребуется, перенести на отдельную машину.

Доступ к ресурсам

Приложение нередко нуждается в доступе к ресурсам в самых разных формах. В частности, может понадобиться доступ к конфигурационным данным из файла, находящегося в файловой системе, к некоторым данным образа из архивного JAR-файла по пути к классам или каким-нибудь другим данным, хранящимся на произвольном сервере. В каркасе Spring предоставляется единообразный механизм для доступа к ресурсам независимо от конкретного протокола. Это означает, что приложение может одинаково получить доступ к файловому ресурсу, где бы он ни находился: в файловой системе, по пути к классам или на удаленном сервере.

В основу всей поддержки ресурсов в Spring положен интерфейс `org.springframework.core.io.Resource`. В интерфейсе `Resource` определены десять самоочевидных методов: `contentLength()`, `exists()`, `getDescription()`, `getFile()`, `getFileName()`, `getURI()`, `getURL()`, `isOpen()`, `isReadable()` и `lastModified()`. Но имеется еще один, не столь самоочевидный метод `createRelative()`, в котором получается новый экземпляр типа `Resource` по относительному пути к тому экземпляру, для которого он вызывается. И хотя можно предоставить собственные реализации интерфейса `Resource` (рассмотрение данного вопроса выходит за рамки этой главы), зачастую применяется одна из встроенных в Spring реализаций для доступа к файлам (класс `FileSystemResource`), по пути к классам (класс `ClassPathResource`) или URL-ресурсам (класс `UrlResource`). Для обнаружения и получения экземпляров типа `Resource` в Spring применяется другой интерфейс, `ResourceLoader`, и его стандартная реализация `DefaultResourceLoader`. Но, как правило, вам не придется обращаться непосредственно к реализации

DefaultResourceLoader, а вместо этого пользоваться другой реализацией интерфейса ResourceLoader — ApplicationContext. Ниже приведен пример приложения, которое обращается к трем ресурсам через интерфейс ApplicationContext.

```
package com.apress.prospring5.ch4;

import java.io.File;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support
          .ClassPathXmlApplicationContext;
import org.springframework.core.io.Resource;

public class ResourceDemo {
  public static void main(String... args) throws Exception {
    ApplicationContext ctx =
              new ClassPathXmlApplicationContext();

    File file = File.createTempFile("test", "txt");
    file.deleteOnExit();

    Resource res1 =
             ctx.getResource("file://" + file.getPath());
    displayInfo(res1);

    Resource res2 = ctx.getResource("classpath:test.txt");
    displayInfo(res2);

    Resource res3 =
             ctx.getResource("http://www.google.com");
    displayInfo(res3);
  }

  private static void displayInfo(Resource res)
                throws Exception {
    System.out.println(res.getClass());
    System.out.println(res.getURL().getContent());
    System.out.println("");
  }
}
```

Обратите внимание на то, что при каждом вызове методу getResource() передается URI отдельного ресурса. Для доступа к ресурсам res1 и res3 в этом методе применяются общепринятые протоколы file: и http:, тогда как для доступа к ресурсу res2 — характерный для Spring протокол classpath:, указывающий на то, что загрузчик ресурсов типа ResourceLoader должен искать ресурс по пути к классам. Выполнение исходного кода из данного примера дает следующий результат:

```
class org.springframework.core.io.UrlResource
java.io.BufferedInputStream@3567135c
```

```
class org.springframework.core.io.ClassPathResource
sun.net.www.content.text.PlainTextInputStream@90f6bfd

class org.springframework.core.io.UrlResource
sun.net.www.protocol.http
   .HttpURLConnection$HttpInputStream@735b5592
```

Как видите, для протоколов `file:` и `http:` каркас Spring возвращает экземпляр типа `UrlResource`. И хотя он подключает и класс `FileSystemResource`, в стандартном загрузчике ресурсов типа `DefaultResourceLoader` этот класс вообще не используется. Дело в том, что URL и файл интерпретируются по стандартной стратегии загрузки ресурсов в Spring как один и тот же тип ресурса, но с разными протоколами (т.е. `file:` и `http:`). Если же требуется экземпляр типа `FileSystemResource`, следует воспользоваться загрузчиком ресурсов типа `FileSystemResourceLoader`. Получив экземпляр типа `Resource`, можно свободно обратиться к доступному содержимому ресурса, вызвав метод `getFile()`, `getInputStream()` или `getURL()`. В ряде случаев, когда, например, применяется протокол `http:`, вызов метода `getFile()` приводит к генерированию исключения типа `FileNotFoundException`. По этой причине для доступа к содержимому ресурсов рекомендуется использовать метод `getInputStream()`, поскольку он будет, вероятнее всего, нормально взаимодействовать со всеми возможными типами ресурсов.

Конфигурирование с помощью классов Java

Кроме формата XML, для конфигурирования контекста типа `ApplicationContext` в Spring можно также применять классы Java. Ранее, здесь и далее приведены примеры исходного кода, помогающие освоить стиль конфигурирования с помощью аннотаций. Ранее для подобных целей существовал отдельный проект Spring JavaConfig, но в версии Spring 3.0 его основные средства, касающиеся конфигурирования с помощью классов Java, были объединены с ядром каркаса Spring Framework. В этом разделе поясняется, каким образом применять классы Java для конфигурирования контекста типа `ApplicationContext`, а для сравнения приводится равнозначная конфигурация в формате XML.

Конфигурирование контекста типа `ApplicationContext` на Java

Итак, рассмотрим конфигурирование контекста типа `ApplicationContext` с помощью классов Java на том же самом примере поставщика и средства воспроизведения сообщений, который был представлен в главах 2 и 3. Для удобства ниже еще раз приведен исходный код интерфейса и класса конфигурируемого поставщика сообщений[5].

[5] Упоминаемые здесь классы не воссоздаются снова в пакете `com.apress.prospring5.ch4`, но проекты, в которых они определены, внедряются как зависимости в проект `java-config`.

```
// chapter02/hello-world/src/main/java/com/apress
//          /prospring5/ch2/decoupled/MessageProvider.java
package com.apress.prospring5.ch2.decoupled;

public interface MessageProvider {
  String getMessage();
}

//chapter03/constructor-injection/src/main/java
//         /com/apress/prospring5/ch3/xml
//         /ConfigurableMessageProvider.java
package com.apress.prospring5.ch3.xml;

import com.apress.prospring5.ch2.decoupled.MessageProvider;

public class ConfigurableMessageProvider
       implements MessageProvider {
  private String message = "Default message";

  public ConfigurableMessageProvider() {
  }

  public ConfigurableMessageProvider(String message) {
    this.message = message;
  }

  public void setMessage(String message) {
    this.message = message;
  }

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

Ниже приведен интерфейс `MessageRenderer` и его реализация в классе `StandardOutMessageRenderer`.

```
//chapter02/hello-world/src/main/java/com/apress
//        /prospring5/ch2/decoupled/MessageRenderer.java
package com.apress.prospring5.ch2.decoupled;

public interface MessageRenderer {
  void render();
  void setMessageProvider(MessageProvider provider);
  MessageProvider getMessageProvider();
}

//chapter02/hello-world/src/main/java/com/apress
```

```
//         /prospring5/ch2/decoupled
//        /StandardOutMessageRenderer.java
package com.apress.prospring5.ch2.decoupled;

public class StandardOutMessageRenderer
       implements MessageRenderer {
  private MessageProvider messageProvider;

  public StandardOutMessageRenderer() {
    System.out.println(" --> StandardOutMessageRenderer:
                            constructor called");
  }

  @Override
  public void render() {
    if (messageProvider == null) {
      throw new RuntimeException("You must set the "
           + "property messageProvider of class:"
           + StandardOutMessageRenderer.class.getName());
    } System.out.println(messageProvider.getMessage());
  }

  @Override
  public void setMessageProvider(MessageProvider provider) {
    System.out.println(" --> StandardOutMessageRenderer:
                            setting the provider");
    this.messageProvider = provider;
  }

  @Override
  public MessageProvider getMessageProvider() {
    return this.messageProvider;
  }
}
```

Соответствующая конфигурация в формате XML из файла `app-context-xml.xml` выглядит следующим образом:

```
<beans ...>

  <bean id="messageRenderer"
    class="com.apress.prospring5.ch2.decoupled
              .StandardOutMessageRenderer"
    p:messageProvider-ref="messageProvider"/>

  <bean id="messageProvider"
    class="com.apress.prospring5.ch3.xml
              .ConfigurableMessageProvider"
    c:message="This is a configurable message"/>
</beans>
```

И, наконец, ниже приведен исходный код класса для тестирования данного примера.

```
package com.apress.prospring5.ch4;

import com.apress.prospring5.ch2.decoupled.MessageRenderer;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support
          .ClassPathXmlApplicationContext;

public class JavaConfigXMLExample {
  public static void main(String... args) {
    ApplicationContext ctx =
                new ClassPathXmlApplicationContext(
                    "classpath:spring/app-context-xml.xml");

    MessageRenderer renderer =
           ctx.getBean("messageRenderer",
                       MessageRenderer.class);
    renderer.render();
  }
}
```

Выполнение исходного кода из данного примера даст следующий результат:

```
--> StandardOutMessageRenderer: constructor called
--> StandardOutMessageRenderer: setting the provider
This is a configurable message[6]
```

Чтобы избавиться от конфигурации в формате XML, придется заменить файл `app-context-xml.xml` специальным классом, который называется *конфигурационным* и снабжен аннотацией `@Configuration`, которая извещает Spring, что данный файл конфигурации построен на основе классов Java. Этот конфигурационный класс будет содержать методы с аннотациями `@Bean`, представляющими определения компонентов Spring Beans. С помощью аннотации `@Bean` объявляется компонент Spring Bean и требования к внедрению зависимостей. Аннотация `@Bean` равнозначна дескриптору разметки `<bean>`, а имя метода — атрибуту `id` в дескрипторе `<bean>`, и когда получается экземпляр компонента Spring Bean типа `MessageRender`, внедрение зависимостей через метод установки достигается благодаря вызову соответствующего метода для получения поставщика сообщений. А это равнозначно тому, чтобы воспользоваться дескриптором разметки `<ref>` при конфигурировании в формате XML. Упомянутые выше аннотации и стиль конфигурирования были представлены в предыдущих главах лишь для ознакомления, но они не были до сих пор рассмотрены

---

[6] `--> StandardOutMessageRenderer: вызван конструктор`
`--> StandardOutMessageRenderer: задан поставщик`
`Это конфигурируемое сообщение`

подробно. Поэтому ниже приведен исходный код конфигурационного класса `AppConfig`, равнозначного представленной ранее конфигурации в формате XML.

```
package com.apress.prospring5.ch4;

import com.apress.prospring5.ch2.decoupled.MessageProvider;
import com.apress.prospring5.ch2.decoupled.MessageRenderer;
import com.apress.prospring5.ch2.decoupled
          .StandardOutMessageRenderer;
import com.apress.prospring5.ch3.xml
          .ConfigurableMessageProvider;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {
  @Bean
  public MessageProvider messageProvider() {
    return new ConfigurableMessageProvider();
  }

  @Bean
  public MessageRenderer messageRenderer() {
    MessageRenderer renderer =
           new StandardOutMessageRenderer();
    renderer.setMessageProvider(messageProvider());

    return renderer;
  }
}
```

В следующем фрагменте кода демонстрируется порядок инициализации экземпляра контекста типа `ApplicationContext` в файле конфигурации на языке Java:

```
package com.apress.prospring5.ch4;

import com.apress.prospring5.ch2.decoupled.MessageRenderer;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;

public class JavaConfigExampleOne {
  public static void main(String... args) {
    ApplicationContext ctx = new
      AnnotationConfigApplicationContext(AppConfig.class);

    MessageRenderer renderer =
     ctx.getBean("messageRenderer", MessageRenderer.class);
```

```
    renderer.render();
  }
}
```

Как следует из приведенного выше фрагмента кода, для инициализации экземпляра контекста типа `ApplicationContext` вызывается конструктор класса `AnnotationConfigApplicationContext`, которому в качестве аргумента передается упомянутый выше конфигурационный класс (используя аргументы переменной длины, этому конструктору можно передать несколько конфигурационных классов). После этого возвращаемым экземпляром контекста типа `ApplicationContext` можно воспользоваться, как обычно. Иногда для целей тестирования конфигурационный класс может быть объявлен как внутренний статический следующим образом:

```
package com.apress.prospring5.ch4;

import com.apress.prospring5.ch2.decoupled.MessageProvider;
import com.apress.prospring5.ch2.decoupled.MessageRenderer;
import com.apress.prospring5.ch2.decoupled
           .StandardOutMessageRenderer;
import com.apress.prospring5.ch3.xml
          .ConfigurableMessageProvider;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

public class JavaConfigSimpleExample {

  @Configuration
  static class AppConfigOne {
    @Bean
    public MessageProvider messageProvider() {
      return new ConfigurableMessageProvider();
    }

    @Bean
    public MessageRenderer messageRenderer() {
      MessageRenderer renderer =
             new StandardOutMessageRenderer();
      renderer.setMessageProvider(messageProvider());
      return renderer;
    }
  }

  public static void main(String... args) {
    ApplicationContext ctx = new
       AnnotationConfigApplicationContext(AppConfig.class);
```

```
    MessageRenderer renderer =
      ctx.getBean("messageRenderer", MessageRenderer.class);

      renderer.render();
  }
}
```

Возвращаемый экземпляр контекста типа `ApplicationContext` может быть использован, как обычно, а выводимый результат окажется таким же, как и при конфигурировании приложения в формате XML, что и показано ниже.

```
--> StandardOutMessageRenderer: constructor called
--> StandardOutMessageRenderer: setting the provider
Default message
```

Рассмотрев простейшее применение конфигурационного класса, перейдем к обсуждению других вариантов конфигурирования на языке Java. Допустим, сообщение требуется вынести из поставщика сообщений во внешний файл свойств (`message.properties`) и затем внедрить его обратно через конструктор класса `ConfigurableMessageProvider`. Содержимое файла свойств `message.properties` таково:

```
message=Only hope can keep me together
```

А ниже приведен исходный код переделанной тестовой программы, сначала загружающей файлы свойств с помощью аннотации `@PropertySource`, а затем внедряющей их в реализацию поставщика сообщений.

```
package com.apress.prospring5.ch4;

import com.apress.prospring5.ch2.decoupled.MessageProvider;
import com.apress.prospring5.ch2.decoupled.MessageRenderer;
import com.apress.prospring5.ch2.decoupled
          .StandardOutMessageRenderer;
import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.context.annotation
          .PropertySource;
import org.springframework.core.env.Environment;

@Configuration
@PropertySource(value = "classpath:message.properties")
public class AppConfigOne {

  @Autowired
  Environment env;
```

```
  @Bean
  public MessageProvider messageProvider() {
    return new ConfigurableMessageProvider(
               env.getProperty("message"));
  }

  @Bean(name = "messageRenderer")
  public MessageRenderer messageRenderer() {
    MessageRenderer renderer =
           new StandardOutMessageRenderer();
    renderer.setMessageProvider(messageProvider());
    return renderer;
  }
}
```

Конфигурационные классы были представлены в главе 3 с целью продемонстрировать способ конфигурирования, равнозначный элементам и атрибутам разметки в формате XML. Объявления компонентов Spring Beans могут быть снабжены и другими аннотациями, связанными с областями видимости компонентов, типом загрузки и зависимостями. Так, в следующем фрагменте кода конфигурационный класс `AppConfigOne` дополняется аннотациями в объявлениях компонентов Spring Beans:

```
package com.apress.prospring5.ch4;

import com.apress.prospring5.ch2.decoupled.MessageProvider;
import com.apress.prospring5.ch2.decoupled.MessageRenderer;
import com.apress.prospring5.ch2.decoupled
          .StandardOutMessageRenderer;
import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.context.annotation.*;
import org.springframework.core.env.Environment;

@Configuration
@PropertySource(value = "classpath:message.properties")
public class AppConfig {

  @Autowired
  Environment env;

  @Bean
  @Lazy
  public MessageProvider messageProvider() {
    return new ConfigurableMessageProvider(
           env.getProperty("message"));
  }

  @Bean(name = "messageRenderer")
  @Scope(value="prototype")
  @DependsOn(value="messageProvider")
```

```
  public MessageRenderer messageRenderer() {
    MessageRenderer renderer =
          new StandardOutMessageRenderer();
    renderer.setMessageProvider(messageProvider());
    return renderer;
  }
}
```

В приведенном выше фрагменте кода был внедрен ряд аннотаций, назначение которых вкратце поясняется в табл. 4.3. Те компоненты Spring Beans, которые определяются с помощью стереотипной аннотации вроде `@Component`, `@Service` и прочих, могут быть использованы в конфигурационном классе Java. С этой целью можно активизировать там, где потребуется, режимы просмотра и автосвязывания компонентов Spring Beans.

Таблица 4.3. Аннотации для конфигурирования на языке Java

Аннотация	Описание
`@PropertySource`	Служит для загрузки файлов свойств в контекст типа **`ApplicationContext`** и принимает в качестве аргумента местоположение (допускается указывать не одно местоположение). В формате XML той же самой цели служит дескриптор разметки **`<context:property-placeholder>`**
`@Lazy`	Предписывает каркасу Spring получать экземпляр компонента Spring Bean лишь по требованию (то же, что и дескриптор разметки **`lazy-init="true"`** в формате XML). У этой аннотации имеется атрибут **`value`**, который по умолчанию принимает логическое значение **`true`**. Поэтому аннотация **`@Lazy(value=true)`** равнозначна по своему действию аннотации **`@Lazy`**
`@Scope`	Служит для определения области видимости компонента Spring, если она должна отличаться от области видимости одиночного компонента
`@DependsOn`	Сообщает каркасу Spring, что одни компоненты Spring Beans зависят от других компонентов, и поэтому необходимо сначала получить экземпляры последних
`@Autowired`	Применяется в приведенном выше примере кода для аннотирования переменной **`env`**, относящейся к типу **`Environment`**, предоставляемому в Spring как средство абстракции среды в интерфейсе **`Environment`**, как поясняется далее в этой главе

В следующем примере кода класс `ConfigurableMessageProvider` объявляется как компонент Spring Bean, реализующего поставщика услуг:

```
package com.apress.prospring5.ch4.annotated;

import com.apress.prospring5.ch2.decoupled.MessageProvider;
import org.springframework.beans.factory
          .annotation.Autowired;
```

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;

@Service("provider")
public class ConfigurableMessageProvider
       implements MessageProvider {

  private String message;

  public ConfigurableMessageProvider(
         @Value("Love on the weekend")String message) {
    this.message = message;
  }

  @Override
  public String getMessage() {
    return this.message;
  }
}
```

Ниже приведен исходный код конфигурационного класса.

```
package com.apress.prospring5.ch4;

import com.apress.prospring5.ch2.decoupled.MessageProvider;
import com.apress.prospring5.ch2.decoupled.MessageRenderer;
import com.apress.prospring5.ch2.decoupled
          .StandardOutMessageRenderer;
import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.context.annotation.*;

@Configuration
@ComponentScan(basePackages=
               {"com.apress.prospring5.ch4.annotated"})
public class AppConfigTwo {

  @Autowired
  MessageProvider provider;

  @Bean(name = "messageRenderer")
  public MessageRenderer messageRenderer() {
    MessageRenderer renderer =
           new StandardOutMessageRenderer();
    renderer.setMessageProvider(provider);
    return renderer;
  }
}
```

В аннотации `@ComponentScan` определяются пакеты, в которых каркас Spring должен просмотреть аннотации к определениям компонентов Spring Beans. По своему действию она равнозначна дескриптору разметки `<context:component-scan>` при конфигурировании в формате XML. Если выполнить исходный код из следующего примера:

```
package com.apress.prospring5.ch4;

import com.apress.prospring5.ch2.decoupled.MessageRenderer;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;

public class JavaConfigExampleTwo {
  public static void main(String... args) {
    ApplicationContext ctx =
               new AnnotationConfigApplicationContext(
                   AppConfigTwo.class);

    MessageRenderer renderer =
           ctx.getBean("messageRenderer",
                       MessageRenderer.class);

    renderer.render();
  }
}
```

то в конечном счете будет получен такой результат:

```
--> StandardOutMessageRenderer: constructor called
--> StandardOutMessageRenderer: setting the provider
Love on the weekend
```

В приложении можно также применять несколько конфигурационных классов для развязки конфигурации и организации компонентов Spring Beans по назначению (например, один такой класс может быть выделен для объявления компонентов DAO, а другой — для объявления компонентов служб и т.д.). Определим для примера компонент `provider` с помощью конфигурационного класса `AppConfigFour`. Этот компонент может быть доступен из другого конфигурационного класса благодаря импорту компонентов Spring Beans, определенных в данном классе. И для этого целевой конфигурационный класс `AppConfigThree` снабжается аннотацией `@Import`, как показано ниже.

```
//AppConfigFour.java
package com.apress.prospring5.ch4.multiple;

import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
```

```
        .Configuration;

@Configuration
@ComponentScan(basePackages=
              {"com.apress.prospring5.ch4.annotated"})

public class AppConfigFour { }

package com.apress.prospring5.ch4.multiple;
import com.apress.prospring5.ch2.decoupled.MessageProvider;
import com.apress.prospring5.ch2.decoupled.MessageRenderer;
import com.apress.prospring5.ch2.decoupled
        .StandardOutMessageRenderer;
import org.springframework.beans.factory
        .annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;

@Configuration
@Import(AppConfigFour.class)
public class AppConfigThree {
  @Autowired
  MessageProvider provider;

  @Bean(name = "messageRenderer")
  public MessageRenderer messageRenderer() {
    MessageRenderer renderer =
           new StandardOutMessageRenderer();
    renderer.setMessageProvider(provider);
    return renderer;
  }
}
```

Если в главном методе из класса `JavaConfigExampleTwo` заменить класс `AppConfigTwo` на класс `AppConfigThree`, то выполнение исходного кода из данного примера приведет к тому же самому результату.

Смешанное конфигурирование в Spring

Но конфигурирование в Spring допускает намного больше возможностей. В частности, формат XML можно сочетать с конфигурационными классами. Это удобно в том случае, если имеются приложения с унаследованным кодом, который нельзя изменить по той или иной причине. Чтобы импортировать объявления компонентов Spring Beans из XML-файла конфигурации, достаточно воспользоваться аннотацией `@ImportResource`. В следующем фрагменте кода демонстрируется объявление компонента Spring Bean в XML-файле конфигурации `app-context-xml-01.xml`:

```
<beans ...>

  <bean id="provider"
     class="com.apress.prospring5.ch4
               .ConfigurableMessageProvider"
     p:message="Love on the weekend" />
</beans>
```

Ниже приведен исходный код конфигурационного класса `AppConfigFive`, где компоненты Spring Beans объявляются в импортируемом XML-файле конфигурации. Если в главном методе из класса `classJavaConfigExampleTwo` заменить класс `AppConfigTwo` на класс `AppConfigFive`, то выполнение исходного кода из данного примера приведет к тому же самому результату.

```
package com.apress.prospring5.ch4.mixed;

import com.apress.prospring5.ch2.decoupled.MessageProvider;
import com.apress.prospring5.ch2.decoupled.MessageRenderer;
import com.apress.prospring5.ch2.decoupled
          .StandardOutMessageRenderer;
import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context
          .annotation.Configuration;
import org.springframework.context
          .annotation.ImportResource;

@Configuration
@ImportResource(value=
                "classpath:spring/app-context-xml-01.xml")
public class AppConfigFive {
  @Autowired
  MessageProvider provider;

  @Bean(name = "messageRenderer")
  public MessageRenderer messageRenderer() {
    MessageRenderer renderer =
           new StandardOutMessageRenderer();
    renderer.setMessageProvider(provider);
    return renderer;
  }
}
```

Можно поступить и наоборот, определив сначала компоненты Spring Beans в конфигурационных классах Java, а затем импортировав их в XML-файлах конфигурации. Так, в следующем примере компонент `messageRenderer` определяется в XML-файле конфигурации, а его зависимость (компонент `provider`) — в конфигурацион-

ном классе AppConfigSix. Ниже приведено содержимое XML-файла app-context-xml-02.xml.

```
<beans ...>

<context:annotation-config/>

<bean
  class="com.apress.prospring5.ch4.mixed.AppConfigSix"/>

<bean id="messageRenderer"
  class="com.apress.prospring5.ch2.decoupled
           .StandardOutMessageRenderer"
  p:messageProvider-ref="provider"/>
</beans>
```

В дескрипторе разметки <context:annotation-config/> должен быть объявлен тип конфигурационного класса, а также разрешена поддержка аннотированных методов. Это даст возможность настроить компоненты Spring Beans, объявленные в конфигурационного классе, как зависимости от компонентов, объявленных в XML-файле конфигурации. В данном случае конфигурационный класс AppConfigSix оказывается довольно простым:

```
package com.apress.prospring5.ch4.mixed;

import com.apress.prospring5.ch2.decoupled.MessageProvider;
import com.apress.prospring5.ch4.annotated
         .ConfigurableMessageProvider;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
         .Configuration;

@Configuration
public class AppConfigSix {

  @Bean
  public MessageProvider provider() {
    return new ConfigurableMessageProvider(
           "Love on the weekend");
  }
}
```

Экземпляр контекста типа ApplicationContext получается, как показано ниже, с помощью класса ClassPathXmlApplicationContext, который уже не раз применялся в приведенных до сих пор примерах кода.

```
package com.apress.prospring5.ch4;

import com.apress.prospring5.ch2.decoupled.MessageRenderer;
```

```
import com.apress.prospring5.ch4.mixed.AppConfigFive;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .ClassPathXmlApplicationContext;

public class JavaConfigExampleThree {
  public static void main(String... args) {
    ApplicationContext ctx =
              new ClassPathXmlApplicationContext
              ("classpath:spring/app-context-xml-02.xml");

    MessageRenderer renderer =
            ctx.getBean("messageRenderer",
                        MessageRenderer.class);

    renderer.render();
  }
}
```

Выполнение исходного кода из данного примера приведет к тому же самому результату.

На заметку Службы прикладной инфраструктуры могут быть также определены в конфигурационных классах Java. Например, в аннотации **`@EnableTransactionManagement`** можно указать, что в Spring следует использовать средство управления транзакциями, рассматриваемое в главе 9, а в аннотациях **`@EnableWebSecurity`** и **`@EnableGlobalMethodSecurity`** — активизировать контекст из проекта Spring Security, обсуждаемого в главе 16.

Выбор между конфигурированием на Java и в формате XML

Как вам должно быть уже известно, с помощью классов Java можно обеспечить тот же уровень конфигурирования контекста типа `ApplicationContext`, что и в формате XML. Так какой же стиль конфигурирования выбрать? Соображение по этому поводу очень похоже на то, которое принималось во внимание при выборе стиля конфигурирования внедрения зависимостей в формате XML или с помощью аннотаций Java. Очевидно, что каждому из этих стилей присущи свои достоинства и недостатки. Тем не менее рекомендация остается той же: решив в своей команде разработчиков, какой стиль конфигурирования применять, старайтесь придерживаться и соханять его неизменным, не разрываясь между классами Java и XML-файлами. Применение единого стиля конфигурирования существенно упростит задачу сопровождения прикладного кода.

Профили

Еще одну интересную возможность предоставляет в Spring понятие профилей конфигурации. В сущности, *профиль* вынуждает Spring конфигурировать только тот экземпляр контекста типа `ApplicationContext`, который был определен, когда указанный профиль стал активным. В этом разделе применение профилей будет продемонстривано на примере простой программы.

Пример применения профилей в Spring

Допустим, имеется служба `FoodProviderService`, которая отвечает за обеспечение едой учебных заведений, включая детский сад и среднюю школу. В интерфейсе `FoodProviderService` определен только один метод `provideLunchSet()`, который формирует комплексный обед для каждого учащегося вызываемого учебного заведения. Комплексный обед представляет собой список объектов класса `Food`. Это очень простой класс с единственным свойством `name`:

```
package com.apress.prospring5.ch4;

public class Food {
  private String name;

  public Food() {
  }

  public Food(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

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

Ниже приведен интерфейс `FoodProviderService`.

```
package com.apress.prospring5.ch4;

import java.util.List;

public interface FoodProviderService {
List<Food> provideLunchSet();
}
```

А теперь допустим, что имеются два поставщика комплексных обедов: один — для детского сада, другой — для средней школы. И хотя изготовляемые ими ком-

плексные обеды отличаются, предоставляемые услуги одинаковы — доставка комплексных обедов для учащихся. Итак, создадим две разные реализации интерфейса FoodProviderService, присвоив их классам одно и то же имя, но разместив их в разных пакетах для обозначения целевых учебных заведений. Ниже приведен исходный код этих двух классов.

```
//chapter04/profiles/src/main/java/com/apress/prospring5
//        /ch4/highschool/FoodProviderServiceImpl.java
package com.apress.prospring5.ch4.highschool;

import java.util.ArrayList;
import java.util.List;

import com.apress.prospring5.ch4.Food;
import com.apress.prospring5.ch4.FoodProviderService;

public class FoodProviderServiceImpl
       implements FoodProviderService {
  @Override
  public List<Food> provideLunchSet() {
    List<Food> lunchSet = new ArrayList<>();
    lunchSet.add(new Food("Coke"));
    lunchSet.add(new Food("Hamburger"));
    lunchSet.add(new Food("French Fries"));
    return lunchSet;
  }
}

//chapter04/profiles/src/main/java/com/apress/prospring5
//         /ch4/kindergarten/FoodProviderServiceImpl.java
package com.apress.prospring5.ch4.kindergarten;

import java.util.ArrayList;
import java.util.List;

import com.apress.prospring5.ch4.Food;
import com.apress.prospring5.ch4.FoodProviderService;

public class FoodProviderServiceImpl
       implements FoodProviderService {
  @Override
  public List<Food> provideLunchSet() {
    List<Food> lunchSet = new ArrayList<>();
    lunchSet.add(new Food("Milk"));
    lunchSet.add(new Food("Biscuits"));
    return lunchSet;
  }
}
```

Как видите, две реализации предоставляют один и тот же интерфейс `FoodProviderService`, но производят разные сочетания блюд в комплексных обедах. А теперь допустим, что в детском саду хотят, чтобы избранный поставщик доставлял комплексные обеды их воспитанникам. Поэтому выясним, как для достижения этой цели сконфигурировать соответствующий профиль в Spring. Рассмотрим сначала конфигурирование в формате XML. Создадим с этой целью два XML-файла конфигурации: один — для профиля детского сада, другой — для профиля средней школы. Ниже приведена конфигурация профилей двух поставщиков комплексных обедов сначала для средней школы, а затем для детского сада.

```
<!-- highschool-config.xml -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation=
          "http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans
               /spring-beans.xsd"
       profile="highschool">

  <bean id="foodProviderService"
     class="com.apress.prospring5.ch4.highschool
               .FoodProviderServiceImpl"/>
</beans>

<!-- kindergarten-config.xml -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org
                                /schema/beans
       http://www.springframework.org/schema/beans
            /spring-beans.xsd"
       profile="kindergarten">

  <bean id="foodProviderService"
     class="com.apress.prospring5.ch4.kindergarten
               .FoodProviderServiceImpl"/>
</beans>
```

Обратите внимание в конфигурации обоих профилей на использование атрибутов `profile="kindergarten"` и `profile="highschool"` в дескрипторе разметки `<beans>`. Этим Spring извещается, что экземпляры компонентов Spring Beans следует получать лишь в том случае, если активен указанный профиль. Поэтому выясним, как активизировать нужный профиль, когда контекст типа `ApplicationContext` из каркаса Spring применяется в автономном приложении. Для этой цели потребуется тестовая программа, исходный код которой приведен ниже.

```
package com.apress.prospring5.ch4;

import java.util.List;
import org.springframework.context.support
          .GenericXmlApplicationContext;

public class ProfileXmlConfigExample {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/*-config.xml");
    ctx.refresh();

    FoodProviderService foodProviderService =
        ctx.getBean("foodProviderService",
                    FoodProviderService.class);

    List<Food> lunchSet =
               foodProviderService.provideLunchSet();

    for (Food food: lunchSet) {
      System.out.println("Food: " + food.getName());
    }
    ctx.close();
  }
}
```

Метод `ctx.load()` загрузит оба файла конфигурации, `kindergarten-config.xml` и `highschoolconfig.xml`, поскольку ему передается метасимвол подстановки в виде префикса. В данном примере будут получены экземпляры только компонентов Spring Beans из файла конфигурации `kindergarten-config.xml`, исходя из значения, заданного в атрибуте `profile`, который активизируется путем передачи аргумента `-Dspring.profiles.active="kindergarten"` виртуальной машине JVM. Выполнение рассматриваемого здесь примера программы с этим аргументом даст следующий результат:

```
Food: Milk
Food: Biscuits
```

Именно такой ассортимент блюд и сформирует реализация поставщика комплексных обедов для детского сада. А теперь изменим упомянутый выше аргумент на `-Dspring.profiles.active="highschool"` для профиля средней школы. В итоге будет получен такой результат:

```
Food: Coke
Food: Hamburger
Food: French Fries
```

Используемый профиль можно также установить программно, вызвав метод `ctx.getEnvironment().setActiveProfiles("kindergarten")`. А средствами Java Config можно зарегистрировать классы, чтобы сделать их доступными через профили, снабдив каждый из них аннотацией `@Profile`.

Конфигурирование профилей Spring на языке Java

Безусловно, профили Spring можно сконфигурировать и средствами Java, чтобы удовлетворить потребности тех разработчиков, которых не устраивает конфигурирование в формате XML. И в этом случае XML-файлы, представленные в предыдущих разделах, придется заменить равнозначными конфигурационными классами Java. Ниже показано, каким образом определяется конфигурационный класс для профиля `kindergarten`.

```
package com.apress.prospring5.ch4.config;

import com.apress.prospring5.ch4.FoodProviderService;
import com.apress.prospring5.ch4.kindergarten
          .FoodProviderServiceImpl;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

@Configuration
@Profile("kindergarten")
public class KindergartenConfig {

  @Bean
  public FoodProviderService foodProviderService() {
    return new FoodProviderServiceImpl();
  }
}
```

Как видите, компонент `foodProviderService` определяется с помощью аннотации `@Bean`, а класс `KindergartenConfig` как специальный для конфигурирования профиля `kindergarten` — с помощью аннотации `@Profile`. Аналогичным образом определяется и специальный класс для конфигурирования профиля `highschool`, за исключением, конечно, типа компонента Spring Bean, имени профиля и класса:

```
package com.apress.prospring5.ch4.config;

import com.apress.prospring5.ch4.FoodProviderService;
import com.apress.prospring5.ch4.highschool
          .FoodProviderServiceImpl;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
import org.springframework.context.annotation.Profile;

@Configuration
@Profile("highschool")
public class HighschoolConfig {

  @Bean
  public FoodProviderService foodProviderService() {
    return new FoodProviderServiceImpl();
  }
}
```

Приведенные выше конфигурационные классы применяются таким же образом, как и XML-файлы конфигурации. Для них объявляется отдельный контекст, но лишь один из них будет фактически использован для получения экземпляра контекста типа `ApplicationContext` в зависимости от значения параметра **`-Dspring.profiles.active`** настройки виртуальной машины JVM.

```
package com.apress.prospring5.ch4.config;

import com.apress.prospring5.ch4.Food;
import com.apress.prospring5.ch4.FoodProviderService;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;

import java.util.List;

public class ProfileJavaConfigExample {

  public static void main(String... args) {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               KindergartenConfig.class,
               HighschoolConfig.class);

    FoodProviderService foodProviderService =
                ctx.getBean("foodProviderService",
                           FoodProviderService.class);

    List<Food> lunchSet =
                foodProviderService.provideLunchSet();
    for (Food food : lunchSet) {
      System.out.println("Food: " + food.getName());
    }
    ctx.close();
  }
}
```

Если выполнить исходный код из предыдущего примера, указав значение `kinder garten` параметра **`-Dspring.profiles.active`** виртуальной машины JVM, то на консоль будет выведен следующий вполне ожидаемый результат:

```
Food: Milk
Food: Biscuits
```

Вместо параметра **`-Dspring.profiles.active`** виртуальной машины JVM можно также воспользоваться специальной аннотацией `@ActiveProfiles` для конфигурирования профилей, хотя это можно сделать только в тестовых классах. Подробнее об этом речь пойдет в главе 13, посвященной тестированию приложений Spring, а ниже приведен простой пример применения такой аннотации в исходном коде.

```
package com.apress.prospring5.ch4.config;

import com.apress.prospring5.ch4.FoodProviderService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test
          .context.ContextConfiguration;
import org.springframework.test.context.junit4
      .SpringJUnit4ClassRunner;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes={KindergartenConfig.class,
                      HighschoolConfig.class})
@ActiveProfiles("kindergarten")
public class ProfilesJavaConfigTest {

  @Autowired FoodProviderService foodProviderService;

  @Test
  public void testProvider() {
    assertTrue(
          foodProviderService.provideLunchSet() != null);
    assertFalse(
          foodProviderService.provideLunchSet().isEmpty());

    assertEquals(2,
          foodProviderService.provideLunchSet().size());
  }
}
```

Как видите, в аннотации `@ActiveProfiles("kindergarten")` указывается конкретный профиль, применяемый для выполнения приведенного выше теста. В сложных приложениях обычно применяется не один профиль для конфигурирования контекста, в котором должен выполняться тест. Приведенный выше тестовый класс может быть выполнен в любом логически развитом редакторе исходного кода Java, а также автоматически по команде сборки **`gradle clean build`**.

О применении профилей

Профили в Spring предоставляют разработчикам приложений еще один способ конфигурирования запуска приложений на выполнение, который обычно реализуется в инструментальных средствах сборки (например, в Maven, где имеется специальная поддержка профилей). Инструментальные средства сборки упаковывают нужные файлы конфигурации/свойств в архив Java (формата JAR или WAR в зависимости от типа приложения), исходя из переданных им аргументов, а затем развертывают его в целевой среде. Разработчики приложений могут самостоятельно определять профили и активизировать их программно или с помощью специального параметра виртуальной машины JVM. Благодаря поддержке профилей в Spring можно иметь один архив приложения и развертывать его в самых разных средах, передавая подходящий профиль в качестве аргумента во время начальной загрузки JVM. Например, приложения могут быть с разными профилями, в том числе (`dev`, `hibernate`), (`prd`, `jdbc`) и т.д., и каждая их комбинация представляет среду выполнения (среду разработки или эксплуатации) и применяемую библиотеку доступа к данным (Hibernate или JDBC). Таким образом, управление профилями приложения переносится на сторону программирования.

Но у такого подхода имеются и свои недостатки. Некоторые могут возразить, что размещение всей конфигурации приложения для разных сред в файлах или классах Java и последующая их сборка чревата ошибками по небрежности или недосмотру (например, администратор может не установить нужный параметр виртуальной машины JVM в среде сервера приложений). А упаковка файлов для всех профилей приводит к увеличению размера пакета сверх обычного. Еще раз напомним, что выбор способа конфигурирования, наиболее подходящего для проекта, следует делать исходя из требований к приложению и его конфигурации.

Абстракция через интерфейсы `Environment` и `PropertySource`

Для установки активного профиля необходимо обратиться к интерфейсу `Environment`, обеспечивающему уровень абстракции для инкапсуляции среды выполняемого приложения Spring.

Помимо профилей, в интерфейсе `Environment` инкапсулируются свойства как главные элементы информации. Свойства служат для сохранения базовой конфигура-

ции среды выполняемого приложения, включая местоположение папки приложения, параметры подключения к базе данных и т.д.

Возможности абстракции через интерфейсы `Environment` и `PropertySource` в Spring содействуют разработчикам в доступе к разнообразной конфигурационной информации, связанной с платформой запуска приложений на выполнение. При такой абстракции все свойства системы, переменные окружения и свойства приложения обслуживаются интерфейсом `Environment`, который наполняется в Spring при начальной загрузке интерфейса `ApplicationContext`. В следующем фрагменте кода демонстрируется простой тому пример:

```
package com.apress.prospring5.ch4;

import java.util.HashMap;
import java.util.Map;

import org.springframework.context.support
          .GenericXmlApplicationContext;
import org.springframework.core.env
          .ConfigurableEnvironment;

import org.springframework.core.env.MapPropertySource;
import org.springframework.core.env.MutablePropertySources;

public class EnvironmentSampleLast {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.refresh();

    ConfigurableEnvironment env = ctx.getEnvironment();
    MutablePropertySources propertySources =
           env.getPropertySources();

    Map<String,Object> appMap = new HashMap<>();
    appMap.put("user.home", "application_home");

    propertySources.addLast(new MapPropertySource(
                              "prospring5_MAP", appMap));

    System.out.println("user.home: "
                     + System.getProperty("user.home"));
    System.out.println("JAVA_HOME: "
                     + System.getenv("JAVA_HOME"));
    System.out.println("user.home: "
                     + env.getProperty("user.home"));
    System.out.println("JAVA_HOME: "
                     + env.getProperty("JAVA_HOME"));
```

```
    ctx.close();
    System.out.println("application.home: "
              + env.getProperty("application.home"));
  }
}
```

Как следует из приведенного выше фрагмента кода, после инициализации контекста типа `ApplicationContext` получается ссылка на интерфейс `Configurable Environment`. Через этот интерфейс получается ссылка на класс `MutableProperty Sources` (стандартную реализацию интерфейса `PropertySources`, которая позволяет манипулировать содержащимися источниками свойств). Затем строится отображение, в котором размещаются свойства приложения и получается экземпляр класса `MapPropertySource`, производного от класса `PropertySource` и предназначенного для чтения ключей и значений из экземпляра данного отображения. И, наконец, новый объект класса `MapPropertySource` добавляется к источникам свойств типа `MutablePropertySources` с помощью метода `addLast()`. Если выполнить исходный код из данного примера, то в конечном итоге будет получен следующий результат:

```
user.home: /home/jules
JAVA_HOME: /home/jules/bin/java
user.home: /home/jules
JAVA_HOME: /home/jules/bin/java
application.home: application_home
```

В первых двух строках приведенного выше результата свойство `user.home` и переменная окружения `JAVA_HOME` извлекаются, как и прежде, средствами класса `System` из системы JVM. Но, как следует из трех последних строк, доступ ко всем свойствам системы, переменным окружения и свойствам приложения может быть осуществлен через интерфейс `Environment`. Как видите, абстракция через интерфейс `Environment` может оказать помощь в управлении и доступе к различным свойствам среды выполнения конкретного приложения.

Благодаря абстракции через интерфейс `PropertySource` в каркасе Spring можно обратиться к свойствам в следующем стандартном порядке.

- Свойства системы для выполнения виртуальной машины JVM.
- Переменные окружения.
- Свойства, определяемые в приложении.

Попробуйте для примера определить то же самое свойство `user.home` приложения и ввести его в интерфейс `Environment` через класс `MutablePropertySources`. Выполнив исходный код данного примера, вы обнаружите, что свойство `user.home` по-прежнему извлекается из свойств виртуальной машины JVM вместо определенного вами. Но в Spring можно установить порядок, в котором свойства извлекаются через интерфейс `Environment`, как демонстрируется в следующем переделанном варианте исходного кода из предыдущего примера:

```
package com.apress.prospring5.ch4;

import java.util.HashMap;
import java.util.Map;

import org.springframework.context.support
          .GenericXmlApplicationContext;
import org.springframework.core
          .env.ConfigurableEnvironment;
import org.springframework.core.env.MapPropertySource;
import org.springframework.core.env.MutablePropertySources;

public class EnvironmentSampleFirst {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.refresh();

    ConfigurableEnvironment env = ctx.getEnvironment();
    MutablePropertySources propertySources =
                           env.getPropertySources();

    Map<String,Object> appMap = new HashMap<>();
    appMap.put("user.home", "application_home");

    propertySources.addFirst(new MapPropertySource(
                             "prospring5_MAP", appMap));

    System.out.println("user.home: "
                     + System.getProperty("user.home"));
    System.out.println("JAVA_HOME: "
                     + System.getenv("JAVA_HOME"));

    System.out.println("user.home: "
                     + env.getProperty("user.home"));
    System.out.println("JAVA_HOME: "
                     + env.getProperty("JAVA_HOME"));

    ctx.close();
  }
}
```

В приведенном выше фрагменте кода было также определено свойство приложения `user.home`, которое было затем введено первым для поиска с помощью метода `addFirst()` из класса `MutablePropertySources`. Если выполнить исходный код из данного примера, то в конечном итоге будет получен следующий результат:

```
user.home: /home/jules
JAVA_HOME: /home/jules/bin/java
```

```
user.home: application_home
JAVA_HOME: /home/jules/bin/java
```

Первые две строки в приведенном выше результате остались прежними, поскольку для извлечения их значений по-прежнему вызываются методы `getProperty()` и `getenv()` класса `System` из системы JVM. Но если воспользоваться интерфейсом `Environment`, то специально определенное свойство `user.home` получает преимущество, поскольку оно определено первым для поиска значений свойств.

В реальных приложениях необходимость непосредственного взаимодействия с интерфейсом `Environment` возникает редко. Чаще всего вместо него применяется заполнитель свойства в форме `${}` (например, `${application.home}`), а разрешенное его значение внедряется в компоненты Spring Beans. Покажем это на конкретном примере. Допустим, имеется класс `AppProperty`, предназначенный для хранения всех свойств приложения, загружаемых из файла свойств. Этот класс определяется следующим образом:

```
package com.apress.prospring5.ch4;

public class AppProperty {
  private String applicationHome;
  private String userHome;

  public String getApplicationHome() {
    return applicationHome;
  }

  public void setApplicationHome(String applicationHome) {
    this.applicationHome = applicationHome;
  }

  public String getUserHome() {
    return userHome;
  }

  public void setUserHome(String userHome) {
    this.userHome = userHome;
  }
}
```

Ниже приведено содержимое файла `application.properties` свойств данного приложения.

```
application.home=application_home
user.home=/home/jules-new
```

Обратите внимание на то, что в этом файле свойств объявляется также свойство `user.home`. А теперь перейдем к конфигурированию данного приложения в формате

XML, как показано в следующем фрагменте кода из файла конфигурации app-context-xml.xml:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi=
            "http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context=
            "http://www.springframework.org/schema/context"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation=
          "http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans
                 /spring-beans.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context
                /spring-context.xsd">

  <context:property-placeholder
      location="classpath:application.properties"/>

  <bean id="appProperty"
     class="com.apress.prospring5.ch4.AppProperty"
     p:applicationHome="${application.home}"
     p:userHome="${user.home}"/>
</beans>
```

Для загрузки свойств в интерфейс Environment, заключенный в оболочку интерфейса ApplicationContext, здесь применяется дескриптор <context:property-placeholder>, а для внедрения значений в компонент Spring Bean типа App Property — заполнители SpEL. В следующем фрагменте кода приведен исходный код тестовой программы:

```
package com.apress.prospring5.ch4;

import org.springframework.context.support
          .GenericXmlApplicationContext;

public class PlaceHolderDemo {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();

    AppProperty appProperty = ctx.getBean("appProperty",
                              AppProperty.class);

    System.out.println("application.home: "
                    + appProperty.getApplicationHome());
```

```
    System.out.println("user.home: "
                       + appProperty.getUserHome());

    ctx.close();
  }
}
```

Если выполнить исходный код тестовой программы из данного примера, то будет получен следующий результат:

```
application.home: application_home
user.home: /Users/jules
```

Как видите, заполнитель свойства `application.home` был соответствующим образом разрешен, в то время как свойство `user.home` по-прежнему извлекалось из свойств JVM, что вполне допустимо, поскольку таково стандартное поведение абстракции через интерфейс `PropertySource`. Чтобы дать каркасу Spring команду назначить более высокий приоритет для значений из файла свойств `application.properties`, в дескриптор разметки `<context:property-placeholder>` введен атрибут `local-override="true"`:

```
<context:property-placeholder local-override="true"
    location="classpath:env/application.properties"/>
```

Атрибут `local-override` сообщает каркасу Spring о необходимости переопределить существующие свойства с помощью свойств, определенных в данном заполнителе. Выполнив тестовую программу из данного примера, можно обнаружить, что свойство `user.home` теперь извлекается из файла `application.properties`:

```
application.home: application_home
user.home: /home/jules-new
```

Конфигурирование с помощью аннотаций JSR-330

Как пояснялось в главе 1, на платформе JEE 6 предоставляется поддержка спецификации JSR-330 (Dependency Injection for Java — Внедрение зависимостей для Java), в которой определяется коллекция аннотаций, с помощью которых можно конфигурировать внедрение зависимостей приложения в контейнере JEE 6 или другом совместимом контейнере инверсии управления. Эти аннотации поддерживаются и распознаются и в каркасе Spring, поэтому их можно применять в приложениях Spring, даже если последние и не предназначены для выполнения в контейнере JEE 6. Аннотации JSR-330 помогают упростить перенос приложения из Spring в контейнер JEE 6 или другой совместимый контейнер инверсии управления (например, Google Guice).

Обратимся снова к примеру поставщика и средства воспроизведения сообщений, чтобы реализовать их с помощью аннотаций JSR-330. Для поддержки аннотаций JSR-330 в текущий проект придется добавить зависимость от пакета `javax.inject`.[7]

В следующем фрагменте кода демонстрируется интерфейс `MessageProvider` и его реализация в классе `ConfigurableMessageProvider`:

```
// chapter04/jsr330/src/main/java/com/apress/prospring5
            /ch4/MessageProvider.java
package com.apress.prospring5.ch4;

public interface MessageProvider {
  String getMessage();
}

// chapter04/jsr330/src/main/java/com/apress/prospring5
            /ch4/ConfigurableMessageProvider.java
package com.apress.prospring5.ch4;

import javax.inject.Inject;
import javax.inject.Named;

@Named("messageProvider")
public class ConfigurableMessageProvider
       implements MessageProvider {
  private String message = "Default message";
  public ConfigurableMessageProvider() {
  }

  @Inject
  @Named("message")
  public ConfigurableMessageProvider(String message) {
    this.message = message;
  }

  public void setMessage(String message) {
    this.message = message;
  }

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

[7] Характеристики этой зависимости (например, идентификаторы групп и последние версии) можно найти в открытом информационном хранилище Maven, и в частности, на специально выделенной для пакета `java.inject` странице, доступной по адресу `https://mvnrepository.com/artifact/javax.inject/javax.inject`.

Как видите, все аннотации относятся к пакету `javax.inject`, в котором реализован стандарт JSR-330. В классе `ConfigurableMessageProvider` аннотация `@Named` применяется в двух местах. Во-первых, она применяется для объявления внедряемого компонента Spring Bean (аналогично аннотации `@Component` или другим стереотипным аннотациям в Spring). В частности, аннотация `@Named("messageProvider")` указывает на то, что класс `ConfigurableMessageProvider` является внедряемым компонентом Spring Bean, которому присваивается имя `messageProvider`, т.е. эта аннотация делает то же самое, что и атрибут `name` в дескрипторе `<bean>` разметки компонентов Spring Beans. И во-вторых, аннотация `@Inject` служит для внедрения зависимостей через конструктор, и поэтому она размещается перед конструктором, принимающим строковое значение. В этом случае аннотация `@Named` указывает на необходимость внедрить значение, которому присвоено имя `message`. А теперь рассмотрим интерфейс `MessageRenderer` и его реализацию в классе `StandardOutMessageRenderer`:

```
//chapter04/jsr330/src/main/java/com/apress/prospring5
          /ch4/MessageRenderer.java
package com.apress.prospring5.ch4;

public interface MessageRenderer {
  void render();
  void setMessageProvider(MessageProvider provider);
  MessageProvider getMessageProvider();
}

//chapter04/jsr330/src/main/java/com/apress/prospring5
           /ch4/StandardOutMessageRenderer.java
package com.apress.prospring5.ch4;

import javax.inject.Inject;
import javax.inject.Named;
import javax.inject.Singleton;

@Named("messageRenderer")
@Singleton
public class StandardOutMessageRenderer
       implements MessageRenderer {
  @Inject
  @Named("messageProvider")
  private MessageProvider messageProvider = null;

  public void render() {
    if (messageProvider == null) {
      throw new RuntimeException("You must set the "
           + "property messageProvider of class:"
           + StandardOutMessageRenderer.class.getName());
    }
```

```
    System.out.println(messageProvider.getMessage());
  }

  public void setMessageProvider(
              MessageProvider provider) {
    this.messageProvider = provider;
  }

  public MessageProvider getMessageProvider() {
    return this.messageProvider;
  }
}
```

В приведенном выше фрагменте кода аннотация `@Named` служит для определения внедряемого компонента Spring Bean. Обратите внимание на аннотацию `@Singleton`. Стоит отметить, что в стандарте JSR-330 по умолчанию компонент Spring Bean является неодиночным, что похоже на область видимости на уровне прототипа в Spring. Но если требуется, чтобы в среде по стандарту JSR-330 компонент Spring Bean был одиночным, то следует применить аннотацию `@Singleton`. Впрочем, применение этой аннотации в Spring не дает никакого эффекта, потому что стандартной для получения экземпляров компонентов Spring Beans является область видимости одиночного компонента. В данном примере аннотация `@Singleton` служит лишь для демонстрационных целей, чтобы подчеркнуть отличие Spring от других контейнеров, совместимых со стандартом JSR-330.

На этот раз в свойстве `messageProvider` применяется аннотация `@Inject` для внедрения зависимостей через метод установки и при этом указывается, что для такого внедрения должен быть выбран компонент `messageProvider`. В следующем фрагменте кода из файла `app-context-annotation.xml` определяется простая конфигурация приложения Spring в формате XML:

```
<beans ...>

  <context:component-scan
     base-package="com.apress.prospring5.ch4"/>

  <bean id="message" class="java.lang.String">
    <constructor-arg value=
                    "Gravity is working against me"/>
  </bean>
</beans>
```

Для применения аннотаций JSR-330 не требуются какие-то особые дескрипторы — достаточно сконфигурировать свое приложение подобно обычному приложению Spring. Дескриптор `<context:component-scan>` вынуждает Spring искать аннотации, относящиеся к внедрению зависимостей, и тогда Spring распознает заданные аннотации JSR-330. Кроме того, в приведенной выше конфигурации объявлен

компонент Spring Bean под названием `message` для внедрения через конструктор в класс `ConfigurableMessageProvider`. Ниже приведен исходный код тестовой программы.

```
package com.apress.prospring5.ch4;

import org.springframework.context.support
          .GenericXmlApplicationContext;

public class Jsr330Demo {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-annotation.xml");
    ctx.refresh();

    MessageRenderer renderer = ctx.getBean("messageRenderer",
                               MessageRenderer.class);
    renderer.render();

    ctx.close();
  }
}
```

Если выполнить исходный код тестовой программы из данного примера, то будет получен следующий результат:

```
Gravity is working against me
```

Благодаря аннотациям JSR-330 можно упростить переход к другим контейнерам инверсии управления, совместимыи со стандартом JSR-330, в том числе к серверам приложений, совместимым с платформой JEE 6, или контейнерам внедрения зависимостей вроде Google Guice. Тем не менее аннотации Spring обладают намного бо́льшими возможностями и гибкостью, чем аннотации JSR-330. Ниже перечислены их основные отличия.

- При использовании аннотации `@Autowired` из Spring можно задавать атрибут `required`, указывающий на то, что внедрение зависимостей должно быть выполнено (для объявления этого требования можно также применять аннотацию `@Required` из Spring), тогда как для аннотации `@Inject` из JSR-330 эквивалент отсутствует. Более того, в Spring предоставляется аннотация `@Qualifier`, обеспечивающая более точное управление автосвязыванием зависимостей на основе имени описателя.
- В спецификации JSR-330 поддерживается только область видимости экземпляров одиночных и неодиночных компонентов Spring Beans, тогда как в Spring — большее разнообразие областей видимости, что очень полезно для веб-приложений.

- В каркасе Spring допускается применять аннотацию `@Lazy`, чтобы принудить его получать экземпляры компонентов Spring Beans только по запросам из приложений. А в спецификации JSR-330 аналогичная возможность отсутствует.

Аннотации Spring и JSR-330 можно также сочетать в одном приложении. Тем не менее рекомендуется выбрать какой-то один вид аннотаций, чтобы придерживаться согласованного стиля конфигурирования приложений. Один из возможных подходов предусматривает применение аннотаций JSR-330 как можно больше, тогда как аннотаций Spring по мере необходимости. Но такой подход принесет лишь незначительные выгоды, поскольку придется выполнить довольно большой объем работы при переходе к другому контейнеру внедрения зависимостей. И в заключение рекомендуется отдавать предпочтение аннотациям Spring над аннотациями JSR-330, поскольку аннотации Spring оказываются намного более эффективными, если только не требуется, чтобы приложение было независимым от контейнеров инверсии управления.

Конфигурирование средствами Groovy

В версии Spring Framework 4.0 появилась возможность конфигурировать определения компонентов Spring Beans и контекст типа `ApplicationContext` средствами языка Groovy. Это дает разработчикам еще одну возможность для замены или дополнения конфигурации компонентов Spring Beans в формате XML и/или с помощью аннотаций. Контекст типа `ApplicationContext` может быть создан непосредственно в сценарии Groovy или загружен из кода Java, причем в обоих случаях через класс `GenericGroovyApplicationContext`. Рассмотрим сначала порядок определения компонентов Spring Beans из внешнего сценария Groovy и последующей их загрузки в коде Java. В предыдущих разделах и главах упоминались различные классы компонентов Spring Beans, поэтому для содействия в какой-то мере повторному использованию кода в рассматриваемом здесь примере будет использован класс `Singer`, представленный в главе 3. Ниже приведено определение этого класса.

```
package com.apress.prospring5.ch3.xml;

public class Singer {
  private String name;
  private int age;

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

  public void setAge(int age) {
    this.age = age;
  }

  public String toString() {
```

```
    return "\tName: " + name + "\n\t" + "Age: " + age;
  }
}
```

Как видите, это всего лишь класс Java с парой свойств, описывающих певца. Воспользуемся этим простым классом Java, чтобы продемонстрировать, что для конфигурирования компонентов Spring Beans средствами Groovy совсем не обязательно переписывать всю кодовую базу на языке Groovy. Более того, классы Java можно импортировать из зависимостей и применять в сценариях Groovy. А теперь перейдем к созданию сценария Groovy (в файле `beans.groovy`), предназначенного для определения компонента Spring Bean, как показано ниже.

```
package com.apress.prospring5.ch4

import com.apress.prospring5.ch3.xml.Singer

beans {
  singer(Singer, name: 'John Mayer', age: 39)
}
```

Сценарий Groovy начинается с замыкания верхнего уровня по имени `beans`, которое предоставляет определения компонентов Spring Beans. Прежде всего здесь указывается имя компонента Spring Bean (`singer`), затем в качестве аргументов передается тип класса (`Singer`), после которого следуют имена свойств и значения для установки этих свойств. Создадим простую тестовую программу на Java, загружающую определения компонентов Spring Beans из сценария Groovy, как показано ниже.

```
package com.apress.prospring5.ch4;

import com.apress.prospring5.ch3.xml.Singer;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support
          .GenericGroovyApplicationContext;

public class GroovyBeansFromJava {
  public static void main(String... args) {
    ApplicationContext context =
              new GenericGroovyApplicationContext(
                  "classpath:beans.groovy");
    Singer singer = context.getBean("singer", Singer.class);
    System.out.println(singer);
  }
}
```

Как видите, создание контекста типа `ApplicationContext` осуществляется обычным способом, но это делается благодаря применению класса `GenericGroovy ApplicationContext` и предоставлению написанного ранее сценария Groovy, в котором составляется определение компонентов Spring Beans.

Прежде чем выполнить примеры исходного кода, приведенные в этом разделе, необходимо внедрить в проект `groovy-config-java` библиотеку `groovy-all` в качестве зависимости. Ниже приведено содержимое файла конфигурации `build.gradle` для данного проекта.

```
apply plugin: 'groovy'

dependencies {
    compile misc.groovy
    compile project(':chapter03:bean-inheritance')
}
```

В строке кода `compile project(':chapter03:bean-inheritance')` указывается, что проект `beaninheritance` из главы 3 необходимо скомпилировать и внедрить в качестве зависимости в данный проект. Этот проект содержит класс `Singer`. В файлах конфигурации Gradle употребляется синтаксис Groovy, а в файле `misc.groovy` делается ссылка на свойство `groovy` из массива `misc`, определенного в файле `build.gradle` родительского проекта. Ниже приведен фрагмент содержимого данного файла, где определяется конфигурация, относящаяся непосредственно к Groovy.

```
ext {
      springVersion = '5.0.0.M4'
      groovyVersion = '2.4.5'
...
    misc = [
            ...
                groovy: "org.codehaus.groovy:
                        groovy-all:$groovyVersion"
    ]
...
}
```

Выполнение исходного кода из класса `GroovyBeansFromJava` даст следующий результат:

```
Name: John Mayer
Age: 39
```

А теперь, когда было показано, как загружать определения компонентов Spring Beans из кода Java через внешний сценарий Groovy, возникает вопрос: каким образом создается контекст типа `ApplicationContext` и определения компонентов Spring Beans только в сценарии Groovy? Рассмотрим с этой целью приведенный ниже код Groovy из файла `GroovyConfig.groovy`.

```
package com.apress.prospring5.ch4

import com.apress.prospring5.ch3.xml.Singer
import org.springframework.context.support
```

```
          .GenericApplicationContext
import org.springframework.beans.factory.groovy
          .GroovyBeanDefinitionReader

def ctx = new GenericApplicationContext()
def reader = new GroovyBeanDefinitionReader(ctx)

reader.beans {
  singer(Singer, name: 'John Mayer', age: 39)
}

ctx.refresh()

println ctx.getBean("singer")
```

Выполнение этого кода приведет к такому же результату. На этот раз получается экземпляр типичного контекста типа `GenericApplicationContext`, но применяется класс `GroovyBeanDefinitionReader`, который будет использован для передачи определений компонентов Spring Beans. Затем, как и в предыдущем примере, создается компонент Spring Bean из простого объекта POJO, обновляется контекст типа `ApplicationContext` и выводится строковое представление компонента Spring Bean типа `Singer`. Как говорится, проще не бывает!

Очевидно, что мы лишь слегка коснулись того, что можно делать благодаря поддержке Groovy в Spring. Имея в своем распоряжении весь потенциал языка Groovy, при создании определений компонентов Spring Beans можно сделать много интересного. А благодаря полному доступу к контексту типа `ApplicationContext` можно не только конфигурировать компоненты Spring Beans, но и работать с поддерживаемыми профилями, файлами свойств и т.д. Не следует, однако, забывать, что большой потенциал влечет за собой и немалую ответственность.

Модуль Spring Boot

Итак, мы рассмотрели не один способ конфигурирования приложений Spring. Теперь вы должны ясно представлять, как это делается, будь то в формате XML, с помощью аннотаций, конфигурационных файлов Java, сценариев Groovy или всех этих способов вместе взятых. Но оказывается, что имеется еще более совершенный способ конфигурирования приложений Spring.

Проект Spring Boot и одноименный модуль предназначен для того, чтобы упростить приобретение начального опыта построения приложений с помощью каркаса Spring. В частности, модуль Spring Boot избавляет от ручного сбора зависимостей по наитию и предоставляет самые необходимые средства для приложений, включая измерение количественных показателей и контроль рабочего состояния.

В модуле Spring Boot принят "компетентный" подход к достижению цели упростить задачу разработчиков, предоставив им начальные проекты разнотипных прило-

жений, которые уже содержат надлежащие зависимости и версии, чтобы сэкономить время на первоначальной стадии разработки. А от тех, кто стремится полностью отойти от конфигурирования приложений в формате XML, модуль Spring Boot совсем не требует размечать конфигурацию в XML.

В рассматриваемом здесь примере будет продемонстрировано, насколько просто создать с помощью Spring Boot веб-приложение с традиционным приветствием "Hello World!" и некоторой изюминкой. Вас приятно удивит минимальный объем требующегося для этого кода по сравнению типичной установкой веб-приложений на Java. Как правило, подобные примеры приходится начинать с определения тех зависимостей, которые требуется внедрить в проект. Отчасти принятая в Spring Boot модель упрощения разработки приложений нацелена на автоматическую подготовку всех требующихся зависимостей. И если разработчик пользуется инструментальными средствами Maven, то может употребить родительскую объектную модель проекта (POM), чтобы получить в свое распоряжение подобные функциональные возможности. А если он пользуется инструментальными средствами Gradle, то дело обстоит еще проще, поскольку для этого не потребуется родительская модель POM, кроме начальной зависимости и модуля, подключаемого к Gradle. В рассматриваемом здесь примере будет создано приложение Spring, перечисляющее сначала все компоненты Spring Beans в текущем контексте, а затем получающее доступ к компоненту `hello World`. Ниже приведена конфигурация проекта `boot-simple` для построения в Gradle.

```
buildscript {
  repositories {
    mavenLocal()
    mavenCentral()
    maven { url "http://repo.spring.io/release" }
    maven { url "http://repo.spring.io/milestone" }
    maven { url "http://repo.spring.io/snapshot" }
    maven { url "https://repo.spring.io/libs-snapshot" }
  }
  dependencies {
    classpath boot.springBootPlugin
  }
}

apply plugin: 'org.springframework.boot'

dependencies {
  compile boot.starter
}
```

В строке кода `boot.springBootPlugin` делается ссылка на свойство `spring BootPlugin` из массива `boot`, определенного в файле `build.gradle` родительского проекта. Ниже приведен фрагмент содержимого данного файла, где определяется конфигурация, относящаяся непосредственно к модулю Spring Boot.

```
ext {
  bootVersion = '2.0.0.BUILD-SNAPSHOT'
  ...
  boot = [
    springBootPlugin:
          "org.springframework.boot:
               spring-boot-gradle-plugin:
               $bootVersion", starter :
          "org.springframework.boot:
               spring-boot-starter:
               $bootVersion", starterWeb :
          "org.springframework.boot:
               spring-boot-starter-web:$bootVersion"
  ]
  ...
}
```

На момент написания данной книги версия 2.0.0 модуля Spring Boot еще не была выпущена. Именно поэтому выше было указано значение `2.0.0.BUILD-SNAPSHOT` в переменной `bootVersion`, а в конфигурацию придется ввести гиперссылку `https://repo.spring.io/libs-snapshot` на информационное хранилище Spring Snapshot. Вполне возможно, что официальная версия Spring Boot появится после выхода данной книги в свет[8].

С каждым выпуском модуля Spring Boot предоставляется рекомендованный список поддерживаемых зависимостей. По этому списку выбираются версии требующихся библиотек для идеального соответствия прикладного интерфейса API, что делается в Spring Boot автоматически, а следовательно, необходимость в ручном конфигурировании версий зависимостей отпадает. При переходе к новой версии Spring Boot гарантируется также автоматическое обновление этих зависимостей. В этом случае предыдущая конфигурация проекта дополняется рядом зависимостей, чтобы обеспечить совместимость их версий с прикладным интерфейсом API. В таком логически развитом редакторе исходного кода, как IDE IntelliJ IDEA, имеется представление проектов Gradle (Gradle Projects), где можно расширить каждый модуль и осмотреть имеющиеся задачи и зависимости (рис. 4.3).

Итак, произведя установку приложения из рассматриваемого здесь примера, перейдем к созданию классов. Ниже приведено определение класса `HelloWorld`.

```
package com.apress.prospring5.ch4;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

@Component
```

[8] Версия Spring Boot 2.0 была официально выпущена в самом начале марта 2018 года. — *Примеч. ред.*

```
public class HelloWorld {

  private static Logger logger =
          LoggerFactory.getLogger(HelloWorld.class);

  public void sayHi() {
    logger.info("Hello World!");
  }
}
```

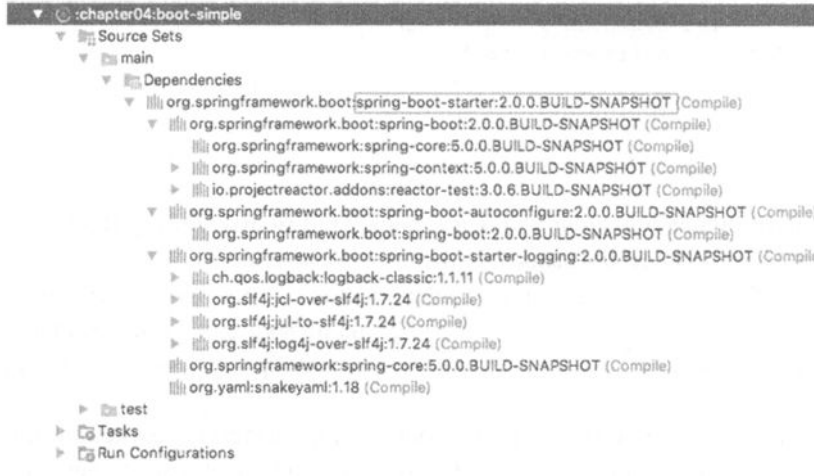


Рис. 4.3. Представление Gradle Projects проекта **`boot-simple`** в IDE IntelliJ IDEA

В этом классе нет ничего особенного и сложного. Он просто содержит метод и аннотацию, объявляющую его как компонент Spring Bean. А теперь покажем, как построить приложение Spring с помощью модуля Spring Boot и создать контекст типа `ApplicationContext`, содержащий данный компонент Spring Bean.

```
package com.apress.prospring5.ch4;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure
          .SpringBootApplication;
import org.springframework.context
          .ConfigurableApplicationContext;

import java.util.Arrays;

@SpringBootApplication
public class Application {
```

```
  private static Logger logger =
          LoggerFactory.getLogger(Application.class);

  public static void main(String args) throws Exception {
    ConfigurableApplicationContext ctx =
            SpringApplication.run(Application.class, args);
    assert (ctx != null);
    logger.info("The beans you were looking for:");

    // перечислить имена всех определяемых
    // компонентов Spring Beans
    Arrays.stream(ctx.getBeanDefinitionNames())
          .forEach(logger::info);

    HelloWorld hw = ctx.getBean(HelloWorld.class);
    hw.sayHi();

    System.in.read();
    ctx.close();
  }
}
```

Вот, собственно, и все. Этот класс мог бы, конечно, быть и более компактным, но в данном примере нам нужно было продемонстрировать, как добиться чего-то большего и, в частности, следующего.

- **Проверить наличие контекста.** Воспользоваться оператором `assert`, чтобы проверить допущение, что значение переменной `ctx` не равно `null`.
- **Установить режим протоколирования.** В состав Spring Boot входит целый ряд библиотек протоколирования, и чтобы воспользоваться ими, достаточно разместить одну из них в каталоге `resources` конкретной конфигурации. В данном случае выбрана библиотека `logback`.
- **Перечислить определения всех компонентов Spring Beans в данном контексте.** Используя лямбда-выражения, внедренные в версии Java 8, определения всех компонентов Spring Beans в текущем контексте можно перечислить в одной строке кода. С этой целью была введена строка кода, где перечисляются все компоненты Spring Beans, автоматически сконфигурированные в Spring Boot. Среди них можно обнаружить и компонент `helloWorld`.
- **Подтвердить выход из приложения.** Если не сделать вызов `System.in.read()`, то после вывода на консоль имен компонентов Spring Beans и содержимого объекта типа `HelloWorld` произойдет автоматический выход из данного приложения. Этот вызов был введен для того, чтобы принудить данное приложение ожидать до тех пор, пока не будет нажата любая клавиша, а затем завершить свое выполнение.

Новшеством здесь является наличие аннотации `@SpringBootApplication`. Это аннотация верхнего уровня, предназначенная для применения только на уровне класса. Она удобна тем, что равнозначна объявлению следующих аннотаций.

- **`@Configuration`.** Помечает данный класс как конфигурационный, в котором можно объявить компоненты Spring Beans с помощью аннотации `@Bean`.
- **`@EnableAutoConfiguration`.** Характерна только для Spring Boot. Она входит в состав пакета `org.springframework.boot.autoconfigure` и способна активизировать контекст типа `ApplicationContext` в Spring, пытаясь предугадать и сконфигурировать компоненты Spring Beans, которые, вероятнее всего, должны основываться на указанных зависимостях.
- Аннотация `@EnableAutoConfiguration` вполне справляется с начальными зависимостями, предоставляемыми в Spring, хотя и не привязана к ним непосредственно, что дает возможность пользоваться другими зависимостями, помимо начальных. Так, если по пути к классам указан конкретный встраиваемый сервер, он будет использован, при условии, что в конфигурации проекта не указан другой фабричный класс `EmbeddedServletContainerFactory`.
- **`@ComponentScan`.** Классы, снабженные стереотипными аннотациями, могут быть объявлены таким образом, чтобы стать определенного рода компонентами Spring Beans. Для перечисления пакетов, которые должны быть просмотрены с помощью аннотации `@SpringBootApplication`, служит атрибут `basePackages`. Другой атрибут, `basePackageClasses`, был внедрен в версии Spring 1.3.0 для просмотра пакетов. Этот атрибут служит типизированной альтернативой атрибуту `basePackages` для указания тех пакетов, в которых требуется просмотреть аннотированные компоненты. И в этом случае пакет будет просмотрен на предмет наличия в нем каждого указанного класса.

Если в аннотации `@SpringBootApplication` не определен атрибут для просмотра компонентов, то просмотрен будет только тот пакет, в котором находится класс, снабженный данной аннотацией. Именно поэтому в рассматриваемом здесь примере находится определение компонента `helloWorld` и происходит его создание.

В данном примере продемонстрировано построение средствами Spring Boot простого консольного приложения с единственным компонентом Spring Bean, определяемым разработчиком, и полностью готовой средой. Но в модуле Spring Boot предоставляются также начальные зависимости для построения веб-приложений. К их числу относится начальная зависимость от библиотеки `spring-boot-starter-web`, а на рис. 4.4 приведены транзитивные зависимости от этой библиотеки Spring Boot. В проекте `boot-web` класс `HelloWorld` служит в качестве контроллера Spring, т.е. специального класса, предназначенного для создания веб-компонентов Spring Beans, как показано ниже. Это типичный для проектного шаблона MVC класс контроллера Spring, более подробно рассматриваемого в главе 16.

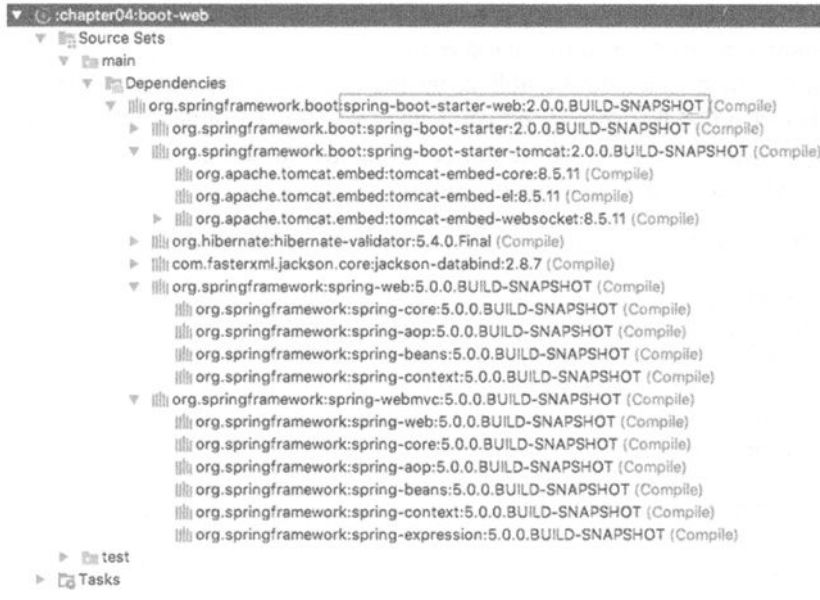


Рис. 4.4. Представление Gradle Projects проекта **`boot-web`**

```
package com.apress.prospring5.ch4.ctrl;

import org.springframework.web.bind
          .annotation.RequestMapping;
import org.springframework.web.bind
          .annotation.RestController;

@RestController
public class HelloWorld {

  @RequestMapping("/")
  public String sayHi() {
    return "Hello World!";
  }
}
```

Для объявления веб-компонентов Spring Beans применяется аннотация `@Cont roller` как особая разновидность аннотации `@Component`. Классы типа контроллеров содержат методы, снабженные аннотацией `@RequestMapping`, благодаря которой они отображаются на определенный URL запроса. В приведенном выше примере аннотация `@RestController` служит конкретным практическим целям. Это разновидность аннотации `@Controller`, предназначенная для объявления REST-служб. В данном случае целесообразно сделать компонент `helloWorld` доступным в качестве REST-службы, чтобы не создавать полноценное веб-приложение с пользовательским интерфейсом и прочими веб-компонентами, способными заслонить собой ос-

новную идею, поясняемую в этом разделе. Все упоминаемые здесь веб-компоненты более подробно рассматриваются в главе 16.

Создадим далее класс начальной загрузки, воспользовавшись простым методом `main()`. В связи с тем что контроллер типа `HelloWorld` объявлен не в том пакете, где класс `WebApplication`, ниже наглядно показано, как пользоваться атрибутом `scanBasePackageClasses` в прикладном коде.

```
package com.apress.prospring5.ch4;

import com.apress.prospring5.ch4.ctrl.HelloWorld;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure
          .SpringBootApplication;
import org.springframework.context
          .ConfigurableApplicationContext;

import java.util.Arrays;

@SpringBootApplication(
           scanBasePackageClasses = HelloWorld.class)
public class WebApplication {

  private static Logger logger =
          LoggerFactory.getLogger(WebApplication.class);

  public static void main(String args) throws Exception {
    ConfigurableApplicationContext ctx =
         SpringApplication.run(WebApplication.class, args);
    assert (ctx != null);
    logger.info("Application started...");

    System.in.read();
    ctx.close();
  }
}
```

На данном этапе у вас может возникнуть следующий вопрос: где находится файл конфигурации `web.xml` и все остальные компоненты, которые необходимо создать для столь простого веб-приложения? Оказывается, что в приведенных выше фрагментах кода мы уже определили все, что требуется! Если не верите, скомпилируйте данный проект и выполняйте исходный код класса `Application` до тех пор, пока не увидите протокольное сообщение, извещающее о запуске приложения на выполнение. Если вы проанализируете протокольные файлы, то обнаружите, сколько всего происходит в столь небольшом объеме кода. Но самое примечательное, что контейнер сервлетов Tomcat работает, а различные конечные точки, где, например, произво-

дится контроль работоспособности, выводятся сведения об окружении и измеряются количественные показатели, определяются автоматически. Перейдите сначала по адресу `http://localhost:8080`, чтобы увидеть веб-страницу с приветствием "Hello World!", отображаемую, как и предполагалось. Затем проанализируйте предварительно сконфигурированные конечные точки (например, по ссылке `http://localhost:8080/health` возвращается представление состояния работающего приложения в формате JSON). Далее выполните загрузку по адресу `http://localhost:8080/health`, чтобы составить более ясное представление о различных накопленных количественных показателях, включая объем динамической памяти типа "кучи", сборку "мусора" и т.д.

Из одного только рассмотренного выше примера можно сделать вывод, что модуль Spring Boot существенно упрощает процесс создания приложений любого типа. Прошли те времена, когда приходилось вручную создавать многочисленные файлы конфигурации, чтобы настроить и привести в действие простое веб-приложение. При наличии встроенного контейнера сервлетов, готового к обслуживанию веб-приложения, все работает как часы.

Несмотря на всю простоту приведенного выше примера, не следует забывать, что модуль Spring Boot совсем не ограничивает возможности построения приложений тем, что он выбирает сам. Он просто применяет "компетентный" подход к автоматическому выбору необходимых средств по умолчанию. Так, если вместо контейнера сервлетов Tomcat потребуется контейнер Jetty, для этого достаточно внести соответствующие коррективы в файл конфигурации, чтобы исключить начальный модуль Tomcat из зависимости `spring-boot-starter-web`. А с помощью представления Gradle Projects можно воспроизвести те зависимости, которые внедрены в текущий проект. В модуле Spring Boot предоставляется немало других зависимостей для разнотипных приложений, поэтому настоятельно рекомендуется обратиться за дополнительной справкой к его документации, оперативно доступной по адресу `http://projects.spring.io/spring-boot/`.

Резюме

В этой главе был представлен обширный ряд средств, дополняющих основные возможности инверсии управления в Spring. В ней было показано, как подключаться к жизненному циклу компонентов Spring Beans и информировать их о среде Spring. В качестве решения для реализации инверсии управления в целом ряде классов здесь были представлены фабрики компонентов Spring Beans, а также пояснялось, как применять редакторы свойств с целью упростить конфигурирование приложений и исключить потребность в искусственных свойствах типа `String`. Кроме того, в этой главе были продемонстрированы разные способы определения компонентов Spring Beans в формате XML, с помощью аннотаций и конфигурационных классов Java. Здесь было завершено подробное рассмотрение ряда дополнительных средств, пред-

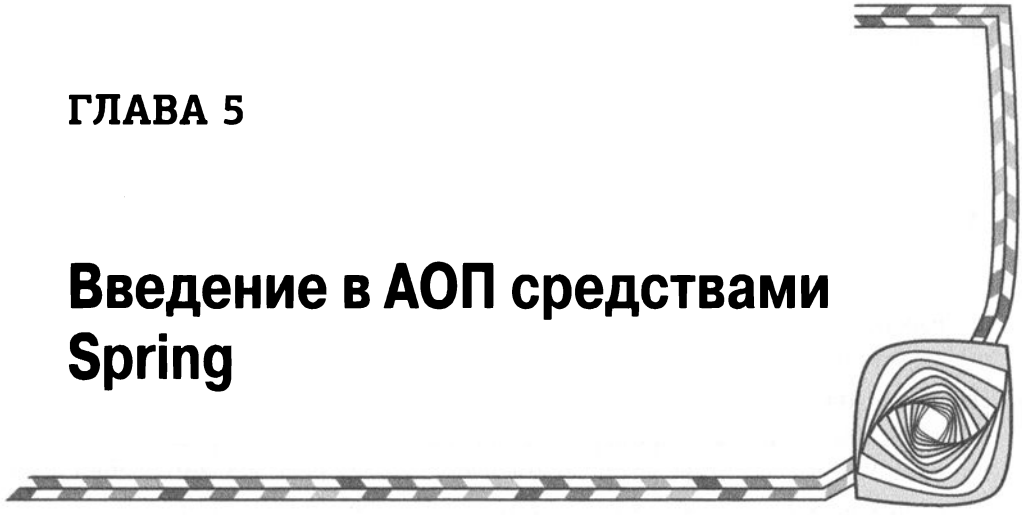
лагаемых в интерфейсе `ApplicationContext`, включая интернационализацию, публикацию событий и доступ к ресурсам.

Помимо этого, были раскрыты такие возможности, как применение классов Java и нового синтаксиса Groovy вместо конфигурирования в формате XML, поддержка профилей, а также уровень абстракции среды и источников свойств. И, наконец, в этой главе было рассмотрено применение в Spring аннотаций по спецификации JSR-330. А в качестве приятного дополнения в завершение главы было показано, как пользоваться модулем Spring Boot для конфигурирования компонентов Spring Beans и начальной загрузки приложений при первой же возможности и с минимальными затратами труда.

До сих пор мы рассматривали основные принципы, понятия и такие средства Spring Framework, как контейнер инверсии управления и другие услуги, предоставляемые ядром Spring Framework. Начиная со следующей главы, мы обсудим применение Spring в отдельных областях, включая АОП, доступ к данным, обработку транзакций и поддержку веб-приложений.

ГЛАВА 5

Введение в АОП средствами Spring



Помимо внедрения зависимостей (Dependency Injection — DI), в каркасе Spring Framework обеспечивается поддержка аспектно-ориентированного программирования (АОП). Термином *АОП* нередко обозначают инструментальные средства для реализации сквозной функциональности. Понятие *сквозной функциональности* имеет отношение к логике, которая не может быть отделена от остальной части приложения, что в конечном итоге приводит к дублированию кода и тесной связанности. Модуляризация средствами АОП отдельных частей логики, называемых *функциональностью*, позволяет применять их в различных частях приложения, не дублируя код и не создавая жесткие зависимости. Типичными примерами сквозной функциональности служат протоколирование и обеспечение безопасности, характерные для многих приложений.

Допустим, имеется приложение, протоколирующее в целях отладки начало и завершение каждого метода. Даже если вынести исходный код протоколирования в отдельный класс, то впоследствии все равно придется вызывать методы этого класса для целей протоколирования по два раза на каждый метод, вызываемый из приложения. А пользуясь АОП, достаточно указать, что методы протоколирующего класса должны вызываться до и после вызова каждого метода, вызываемого из приложения.

Важно понимать, что АОП дополняет ООП, а не соперничает с ним. ООП очень хорошо подходит для обширного ряда задач, которые приходится решать программистам. Но если снова обратиться к примеру протоколирования, то станет очевидно, что когда дело доходит до реализации сквозной функциональности в крупных масштабах, то возможностей ООП недостаточно. С одной стороны, применять только АОП для разработки целого приложения практически нереально, учитывая, что функции АОП основаны на ООП. А с другой стороны, более разумный подход предусматривает применение АОП для решения определенных задач, связанных со сквоз-

ной функциональностью, несмотря на то, что построить целые приложения только средствами ООП вполне возможно.

В этой главе будут рассмотрены следующие вопросы.

- **Основы АОП.** Прежде чем обсуждать реализацию АОП в Spring, в этой части будут изложены основы технологии АОП. Большинство понятий, описанных здесь, не являются характерными только для Spring и могут быть обнаружены в любой реализации АОП. Если вы уже знакомы с другими реализациями АОП, можете пропустить эту часть главы.
- **Типы АОП.** Различают два типа АОП: статическое и динамическое. При статическом АОП, как, например, в механизмах связывания во время компиляции кода AspectJ[1], сквозная логика применяется к коду на этапе компиляции, и ее нельзя изменить, не внеся соответствующие коррективы в исходный код и не скомпилировав его снова. А при динамическом АОП, как, например, в Spring, сквозная логика применяется динамически во время выполнения. Это позволяет вносить изменения в конфигурацию АОП, не компилируя прикладной код повторно. Эти типы АОП дополняют друг друга, а при совместном использовании составляют эффективную комбинацию для применения в приложениях.
- **Архитектура АОП в Spring.** Реализация АОП в Spring является лишь подмножеством полного набора функциональных средств АОП, доступного в других реализациях, включая AspectJ. В этой части будет сделан общий обзор средств АОП, доступных в Spring, и особенностей их реализации, а также пояснено, почему некоторые из этих средств исключены из реализации в Spring.
- **Заместители АОП в Spring.** Заместители составляют львиную долю реализации АОП в Spring, и в них нужно хорошо разбираться, чтобы извлечь наибольшую пользу из данной реализации АОП. В этой части главы будут рассмотрены два вида заместителей: динамический заместитель JDK и заместитель CGLIB. В частности, мы рассмотрим различные сценарии применения каждой из этих разновидностей заместителей, оценим их производительность и дадим ряд простых рекомендаций, следуя которым в приложении можно извлечь наибольшую пользу из реализации АОП в Spring.
- **Применение АОП в Spring.** В этой части будет представлен ряд практических примеров применения АОП. И начнем мы с простого примера вывода сообщения "Hello World!", чтобы упростить понимание кода АОП, а продолжим подробным описанием функциональных средств АОП, доступных в Spring, сопровождая их подходящими примерами.
- **Расширенное использование срезов.** В этой части будут исследованы классы `ComposablePointcut` и `ControlFlowPointcut`, введения и соответствующие технологии, которые придется задействовать, применяя срезы в приложении.

[1] См. по адресу `http://eclipse.org/aspectj/`.

- **Службы каркаса АОП.** В каркасе Spring полностью поддерживается конфигурирование АОП прозрачным и декларативным образом. В этой части будут рассмотрены три способа (класс `ProxyFactoryBean`, пространство имен `aop` и аннотации в стиле `@AspectJ`) внедрения декларативно определяемых заместителей АОП в объекты приложения как взаимодействующих с ними объектов. В итоге приложение даже не подозревает, что оно оперирует объектами, снабженными советами.
- **Интеграция с AspectJ.** Язык AspectJ является расширением Java, в котором полностью реализованы принципы АОП. Основное отличие реализации АОП в AspectJ и Spring состоит в том, что совет применяется в AspectJ к целевым объектам через связывание (во время компиляции или же во время выполнения), тогда как реализация АОП в Spring основывается на заместителях. Набор функциональных средств AspectJ намного обширнее, чем у реализации АОП в Spring, но им труднее пользоваться по сравнению с аналогичным набором в Spring. Язык AspectJ оказывается подходящим решением в том случае, когда необходимых средств АОП, требующихся в приложении, становится в Spring недостаточно.

Основные понятия АОП

Как и в большинстве других технологий, в АОП имеется свой особый ряд понятий и терминов, смысл которых важно понимать. Ниже перечислены и вкратце описаны основные понятия АОП.

- **Точки соединения** (joinpoint). Это вполне определенная точка во время выполнения приложения. Характерными примерами точек соединения служат вызов метода, обращение к методу, инициализация класса и получение экземпляра объекта. Точки соединения относятся к базовым понятиям АОП и определяют места в приложении, где можно вставлять дополнительную логику средствами АОП.
- **Советы.** Фрагмент кода, который должен выполняться в отдельной точке соединения, представляет собой *совет* (advice), определенный в методе класса. Существует много разновидностей советов, в том числе предшествующий совет, когда он выполняется до точки соединения, а также последующий совет, когда он выполняется после точки соединения.
- **Срезы.** *Срез* (pointcut) — это совокупность точек соединения, предназначенная для определения момента, когда следует выполнить совет. Создавая срезы, можно приобрести очень точный контроль над тем, как применять совет к компонентам приложения. Как упоминалось ранее, типичной точкой соединения служит вызов метода или же вызовы всех методов из отдельного класса. Зачастую между срезами можно устанавливать сложные отношения, чтобы наложить дополнительное ограничение на момент, когда следует выполнить совет.

- **Аспекты.** *Аспект* (aspect) — это сочетание совета и срезов, инкапсулированных в классе. Такое сочетание приводит в итоге к определению логики, которую следует внедрить в приложение, а также тех мест, где она должна выполняться.
- **Связывание** (weawing) — это процесс вставки аспектов в определенном месте прикладного кода. Для решений АОП на стадии компиляции связывание обычно делается статически во время сборки. А для решений АОП на стадии выполнения связывание происходит динамически во время выполнения. В языке AspectJ поддерживается еще один механизм связывания, называемый *связыванием во время загрузки* (load-time weaving — LTW), при котором перехватывается базовый загрузчик классов виртуальной машины JVM и обеспечивается связывание с байт-кодом, когда загрузчик классов загружает его.
- **Цель** (target) — это объект, поток исполнения которого изменяется каким-нибудь процессом АОП. Зачастую целевой объект обозначается как *снабженный советом* объект.
- **Введение** (introduction) — это процесс, посредством которого можно изменить структуру объекта, введя в него дополнительные методы или поля. Введение можно применять в АОП для того, чтобы реализовать в любом объекте определенный интерфейс, не прибегая к явной его реализации в классе данного объекта.

Не отчаивайтесь, если эти понятия пока что не совсем ясны вам. Они станут яснее, когда мы представим ряд наглядных примеров. Имейте также в виду, что вам не придется непосредственно иметь дело со многими из перечисленных выше понятий благодаря особенностям реализации АОП в Spring, где некоторые из этих понятий не так уж и важны. По ходу изложения материала этой главы мы обсудим все эти понятия в контексте Spring.

Типы АОП

Как упоминалось ранее, имеются два разных типа АОП: статическое и динамическое. Отличаются они лишь местом, где происходит процесс связывания, а также тем, как этот процесс достигается.

Реализация статического АОП

При статическом АОП связывание образует еще одну стадию в процессе сборки приложения. В терминах Java процесс связывания при статической реализации АОП предусматривает модификацию действительного байт-кода приложения, изменяя и расширяя прикладной код должным образом. Это эффективный механизм достижения связывания, поскольку конечным результатом оказывается просто байт-код Java, избавляя от необходимости предпринимать специальные меры на стадии выполнения, чтобы определить тот момент, когда должен быть выполнен совет. Недостаток

такого механизма заключается в том, что любые изменения, вносимые в аспекты, даже если они касаются только ввода еще одной точки соединения, требуют перекомпиляции всего приложения в целом. Отличным примером статической реализации АОП может служить связывание во время компиляции кода AspectJ.

Реализация динамического АОП

Динамические реализации АОП, как, например, в Spring, отличаются от статических реализаций тем, что процесс связывания происходит динамически во время выполнения. Добиться этого можно по-разному в зависимости от конкретной реализации, но, как станет ясно в дальнейшем, применяемый в Spring подход состоит в создании заместителя для всех целевых объектов, что дает возможность вызывать совет по мере надобности. Недостаток динамического АОП заключается в том, что оно обычно не выполняется так же хорошо, как и статическое АОП, хотя производительность при этом неуклонно растет. А главным преимуществом динамических реализаций АОП является простота, с которой можно изменить целый ряд аспектов в приложении, не прибегая к повторной компиляции основного прикладного кода.

Выбор типа АОП

Выбор между статическим и динамическим АОП — довольно трудное решение. Оба типа реализации АОП обладают своими достоинствами, а разработчики не ограничены в применении лишь какого-то одного типа. В общем, статические реализации АОП существуют дольше и обычно обладают более богатым набором функциональных средств и большим количеством доступных точек соединения. И если производительность особенно важна или же требуется средство АОП, которое в Spring не реализовано, то, как правило, следует применять AspectJ. А в большинстве других случаев реализация АОП в Spring оказывается идеальным выбором. Однако многие решения, основанные на АОП, включая управление транзакциями, предлагаются каркасом Spring в готовом виде, поэтому убедитесь в наличии требующихся функциональных возможностей в Spring, прежде чем реализовывать их самостоятельно!

Как обычно, руководствуйтесь требованиями к приложению, выбирая конкретную реализацию АОП, и не ограничивайтесь единственной реализацией, если для приложения больше подходит определенное сочетание разных технологий. В целом, реализация АОП в Spring отличается меньшей сложностью, чем AspectJ, поэтому именно ее и следует рассматривать в качестве первого варианта выбора.

АОП в Spring

Реализацию АОП в Spring можно представить как состоящую из двух логических частей. Первая часть содержит ядро АОП, обеспечивающее полностью развязанные, чисто программные функциональные возможности АОП, иначе называемые *Spring AOP API*, т.е. прикладным интерфейсом АОП в Spring. А вторая часть реализации

АОП содержит ряд служб каркаса, упрощающих применение АОП в приложениях. Помимо этого, другие компоненты Spring, в том числе диспетчер транзакций и вспомогательные классы EJB, предоставляют ориентированные на АОП службы, упрощающие разработку приложений.

Альянс АОП

Альянс АОП (AOP Alliance; `http://aopalliance.sourceforge.net/`) — это результат совместных усилий участников многих проектов АОП с открытым кодом по определению стандартного набора интерфейсов для реализаций АОП. Там, где это возможно, в Spring вместо определения собственных интерфейсов применяются интерфейсы, определенные Альянсом АОП. Это позволяет многократно использовать вполне определенный совет во многих реализациях АОП, где поддерживаются интерфейсы от Альянса АОП.

Пример вывода обращения в АОП

Прежде чем вдаваться в подробности реализации АОП в Spring, обратимся к конкретному примеру, в котором будет продемонстрировано, как вместо традиционного для программирования приветствия "Hello World!" вывести обращения по Ф.И.О. известного киногероя Джеймса Бонда. Итак, определим и реализуем класс `Agent`, чтобы вывести фамилию Bond. А в процессе применения АОП экземпляр данного класса будет преобразован на стадии выполнения для вывода обращения "James Bond!". Ниже показано, каким образом определяется класс `Agent`.

```
package com.apress.prospring5.ch5;

public class Agent {
  public void speak() {
    System.out.print("Bond");
  }
}
```

Реализовав метод вывода фамилии киногероя в приведенном выше классе, добавим к этому методу *совет*. В терминологии АОП это означает *снабдить* метод советом. В итоге вместо фамилии Bond в методе `speak()` будет выведено обращение "James Bond!".

Чтобы добиться этого, прежде чем выполнять тело метода `speak()`, необходимо выполнить один код (для вывода имени киногероя `James`), а после выполнения тела данного метода — другой код (для вывода знака `!`). В терминологии АОП для этой цели потребуется *окружающий* совет, который выполняется вокруг точки соединения. В данном случае точкой соединения служит вызов метода `speak()`. Ниже приведен исходный код класса `AgentDecorator`, действующего в качестве реализации окружающего совета.

```
package com.apress.prospring5.ch5;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class AgentDecorator implements MethodInterceptor {
  public Object invoke(MethodInvocation invocation)
        throws Throwable {
    System.out.print("James ");

    Object retVal = invocation.proceed();

    System.out.println("!");
    return retVal;
  }
}
```

`MethodInterceptor` — это стандартный интерфейс от Альянса АОП, предназначенный для реализации совета, окружающего точки соединения вызовов методов. А объект типа `MethodInvocation` представляет вызов метода, снабжаемого советом, и с помощью этого объекта можно контролировать момент, когда разрешается продолжить вызов метода. А поскольку это окружающий совет, то можно выполнить определенные действия как до, так и после вызова метода, но до возврата из него. В приведенном выше фрагменте кода сначала на консоль выводится строка `"James "`, затем вызывается метод `speak()` через вызов `MethodInvocation.proceed()`, а после этого на консоль выводится строка `"!"`.

Завершающая стадия данного примера состоит в связывании совета типа `AgentDecorator` (а точнее — метода `invoke()`) с прикладным кодом. Для этого сначала получается экземпляр типа `Agent`, т.е. целевой объект, а затем его заместитель, давая тем самым команду фабрике заместителей привязать совет типа `AgentDecorator`, как показано ниже.

```
package com.apress.prospring5.ch5;

import org.springframework.aop.framework.ProxyFactory;

public class AgentAOPDemo {
  public static void main(String... args) {
    Agent target = new Agent();

    ProxyFactory pf = new ProxyFactory();
    pf.addAdvice(new AgentDecorator());
    pf.setTarget(target);

    Agent proxy = (Agent) pf.getProxy();

    target.speak();
```

```
    System.out.println("");
    proxy.speak();
  }
}
```

В приведенном выше фрагменте кода очень важно отметить применение класса `ProxyFactory` для создания заместителя целевого объекта и одновременной привязки к нему совета. Совет типа `AgentDecorator` передается объекту типа `ProxyFactory` через вызов `addAdvice()`, а цель для связывания указывается через вызов метода `setTarget()`. Как только цель будет установлена, а совет введен в объект типа `ProxyFactory`, в результате вызова метода `getProxy()` сформируется заместитель. И, наконец, для исходного целевого объекта, а затем и для его заместителя вызывается метод `speak()`.

Выполнение приведенного выше фрагмента кода дает следующий результат:

```
Bond
James Bond!
```

Как видите, вызов метода `speak()` для незатронутого целевого объекта приводит к стандартному обращению к нему без вывода дополнительной информации на консоль. Но при вызове данного метода для заместителя выполняется код из класса `AgentDecorator`, фомируя вывод требуемого обращения "James Bond!". В приведенном здесь примере у целевого класса, снабженного советом, отсутствуют зависимости от каркаса Spring или интерфейсов от Альянса АОП. Прелесть реализации АОП в Spring в частности и методики АОП вообще состоит в том, что советом можно снабдить почти любой класс, даже если этот класс создавался без учета АОП. Единственное ограничение, по крайней мере, для реализации АОП в Spring, состоит в невозможности снабдить советом оконечные классы, поскольку они не могут быть расширены, а следовательно, их нельзя заместить. Оконечный класс нельзя заместить и в том случае, если он не реализует интерфейс.

Архитектура АОП в Spring

Базовая архитектура АОП в Spring основана на заместителях. Так, если требуется получить экземпляр класса, снабженного советом, то с этой целью придется получить экземпляр заместителя типа `ProxyFactory`, предоставив прежде всего конструктору класса `ProxyFactory` все аспекты, которые необходимо привязать к заместителю. Применение класса `ProxyFactory` — это чисто программный подход к созданию заместителей в АОП. Зачастую применять такой подход в своем приложении совсем не обязательно. Вместо этого можно положиться на механизмы декларативного конфигурирования АОП, предоставляемые в Spring (класс `ProxyFactory Bean`, пространство имен `aop` и аннотации в стиле `@AspectJ`), которые обеспечат декларативное создание заместителей. Тем не менее важно понимать, каким образом действует механизм создания заместителей, поэтому мы применим сначала про-

граммный подход к созданию заместителей, а затем приступим к исследованию декларативных конфигураций АОП в Spring.

На стадии выполнения каркас Spring анализирует сквозную функциональность, определенную для компонентов Spring Beans в контексте типа `ApplicationContext`, и динамически формирует замещающие компоненты Spring Beans, которые служат оболочками для целевых компонентов. Вместо непосредственного обращения к целевому компоненту Spring Bean вызывающие объекты внедряются с помощью замещающего компонента Spring Bean. Затем замещающий компонент Spring Bean анализирует текущие условия (т.е. точку соединения, срез или совет) и соответственно привязывает подходящий совет. На рис. 5.1 схематически показано, каким образом действует заместитель при реализации АОП в Spring.

Рис. 5.1. Принцип действия заместителя при реализации АОП в Spring

В самом каркасе Spring поддерживаются две реализации заместителей: динамические заместители JDK и заместители CGLIB. По умолчанию, когда целевой объект, снабженный советом, реализует какой-нибудь интерфейс, для получения экземпляров заместителя целевого объекта в Spring будет использован динамический заместитель JDK. Но если целевой объект, снабженный советом, не реализует интерфейс (например, потому, что он представляет конкретный класс), то для получения экземпляров заместителей будет применяться библиотека CGLIB. И объясняется это в основном тем, что динамический заместитель JDK поддерживает создание заместителей только для интерфейсов. Более подробно заместители обсуждаются далее в разделе "Общее представление о заместителях".

Точки соединения в Spring

К числу самых заметных упрощений АОП в Spring следует отнести поддержку только одного типа точек соединения, которым является вызов метода. На первый взгляд это может показаться серьезным ограничением тем, кто знаком с другими реализациями АОП (например, в AspectJ), где поддерживается намного больше типов точек соединения, но в действительности это делает каркас Spring более доступным.

В настоящее время точка соединения типа вызова метода употребляется чаще всего из всех доступных точек соединения, и с ее помощью можно решить многие за-

дачи, которые делают АОП полезным в повседневной разработке. Не следует, однако, забывать, что если требуется снабдить советом какой-нибудь код в точке соединения, отличающейся от вызова метода, то всегда можно воспользоваться Spring вместе с AspectJ.

Аспекты в Spring

При реализации АОП в Spring аспект представлен экземпляром класса, реализующего интерфейс `Advisor`. В каркасе Spring предоставляются удобные реализации интерфейса `Advisor`, которые можно применять в своих приложениях, не прибегая к необходимости самостоятельно создавать специальные его реализации. У интерфейса `Advisor` имеются два подчиненных интерфейса: `IntroductionAdvisor` и `PointcutAdvisor`.

Интерфейс `PointcutAdvisor` реализуется во всех реализациях интерфейса `Advisor`, где срезы служат для управления процессом применения совета в точках соединения. Введения трактуются в Spring как особые разновидности советов, а с помощью интерфейса `IntroductionAdvisor` можно управлять теми классами, в которых применяется введение. Различные реализации интерфейса `PointcutAdvisor` более подробно обсуждаются в далее разделе "Советники и срезы в Spring".

Описание класса ProxyFactory

Класс `ProxyFactory` управляет процессом связывания и создания заместителей при реализации АОП в Spring. Прежде чем создавать заместитель, необходимо указать снабженный советом, или целевой, объект. Как пояснялось ранее, это можно сделать с помощью метода `setTarget()`. В самом классе `ProxyFactory` процесс создания заместителя поручается экземпляру типа `DefaultAopProxyFactory`, который, в свою очередь, поручает его экземпляру типа `Cglib2AopProxy` или `JdkDynamicAopProxy` в зависимости от параметров настройки приложения. Порядок создания заместителей более подробно рассматривается далее в этой главе.

Метод `addAdvice()`, задействованный в исходном коде из рассмотренного ранее примера, предоставляется в классе `ProxyFactory` на тот случай, если совет должен применяться в вызовах всех, а не только избранных методов из класса. Совет, передаваемый методу `addAdvice()`, заключается в оболочку экземпляра типа `DefaultPointcutAdvisor`, который является стандартной реализацией интерфейса `PointcutAdvisor`, и конфигурирует его по срезу, включающему по умолчанию все методы. Если же требуется приобрести дополнительный контроль над созданием советника типа `Advisor` или дополнить заместитель введением, то придется самостоятельно создать реализацию интерфейса `Advisor` и вызвать метод `addAdvisor()` из класса `ProxyFactory`.

Один и тот же экземпляр типа `ProxyFactory` можно применять для создания многих заместителей с разными аспектами. Для оказания помощи в этом деле в классе `ProxyFactory` предусмотрены методы `removeAdvice()` и `removeAdvisor()`,

позволяющие удалить из экземпляра типа `ProxyFactory` любой введенный ранее совет или советник. А для того чтобы проверить, имеется ли в экземпляре типа `ProxyFactory` конкретный присоединенный к нему совет, следует вызвать метод `adviceIncluded()`, передав ему объект проверяемого совета.

Создание совета в Spring

В каркасе Spring поддерживается шесть разновидностей советов, перечисленных в табл. 5.1.

Некоторые считают, что этих разновидностей только четыре, поскольку большинство разработчиков исключают класс `IntroductionInterceptor`, а все реализации последующего совета сводят в одно семейство. Ради простоты можно было бы свести описание советов только к трем из них: предшествующему, окружающему и последующему, что было бы так же верно. Хотя это зависит от того, что именно интересует конкретного разработчика.

В этой книге рассматриваются шесть разновидностей советов в силу их особой важности для методов. Они представлены отдельными интерфейсами, которые расширяют основной интерфейс `Advice` за исключением интерфейса `IntroductionInterceptor`, который расширяет не только интерфейс `MethodInterceptor`, предназначенный для окружающего совета, но и интерфейс `DynamicIntroductionAdvice`, предназначенный для специальной разновидности совета, позволяющей реализовывать заранее неизвестные дополнительные интерфейсы. Можно было бы также упомянуть интерфейс `ConstructorInterceptor`, но поскольку конструкторы относятся к особому типу методов, то рассматривать здесь этот интерфейс, по-видимому, неуместно.

Таблица 5.1. Типы советов в Spring

Наименование совета	Интерфейс	Описание
Предшествующий	`org.springframework.aop.MethodBeforeAdvice`	Используя предшествующий совет, можно выполнять специальную обработку перед входом в точку соединения. А поскольку в Spring всегда применяется точка соединения типа вызова метода, это, по существу, позволяет реализовать предварительную обработку до выполнения метода. Предшествующий совет имеет полный доступ к цели вызова метода, а также к аргументам, передаваемым методу, но он никак не контролирует выполнение самого метода. Если же предшествующий совет сгенерирует исключение, то дальнейшее выполнение цепочки перехватчиков (а также целевого метода) прекратится, а исключение распространится обратно по цепочке перехватчиков

Продолжение табл. 5.1

Наименование совета	Интерфейс	Описание
Послевозвратный	`org.springframework.aop.AfterReturningAdvice`	Послевозвратный совет выполняется по окончании вызова метода в точке соединения и возврата значения. Послевозвратный совет имеет доступ к цели вызова метода, аргументам, передаваемым методу, а также к возвращаемому значению. А поскольку, метод уже выполнен, когда вызывается совет данного типа, то он никак не контролирует вызов метода. Если целевой метод сгенерирует исключение, то послевозвратный совет не будет выполнен, а исключение распространится вверх по стеку вызовов обычным образом
Последующий (окончательно)	`org.springframework.aop.AfterAdvice`	Послевозвратный совет выполняется только при нормальном завершении метода, снабженного советом. Но последующий (окончательно) совет будет выполнен независимо от результата выполнения метода, снабженного советом. Совет данного типа выполняется даже в том случае, если метод, снабженный советом, завершается неудачно или генерируется исключение. Указанный интерфейс служит лишь маркерным для обоих разновидностей последующего совета, поддерживаемых в Spring. Но поскольку в Spring отсутствует собственная поддержка последующего (окончательно) совета, то его реализацию приходится заимствовать из внешней библиотеки вроде AspectJ
Окружающий	`org.aopalliance.intercept.ethodInterceptor`	Окружающий совет моделируется в Spring по стандарту Альянса АОП на перехватчики методов. Такому совету разрешается выполняться до и после вызова метода. Имеется также возможность контролировать момент, когда вызов метода может быть продолжен. Если требуется, можно вообще пропустить выполнение метода, предоставив собственную реализацию его логики
Перехватывающий	`org.springframework.aop.ThrowsAdvice`	Перехватывающий совет выполняется после возврата из вызванного метода, но только в том случае, если во время вызова было сгенерировано исключение. Совет данного типа может перехватывать только конкретные исключения, и тогда возможен доступ к методу, сгенерировавшему исключение, аргументам, передаваемым при вызове, а также к цели вызова

Окончание табл. 5.1

Наименование совета	Интерфейс	Описание
Введение	`org.springframework.aop.IntroductionInterceptor`	Введения моделируются в Spring как специальные типы перехватчиков. Используя перехватчик введений, можно указать реализацию методов, вводимых по совету

Интерфейсы для советов

Как следует из приведенного ранее описания класса `ProxyFactory`, совет вводится в заместитель прямо через вызов метода `addAdvice()` или косвенно через вызов метода `addAdvisor()` из класса, реализующего интерфейс `Advisor`. Главное отличие совета от советника состоит в том, что советник несет в себе совет и связанный с ним срез, благодаря чему обеспечивается более точный контроль над тем, какие именно точки соединения будет перехватывать совет. В каркасе Spring предусмотрена четко определенная иерархия интерфейсов для советов, основанная на интерфейсах от Альянса АОП (рис. 5.2).

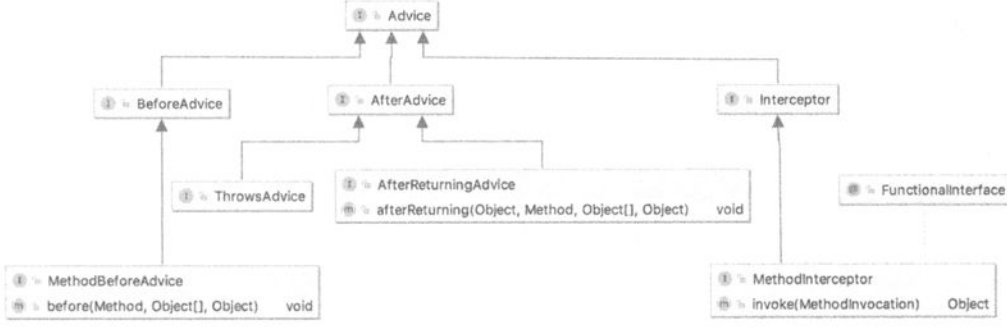


Рис. 5.2. Схематическое представление иерархии интерфейсов для разных типов советов в IDE IntelliJ IDEA

Такая иерархия обладает не только преимуществом явно выраженной объектно-ориентированной структуры, но и возможностью оперировать разными типами советов обобщенно, используя, например, единственный метод `addAdvice()` из класса `ProxyFactory`. А кроме того, новые типы советов можно легко внедрять, не внося необходимые коррективы в класс `ProxyFactory`.

Создание предшествующего совета

Предшествующий совет относится к числу наиболее употребительных типов советов, доступных в Spring. Этот совет способен изменить аргументы, передаваемые методу, а также предотвратить выполнение метода, сгенерировав исключение. В этом разделе будут представлены два простых примера, демонстрирующих применение

предшествующего совета: для вывода на консоль сообщения, содержащего имя метода, перед его выполнением, а также для ограничения доступа к методам объекта. Ниже приведен исходный код класса `SimpleBeforeAdvice`, реализующего простой предшествующий совет.

```
package com.apress.prospring5.ch5;

import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;
import org.springframework.aop.framework.ProxyFactory;

public class SimpleBeforeAdvice
       implements MethodBeforeAdvice {
  public static void main(String... args) {
    Guitarist johnMayer = new Guitarist();

    ProxyFactory pf = new ProxyFactory();
    pf.addAdvice(new SimpleBeforeAdvice());
    pf.setTarget(johnMayer);

    Guitarist proxy = (Guitarist) pf.getProxy();

    proxy.sing();
  }

  @Override
  public void before(Method method, Object[] args,
                     Object target) throws Throwable {
    System.out.println("Before '" + method.getName()
                     + "', tune guitar.");
  }
}
```

Класс `Guitarist` содержит единственный метод `sing()`, выводящий текст песни на консоль, и реализует интерфейс `Singer`, применяемый повсюду в примерах из данной книги:

```
package com.apress.prospring5.ch5;

import com.apress.prospring5.ch2.common.Singer;

public class Guitarist implements Singer {
  private String lyric="You're gonna live forever in me";

  @Override
  public void sing(){
   System.out.println(lyric);
  }
}
```

По существу, совет служит для того, чтобы убедиться, что исполнитель, представленный объектом `johnMayer`, настроил свою гитару перед тем, как петь. В рассматриваемом здесь примере кода советом снабжен экземпляр класса `Guitarist`, предварительно полученный с помощью экземпляра класса `SimpleBeforeAdvice`. В интерфейсе `MethodBeforeAdvice`, который реализуется в упомянутом выше классе `SimpleBeforeAdvice`, определяется единственный метод `before()`, вызываемый каркасом AOP перед вызовом метода в точке соединения. Напомним, что мы пользуемся пока что срезом, предоставляемым по умолчанию методом `addAdvice()` и соответствующим всем методам в данном классе. Методу `before()` передаются три аргумента: вызываемый метод, передаваемые ему аргументы, а также целевой объект вызова типа `Object`. Аргумент типа `Method` из метода `before()` используется в классе `SimpleBeforeAdvice` для вывода на консоль сообщения, содержащего имя вызываемого метода. Выполнение исходного кода из данного примера приведет к приведенному ниже результату. Как видите, прежде вывода текста песни из вызываемого метода `sing()` на консоль направляется сообщение, сформированное в классе `SimpleBeforeAdvice`.

```
Before 'sing', tune guitar.
You're gonna live forever in me
```

Защита доступа к методам с помощью предшествующего совета

В этом разделе мы реализуем предшествующий совет, который проверяет учетные данные пользователя, прежде чем разрешить продолжение вызова метода. Если учетные данные пользователя неверны, совет генерирует исключение, предотвращая выполнение метода. Пример, рассматриваемый в этом разделе, несколько упрощен. Он дает пользователям возможность пройти аутентификацию с произвольным паролем и предоставляет доступ к защищенным методам только одному пользователю с жестко закодированными учетными данными. Тем не менее данный пример наглядно демонстрирует простоту применения АОП для реализации сквозной функциональности в целях безопасности.

На заметку Это всего лишь пример, демонстрирующий применение предшествующего совета. Полная поддержка защиты выполнения методов из компонентов Spring Beans уже предоставляется в проекте Spring Security, поэтому вам не придется самостоятельно реализовывать подобные функциональные средства.

Ниже приведен исходный код класса `SecureBean`, который будет защищен средствами АОП.

```
package com.apress.prospring5.ch5;

public class SecureBean {
```

```
  public void writeSecureMessage() {
    System.out.println("Every time I learn something new, "
            + "it pushes some old stuff out of my brain");
  }
}
```

Класс `SecureBean` лишь сообщает один из перлов премудрости Гомера Симпсона, который мы не хотим показывать абсолютно всем. А поскольку в данном примере требуется аутентификация пользователей, мы должны каким-то образом предусмотреть сохранение сведений о них. Ниже приведен код класса `UserInfo`, предназначенного для хранения учетных данных пользователя.

```
package com.apress.prospring5.ch5;

public class UserInfo {
  private String userName;
  private String password;

  public UserInfo(String userName, String password) {
    this.userName = userName;
     this.password = password;
  }

  public String getPassword() {
    return password;
  }

  public String getUserName() {
    return userName;
  }
}
```

Этот класс сохраняет сведения о пользователе, с которыми в дальнейшем можно сделать что-нибудь полезное. Ниже представлен исходный код класса `Security Manager`, отвечающего за аутентификацию пользователей и сохранение их учетных данных для последующего извлечения.

```
package com.apress.prospring5.ch5;

public class SecurityManager {
  private static ThreadLocal<UserInfo>
          threadLocal = new ThreadLocal<>();

  public void login(String userName, String password) {
    threadLocal.set(new UserInfo(userName, password));
  }

  public void logout() {
    threadLocal.set(null);
```

```
  }

  public UserInfo getLoggedOnUser() {
    return threadLocal.get();
  }
}
```

В данном примере класс `SecurityManager` служит для аутентификации пользователя и последующего извлечения учетных данных пользователя, проходящего аутентификацию, которая выполняется с помощью метода `login()`. В реальном приложении метод `login()` должен быть запрограммирован на проверку учетных данных пользователя по базе данных или каталогу LDAP, но здесь предполагается, что все пользователи допускаются к аутентификации. Для пользователя в методе `login()` создается объект типа `UserInfo`, который сохраняется в текущем потоке средствами класса `ThreadLocal`. А в методе `logout()` устанавливается пустое значение `null` в любых учетных данных, которые могут быть сохранены в объекте типа `ThreadLocal`. И, наконец, метод `getLoggedOnUser()` возвращает объект типа `UserInfo` для пользователя, проходящего аутентификацию. Если ни один из пользователей не пройдет аутентификацию, этот метод возвратит пустое значение `null`.

Чтобы проверить, прошел ли пользователь аутентификацию, а если это так, то разрешен ли ему доступ к методам из класса `SecureBean`, мы должны создать совет, который выполняется перед методом и сравнивает объект типа `UserInfo`, возвращаемый из метода `SecurityManager.getLoggedOnUser()`, с набором учетных данных для разрешенных пользователей. Ниже приведен исходный код класса `SecurityAdvice`, представляющего этот совет.

```
package com.apress.prospring5.ch5;

import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;

public class SecurityAdvice implements MethodBeforeAdvice {
  private SecurityManager securityManager;

  public SecurityAdvice() {
    this.securityManager = new SecurityManager();
  }

  @Override
  public void before(Method method, Object[] args,
                     Object target) throws Throwable {
    UserInfo user = securityManager.getLoggedOnUser();

    if (user == null) {
      System.out.println("No user authenticated");
          + "throw new SecurityException("You must login "
```

```
            + "before attempting to invoke the method: "
            + method.getName());
        } else if ("John".equals(user.getUserName())) {
          System.out.println("Logged in user is John - OKAY!");
        } else {
          System.out.println("Logged in user is "
                   + user.getUserName() + " NOT GOOD :(");
          throw new SecurityException("User "
                   + user.getUserName() + " is not allowed "
                   + "access to method " + method.getName());
        }
      }
    }
```

В конструкторе класса `SecurityAdvice` получается экземпляр типа `SecurityManager`, который затем сохраняется в поле. Следует, однако, иметь в виду, что совместное использование одного и того же экземпляра типа `SecurityManager` в рассматриваемом здесь примере приложения и в классе `SecurityAdvice` не требуется, поскольку все данные сохранены в текущем потоке исполнения средствами класса `ThreadLocal`. В методе `before()` просто проверяется, является ли `John` именем пользователя, прошедшего аутентификацию. И если это так, то пользователю предоставляется доступ, а иначе генерируется исключение. Обратите также внимание на то, что объект типа `UserInfo` проверяется на равенство пустому значению `null`, которое означает, что текущий пользователь не прошел аутентификацию.

Ниже приведен пример приложения, в котором класс `SecurityAdvice` применяется для защиты класса `SecureBean`.

```
package com.apress.prospring5.ch5;

import org.springframework.aop.framework.ProxyFactory;

public class SecurityDemo {
  public static void main(String... args) {
    SecurityManager mgr = new SecurityManager();

    SecureBean bean = getSecureBean();

    mgr.login("John", "pwd");
    bean.writeSecureMessage();
    mgr.logout();

    try {
      mgr.login("invalid user", "pwd");
     bean.writeSecureMessage();
    } catch(SecurityException ex) {
      System.out.println("Exception Caught: "
                         + ex.getMessage());
```

```
    } finally {
      mgr.logout();
    }

    try {
      bean.writeSecureMessage();
    } catch(SecurityException ex) {
      System.out.println("Exception Caught: "
                         + ex.getMessage());
    }
  }

  private static SecureBean getSecureBean() {
    SecureBean target = new SecureBean();

    SecurityAdvice advice = new SecurityAdvice();

    ProxyFactory factory = new ProxyFactory();
    factory.setTarget(target);
    factory.addAdvice(advice);

    SecureBean proxy = (SecureBean)factory.getProxy();

    return proxy;
  }
}
```

В методе `getSecureBean()` создается заместитель класса `SecureBean`, который снабжается советом с помощью экземпляра класса `SecurityAdvice` и затем возвращается вызывающему объекту. Когда вызывающий объект вызывает любой метод для этого заместителя, вызов сначала направляется экземпляру класса `SecurityAdvice` для проверки на безопасность. В методе `main()` проверяются три возможных варианта, и для этого метод `SecureBean.writeSecureMessage()` вызывается сначала с двумя наборами пользовательских учетных данных, а затем без учетных данных. А поскольку в классе `SecurityAdvice` разрешается продолжить вызовы методов только в том случае, если прошедшим аутентификацию окажется пользователь `John`, то можно предположить, что удачным в приведенном выше коде станет лишь первый проверяемый вариант. Выполнение исходного кода из данного примера дает следующий результат:

```
Logged in user is John - OKAY!
Every time I learn something new,
it pushes some old stuff out of my brain
Logged in user is invalid user NOT GOOD :(
Exception Caught: User invalid user is not allowed access
to method writeSecureMessage
No user authenticated
```

```
Exception Caught: You must login before attempting
to invoke the method: writeSecureMessage[2]
```

Как видите, только первому вызову метода `SecureBean.writeSecureMessage()` было разрешено продолжаться, а остальные вызовы были прекращены по исключению типа `SecurityException`, сгенерированному в классе `SecurityAdvice`. Несмотря на всю простоту данного примера, он наглядно демонстрирует полезность предшествующего совета. И хотя обеспечение безопасности служит типичным примером употребления предшествующего совета, он может оказаться полезным и в том случае, если потребуется внести коррективы в аргументы, передаваемые методу.

Создание послевозвратного совета

Послевозвратный совет выполняется после возврата из метода, вызываемого в точке соединения. Учитывая, что метод уже выполнен, переданные ему аргументы изменить нельзя. И хотя эти аргументы можно прочитать, путь выполнения изменить нельзя, как, впрочем, и воспрепятствовать выполнению самого метода. Это вполне ожидаемые ограничения, хотя и несколько неожиданным оказывается то обстоятельство, что в послевозвратном совете нельзя модифицировать возвращаемое значение. Применяя послевозвратный совет, приходится довольствоваться лишь дополнительной обработкой. Несмотря на то что послевозвратный совет не позволяет изменять значение, возвращаемое из вызываемого метода, можно сгенерировать исключение, которое будет передано вверх по стеку вместо возвращаемого значения.

В этом разделе будут представлены два примера употребления послевозвратного совета в приложении. В первом примере после вызова метода просто выполняется вывод сообщения на консоль. А во втором примере демонстрируется, как с помощью послевозвратного совета внедрить проверку ошибок в метод. Допустим, имеется класс `KeyGenerator`, генерирующий ключи для криптографических целей. Многие криптографические алгоритмы страдают недостатком, заключающимся в том, что небольшое количество ключей считаются слабыми. Слабым оказывается любой ключ, характеристики которого значительно упрощают выведение исходного сообщения, даже не зная этот ключ. Для алгоритма шифрования DES существует всего 256 возможных ключей. В этом пространстве ключей 4 ключа считаются слабыми, а еще 12 ключей — полуслабыми. И хотя шансы получить один из этих ключей во время их

---

[2] Зарегистрирован пользователь Джон - ХОРОШО!
Всякий раз, когда я узнаю что-то новое, старое вытесняется
из моих мозгов
Зарегистрирован недостоверный пользователь
- ПЛОХО:(
Перехвачено исключение: недостоверному пользователю не
разрешен доступ к методу writeSecureMessage()
Отсутствует пользователь, прошедший аутентификацию
Перехвачено исключение: вы должны зарегистрироваться,
прежде чем пытаться вызвать метод: writeSecureMessage()

случайной генерации невелики (1 из 252), проверка ключей настолько проста, что ее стоит, конечно, выполнить. Во втором примере из этого раздела будет составлен послевозвратный совет для проверки ключей, сгенерированных в классе `KeyGenerator`, на предмет их слабости и генерирования соответствующего исключения при обнаружении слабого ключа.

На заметку За более подробными сведениями о слабых ключах в частности и криптографии вообще рекомендуется обратиться на веб-сайт Уильяма Столлингза (William Stallings) по адресу **`http://williamstallings.com/Cryptography/`**.

В следующем фрагменте кода приведен исходный код класса `SimpleAfterReturningAdvice`, демонстрирующий применение послевозвратного совета для вывода сообщения на консоль после возврата из метода. В данном примере снова применяется упоминавшийся ранее класс `Guitarist`.

```
package com.apress.prospring5.ch5;

import java.lang.reflect.Method;
import org.springframework.aop.AfterReturningAdvice;
import org.springframework.aop.framework.ProxyFactory;

public class SimpleAfterReturningAdvice
       implements AfterReturningAdvice {
  public static void main(String... args) {
    Guitarist target = new Guitarist();

    ProxyFactory pf = new ProxyFactory();

    pf.addAdvice(new SimpleAfterReturningAdvice());
    pf.setTarget(target);

    Guitarist proxy = (Guitarist) pf.getProxy();
    proxy.sing();
  }

  @Override
  public void afterReturning(Object returnValue,
             Method method, Object[] args, Object target)
             throws Throwable {
    System.out.println("After '" + method.getName()
                    + "' put down guitar.");
  }
}
```

Обратите внимание, что в интерфейсе `AfterReturningAdvice` объявлен единственный метод `afterReturning()`, которому передается значение, возвращаемое из вызываемого метода, ссылка на вызываемый метод, аргументы, передаваемые это-

му методу, а также цель вызова. Выполнение исходного кода из данного примера дает следующий результат:

```
You're gonna live forever in me
After 'sing' put down guitar.
```

Приведенный выше результат очень похож на полученный в предыдущем примере с предшествующим советом, за исключением того, что сообщение, выводимое советом на консоль, оказывается после сообщения, выводимого из метода `write Message()`. Послевозвратный совет удобно применять для проведения дополнительной проверки ошибок, когда это возможно, в отношении метода, возвращающего недопустимое значение. В описанном ранее случае генератор криптографических ключей может выдать ключ, который считается слабым для определенного алгоритма. В идеальном случае генератор ключей должен самостоятельно проверять ключи на предмет слабости, но поскольку шансы получить слабые ключи весьма малы, многие генераторы не оснащены такой проверкой. Снабдив послевозвратным советом метод, генерирующий ключи, можно обеспечить их дополнительную проверку. Ниже приведен пример очень простого генератора ключей.

```
package com.apress.prospring5.ch5;

import java.util.Random;

public class KeyGenerator {
  protected static final long
            WEAK_KEY = 0xFFFFFFF0000000L;
  protected static final long
            STRONG_KEY = 0xACDF03F590AE56L;
  private Random rand = new Random();
  public long getKey() {
    int x = rand.nextInt(3);

    if (x == 1) {
      return WEAK_KEY;
    }

    return STRONG_KEY;
  }
}
```

Этот генератор ключей нельзя считать безопасным. Он умышленно сделан простым для данного примера, и в одном случае из трех генерирует слабый ключ. Ниже приведен исходный код класса `WeakKeyCheckAdvice`, где проверяется, не является ли результат, возвращаемый методом `getKey()`, слабым ключом.

```
package com.apress.prospring5.ch5;

import java.lang.reflect.Method;
import org.springframework.aop.AfterReturningAdvice;
```

```
import static com.apress.prospring5.ch5
                .KeyGenerator.WEAK_KEY;

public class WeakKeyCheckAdvice
       implements AfterReturningAdvice {
  @Override
  public void afterReturning(Object returnValue,
              Method method, Object args, Object target)
              throws Throwable {
    if ((target instanceof KeyGenerator)
         && ("getKey".equals(method.getName()))) {
      long key = ((Long) returnValue).longValue();

      if (key == WEAK_KEY) {
        throw new SecurityException("Key Generator "
                + "generated a weak key. Try again");
      }
    }
  }
}
```

В методе `afterReturning()` сначала проверяется, был ли в точке соединения выполнен именно метод `getKey()`. Если это так, то результирующее значение проверяется на предмет слабого ключа. И если обнаружится, что в результате выполнения метода `getKey()` получен слабый ключ, то сгенерируется исключение типа `SecurityException`, чтобы известить об этом вызывающий код. В следующем фрагменте кода приведено простое приложение, демонстрирующее применение послевозвратного совета:

```
package com.apress.prospring5.ch5;

import org.springframework.aop.framework.ProxyFactory;

public class AfterAdviceDemo {
  private static KeyGenerator getKeyGenerator() {
    KeyGenerator target = new KeyGenerator();

    ProxyFactory factory = new ProxyFactory();
    factory.setTarget(target);
    factory.addAdvice(new WeakKeyCheckAdvice());

    return (KeyGenerator)factory.getProxy();
  }

  public static void main(String... args) {
    KeyGenerator keyGen = getKeyGenerator();

    for(int x = 0; x < 10; x++) {
      try {
        long key = keyGen.getKey();
```

```
            System.out.println("Key: " + key);
        } catch(SecurityException ex) {
            System.out.println("Weak Key Generated!");
        }
    }
  }
}
```

После создания снабженного советом заместителя целевого объекта типа `KeyGenerator` в классе `AfterAdviceExample` предпринимается попытка сгенерировать десять ключей. Если в процессе генерирования одного ключа возникает исключение типа `SecurityException`, на консоль выводится сообщение, извещающее пользователя, что был получен слабый ключ, а иначе отображается сгенерированный ключ. Однократный запуск данного приложения на нашей машине дал следующий результат:

```
Key: 48658904092028502
Weak Key Generated![3]
Key: 48658904092028502
Weak Key Generated!
Weak Key Generated!
Weak Key Generated!
Key: 48658904092028502
Key: 48658904092028502
Key: 48658904092028502
Key: 48658904092028502
```

Как следует из приведенного выше результата, в классе `KeyGenerator`, как и ожидалось, периодически генерируются слабые ключи, а в классе `WeakKeyCheckAdvice` гарантируется, что при каждом обнаружении слабого ключа будет сгенерировано исключение типа `SecurityException`.

Создание окружающего совета

Окружающий совет функционирует как сочетание предшествующих и последующих советов, но с одним существенным отличием: возвращаемое значение можно модифицировать. Кроме того, можно предотвратить фактическое выполнение метода. Это означает, что, применяя окружающий совет, можно, по существу, заменить всю реализацию метода новым кодом. Окружающий совет моделируется в Spring как перехватчик с помощью интерфейса `MethodInterceptor`. Существует немало практических примеров применения окружающего совета, и оказывается, что многие функциональные средства Spring, включая поддержку удаленных заместителей и управление транзакциями, реализованы с помощью перехватчиков методов. Перехватчики методов служат удобным механизмом для профилирования порядка выполнения приложений, а также основанием для примеров, рассматриваемых в этом разделе.

---

[3] `Сгенерирован слабый ключ!`

Вместо того чтобы приводить простой пример для перехвата метода, обратимся снова к первому примеру применения класса `Agent`, где демонстрируется элементарный перехватчик метода для вывода сообщения по обе стороны от вызова метода. Обратите внимание на то, что метод `invoke()` из интерфейса `MethodInterceptor` не обладает таким же набором аргументов, как и у аналогичных методов из интерфейсов `MethodBeforeAdvice` и `AfterReturningAdvice`, т.е. ему не передается цель вызова, вызываемый метод и аргументы для вызываемого метода. Тем не менее доступ к этим данным можно получить через объект типа `MethodInvocation`, который передается методу `invoke()`. В следующем примере будет показано, как это осуществляется на практике.

В данном примере требуется получить основные сведения о производительности методов класса и, в частности, выяснить, сколько времени выполняется тот или иной метод, снабдив проверяемый класс соответствующим советом. Чтобы достичь этой цели, воспользуемся классом `StopWatch`, входящим в состав Spring, а также классом, реализующим интерфейс `MethodInterceptor`, поскольку необходимо каким-то образом запустить хронометр типа `StopWatch` перед вызовом метода и остановить его сразу после вызова метода.

Ниже приведен исходный код класса `WorkerBean`, который мы собираемся профилировать с помощью класса `StopWatch` и окружающего совета.

```
package com.apress.prospring5.ch5;

public class WorkerBean {
  public void doSomeWork(int noOfTimes) {
    for(int x = 0; x < noOfTimes; x++) {
      work();
    }
  }

  private void work() {
    System.out.print("");
  }
}
```

Как видите, класс `WorkerBean` довольно прост. Метод `doSomeWork()` принимает единственный аргумент `noOfTimes` и вызывает метод `work()` указанное в аргументе `noOfTimes` количество раз. Метод `work()` просто содержит фиктивный вызов метода `System.out.print()`, которому передается пустая строка. Благодаря этому предотвращается оптимизация, проводимая компилятором с целью исключить исходный код метода `work()`, а следовательно, этот метод вызывается.

Ниже приведен исходный код класса `ProfilingInterceptor`, в котором класс `StopWatch` используется для профилирования отдельных моментов времени вызова метода. Мы воспользуемся этим классом перехватчика для профилирования упомянутого выше класса `WorkerBean`.

```
package com.apress.prospring5.ch5;

import java.lang.reflect.Method;
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
import org.springframework.util.StopWatch;

public class ProfilingInterceptor
      implements MethodInterceptor {
  @Override
  public Object invoke(MethodInvocation invocation)
        throws Throwable {
    StopWatch sw = new StopWatch();
    sw.start(invocation.getMethod().getName());

    Object returnValue = invocation.proceed();

    sw.stop();
    dumpInfo(invocation, sw.getTotalTimeMillis());
    return returnValue;
  }

  private void dumpInfo(MethodInvocation invocation,
                        long ms) {
    Method m = invocation.getMethod();
    Object target = invocation.getThis();
    Object args = invocation.getArguments();

    System.out.println("Executed method: " + m.getName());
    System.out.println("On object of type: "
                    + target.getClass().getName());
    System.out.println("With arguments:");

    for (int x = 0; x < args.length; x++) {
      System.out.print(" > " + argsx);
    }

    System.out.print("\n");
    System.out.println("Took: " + ms + " ms");
  }
}
```

В методе `invoke()`, который объявлен единственным в интерфейсе `MethodInterceptor`, получается экземпляр хронометра типа `StopWatch`, который сразу же запускается, а затем разрешается продолжение вызова метода через вызов `MethodInvocation.proceed()`. Как только вызываемый метод завершится и зафиксируется возвращаемое им значение, хронометр типа `StopWatch` остановится и общее количество миллисекунд будет передано вместе с объектом типа `MethodInvocation` методу `dumpInfo()`. И, наконец, возвращается экземпляр типа `Object`, полученный в результате вызова `MethodInvocation.proceed()`, чтобы предоставить вызыва-

ющему коду правильное возвращаемое значение. В данном случае мы не стремились каким-то образом нарушить стек вызовов, а просто реализовали код, действующий как перехватчик вызовов метода. При желании можно было бы изменить стек вызовов полностью, переадресовав вызов метода другому объекту или удаленной службе, или же просто реализовать иначе логику метода в самом перехватчике и возвратить другое значение.

Метод `dumpInfo()` выводит на консоль некоторые сведения о вызове метода, а также время, затраченное на его выполнение. В первых трех строках исходного кода метода `dumpInfo()` показано, как воспользоваться объектом типа `MethodInvocation`, чтобы выяснить, какой именно метод был вызван, а также исходную цель вызова и переданные аргументы.

Ниже приведен исходный код класса `ProfilingExample`, который сначала снабжает советом экземпляр типа `WorkerBean` с помощью класса `ProfilingInterceptor`, а затем профилирует метод `doSomeWork()`.

```
package com.apress.prospring5.ch5;

import org.springframework.aop.framework.ProxyFactory;

public class ProfilingDemo {
  public static void main(String... args) {
    WorkerBean bean = getWorkerBean();
    bean.doSomeWork(10000000);
  }

  private static WorkerBean getWorkerBean() {
    WorkerBean target = new WorkerBean();

    ProxyFactory factory = new ProxyFactory();
    factory.setTarget(target);
    factory.addAdvice(new ProfilingInterceptor());

    return (WorkerBean)factory.getProxy();
  }
}
```

Выполнение исходного кода из данного примера на нашей машине дал следующий результат:

```
Executed method: doSomeWork
On object of type: com.apress.prospring5.ch5.WorkerBean
With arguments:
> 10000000
Took: 1139 ms
```

Из приведенного выше результата ясно видно, какой именно метод был выполнен, какой класс послужил целью, какие аргументы были переданы методу и сколько времени (в миллисекундах) отняло его выполнение.

Создание перехватывающего совета

Перехватывающий совет подобен послевозвратному совету тем, что он выполняется после точки соединения, которая всегда является вызовом метода, но перехватывающий совет инициируется лишь в том случае, если метод генерирует исключение. Перехватывающий совет подобен послевозвратному совету и тем, что он имеет незначительный контроль над выполнением программы. Используя перехватывающий совет, нельзя проигнорировать возникшее исключение и вместо этого просто возвратить значение из метода. Единственная корректива, которую можно внести в поток управления программой, состоит в изменении типа сгенерированного исключения. Это довольно эффективный механизм, существенно упрощающий разработку приложений. Допустим, имеется прикладной интерфейс API, генерирующий целый ряд недостаточно определенных исключений. Перехватывающим советом можно снабдить все классы из этого прикладного интерфейса, превратив иерархию исключений в нечто более удобное в управлении и описательное. Разумеется, с помощью перехватывающего совета можно также организовать централизованную регистрацию ошибок в приложении, чтобы сократить объем требующегося для этой цели кода, разбросанного по всему приложению.

Как было схематически показано на рис. 5.2, перехватывающий совет реализуется в интерфейсе `ThrowsAdvice`. В отличие от рассмотренных ранее интерфейсов, в интерфейсе `ThrowsAdvice` не определено ни одного метода, а следовательно, он просто служит в Spring маркерным интерфейсом. Дело в том, что в Spring допускается типизированный перехватывающий совет, позволяющий точно определить, какие именно типы исключений должен перехватывать этот совет. С этой целью в Spring через рефлексию обнаруживаются методы с определенными сигнатурами и, в частности, две отличающихся сигнатуры методов. Это лучше всего продемонстрировать на простом примере. Так, ниже представлен простой компонент Spring Bean с двумя методами, генерирующими разнотипные исключения.

```
package com.apress.prospring5.ch5;

public class ErrorBean {
  public void errorProneMethod() throws Exception {
    throw new Exception("Generic Exception");
  }

  public void otherErrorProneMethod()
              throws IllegalArgumentException {
    throw new IllegalArgumentException(
              "IllegalArgument Exception");
  }
}
```

Ниже приведен исходный код класса `SimpleThrowsAdvice`, отображающего обе сигнатуры методов, которые Spring ищет по перехватывающему совету.

```
package com.apress.prospring5.ch5;

import java.lang.reflect.Method;

import org.springframework.aop.ThrowsAdvice;
import org.springframework.aop.framework.ProxyFactory;

public class SimpleThrowsAdvice implements ThrowsAdvice {
  public static void main(String... args) throws Exception {
    ErrorBean errorBean = new ErrorBean();

    ProxyFactory pf = new ProxyFactory();
    pf.setTarget(errorBean);
    pf.addAdvice(new SimpleThrowsAdvice());

    ErrorBean proxy = (ErrorBean) pf.getProxy();
    try {
      proxy.errorProneMethod();
    } catch (Exception ignored) {
    }

    try {
      proxy.otherErrorProneMethod();
    } catch (Exception ignored) {
    }
  }

  public void afterThrowing(Exception ex) throws Throwable {
    System.out.println("***");
     System.out.println("Generic Exception Capture");
     System.out.println("Caught: "
                     + ex.getClass().getName());
     System.out.println("***\n");
  }

  public void afterThrowing(Method method, Object[] args,
              Object target, IllegalArgumentException ex)
              throws Throwable {
    System.out.println("***");
    System.out.println("IllegalArgumentException Capture");
    System.out.println("Caught: "
                    + ex.getClass().getName());
    System.out.println("Method: " + method.getName());
    System.out.println("***\n");
  }
}
```

Первое, что Spring ищет по перехватывающему совету, — это один или больше открытых методов по имени `afterThrowing()`. Возвращаемый этими методами тип

не особенно важен, хотя мы считаем, что лучше указывать для них возвращаемый тип `void`, поскольку такие методы не могут возвращать сколько-нибудь значащие данные. Первый метод `afterThrowing()` из класса `SimpleThrowsAdvice` принимает единственный аргумент типа `Exception`, в котором можно указать любой тип исключения. Этот метод идеально подходит, когда не так уж и важно, какой именно метод сгенерировал исключение и какие аргументы были ему переданы. Обратите внимание на то, что в данном методе перехватывается исключение типа `Exception` и любые его подтипы, если только для них не предусмотрены свои методы `afterThrowing()`.

Во втором методе `afterThrowing()` объявлены четыре аргумента для указания метода, сгенерировавшего исключение, для аргументов, передаваемых этому методу, а также цели вызова метода. Порядок следования аргументов в данном методе имеет значение, поэтому должны быть указаны все четыре аргумента. Обратите внимание на то, что во втором методе `afterThrowing()` перехватываются исключения типа `IllegalArgumentException` (или его подтипа). Выполнение исходного кода из данного примера дает следующий результат:

```
***
Generic Exception Capture
Caught: java.lang.Exception
***

***
IllegalArgumentException Capture
Caught: java.lang.IllegalArgumentException
Method: otherErrorProneMethod
***
```

Как видите, когда генерируется простое исключение типа `Exception`, вызывается первый метод `afterThrowing()`, а если генерируется исключение типа `IllegalArgumentException`, то вызывается второй метод `afterThrowing()`. Метод `afterThrowing()` вызывается в Spring для каждого исключения типа `Exception`, а затем, как следует из приведенного выше примера, в классе `SimpleThrowsAdvice` выбирается тот метод, сигнатура которого в наибольшей степени соответствует типу исключения. В том же случае, когда у перехватывающего совета имеются два метода `afterThrowing()`, объявленных с одним и тем же исключением типа `Exception`, но один из них принимает единственный аргумент, а другой — четыре аргумента, в Spring вызывается метод `afterThrowing()` с четырьмя аргументами.

Перехватывающий совет приносит пользу во многих случаях. В частности, он позволяет заново классифицировать целые иерархии исключений, а также организовать централизованную регистрацию исключений в приложении. Мы также обнаружили, что перехватывающий совет удобен для отладки готовых приложений, поскольку он дает возможность вводить дополнительный код регистрации, не изменяя прикладной код.

Выбор типа совета

В общем, выбор типа совета определяется требованиями к приложению, но выбирать следует наиболее подходящий тип совета. Иными словами, не следует пользоваться окружающим советом, если лучше подходит предшествующий совет. Как правило, окружающий совет может обеспечить все, что поддерживают остальные три типа советов, но он может оказаться излишним для реализации задуманного. Если применяется наиболее подходящий тип совета, то назначение кода становится более ясным, а также уменьшается вероятность возникновения ошибок. Допустим, в совете должно подсчитываться количество вызовов метода. Если выбрать для этой цели предшествующий совет, то достаточно лишь запрограммировать соответственно счетчик. Но если выбрать окружающий совет, то нужно не забыть о вызове самого метода и возврате значения вызывающему коду. Такие мелочи способствуют проникновению случайных ошибок в приложение. Сохраняя в максимальной степени целенаправленным выбранный тип совета, можно уменьшить вероятность возникновения подобных ошибок.

Советники и срезы в Spring

Во всех рассмотренных до сих пор примерах применялся класс `ProxyFactory`, в котором предоставляется простой способ получения и конфигурирования экземпляров заместителей АОП в специальном пользовательском коде. В частности, метод `ProxyFactory.addAdvice()` служит для конфигурирования совета, предназначенного для заместителя. Этот метод подспудно делегирует свои полномочия методу `addAdvisor()`, получая экземпляр типа `DefaultPointcutAdvisor` и конфигурируя его по срезу, указывающему на все методы. Таким образом, можно считать, что совет применяется ко всем методам целевого объекта. В одних случаях, когда, например, АОП применяется для протоколирования, такое поведение может оказаться желательным. Но в других случаях может возникнуть потребность ограничить круг методов, к которым применяется совет.

Можно было бы, конечно, проверить в самом совете, подходит ли метод, но такому подходу присущи определенные недостатки. Один из них заключается в том, что жесткое кодирование приемлемых в совете методов снижает возможность его неоднократного использования. С помощью срезов можно сконфигурировать перечень методов, к которым будет применяться совет, не внедряя этот код в сам совет. Очевидно, что благодаря этому повышается степень повторного использования совета. Другие недостатки жесткого кодирования списка методов в совете связаны с производительностью. Если снабжаемый советом метод проверяется в самом совете, это происходит всякий раз, когда вызывается любой метод целевого объекта. В итоге производительность приложения снижается. Когда применяются срезы, проверка выполняется по одному разу для каждого метода, а результаты кешируются для дальнейшего использования. Еще один недостаток отказа от применения срезов для огра-

ничения списка снабжаемых советом методов также связан с производительностью и заключается в том, что при создании заместителей каркас Spring может оптимизировать методы, не снабженные советом, чтобы ускорить тем самым их вызовы. Более подробно вопросы, касающиеся такой оптимизации, будут рассматриваться далее в этой главе, когда речь пойдет о заместителях.

Настоятельно рекомендуется не поддаваться искушению жестко кодировать проверки методов в самом совете, а вместо этого пользоваться срезами везде, где только возможно, для контроля над применимостью совета к методам целевого объекта. Но иногда жесткое кодирование проверок в совете оказывается просто необходимым. Вспомните рассмотренный ранее пример употребления послевозвратного совета для перехвата слабых ключей, генерируемых в классе `KeyGenerator`. Такой вид совета тесно связан с классом, который им снабжается, и поэтому было бы вполне разумно проверить в самом совете, к правильному ли типу он применяется. Подобное связывание совета и целевого объекта называется *привязкой к цели*. В общем, срезы следует применять в том случае, когда совет имеет слабую или полностью отсутствующую привязку к цели, т.е. совет может быть применен к любому типу или к широкому диапазону типов. Если же совет имеет привязку к цели, следует проверить правильность его употребления в нем самом. Это поможет уменьшить неприятные ошибки, связанные с неправильным применением совета. Рекомендуется также не снабжать методы советами без особой нужды. Как станет ясно в дальнейшем, это приведет к заметному снижению скорости вызовов, а следовательно, может оказать заметное влияние на общую производительность приложения.

Интерфейс `Pointcut`

Для создания срезов в Spring необходимо реализовать интерфейс `Pointcut`, который определяется следующим образом:

```
package org.springframework.aop;

public interface Pointcut {
  ClassFilter getClassFilter ();
  MethodMatcher getMethodMatcher();
}
```

Как видите, в интерфейсе `Pointcut` определяются два метода, `getClassFilter()` и `getMethodMatcher()`, возвращающие экземпляры типа `ClassFilter` и `MethodMatcher` соответственно. Очевидно, что если решиться реализовать интерфейс `Pointcut`, то придется реализовать и эти методы. Правда, как будет показано в следующем разделе, делать это, как правило, не обязательно, поскольку в Spring на выбор предоставляются реализации интерфейса `Pointcut`, охватывающие большинство, если не все возможные варианты применения.

Когда в Spring выясняется, применим ли интерфейс `Pointcut` к конкретному методу, сначала проверяется, применим ли этот интерфейс к классу данного метода.

И для этой цели используется экземпляр типа `ClassFilter`, который возвращается в результате вызова `Pointcut.getClassFilter()`. Ниже показано, каким образом определяется интерфейс `ClassFilter`.

```
org.springframework.aop;

public interface ClassFilter {
  boolean matches(Class<?> clazz);
}
```

Как видите, в интерфейсе `ClassFilter` определяется единственный метод `matches()`, принимающий экземпляр типа `Class`, который представляет проверяемый класс. Метод `matches()` возвращает логическое значение `true`, если срез применим к классу, а иначе — логическое значение `false`.

Ниже приведен интерфейс `MethodMatcher`, который оказывается более сложным, чем интерфейс `ClassFilter`.

```
package org.springframework.aop;

public interface MethodMatcher {
  boolean matches(Method m, Class<?> targetClass);
  boolean isRuntime();
  boolean matches(Method m, Class<?> targetClass,
                  Object[] args);
}
```

В каркасе Spring поддерживаются две разновидности интерфейса `MethodMatcher`, статическая и динамическая, что определяется по значению, возвращаемому из метода `isRuntime()`. Перед тем как применять интерфейс `MethodMatcher`, в каркасе Spring вызывается метод `isRuntime()`, чтобы выяснить, является ли интерфейс `MethodMatcher` статическим, на что указывает возвращаемое логическое значение `false`, или же динамическим, на что указывает возвращаемое логическое значение `true`.

Для статического среза в Spring метод `matches(Method, Class<T>)` вызывается из интерфейса `MethodMatcher` по одному разу для каждого метода целевого объекта, кешируя возвращаемое значение для последующих обращений к этому методу. Таким образом, проверка применимости каждого метода производится лишь один раз, а последующие обращения к методу не приводят к вызову метода `matches()`.

Для динамических срезов в Spring по-прежнему выполняется статическая проверка с помощью метода `matches(Method, Class<T>)`, когда метод вызывается в первый раз, чтобы определить общую применимость данного метода. Но если в результате статической проверки возвращается логическое значение `true`, то в Spring производится дополнительная проверка с помощью метода `matches(Method, Class<T>, Object[])` при каждом вызове метода. Таким образом, динамический интерфейс `MethodMatcher` позволяет выяснить, должен ли применяться срез, на основании конкретного вызова метода, а не только самого метода. Например, срез

следует применять лишь в том случае, если аргумент представляет значение типа `Integer`, которое больше `100`. В этом случае в методе `matches(Method, Class<T>, Object[])` может быть предусмотрен код для дополнительной проверки аргумента при каждом вызове.

Очевидно, что статические срезы выполняются намного быстрее динамических, поскольку они не требуют дополнительной проверки при каждом вызове. Динамические срезы обеспечивают больший уровень гибкости для принятия решения, следует ли применять совет. В общем, статические срезы рекомендуется применять везде, где только возможно. Но в тех случаях, когда совет влечет за собой значительные издержки, возможно, будет разумнее исключить любые излишние обращения к совету, применяя динамический срез.

В целом вам редко придется создавать собственные реализации интерфейса `Pointcut` заново, потому что в Spring предоставляются абстрактные базовые классы как для статических, так и для динамических срезов. Эти базовые классы, а также другие реализации интерфейса `Pointcut` рассматриваются в последующих разделах.

Доступные реализации интерфейса `Pointcut`

В версии Spring 4.0 предлагаются восемь реализаций интерфейса `Pointcut`: два абстрактных класса, служащие в качестве вспомогательных классов для создания статических и динамических срезов, а также шесть конкретных классов, предназначенных для решения следующих задач.

- Объединение нескольких срезов в одно целое.
- Обработка срезов потока управления.
- Простое сопоставление по имени.
- Определение срезов с помощью регулярных выражений.
- Определение срезов с помощью выражений AspectJ.
- Определение срезов, обнаруживающих конкретные аннотации на уровне классов или методов.

Все восемь реализаций интерфейса `Pointcut` сведены в табл. 5.2.

Таблица 5.2. Сводка реализаций интерфейса `Pointcut`

Класс реализации	Описание
`org.springframework.aop.support.annotation.AnnotationMatchingPointcut`	В данной реализации обнаруживается конкретная аннотация Java в классе или методе. Этот класс требует применения JDK, начиная с версии 5
`org.springframework.aop.aspectj.AspectJExpressionPointcut`	В данной реализации применяется средство связывания AspectJ для вычисления конкретного выражения со срезом, представленного в синтаксисе языка AspectJ

Окончание табл. 5.2

Класс реализации	Описание
`org.springframework.aop.support.ComposablePointcut`	Класс `ComposablePointcut` служит для объединения двух и более срезов с помощью таких операций, как `union()` и `intersection()`
`org.springframework.aop.support.ControlFlowPointcut`	Класс `ControlFlowPointcut` представляет срез, предназначенный для особого случая, который соответствует всем методам в потоке управления другого метода, т.е. любого метода, вызваемого прямо или косвенно в результате выполнения другого метода
`org.springframework.aop.support.DynamicMethodMatcherPointcut`	Данная реализация служит в качестве базового класса для построения динамических срезов
`org.springframework.aop.support.JdkRegexpMethodPointcut`	Данная реализация позволяет определить срезы благодаря поддержке регулярных выражений в версии JDK 1.4. Этот класс требует применения JDK, начиная с версии 1.4
`org.springframework.aop.support.NameMatchMethodPointcut`	С помощью класса `NameMatchMethodPointcut` можно создать срез, выполняющий простое сопоставление со списком имен методов
`org.springframework.aop.support.StaticMethodMatcherPointcut`	Класс `StaticMethodMatcherPointcut` служит в качестве базового класса для построения статических срезов

На рис. 5.3 показана диаграмма, составленная на языке UML (Unified Modeling Language — унифицированный язык моделирования)[4] для классов, реализующих интерфейс `Pointcut`.

Применение класса `DefaultPointcutAdvisor`

Прежде чем воспользоваться любой реализацией интерфейса `Pointcut`, придется реализовать интерфейс `Advisor`, а точнее говоря, интерфейс `PointcutAdvisor`. Как упоминалось в разделе "Аспекты в Spring", интерфейс `Advisor` служит для представления в Spring аспекта, связывающего совет со срезами и определяющего, какие именно методы должны снабжаться советом и как это должно делаться. В каркасе Spring предоставляется несколько реализаций интерфейса `PointcutAdvisor`, но мы рассмотрим пока что одну из них в классе `DefaultPointcutAdvisor`. Класс

[4] Язык UML крайне важен для разработки программного обеспечения, поскольку он заметно упрощает логику приложения и позволяет наглядно представить и легко выявить недостатки в ней еще до написания кода. Он служит также для документирования проекта, который необходимо представить новым членам команды разработчиков, чтобы они могли работать как можно более продуктивно. Подробнее ознакомиться с языком UML можно по адресу `www.uml.org/`.

`DefaultPointcutAdvisor` служит простой реализацией интерфейса `Pointcut Advisor` для связывания одного среза типа `Pointcut` с одним советом типа `Advice`.

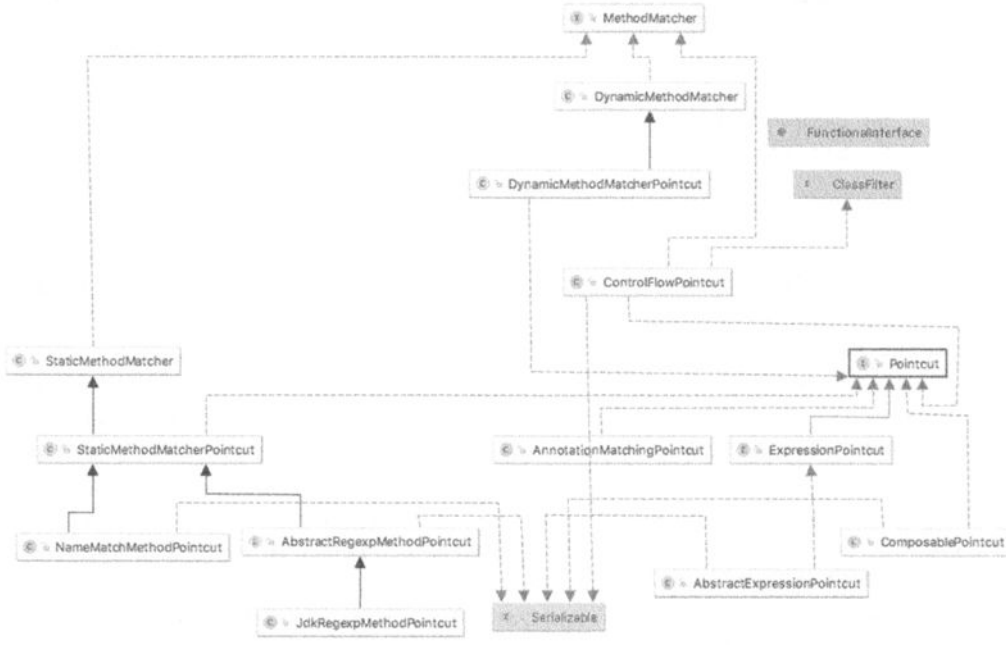


Рис. 5.3. Иерархия классов, реализующих интерфейс **`Pointcut`**, схематически представленная на языке UML в IDE Intellij IDEA

Создание статического среза с помощью класса `StaticMethodMatcherPointcut`

В этом разделе мы покажем, как создать простой статический срез, расширив абстрактный класс `StaticMethodMatcherPointcut`. А поскольку класс `Static MethodMatcherPointcut`, в свою очередь, расширяет класс `StaticMethod Matcher`, который также является абстрактным и реализует интерфейс `Method Matcher`, придется реализовать и метод `matches(Method, Class<?>)`, а остальная часть интерфейса `Pointcut` реализуется автоматически. И хотя это единственный метод, который должен быть реализован в данном интерфейсе при расширении класса `StaticMethodMatcherPointcut`, возможно, потребуется переопределить и метод `getClassFilter()`, как демонстрируется в рассматриваемом здесь примере, чтобы советом были снабжены только методы подходящего типа. Для данного примера воспользуемся двумя классами, `GoodGuitarist` и `GreatGuitarist`, в которых определены одинаковые методы, реализующие один и тот же метод `sing()` из интерфейса `Singer`:

```
package com.apress.prospring5.ch5;
import com.apress.prospring5.ch2.common.Singer;
```

```
public class GoodGuitarist implements Singer {
  @Override public void sing() {
    System.out.println("Who says I can't be free \n"
             + "From all of the things that I used to be");
  }
}

public class GreatGuitarist implements Singer {
  @Override public void sing() {
    System.out.println("I shot the sheriff, \n"
                    + "But I did not shoot the deputy");
  }
}
```

В данном примере нам нужно воспользоваться возможностью создать заместитель обоих классов с помощью класса `DefaultPointcutAdvisor`, но в то же время применить совет только к методу `sing()` из класса `GoodGuitarist`. С этой целью создадим приведенный ниже класс `SimpleStaticPointcut`.

```
package com.apress.prospring5.ch5;

import java.lang.reflect.Method;
import org.springframework.aop.ClassFilter;
import org.springframework.aop.support
          .StaticMethodMatcherPointcut;

public class SimpleStaticPointcut
       extends StaticMethodMatcherPointcut {
  @Override
  public boolean matches(Method method, Class<?> cls) {
    return ("sing".equals(method.getName()));
  }

  @Override
  public ClassFilter getClassFilter() {
    return cls -> (cls == GoodGuitarist.class);
  }
}
```

Как видите, мы реализовали метод `matches(Method, Class<?>)`, требующийся для абстрактного класса `StaticMethodMatcher`. Его реализация просто возвращает логическое значение `true`, если метод называется `sing`, а иначе — логическое значение `false`. В приведенном выше примере кода создание анонимного класса, реализующего интерфейс `ClassFilter` в методе `getClassFilter()`, скрывается благодаря применению лямбда-выражения. В развернутом виде это лямбда-выражение выглядит следующис образом:

```
public ClassFilter getClassFilter() {
  return new ClassFilter() {
```

```
    public boolean matches(Class<?> cls) {
      return (cls == GoodGuitarist.class);
    }
  };
}
```

Обратите внимание на то, что мы переопределили также метод `getClassFilter()`, чтобы возвратить экземпляр типа `ClassFilter`, метод `matches()` которого возвращает логическое значение `true` только для класса `GoodGuitarist`. С помощью этого статического среза мы утверждаем, что с критерием поиска совпадут только методы из класса `GoodGuitarist`, а точнее, только метод `sing()`.

Ниже приведен исходный код класса `SimpleAdvice`, который выводит сообщения по обе стороны от вызова метода.

```
package com.apress.prospring5.ch5;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class SimpleAdvice implements MethodInterceptor {
  @Override
  public Object invoke(MethodInvocation invocation)
                throws Throwable {
    System.out.println(">> Invoking "
                    + invocation.getMethod().getName());
    Object retVal = invocation.proceed();
    System.out.println(">> Done\n");
    return retVal;
  }
}
```

Ниже представлен исходный код простого тестового приложения для данного примера, где получается экземпляр класса `DefaultPointcutAdvisor` с помощью классов `SimpleAdvice` и `SimpleStaticPointcut`. А поскольку оба эти класса реализуют один и тот же интерфейс, то заместители, очевидно, могут быть созданы на основании этого интерфейса, а не конкретных классов.

```
package com.apress.prospring5.ch5;

import com.apress.prospring5.ch2.common.Singer;
import org.aopalliance.aop.Advice; import org
          .springframework.aop.Advisor;
import org.springframework.aop.Pointcut;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop
          .support.DefaultPointcutAdvisor;

public class StaticPointcutDemo {
  public static void main(String... args) {
```

```
    GoodGuitarist johnMayer = new GoodGuitarist();
    GreatGuitarist ericClapton = new GreatGuitarist();

    Singer proxyOne;
    Singer proxyTwo;

    Pointcut pc = new SimpleStaticPointcut();
    Advice advice = new SimpleAdvice();
    Advisor advisor =
            new DefaultPointcutAdvisor(pc, advice);

    ProxyFactory pf = new ProxyFactory();
    pf.addAdvisor(advisor);
    pf.setTarget(johnMayer);
    proxyOne = (Singer)pf.getProxy();

    pf = new ProxyFactory();
    pf.addAdvisor(advisor);
    pf.setTarget(ericClapton);
    proxyTwo = (Singer)pf.getProxy();
    proxyOne.sing();
    proxyTwo.sing();
  }
}
```

Обратите внимание на то, что экземпляр типа `DefaultPointcutAdvisor` применяется далее для создания двух заместителей объектов типа `GoodGuitarist`: одного — для экземпляра `johnMayer`, а другого — для экземпляра `ericClapton`. И, наконец, метод `sing()` вызывается для обоих заместителей. Выполнение исходного кода из данного примера дает следующий результат:

```
>> Invoking sing
Who says I can't be free
From all of the things that I used to be
>> Done

I shot the sheriff,
But I did not shoot the deputy[5]
```

Как видите, совет типа `SimpleAdvice` был применен только к методу `sing()` из класса `GoodGuitarist`, как и предполагалось. Ограничение круга методов, к кото-

---

[5] >> Вызов метода sing()
Кто говорит, что не могу быть свободным
От всего того, чем я был раньше
>> Готово

Я убил шерифа,
Но не убивал депутата.

рым применяется совет, реализуется довольно просто и, как будет показано далее при обсуждении вариантов заместителей, служит ключевым фактором для достижения максимальной производительности в приложении.

Создание динамического среза с помощью класса DynamicMethodMatcherPointcut

Порядок создания динамического среза не особенно отличается от создания статического среза, поэтому в рассматриваемом здесь примере мы создадим динамический срез для класса, исходный код которого приведен ниже. В данном примере требуется снабдить советом только метод `foo()`, но сделать это, в отличие от предыдущего примера, в тот момент, когда данному методу передается аргумент `int` со значением, которое не равно `100`.

```
package com.apress.prospring5.ch5;

public class SampleBean {
  public void foo(int x) {
    System.out.println("Invoked foo() with: " + x);
  }

  public void bar() {
    System.out.println("Invoked bar()");
  }
}
```

Аналогично статическим срезам, для создания динамических срезов в каркасе Spring предоставляется удобный базовый класс `DynamicMethodMatcherPointcut`. В этом классе имеется единственный абстрактный метод `matches(Method, Class<?>, Object[])` (благодаря реализуемому в нем интерфейсу `MethodMatcher`), который должен быть реализован, но, как станет ясно в дальнейшем, благоразумно также реализовать метод `matches(Method, Class<?>)`, чтобы управлять поведением статических проверок. Ниже приведен исходный код класса `SimpleDynamicPointcut`.

```
package com.apress.prospring5.ch5;

import java.lang.reflect.Method;
import org.springframework.aop.ClassFilter;
import org.springframework.aop
          .support.DynamicMethodMatcherPointcut;

public class SimpleDynamicPointcut
             extends DynamicMethodMatcherPointcut {
  @Override
  public boolean matches(Method method, Class<?> cls) {
    System.out.println("Static check for " + method.getName());
```

```
    return ("foo".equals(method.getName()));
  }

  @Override
  public boolean matches(Method method, Class<?> cls, Object[] args) {
    System.out.println("Dynamic check for "
                       + method.getName());
    int x = ((Integer) args0).intValue();
    return (x != 100);
  }

  @Override
  public ClassFilter getClassFilter() {
    return cls -> (cls == SampleBean.class);
  }
}
```

Нетрудно заметить, что в приведенном выше примере кода метод `getClassFilter()` переопределяется таким же образом, как и в примере из предыдущего раздела. Этим устраняется необходимость проверять класс в методах сопоставления имен методов, что особенно важно для динамической проверки. И хотя обязательной является реализация лишь динамической проверки, в данном примере реализуется и статическая проверка. Дело в том, что метод `bar()`, как известно, вообще не будет снабжен советом. И если указать на этот факт с помощью статической проверки, то в Spring так и не придется выполнять динамическую проверку для данного метода. Ведь если статическая проверка метода реализована, то каркас Spring воспользуется сначала именно ею, и если она выявит в итоге несоответствие, то дальнейшая динамическая проверка не будет произведена. Кроме того, результат статической проверки кешируется для повышения производительности. Но если пренебречь статической проверкой, то Spring будет выполнять динамическую проверку при каждом вызове метода `bar()`. На практике рекомендуется выполнять проверку класса в методе `getClassFilter()`, проверку метода — в методе `matches(Method, Class<?>)`, а проверку аргумента — в методе `matches(Method, Class<?>, Object[])`. Благодаря этому срез станет намного более простым для понимания и сопровождения, а показатели производительности улучшатся.

Из метода `matches(Method, Class<?>, Object[])` возвращается логическое значение `true`, если значение, переданное в аргументе типа `int` методу `foo()`, не равно `100`, а иначе — логическое значение `false`. Обратите внимание на то, что когда производится динамическая проверка, то известно, что она касается метода `foo()`, поскольку ни один другой метод не прошел статическую проверку. Ниже приведен пример данного среза в действии.

```
package com.apress.prospring5.ch5;

import org.springframework.aop.Advisor;
import org.springframework.aop.framework.ProxyFactory;
```

```
import org.springframework.aop
          .support.DefaultPointcutAdvisor;

public class DynamicPointcutDemo {
  public static void main(String... args) {
    SampleBean target = new SampleBean();

    Advisor advisor = new DefaultPointcutAdvisor(
      new SimpleDynamicPointcut(), new SimpleAdvice());

    ProxyFactory pf = new ProxyFactory();
    pf.setTarget(target);
    pf.addAdvisor(advisor);
    SampleBean proxy = (SampleBean)pf.getProxy();

    proxy.foo(1);
    proxy.foo(10);
    proxy.foo(100);

    proxy.bar();
    proxy.bar();
    proxy.bar();
  }
}
```

В данном случае используется тот же самый класс совета, что и в примере со статическим срезом. Но в данном примере должны быть снабжены советом только два первых вызова метода `foo()`. Динамическая проверка препятствует тому, чтобы советом был снабжен третий вызов метода `foo()`, а статическая проверка не позволяет снабдить советом метод `bar()`. Выполнение исходного кода из данного примера дает следующий результат:

```
Static check for[6] bar
Static check for foo
Static check for toString
Static check for clone
Static check for foo
Dynamic check for[7] foo
>> Invoking foo
Invoked foo() with: 1
>> Done
Dynamic check for foo
>> Invoking foo
Invoked foo() with: 10
>> Done
```

[6] Статическая проверка метода...

[7] Динамическая проверка метода...

```
Dynamic check for foo
Invoked foo() with: 100
Static check for bar
Invoked bar()
Invoked bar()
Invoked bar()
```

Как и следовало ожидать, советом были снабжены только первые два обращения к методу `foo()`. Благодаря статической проверке метода `bar()` ни один из его вызовов не подвергался динамической проверке. Любопытно также отметить, что метод `foo()` подвергся двум статическим проверкам: одной — на начальной стадии, когда проверялись все методы, а другой — при его вызове в первый раз. Как видите, динамические проверки с помощью срезов обеспечивают бо́льшую степень гибкости, чем их статические аналоги, но из-за дополнительных издержек на стадии выполнения ими следует пользоваться лишь в случае крайней необходимости.

Простое сопоставление имен методов

Зачастую при создании среза требуется выполнять сопоставление на основе лишь имени метода, игнорируя его сигнатуру и возвращаемый тип. В таком случае можно избежать создания подкласса, производного от класса `StaticMethodMatcher Pointcut`, и применять вместо этого его подкласс `NameMatchMethodPointcut` для сопоставления со списком имен методов.

Когда применяется класс `NameMatchMethodPointcut`, сигнатура метода во внимание не принимается. Так, если имеются вызовы метода `sing()` и `sing(guitar)`, то оба они совпадают с именем `sing`.

Ниже приведен исходный код класса `GrammyGuitarist`. Это еще одна реализация интерфейса `Singer`, поскольку она представляет обладателя награды "Грэмми", поющего под аккомпанемент гитары, а в свободное от пения время пользующегося своим голосом для общения с другими людьми.

```
package com.apress.prospring5.ch5;

import com.apress.prospring5.ch2.common.Guitar;
import com.apress.prospring5.ch2.common.Singer;

public class GrammyGuitarist implements Singer {

  @Override public void sing() {
    System.out.println("sing: Gravity is working "
                     + "against me\n" + "And gravity "
                     + "wants to bring me down");
    }

  public void sing(Guitar guitar) {
    System.out.println("play: " + guitar.play());
  }
```

```
  public void rest(){
    System.out.println("zzz");
  }

  public void talk(){
    System.out.println("talk");
  }
}

// chapter02/hello-world/src/main/java/com/apress
            /prospring5/ch2/common/Guitar.java
package com.apress.prospring5.ch2.common;

public class Guitar {
  public String play(){
    return "G C G C Am D7";
  }
}
```

В данном примере требуется сопоставить вызовы методов `sing()`, `sing(Guitar)` и `rest()` с помощью класса `NameMatchMethodPointcut`. Это приводит к сопоставлению с именами `sing` и `rest`, как демонстрируется в следующем фрагменте кода:

```
package com.apress.prospring5.ch5;

import com.apress.prospring5.ch2.common.Guitar;
import org.springframework.aop.Advisor;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop
          .support.DefaultPointcutAdvisor;
import org.springframework.aop
          .support.NameMatchMethodPointcut;

public class NamePointcutDemo {

  public static void main(String... args) {
    GrammyGuitarist johnMayer = new GrammyGuitarist();
    NameMatchMethodPointcut pc =
                  new NameMatchMethodPointcut();
    pc.addMethodName("sing");
    pc.addMethodName("rest");

    Advisor advisor = new DefaultPointcutAdvisor(pc,
                      new SimpleAdvice());
    ProxyFactory pf = new ProxyFactory();
    pf.setTarget(johnMayer);
    pf.addAdvisor(advisor);
```

```
    GrammyGuitarist proxy =
                    (GrammyGuitarist) pf.getProxy();
    proxy.sing();
    proxy.sing(new Guitar());
    proxy.rest();
    proxy.talk();
  }
}
```

В данном случае нет потребности в построении отдельного класса для среза, поскольку достаточно получить экземпляр класса `NameMatchMethodPointcut`. Обратите внимание на то, что в срез было введено два имени, `sing` и `rest`, с помощью метода `addMethodName()`. Выполнение исходного кода из данного примера дает приведенный ниже результат. Как и следовало ожидать, вызовы методов `sing`, `sing(Guitar)` и `rest()` снабжаются советом благодаря срезу, тогда как метод `talk()` лишается совета.

```
>> Invoking sing
sing: Gravity is working against me
And gravity wants to bring me down
>> Done

>> Invoking sing
play: G C G C Am D7
>> Done

>> Invoking rest
zzz
>> Done

talk[8]
```

[8] >> Вызов метода sing()
пение: Тяготение действует против меня
И хочет свалить меня
>> Готово

>> Вызов метода sing()
игра на гитаре: соль мажор, до мажор, соль мажор, до мажор,
ля минор, доминантсептаккорд
>> Готово

>> Вызов метода rest()
zzz
>> Готово

Разговор

Создание срезов с помощью регулярных выражений

В предыдущем разделе пояснялось, как выполнить простое сопоставление с предварительно определенным списком имен методов. Но как быть в том случае, если заранее известны имена не всех методов, а только шаблон, по которому эти имена составляются? Что, если требуется, например, сопоставление со всеми методами, имена которых начинаются на **get**? В таком случае можно воспользоваться срезом типа `JdkRegexpMethodPointcut`, который позволяет выполнить сопоставление на основе регулярного выражения. Ниже приведен еще один вариант класса `Guitarist`, содержащий три метода.

```
package com.apress.prospring5.ch5;

import com.apress.prospring5.ch2.common.Singer;

public class Guitarist implements Singer {
  @Override public void sing() {
    System.out.println("Just keep me where the light is");
  }

  public void sing2() {
    System.out.println("Oh gravity, stay the hell away from me");
  }

  public void rest() {
    System.out.println("zzz");
  }
}
```

Используя срез на основе регулярного выражения, можно реализовать сопоставление со всеми методами данного класса, имена которых начинаются на `string`:

```
package com.apress.prospring5.ch5;

import org.springframework.aop.Advisor;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop
          .support.DefaultPointcutAdvisor;
import org.springframework.aop
          .support.JdkRegexpMethodPointcut;

public class RegexpPointcutDemo {
  public static void main(String... args) {
    Guitarist johnMayer = new Guitarist();

    JdkRegexpMethodPointcut pc =
              new JdkRegexpMethodPointcut();
```

```
    pc.setPattern(".*sing.*");
    Advisor advisor = new DefaultPointcutAdvisor(pc,
                      new SimpleAdvice());

    ProxyFactory pf = new ProxyFactory();
    pf.setTarget(johnMayer);
    pf.addAdvisor(advisor);
    Guitarist proxy = (Guitarist) pf.getProxy();

    proxy.sing();
    proxy.sing2();
    proxy.rest();
  }
}
```

Создавать класс для среза совсем не обязательно — достаточно получить экземпляр класса `JdkRegexpMethodPointcut` и указать шаблон для сопоставления. Обратите также внимание на шаблон. При сопоставлении имен методов в Spring употребляется полностью уточненное имя метода, т.е. имя метода `sing1()` сопоставляется с именем `com.apress.prospring5.ch5.Guitarist.sing1`, чем и объясняется наличие знаков `.*` в начале шаблона. Это довольно эффективный механизм, поскольку он позволяет выполнять сопоставление со всеми методами из указанного пакета, даже не зная точно перечень всех входящих в него классов, а также имен их методов. Выполнение исходного кода из данного примера дает приведенный ниже результат.

Как и следовало ожидать, советом были снабжены только методы `sing()` и `sing2()`, поскольку метод `rest()` не совпал с шаблоном регулярного выражения.

```
>> Invoking sing
Just keep me where the light is
>> Done

>> Invoking sing2
Oh gravity, stay the hell away from me
>> Done[9]
```

Создание срезов с помощью выражений AspectJ

Помимо регулярных выражений JDK, срезы можно объявлять и средствами языка AspectJ для выражений со срезами. Как будет показано далее в этой главе, когда срез объявляется в пространстве имен `aop` при конфигурировании в формате XML,

[9] >> Вызов метода sing()
Оставьте меня лишь там, где свет
>> Готово

>> Вызов метода sing2()
О, тяготение, прочь от меня>> Готово

в Spring по умолчанию применяется язык AspectJ. Более того, обращаясь к поддержке АОП в Spring с помощью аннотаций `@AspectJ`, необходимо также применять язык AspectJ. Таким образом, для объявления срезов с помощью выражений лучше всего подойдут языковые средства AspectJ. Для определения срезов на языке выражений AspectJ в Spring предоставляется класс `AspectJExpressionPointcut`. Чтобы воспользоваться в Spring выражениями со срезами на языке AspectJ, в путь к классам текущего проекта необходимо ввести файлы двух библиотек AspectJ: `aspectjrt.jar` и `aspectjweaver.jar`. Зависимости и версии этих библиотек настраиваются в главном файле конфигурации `build.gradle` (в данном случае как зависимости для всех модулей проекта `chapter05`).

```
ext {
    aspectjVersion = '1.9.0.BETA-5'
...
    misc = [
...
    Aspectjweaver    :
          "org.aspectj:aspectjweaver:$aspectjVersion",
    Aspectjrt    : "org.aspectj:aspectjrt:$aspectjVersion"
]
...
```

Если вернуться к предыдущей реализации класса `Guitarist`, то те же самые функциональные средства, реализованные с помощью регулярных выражений из комплекта JDK, можно реализовать и посредством выражений AspectJ. Ниже приведен соответствующий исходный код.

```
package com.apress.prospring5.ch5;

import org.springframework.aop.Advisor;
import org.springframework.aop
          .aspectj.AspectJExpressionPointcut;
import org.springframework.aop
          .framework.ProxyFactory;
import org.springframework.aop
          .support.DefaultPointcutAdvisor;

public class AspectjexpPointcutDemo {
  public static void main(String... args) {
    Guitarist johnMayer = new Guitarist();

    AspectJExpressionPointcut pc =
                    new AspectJExpressionPointcut();
    pc.setExpression("execution(* sing*(..))");
    Advisor advisor = new DefaultPointcutAdvisor(pc,
                     new SimpleAdvice());

    ProxyFactory pf = new ProxyFactory();
```

```
        pf.setTarget(johnMayer);
        pf.addAdvisor(advisor);
        Guitarist proxy = (Guitarist) pf.getProxy();

        proxy.sing();
        proxy.sing2();
        proxy.rest();
    }
}
```

Обратите внимание на вызов метода `setExpression()` из класса `AspectJExpressionPointcut` для установки критерия совпадения. Выражение `execution(* sing*(..))` означает, что совет должен применяться к любым выполняемым методам, имена которых начинаются на `sing`, которые принимают любые аргументы и возвращают значение любого типа. Выполнение приведенного выше кода даст то же результат, что и в предыдущем примере, где употреблялись регулярные выражения из комплекта JDK.

Создание срезов, совпадающих с аннотациями

Если ваше приложение основано на аннотациях, то у вас может возникнуть желание задействовать собственные аннотации для определения срезов, т.е. применить логику совета ко всем методам или типам с конкретными аннотациями. В каркасе Spring для определения срезов с помощью аннотаций предоставляется класс `AnnotationMatchingPointcut`. Обратимся снова к предыдущему примеру, чтобы показать, как употребить аннотацию в качестве среза.

Определим сначала аннотацию `@AdviceRequired`, которая послужит в дальнейшем для объявления среза. Ниже приведен интерфейс для определения этой аннотации.

```
package com.apress.prospring5.ch5;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

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

В приведенном выше фрагменте кода интерфейс объявлен как аннотация, причем аннотация `@interface` обозначает тип, тогда как аннотация `@Target` определяет, что объявляемую аннотацию `@AdviceRequired` можно применять на уровне типа или на уровне метода. Ниже приведен еще один вариант реализации класса `Guitarist`, один из методов которого снабжен данной аннотацией.

```
package com.apress.prospring5.ch5;

import com.apress.prospring5.ch2.common.Guitar;
import com.apress.prospring5.ch2.common.Singer;

public class Guitarist implements Singer {
  @Override public void sing() {
    System.out.println("Dream of ways to throw "
                    + "it all away");
  }

  @AdviceRequired
  public void sing(Guitar guitar) {
    System.out.println("play: " + guitar.play());
  }

  public void rest(){
    System.out.println("zzz");
  }
}
```

В следующем фрагменте кода приведена тестовая программа для рассматриваемого здесь примера приложения:

```
package com.apress.prospring5.ch5;

import com.apress.prospring5.ch2.common.Guitar;
import org.springframework.aop.Advisor;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop
          .support.DefaultPointcutAdvisor;
import org.springframework.aop.support
          .annotation.AnnotationMatchingPointcut;

public class AnnotationPointcutDemo {
  public static void main(String... args) {
    Guitarist johnMayer = new Guitarist();

    AnnotationMatchingPointcut pc =
              AnnotationMatchingPointcut
              .forMethodAnnotation(AdviceRequired.class);
    Advisor advisor = new DefaultPointcutAdvisor(pc,
                      new SimpleAdvice());

    ProxyFactory pf = new ProxyFactory();
    pf.setTarget(johnMayer);
    pf.addAdvisor(advisor);
    Guitarist proxy = (Guitarist) pf.getProxy();
```

```
    proxy.sing(new Guitar());
    proxy.rest();
  }
}
```

В приведенном выше коде получается экземпляр класса `AnnotationMatchingPointcut` в результате вызова его статического метода `forMethodAnnotation()`, которому передается тип аннотации. Это означает, что нам требуется применить совет ко всем методам, снабженным заданной аннотацией. Аннотации можно также указать для применения на уровне типа, вызвав метод `forClassAnnotation()`. Ниже приведен результат, выводимый при выполнении программы из данного примера. Как видите, в данном примере был аннотирован только метод `sing()`, и поэтому только он и был снабжен советом.

```
>> Invoking sing
play: G C G C Am D7
>> Done

zzz
```

Удобные реализации интерфейса `Advisor`

Для многих реализаций интерфейса `Pointcut` в Spring предлагается также удобная реализация интерфейса `Advisor`, которая действует в качестве среза. Например, в предыдущем примере вместо классов `NameMatchMethodPointcut` и `DefaultPointcutAdvisor` можно было бы просто применить класс `NameMatchMethodPointcutAdvisor`, как показано ниже.

```
package com.apress.prospring5.ch5;
...
import org.springframework.aop.support
          .NameMatchMethodPointcutAdvisor;

public class NamePointcutUsingAdvisor {
  public static void main(String... args) {
    GrammyGuitarist johnMayer = new GrammyGuitarist();

    NameMatchMethodPointcutAdvisor advisor =
             new NameMatchMethodPointcutAdvisor(
                 new SimpleAdvice());
    advisor.setMappedNames("sing");
    advisor.setMappedNames("rest");

    ProxyFactory pf = new ProxyFactory();
    pf.setTarget(johnMayer);
    pf.addAdvisor(advisor);

    GrammyGuitarist proxy = (GrammyGuitarist) pf.getProxy();
```

```
        proxy.sing();
        proxy.sing(new Guitar());
        proxy.rest();
        proxy.talk();
    }
}
```

Вместо получения экземпляра типа `NameMatchMethodPointcut` в данном случае конфигурируются подробности среза с помощью экземпляра типа `NameMatchMethodPointcutAdvisor`. При этом класс `NameMatchMethodPointcutAdvisor` действует как советник и как срез.

С подробным описанием различных реализаций интерфейса `Advisor` можно ознакомиться в документации формата Javadoc на пакет `org.springframework.aop.support`. Оба представленных здесь подхода к реализации срезов не проявляют заметных отличий в производительности, кроме чуть меньшего объема кода во втором случае. Мы предпочитаем придерживаться первого подхода, поскольку считаем, что он дает более ясный код. Но в конечном счете выбор конкретного подхода зависит от личных предпочтений.

Общее представление о заместителях

До сих пор мы только вскользь упоминали о заместителях, генерируемых средствами класса `ProxyFactory`. В частности, мы отметили, что в Spring доступны два типа заместителей: те, что создаются с помощью класса `Proxy` из комплекта JDK, и те, что создаются с помощью класса `Enhancer` из библиотеки CGLIB. В чем же отличия этих двух типов заместителей и для чего они применяются в Spring? В этом разделе мы подробно рассмотрим отличия между обоими типами заместителей.

Основное предназначение заместителей — перехват вызовов методов и выполнение там, где это требуется, цепочек вызовов советов, применяемых к отдельному методу. Управление и вызов совета в основном не зависит от заместителя и возложено на каркас АОП в Spring. Тем не менее заместитель отвечает за перехват вызовов всех методов и их передачу по мере надобности каркасу АОП для применения совета.

Помимо этих базовых функциональных возможностей, в заместителе должен также поддерживаться ряд дополнительных средств. Заместитель можно сконфигурировать таким образом, чтобы он был доступен через абстрактный класс `AopContext`. Это позволит извлечь заместитель и вызвать оснащенные советом методы из целевого объекта. Заместитель отвечает за надлежащий доступ к своему классу, если такой режим активизирован через вызов метода `ProxyFactory.setExposeProxy()`. Кроме того, все классы заместителей по умолчанию реализуют интерфейс `Advised`, который, среди прочего, позволяет изменять цепочку вызовов совета после создания заместителя. Заместитель должен также гарантировать, чтобы любой метод, который возвращает ссылку `this` (т.е. возвращает замещаемый целевой объект), фактически возвращал заместитель, а не целевой объект. Как видите, типичный заместитель дол-

жен выполнять немало функций, и поэтому вся необходимая логика реализована в заместителях из комплекта JDK и библиотеки CGLIB.

Применение динамических заместителей из комплекта JDK

Заместители из комплекта JDK — это самый элементарный тип заместителей, доступных в Spring. В отличие от заместителей из библиотеки CGLIB, заместители из комплекта JDK могут генерировать соответствующие заместители только для интерфейсов, но не для классов. Таким образом, любой объект, для которого требуется заместитель, должен реализовывать хотя бы один интерфейс, а получающийся в итоге заместитель будет представлять объект, реализующий данный интерфейс. Абстрактная схема такого заместителя приведена на рис. 5.4.

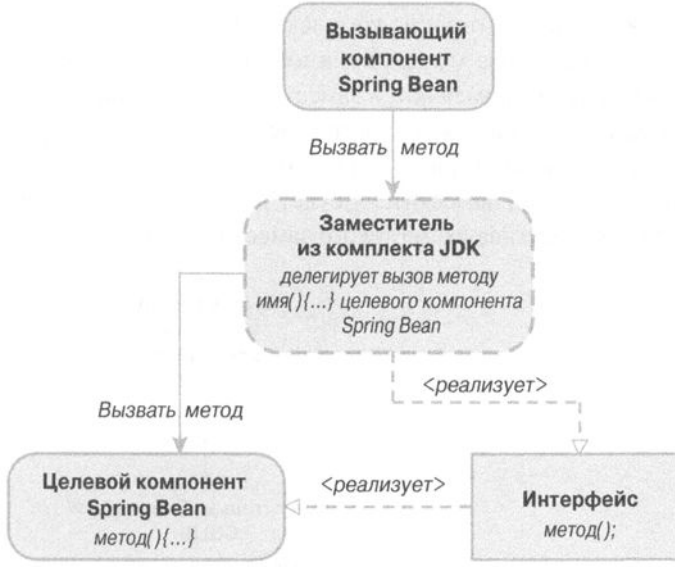


Рис. 5.4. Абстрактная схема заместителя из комплекта JDK

В общем, применение интерфейсов вместо классов является удачным проектным решением, хотя это не всегда возможно, особенно если приходится работать с унаследованным или сторонним кодом. В таком случае должен применяться заместитель из библиотеки CGLIB. Если же используется заместитель из комплекта JDK, то все вызовы методов перехватываются виртуальной машиной JVM и направляются методу `invoke()` данного заместителя. В методе `invoke()` выясняется, снабжен ли вызываемый метод советом (по правилам, определяемым срезом), и если это так, то приводится в действие цепочка вызовов совета, а затем с помощью рефлексии вызывается сам метод. Кроме того, метод `invoke()` выполняет всю необходимую логику, обсуждавшуюся в предыдущем разделе.

Заместитель из комплекта JDK не разделяет методы на снабженные и не снабженные советом вплоть до вызова метода `invoke()`. Это означает, что для методов, не снабженных советом, метод `invoke()` по-прежнему вызывается для заместителя, все проверки выполняются, а сами методы вызываются через рефлексию. Очевидно, что каждый вызов метода влечет за собой дополнительные издержки во время выполнения, несмотря на то, что заместитель зачастую не реализует никакой дополнительной обработки, кроме вызова метода, не снабженного советом, через рефлексию. Чтобы сообщить классу `ProxyFactory` о необходимости использовать заместитель из комплекта JDK, следует указать список интерфейсов для заместителя с помощью метода `setInterfaces()` (из класса `AdvisedSupport`, косвенно расширяемого классом `ProxyFactory`).

Применение заместителей из библиотеки CGLIB

Если применяются заместители из комплекта JDK, то все решения относительно обработки конкретного вызова метода принимаются во время выполнения при каждом таком вызове. А если применяются заместители из библиотеки CGLIB, то в ней динамически генерируется байт-код нового класса для каждого заместителя, а сгенерированные ранее классы повторно используются там, где это возможно. Получающийся в данном случае тип заместителя будет подклассом, производным от класса целевого объекта. Абстрактная схема такого заместителя приведена на рис. 5.5.

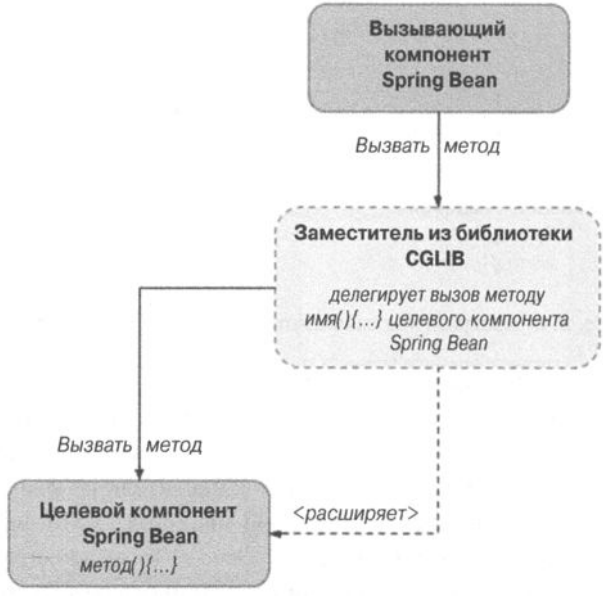


Рис. 5.5. Абстрактная схема заместителя из библиотеки CGLIB

Когда заместитель из библиотеки CGLIB создается в первый раз, библиотека CGLIB запрашивает в Spring, каким образом требуется вызывать каждый метод. Это

означает, что многие решения, принимаемые при каждом вызове метода `invoke()` для заместителя из комплекта JDK, принимаются для заместителя из библиотеки CGLIB только один раз. А поскольку библиотека CGLIB генерирует конкретный байт-код, то вызывать методы можно с намного большей гибкостью. Например, заместитель из библиотеки CGLIB генерирует соответствующий байт-код для непосредственного вызова любых методов, не снабженных советом, сокращая тем самым издержки, вносимые заместителем. Кроме того, заместитель из библиотеки CGLIB определяет, может ли метод возвратить ссылку `this`, и если это невозможно, то вызов метода делается напосредственно, снова сокращая издержки на стадии выполнения.

В заместителе из библиотеки CGLIB поддерживаются также цепочки вызовов фиксированных советов, хотя делается это иначе, чем в заместителе из комплекта JDK. Цепочка вызовов фиксированного совета останется неизменной после того, как будет сформирован заместитель. По умолчанию советники и сам совет можно изменить даже после создания заместителя, хотя такое требуется редко. Заместитель из библиотеки CGLIB обрабатывает цепочки вызовов фиксированных советов особым образом, сокращая издержки на выполнение цепочки вызовов совета.

Сравнение производительности заместителей

До сих пор мы обсуждали в свободной форме отличия в реализации разных типов заместителей. А в этом разделе мы проведем простой тест для сравнения производительности заместителей из библиотеки CGLIB и комплекта JDK.

Создадим класс `DefaultSimpleBean`, чтобы использовать его в качестве целевого объекта для замещения. Ниже приведен интерфейс `SimpleBean` и реализующий его класс `DefaultSimpleBean`.

```
ppackage com.apress.prospring5.ch5;

public interface SimpleBean {
  void advised();
  void unadvised();
}

public class DefaultSimpleBean implements SimpleBean {
  private long dummy = 0;

  @Override
  public void advised() {
    dummy = System.currentTimeMillis();
  }

 @Override
  public void unadvised() {
    dummy = System.currentTimeMillis();
  }
}
```

Далее приведен исходный код класса `TestPointcut`, обеспечивающего статическую проверку метода, снабжаемого советом.

```
package com.apress.prospring5.ch5;

import java.lang.reflect.Method;
import org.springframework.aop
         .support.StaticMethodMatcherPointcut;

public class TestPointcut extends StaticMethodMatcherPointcut {
  @Override
  public boolean matches(Method method, Class cls) {
    return ("advise".equals(method.getName()));
  }
}
```

В следующем фрагменте кода приведен исходный код класса `NoOpBeforeAdvice`, реализующего простой предшествующий совет без операций:

```
package com.apress.prospring5.ch5;

import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

public class NoOpBeforeAdvice
       implements MethodBeforeAdvice {
  @Override
  public void before(Method method, Object args,
                     Object target)
    throws Throwable {
    // без операций
  }
}
```

И, наконец, ниже представлен исходный код для тестирования разных типов заместителей.

```
package com.apress.prospring5.ch5;

import org.springframework.aop.Advisor;
import org.springframework.aop.framework.Advised;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop
          .support.DefaultPointcutAdvisor;

public class ProxyPerfTest {
  public static void main(String... args) {
    SimpleBean target = new DefaultSimpleBean();

    Advisor advisor = new DefaultPointcutAdvisor(
```

```
            new TestPointcut(), new NoOpBeforeAdvice());

        runCglibTests(advisor, target);
        runCglibFrozenTests(advisor, target);
        runJdkTests(advisor, target);
    }

    private static void runCglibTests(Advisor advisor,
                                      SimpleBean target) {
        ProxyFactory pf = new ProxyFactory();
        pf.setProxyTargetClass(true);
        pf.setTarget(target);
        pf.addAdvisor(advisor);
        SimpleBean proxy = (SimpleBean)pf.getProxy();
        System.out.println("Running CGLIB (Standard) Tests");
        test(proxy);
    }

    private static void runCglibFrozenTests(Advisor advisor,
                        SimpleBean target) {
        ProxyFactory pf = new ProxyFactory();
        pf.setProxyTargetClass(true);
        pf.setTarget(target);
        pf.addAdvisor(advisor);
        pf.setFrozen(true);

        SimpleBean proxy = (SimpleBean) pf.getProxy();
        System.out.println("Running CGLIB (Frozen) Tests");
        test(proxy);
    }

    private static void runJdkTests(Advisor advisor,
                        SimpleBean target) {
        ProxyFactory pf = new ProxyFactory();
        pf.setTarget(target);
        pf.addAdvisor(advisor);
        pf.setInterfaces(new Class[]{SimpleBean.class});

        SimpleBean proxy = (SimpleBean)pf.getProxy();
        System.out.println("Running JDK Tests");
        test(proxy);
    }

    private static void test(SimpleBean bean) {
        long before = 0;
        long after = 0;

        System.out.println("Testing Advised Method");
        before = System.currentTimeMillis();
```

```
    for(int x = 0; x < 500000; x++) {
      bean.advised();
    }
    after = System.currentTimeMillis();

    System.out.println("Took " + (after - before) + " ms");

    System.out.println("Testing Unadvised Method");
    before = System.currentTimeMillis();
    for(int x = 0; x < 500000; x++) {
      bean.unadvised();
    }
    after = System.currentTimeMillis();

    System.out.println("Took " + (after - before) + " ms");

    System.out.println("Testing equals() Method");
    before = System.currentTimeMillis();
    for(int x = 0; x < 500000; x++) {
      bean.equals(bean);
    }
    after = System.currentTimeMillis();

    System.out.println("Took " + (after - before) + " ms");
    System.out.println("Testing hashCode() Method");
    before = System.currentTimeMillis();
    for(int x = 0; x < 500000; x++) {
      bean.hashCode();
    }
    after = System.currentTimeMillis();

    System.out.println("Took " + (after - before) + " ms");

    Advised advised = (Advised)bean;

    System.out.println(
           "Testing Advised.getProxyTargetClass() Method");
    before = System.currentTimeMillis();
    for(int x = 0; x < 500000; x++) {
      advised.getTargetClass();
    }
    after = System.currentTimeMillis();

    System.out.println("Took " + (after - before) + " ms");

    System.out.println(">>>\n");
  }
}
```

В приведенном выше коде тестируются следующие типы заместителей.

- Стандартный заместитель из библиотеки CGLIB.
- Заместитель из библиотеки CGLIB с цепочкой вызовов фиксированного совета (т.е. когда заместитель зафиксирован методом `setFrozen()` из класса `ProxyConfig`, который косвенно расширяет класс `ProxyFactory`, библиотека CGLIB выполнит дополнительную оптимизацию, но запретит дальнейшее изменение совета).
- Заместитель из комплекта JDK.

Для каждого типа заместителя выполняются следующие контрольные примеры.

- **Метод, снабженный советом (тест 1).** Проверяется конкретный метод, снабженный советом. В этом тесте используется предшествующий совет, не предусматривающий никакой обработки, что уменьшает влияние совета на тесты производительности.
- **Метод, не снабженный советом (тест 2).** Проверяется конкретный метод заместителя, не снабженный советом. Зачастую у заместителя имеется немало методов, не снабженных советом. Этот тест позволяет выяснить, насколько хорошо методы, не снабженные советом, выполняются для различных заместителей.
- **Метод `equals()` (тест 3).** Этот тест позволяет выяснить издержки, связанные с вызовом метода `equals()`. Это особенно важно, когда заместители применяются в качестве ключей в хеш-отображении типа `HashMap` или подобного типа коллекции.
- **Метод `hashcode()` (тест 4).** Как и метод `equals()`, метод `hashCode()` важен, когда применяется хеш-отображение типа `HashMap` или подобные коллекции.
- **Выполнение методов из интерфейса `Advised` (тест 5).** Как упоминалось ранее, заместитель по умолчанию реализует интерфейс `Advised`, позволяя модифицировать заместитель после создания и запрашивать сведения о нем. Этот тест показывает, насколько быстро происходит обращение к методам из интерфейса `Advised` с помощью различных типов заместителей.

Результаты проведения этих тестов приведены в табл. 5.3.

Таблица 5.3. Результаты тестирования производительности заместителей (значения указаны в миллисекундах)

	Заместитель из библиотеки CGLIB (стандартный)	Заместитель из библиотеки CGLIB (фиксированный)	Заместитель из комплекта JDK
Метод, снабженный советом	245	135	224
Метод, не снабженный советом	92	42	78
Метод `equals()`	9	6	77
Метод `hashCode()`	29	13	23

Окончание табл. 5.3

	Заместитель из библиотеки CGLIB (стандартный)	Заместитель из библиотеки CGLIB (фиксированный)	Заместитель из комплекта JDK
Метод `Advised.getProxyTargetClass()`	9	6	15

Как видите, производительность стандартного заместителя из библиотеки CGLIB и динамического заместителя из комплекта JDK для методов, снабженных и не снабженных советом, не сильно отличается. Как обычно, получаемые результаты будут варьироваться в зависимости от конкретного оборудования и версии комплекта JDK.

Тем не менее заметные отличия проявляются, когда применяются заместители из библиотеки CGLIB с цепочкой вызовов фиксированного совета. Похожие результаты наблюдаются и при тестировании методов `equals()` и `hashCode()`, которые выполняются значительно быстрее, когда применяется стандартный заместитель из библиотеки CGLIB. Методы из интерфейса `Advised` также действуют быстрее, когда применяется фиксированный заместитель из библиотеки CGLIB. Это объясняется тем, что методы из интерфейса `Advised` обрабатываются раньше в методе `intercept()`, поэтому для них не выполняется большая часть логики, которая требуется остальным методам.

Выбор заместителя для практического применения

Принять решение, какой именно заместитель следует применять на практике, обычно не составляет особого труда. В частности, заместитель из библиотеки CGLIB предназначен как для классов, так и для интерфейсов, тогда как заместитель из комплекта JDK — только для интерфейсов. С точки зрения производительности заметных отличий в применении заместителя из комплекта JDK и стандартного заместителя из библиотеки CGLIB не наблюдается (во всяком случае, при выполнении методов, снабженных и не снабженных советом). Если же применяется фиксированный заместитель из библиотеки CGLIB, то цепочка вызовов совета не может быть изменена, а в библиотеке CGLIB проводится дополнительная оптимизация. Когда заместитель создается для замещения класса, по умолчанию устанавливается заместитель из библиотеки CGLIB, поскольку это единственный заместитель, который может быть сформирован для замещения класса. Чтобы воспользоваться заместителем из библиотеки CGLIB для замещения интерфейса, придется установить логическое значение `true` признака `optimize` в классе `ProxyFactory` с помощью метода `setOptimize()`.

Расширенное использование срезов

Ранее в этой главе были рассмотрены шесть основных реализаций интерфейса `Pointcut`, предоставляемых в Spring. Они в основном удовлетворяют требованиям

многих приложений, но иногда возникает потребность в большей гибкости при определении срезов. В каркасе Spring имеются две дополнительных реализации интерфейса `Pointcut` в классах `ComposablePointcut` и `ControlFlowPointcut`, где как раз и обеспечивается требующаяся гибкость.

Применение срезов потока управления

Срезы потока управления, реализуемые в классе `ControlFlowPointcut` из каркаса Spring, подобны конструкции `cflow`, доступной во многих других реализациях АОП, хотя они и не столь эффективные. В сущности, срез потока управления в Spring применяется ко всем вызовам методов из заданного метода или из всех методов, определенных в классе. Это довольно трудно представить наглядно, поэтому покажем на простом примере, как действует такой срез.

Ниже приведен исходный код класса `SimpleBeforeAdvice`, в котором выводится сообщение с описанием метода, снабжаемого советом.

```
package com.apress.prospring5.ch5;

import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

public class SimpleBeforeAdvice
       implements MethodBeforeAdvice {
  @Override
  public void before(Method method, Object args,
                     Object target)
              throws Throwable {
    System.out.println("Before method: " + method);
  }
}
```

Этот класс совета позволяет выяснить, к каким именно методам применяется срез типа `ControlFlowPointcut`. Ниже приведен исходный код простого тестового класса `TestBean`.

```
package com.apress.prospring5.ch5;

public class TestBean {
  public void foo() {
    System.out.println("foo()");
  }
}
```

Как видите, в данном классе определяется простой метод `foo()`, который требуется снабдить советом. Но при этом должно удовлетворяться специальное требование: этот метод необходимо снабдить советом только в том случае, если он вызывается из другого конкретного метода. Ниже приведен исходный код простой тестовой программы для рассматриваемого здесь примера.

```
package com.apress.prospring5.ch5;

import org.springframework.aop.Advisor;
import org.springframework.aop.Pointcut;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop
          .support.ControlFlowPointcut;
import org.springframework.aop
          .support.DefaultPointcutAdvisor;

public class ControlFlowDemo {
  public static void main(String... args) {
     ControlFlowDemo ex = new ControlFlowDemo();
     ex.run();
  }

  public void run() {
    TestBean target = new TestBean();
    Pointcut pc = new ControlFlowPointcut(
                  ControlFlowDemo.class, "test");
    Advisor advisor = new DefaultPointcutAdvisor(pc,
                      new SimpleBeforeAdvice());

    ProxyFactory pf = new ProxyFactory();
    pf.setTarget(target);
    pf.addAdvisor(advisor);

    TestBean proxy = (TestBean) pf.getProxy();
    System.out.println("\tTrying normal invoke");
    proxy.foo();
    System.out.println(
           "\n\tTrying under ControlFlowDemo.test()");
    test(proxy);
  }

  private void test(TestBean bean) {
    bean.foo();
  }
}
```

В приведенном выше фрагменте кода снабженный советом заместитель связывается со срезом типа `ControlFlowPointcut`, а затем метод `foo()` вызывается дважды: один раз — непосредственно из метода `run()`, а другой раз — из метода `test()`. Особый интерес представляет следующая строка кода:

```
Pointcut pc = new ControlFlowPointcut(
              ControlFlowDemo.class, "test");
```

В этой строке кода получается экземпляр типа `ControlFlowPointcut` для метода `test()` из класса `ControlFlowDemo`. По существу, это означает следующее:

"Сделать срез по всем методам, вызываемым из метода `ControlFlowDemo.test()`". Однако фраза "Сделать срез по всем методам" на самом деле означает следующее: "Сделать срез по всем методам объекта-заместителя, снабженного советом с помощью интерфейса `Advisor`, соответствующего данному экземпляру типа `ControlFlowPointcut`". Выполнение исходного кода из данного примера приводит к выводу на консоль следующего результата:

```
    Trying normal invoke[10]
foo()
    Trying under[11] ControlFlowDemo.test()
Before method:[12]
    public void com.apress.prospring5.ch5.TestBean.foo()
foo()
```

Как видите, когда метод `foo()` вызывается в первый раз за пределами потока управления из метода `test()`, он не снабжается советом. А когда метод `foo()` вызывается во второй раз, уже в потоке управления из метода `test()`, экземпляр типа `ControlFlowPointcut` показывает, что связанный с ним совет применяется к методу `foo()`, поэтому данный метод снабжается советом. Но если вызвать в теле метода `test()` еще один метод, не относящийся к заместителю, то он не будет снабжен советом.

Срезы потока управления могут быть исключительно полезны, позволяя снабжать объект советом выборочно, т.е. только тогда, когда этот объект выполняется в контексте другого объекта. Однако применение среза потока управления связано с существенным снижением производительности по сравнению с другими видами срезов.

Обратимся к конкретному примеру. Допустим, имеется система обработки транзакций, включающая в себя интерфейсы `TransactionService` и `AccountService`. В данном примере требуется применить последующий совет, чтобы, в том случае, когда метод `AccountService.updateBalance()` вызывается из метода `TransactionService.reverseTransaction()`, после обновления остатков на счете соответствующее уведомление отправлялось клиенту по электронной почте. Но такое уведомление не должно посылаться ни при каких других обстоятельствах. В таком случае удобно воспользоваться срезом потока управления. На рис. 5.6 приведена диаграмма последовательности действий, составленная на языке UML для данного примера.

Применение составного среза

В предыдущих примерах мы применяли по одному срезу на каждый экземпляр типа `Advisor`. И, как правило, этого оказывается достаточно, но иногда для достижения намеченной цели может возникнуть потребность составить вместе два или еще

---

[10] Попытка сделать обычный вызов

[11] Попытка сделать вызов из метода...

[12] Предшествующий метод:

более среза. Допустим, требуется сделать срез по всем методам получения и установки из класса компонента Spring Bean. При этом имеются два отдельных среза для методов получения и установки, но отсутствует срез для обеих разновидностей методов. Можно было бы, конечно, создать еще один срез с новой логикой, но более совершенный подход предполагает составление двух срезов в единый срез с помощью класса `ComposablePointcut`.

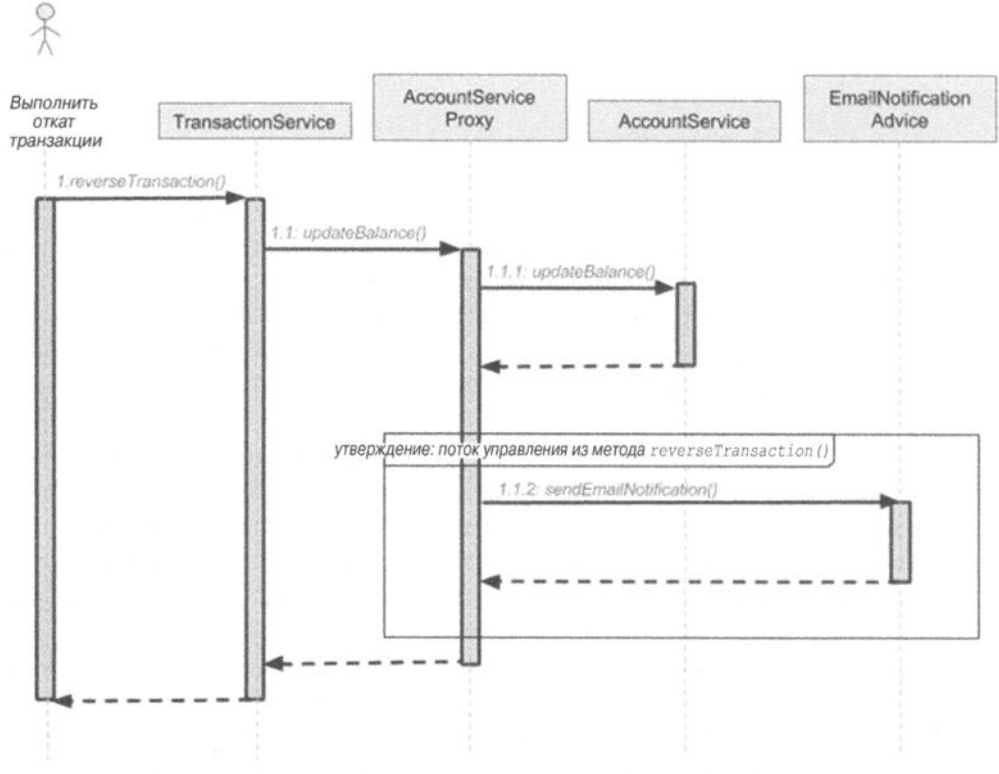


Рис. 5.6. Диаграмма последовательности действий, составленная на языке UML для среза потока управления

В классе `ComposablePointcut` поддерживаются два метода: `union()` и `intersection()`. По умолчанию объект типа `ComposablePointcut` создается с реализацией интерфейса `ClassFilter`, соответствующей всем классам, а также с реализацией интерфейса `MethodMatcher`, соответствующей всем методам, хотя при конструировании данного объекта можно указать и свои первоначальные реализации интерфейсов `ClassFilter` и `MethodMatcher`. Оба метода, `union()` и `intersection()`, перегружаются, чтобы принимать аргументы типа `ClassFilter` и `MethodMatcher`.

Чтобы вызвать метод `ComposablePointcut.union()`, достаточно передать экземпляр класса, реализующего один из интерфейсов `ClassFilter`, `MethodMatcher`

или Pointcut. В результате операции объединения класс ComposablePointcut введет условие "или" в свою цепочку вызовов, чтобы обнаружить совпадение со срезами. То же самое происходит и при вызове метода ComposablePointcut.intersection(), но вместо условия "или" вводится условие "и", которое означает, что все определения интерфейсов ClassFilter, MethodMatcher и Pointcut в классе ComposablePointcut должны совпасть, чтобы совет был применен. Это условие можно наглядно представить как предложение WHERE в операторе запроса SQL, где метод union() действует подобно логической операции ИЛИ, а метод intersection() — подобно логической операции И.

Составные срезы трудно представить наглядно, как и рассмотренные ранее срезы потока управления, поэтому обратимся к конкретному примеру. Так, ниже приведен снова исходный код класса GrammyGuitarist, применявшегося в одном из предыдущих примеров.

```
package com.apress.prospring5.ch5;

import com.apress.prospring5.ch2.common.Guitar;
import com.apress.prospring5.ch2.common.Singer;

public class GrammyGuitarist implements Singer {
  @Override public void sing() {
    System.out.println("sing: Gravity is working "
                + "against me\n"
                + "And gravity wants to bring me down");
  }

  public void sing(Guitar guitar) {
    System.out.println("play: " + guitar.play());
  }

  public void rest() {
    System.out.println("zzz");
  }

  public void talk(){
    System.out.println("talk");
  }
}
```

В данном примере мы собираемся сгенерировать три разных заместителя, используя один и тот же экземпляр типа ComposablePointcut, но каждый раз модифицировать этот экземпляр с помощью метода union() или intersection(). После этого будут вызваны все три метода для заместителя целевого компонента Spring Bean, чтобы выяснить, какие из них будут снабжены советом, как демонстрируется в приведенном ниже коде.

```
package com.apress.prospring5.ch5;

import java.lang.reflect.Method;
import org.springframework.aop.Advisor;
import org.springframework.aop.ClassFilter;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.ComposablePointcut;
import org.springframework.aop.support
          .DefaultPointcutAdvisor;
import org.springframework.aop.support.StaticMethodMatcher;

public class ComposablePointcutExample {
  public static void main(String... args) {
    GrammyGuitarist johnMayer = new GrammyGuitarist();

    ComposablePointcut pc = new ComposablePointcut(
              ClassFilter.TRUE, new SingMethodMatcher());

    System.out.println("Test 1 >> ");
    GrammyGuitarist proxy = getProxy(pc, johnMayer);
    testInvoke(proxy);
    System.out.println();

    System.out.println("Test 2 >> ");
    pc.union(new TalkMethodMatcher());
    proxy = getProxy(pc, johnMayer);
    testInvoke(proxy);
    System.out.println();

    System.out.println("Test 3 >> ");
    pc.intersection(new RestMethodMatcher());
    proxy = getProxy(pc, johnMayer);
    testInvoke(proxy);
  }

  private static GrammyGuitarist getProxy(
          ComposablePointcut pc, GrammyGuitarist target) {
    Advisor advisor = new DefaultPointcutAdvisor(pc,
                      new SimpleBeforeAdvice());
    ProxyFactory pf = new ProxyFactory();
    pf.setTarget(target);
    pf.addAdvisor(advisor);
    return (GrammyGuitarist) pf.getProxy();
  }

  private static void testInvoke(GrammyGuitarist proxy) {
    proxy.sing();
    proxy.sing(new Guitar());
    proxy.talk();
```

```
    proxy.rest();
  }

  private static class SingMethodMatcher
          extends StaticMethodMatcher {
    @Override
    public boolean matches(Method method, Class<?> cls) {
      return (method.getName().startsWith("si"));
    }
  }

  private static class TalkMethodMatcher
          extends StaticMethodMatcher {
    @Override
    public boolean matches(Method method, Class<?> cls) {
      return "talk".equals(method.getName());
    }
  }

  private static class RestMethodMatcher
          extends StaticMethodMatcher {
  @Override
    public boolean matches(Method method, Class<?> cls) {
      return (method.getName().endsWith("st"));
    }
  }
}
```

В данном примере прежде всего обратите внимание на три закрытые реализации интерфейса `MethodMatcher`. Так, в классе `SingMethodMatcher` обнаруживается совпадение со всеми методами, имена которых начинаются на **`si`**. Это стандартная реализация интерфейса `MethodMatcher`, применяемая для сборки среза типа `ComposablePointcut`. В связи с этим предполагается, что при первом обращении к методам из класса `GrammyGuitarist` советом будет снабжен только метод `sing()`.

В классе `TalkMethodMatcher` обнаруживается совпадение со всеми методами, имена которых начинаются на **`talk`**, а с помощью метода `union()` он объединяется с классом `ComposablePointcut` при втором обращении к методам из класса `GrammyGuitarist`. В этот момент происходит объединение двух реализаций интерфейса `MethodMatcher`, одна из которых обнаруживает совпадение со всеми методами, имена которых начинаются на **`si`**, а другая — со всеми методами, имена которых начинаются на **`talk`**. И теперь предполагается, что советом будут снабжены все вызовы методов. Класс `TalkMethodMatcher` имеет весьма конкретное назначение: обнаружить совпадение только с методом `talk()`. С помощью метода `intersection()` эта реализация интерфейса `MethodMatcher` объединяется с его реализацией в классе `ComposablePointcut` при третьем обращении к методам.

В связи с тем что класс `RestMethodMatcher` объединяется с помощью метода `intersection()`, предполагается, что при третьем обращении к методам ни один из

них не будет снабжен советом. Ведь ни в одной из составленных вместе реализаций интерфейса `MethodMatcher` не обнаруживается совпадение с вызываемыми методами. После выполнения исходного кода из данного примера на консоль будет выведен следующий результат:

```
Test 1 >>
Before method: public void com.apress
               .prospring5.ch5.GrammyGuitarist.sing()
sing: Gravity is working against me
      And gravity wants to bring me down
Before method: public void com.apress.prospring5.ch5
               .GrammyGuitarist.sing(
                com.apress.prospring5.ch2.common.Guitar)
play: G C G C Am D7
talk
zzz

Test 2 >>
Before method: public void com.apress
               .prospring5.ch5.GrammyGuitarist.sing()
sing: Gravity is working against me
      And gravity wants to bring me down
Before method: public void com.apress
               .prospring5.ch5.GrammyGuitarist.talk()
Before method: public void com.apress.prospring5.ch5
               .GrammyGuitarist.sing(
                com.apress.prospring5.ch2.common.Guitar)
play: G C G C Am D7
talk
zzz

Test 3 >>
sing: Gravity is working against me
      And gravity wants to bring me down
talk
zzz
```

Несмотря на то что в данном примере демонстрируется применение различных реализаций интерфейса `MethodMatcher` только в процессе составления, при построении среза столь же просто применяется интерфейс `ClassFilter`. На самом деле при построении составного среза можно использовать в любом сочетании реализации интерфейсов `MethodMatcher` и `ClassFilter`.

Составление срезов и интерфейс `Pointcut`

В предыдущем разделе было показано, как составлять срез с помощью различных реализаций интерфейсов `MethodMatcher` и `ClassFilter`. Составлять срезы можно и с помощью других объектов, классы которых реализуют интерфейс `Pointcut`.

Еще один способ построения составного среза предполагает применение класса `org.springframework.aop.support.Pointcuts`, в котором предоставляются три статических метода. С одной стороны, методы `intersection()` и `union()` принимают два среза в качестве аргументов и конструируют составной срез. А с другой стороны, имеется метод `matches(Pointcut, Method, Class, Object[])`, позволяющий быстро проверить, совпадает ли срез с предоставленным методом, классом и аргументам метода.

В классе `Pointcuts` поддерживаются операции только с двумя срезами. Так, если требуется объединить реализации интерфейсов `MethodMatcher`, `ClassFilter` и `Pointcut`, следует воспользоваться классом `ComposablePointcut`. Но если требуется объединить два среза, то удобнее употребить класс `Pointcuts`.

Краткие итоги по срезам

В каркасе Spring предоставляется обширный ряд реализаций интерфейса `Pointcut`, которые должны удовлетворять большинству, если не всем, требованиям приложения. Не следует, однако, забывать, что когда не удается найти подходящий срез для конкретных потребностей, можно самостоятельно создать свой срез, реализовав интерфейсы `Pointcut`, `MethodMatcher` и `ClassFilter`.

Объединять срезы и советы можно, применяя два разных подхода. Первый, применявшийся до сих пор подход предполагает отделение реализации среза от совета. В исходном коде приведенных ранее примеров сначала получались экземпляры реализаций интерфейса `Pointcut`, а затем использовался класс `DefaultPointcutAdvisor` для ввода совета в заместитель вместе со срезом, реализующим интерфейс `Pointcut`.

Второй подход, применяемый во многих примерах из документации на Spring, состоит в том, чтобы инкапсулировать интерфейс `Pointcut` в собственной реализации интерфейса `Advisor`. В итоге получается класс, реализующий интерфейсы `Pointcut` и `PointcutAdvisor`, а также метод `PointcutAdvisor.getPointcut()`, просто возвращающий ссылку `this`. Именно такой подход применяется во многих классах Spring, например `StaticMethodMatcherPointcutAdvisor`. Мы считаем первый подход более гибким, поскольку он позволяет сочетать разные реализации интерфейсов `Pointcut` и `Advisor`. Но второй подход удобен в тех случаях, когда одно и то же сочетание реализаций интерфейсов `Pointcut` и `Advisor` предполагается использовать в различных частях приложения или даже в разных приложениях.

Второй подход оказывается удобным и в том случае, если у каждой реализации интерфейса `Advisor` должен быть отдельный экземпляр реализации интерфейса `Pointcut`. Этого можно добиться, возложив на реализацию интерфейса `Advisor` ответственность за создание реализации интерфейса `Pointcut`. Как упоминалось ранее при обсуждении производительности заместителей, методы, не снабженные советом, выполняются намного эффективнее, чем методы, снабженные советом. Именно по этой причине следует удостовериться, что если применяется класс

`Pointcuts`, то советом снабжаются только те методы, для которых этот совет совершенно необходим. Тем самым сокращаются излишние дополнительные издержки на применение АОП в приложениях.

Основы применения введений

Введения являются важной частью функциональных возможностей АОП, доступных в Spring. Благодаря применению введений можно динамически внедрять новые функциональные возможности в существующий объект. Каркас Spring позволяет ввести в существующий объект реализацию любого интерфейса. В связи с этим возникает следующий закономерный вопрос: зачем внедрять функциональность динамически на стадии выполнения, если это нетрудно сделать на стадии проектирования? Ответить на этот вопрос просто: функциональность внедряется динамически, если является сквозной, и ее очень сложно реализовать с помощью традиционного совета.

Основные положения о введениях

Введения трактуются в Spring как специальный тип совета, а точнее — как специальный тип окружающего совета. А поскольку введения применяются исключительно на уровне классов, то применять срезы вместе с введениями нельзя, так как они не совпадают семантически. Введение внедряет в класс новые реализации интерфейсов, а срез определяет, к каким именно методам применяется совет. Чтобы создать введение, следует реализовать интерфейс `IntroductionInterceptor`, расширяющий интерфейсы `MethodInterceptor` и `DynamicIntroductionAdvice`. Эта иерархия интерфейсов, составленная на языке UML в IDE IntelliJ IDEA, приведена на рис. 5.7 вместе с методами из обоих интерфейсов.

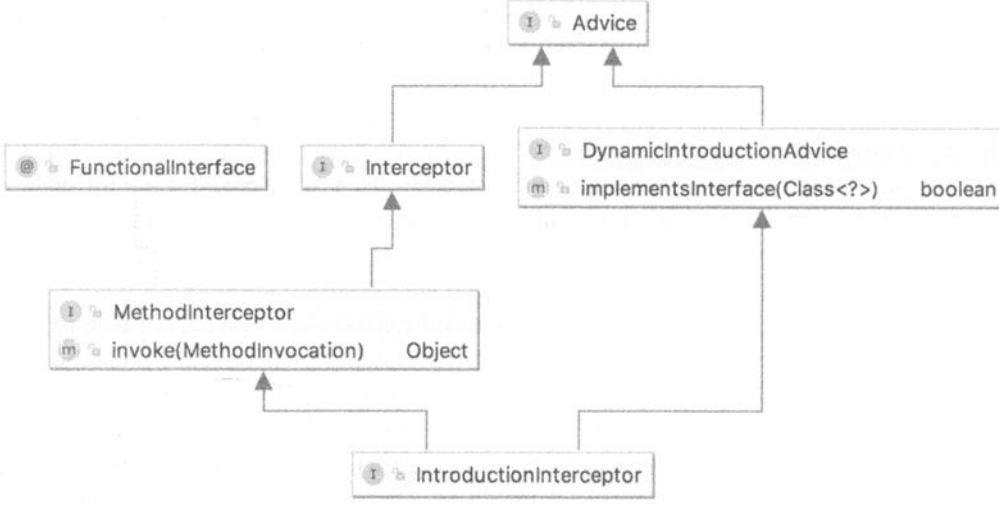


Рис. 5.7. Иерархия интерфейсов для введений

Как видите, в интерфейсе `MethodInterceptor` определен метод `invoke()`, с помощью которого предоставляется реализация для вводимых интерфейсов и выполняется перехват любых дополнительных методов по мере необходимости. Реализация всех методов для интерфейса в одном методе может оказаться затруднительной и, скорее всего, приведет к большому объему кода, с которым придется иметь дело, принимая решение, какой из методов следует вызвать. Правда, в Spring предоставляется стандартная реализация интерфейса `IntroductionInterceptor`, которая называется `DelegatingIntroductionInterceptor` и намного упрощает создание введений. Чтобы построить введение с помощью класса `DelegatingIntroduction Interceptor`, следует создать класс, производный от класса `Delegating IntroductionInterceptor` и реализующий интерфейсы, которые требуется ввести. А в самом классе `DelegatingIntroductionInterceptor` все вызовы введенных методов просто поручаются соответствующему методу. Не особенно переживайте, если такое пояснение покажется вам не особенно понятным; в следующем разделе будет приведен наглядный пример реализации введений.

Подобно тому, как для работы с советом из среза требуется интерфейс `Pointcut Advisor`, для внедрения введений в заместители требуется интерфейс `Introduction Advisor`. Стандартной реализацией интерфейса `IntroductionAdvisor` служит класс `DefaultIntroductionAdvisor`, который должен удовлетворять большинству, если не всем, потребностям введения. Но внедрять введения с помощью метода `ProxyFactory.addAdvice()` не разрешается, поскольку это приводит к генерированию исключения типа `AopConfigException`. Вместо этого следует вызвать метод `addAdvisor()`, передав ему экземпляр класса, реализующего интерфейс `IntroductionAdvisor`.

Если применяется стандартный совет, а не введение, то один и тот же экземпляр совета можно употребить для самых разных объектов. В документации на Spring это называется *жизненным циклом совета на каждый класс*, хотя одиночный экземпляр совета может быть пригодным для многих классов. А для введений совет образует часть состояния объекта, снабженного советом, в результате чего для каждого такого объекта должен быть предусмотрен отдельный экземпляр совета. И это называется *жизненным циклом совета на каждый экземпляр*. А поскольку требуется гарантировать наличие отдельного экземпляра введения для каждого снабженного советом объекта, то зачастую предпочтительнее создать подкласс, производный от класса `Default IntroductionAdvisor` и отвечающий за создание совета из введения. Таким образом, достаточно создать новый экземпляр класса советника для каждого объекта, потому что это автоматически приведет к получению нового экземпляра введения. Допустим, требуется применить предшествующий совет к методу `setFirstName()` для всех экземпляров класса `Contact`. На рис. 5.8 приведен один и тот же совет, применяемый ко всем объектам типа `Contact`.

А теперь допустим, что введение требуется внедрить во все экземпляры класса `Contact`, снабдив его соответствующими сведениями для каждого экземпляра данного класса (например, атрибутом `isModified`, где указывается, был ли экземпляр

модифицирован). В таком случае введение будет создано для каждого экземпляра класса `Contact` и привязано к этому экземпляру, как показано на рис. 5.9. На этом рассмотрение основных положений о введениях завершается. Далее мы обсудим, как решить задачу выявления изменений в объекте с помощью введений.

Рис. 5.8. Жизненный цикл совета на каждый класс

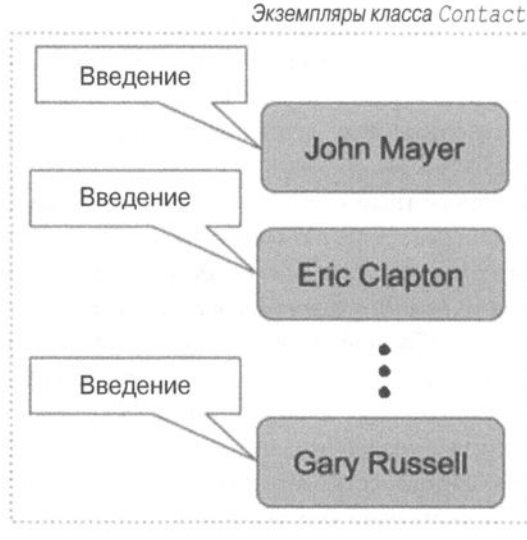


Рис. 5.9. Введение на каждый экземпляр

Выявление изменений в объекте с помощью введений

Выявлять изменения в объекте полезно по многим причинам. Как правило, выявление изменений в объекте позволяет предотвратить излишний доступ к базе данных в процессе сохранения содержимого объекта. Если объект передается методу для ви-

доизменения, но возвращается обратно без изменений, то нет никакого смысла обновлять базу данных. Такой способ проверки изменений в объекте позволяет увеличить пропускную способность приложения, особенно в том случае, когда база данных уже работает с большой нагрузкой или расположена в удаленной сети, увеличивая стоимость обмена данными.

К сожалению, такие функциональные возможности трудно реализовать вручную, потому что они требуют внедрения в каждый метод, способный изменить состояние объекта, проверки факта такого изменения. Если подсчитать все проверки на пустое значение `null` и проверки на изменение значения, то в итоге получится примерно по восемь строк дополнительного кода на каждый метод. Этот код можно, конечно, вынести в отдельный метод, но все равно данный метод придется вызывать всякий раз, когда требуется выполнить проверку. А распространение такого кода по всему приложению, где проверка факта изменений требуется во многих классах, — это своего рода мина замедленного действия.

Именно здесь и приходят на помощь введения. Вряд ли нужно, чтобы каждый класс, в котором требуются проверки изменений, наследовался от какой-то базовой реализации, теряя в итоге свой единственный шанс на наследование, как, впрочем, и внедрять код проверки во все методы, которые могут изменить состояние объекта. А с помощью введений можно найти гибкое решение задачи выявления изменений, не требующее написания большого объема повторяющегося и подверженного ошибкам кода.

В рассматриваемом здесь примере с помощью введений будет построен полноценный каркас проверок изменений. Логика проверки изменений инкапсулирована в интерфейс `IsModified`, реализация которого будет введена в соответствующие объекты вместе с логикой перехвата для автоматического выполнения проверок изменений. Для целей данного примера соблюдаются соглашения, принятые для компонентов JavaBeans и означающие, что изменение происходит при любом обращении к методу установки. Разумеется, не все вызовы метода установки трактуются как изменения. Нужно лишь проверить, отличается ли значение, переданное методу установки, от текущего значения, хранящегося в объекте. Единственный недостаток такого решения заключается в том, что возврат объекта в первоначальное состояние будет считаться изменением, если изменится любое значение в объекте. Допустим, имеется объект типа `Contact` со свойством `firstName`. Допустим также, что в процессе обработки свойство `firstName` изменило свое значение с `Peter` на `John`. В итоге объект будет помечен как модифицированный. Но он будет все равно помечен как модифицированный, даже если при последующей обработке свойство `firstName` вернется к исходному значению `Peter`. Один из способов принять подобную ситуацию во внимание состоит в том, чтобы сохранить предысторию всех изменений в пределах жизненного цикла объекта. Тем не менее предлагаемая здесь реализация нетривиальна и удовлетворяет большинству требований. А реализация более полного решения могла бы привести к излишнему усложнению данного примера.

Применение интерфейса IsModified

Центральное место в решении для проверки изменений в объекте принадлежит интерфейсу IsModified, который в рассматриваемом здесь примере вымышленного приложения служит для принятия благоразумных решений о сохраняемости объектов. Мы не будем касаться того, каким образом интерфейс IsModified применяется в приложении, а вместо этого сосредоточим основное внимание на реализации введения. Ниже показано, каким образом определяется интерфейс IsModified. В нем нет ничего особенно интересного, кроме единственного метода isModified(), указывающего, был ли объект изменен.

```
package com.apress.prospring5.ch5.introduction;

public interface IsModified {
  boolean isModified();
}
```

Создание примеси

Следующий шаг состоит в том, чтобы написать код, реализующий интерфейс IsModified и вводимый в виде так называемой *примеси (mixin)*. Как упоминалось ранее, создавать введения намного проще, определяя подклассы, производные от класса DelegatingIntroductionInterceptor, чем реализуя непосредственно интерфейс IntroductionInterceptor. Класс создаваемой здесь примеси называется IsModifiedMixin, является производным от класса DelegatingIntroduction Interceptor и реализует интерфейс IsModified, как показано ниже.

```
package com.apress.prospring5.ch5.introduction;

import java.lang.reflect.Method;
import java.util.HashMap;
import java.util.Map;
import org.aopalliance.intercept.MethodInvocation;
import org.springframework.aop.support
          .DelegatingIntroductionInterceptor;

public class IsModifiedMixin
       extends DelegatingIntroductionInterceptor
       implements IsModified {
  private boolean isModified = false;

  private Map<Method, Method> methodCache = new HashMap<>();

  @Override
  public boolean isModified() {
    return isModified;
  }
```

```
  @Override
  public Object invoke(MethodInvocation invocation)
         throws Throwable {
    if (!isModified) {
      if ((invocation.getMethod().getName()
                     .startsWith("set"))
           && (invocation.getArguments().length == 1)) {

        Method getter = getGetter(invocation.getMethod());

        if (getter != null) {
          Object newVal = invocation.getArguments()[0];
          Object oldVal = getter.invoke(
                          invocation.getThis(), null);
          if((newVal == null) && (oldVal == null)) {
            isModified = false;
          } else if((newVal == null) && (oldVal != null)) {
            isModified = true;
          } else if((newVal != null) && (oldVal == null)) {
            isModified = true;
          } else {
            isModified = !newVal.equals(oldVal);
          }
        }
      }
    }
    return super.invoke(invocation);
  }

  private Method getGetter(Method setter) {
    Method getter = methodCache.get(setter);
    if (getter != null) {
      return getter;
    }

    String getterName = setter.getName()
                        .replaceFirst("set", "get");
    try {
      getter = setter.getDeclaringClass()
                     .getMethod(getterName, null);
      synchronized (methodCache) {
        methodCache.put(setter, getter);
      }
      return getter;
    } catch (NoSuchMethodException ex) {
      return null;
    }
  }
}
```

Прежде всего, в приведенном выше коде следует обратить внимание на реализацию интерфейса `IsModified`, которая состоит из закрытого поля `isModified` и метода `isModified()`. Данный пример наглядно показывает главную причину, по которой должен существовать только один экземпляр примеси для каждого объекта, снабженного советом: примесь вводит в объект не только методы, но и состояние. Если сделать единственный экземпляр такой примеси общим для нескольких объектов, то общим для них окажется и его состояние. Это означает, что после первого же изменения в каком-нибудь одном объекте измененными будут считаться все остальные объекты.

В действительности реализовывать метод `invoke()` для примеси совсем не обязательно, но в данном случае это позволяет автоматически выявлять изменение, когда оно происходит. Проверка начинается лишь в том случае, если объект все еще остается без изменений. Ведь проверять на предмет изменений объект, о котором известно, что он изменялся, нет никакой нужды. Затем выясняется, является ли методом установки проверяемый метод, и если это так, то извлекается соответствующий метод получения. Обратите внимание на то, что пары методов получения и установки кешируются для ускорения последующих операций извлечения. И, наконец, значение, возвращаемое методом получения, сравнивается со значением, передаваемым методу установки, чтобы выяснить, произошли ли изменения в объекте. При этом выполняются проверки с самыми разными сочетаниями пустого значения `null` и рядом соответствующих изменений. Очень важно помнить, что если при переопределении метода `invoke()` применяется класс `DelegatingIntroductionInterceptor`, то придется вызвать метод `super.invoke()`, потому что класс `DelegatingIntroductionInterceptor` направляет вызов корректному адресату: объекту, снабженному советом, или самой примеси.

В примеси можно реализовать произвольное количество интерфейсов, и каждый из них будет автоматически введен в объект, снабженный советом.

Создание советника

На следующем шаге создается советник типа `Advisor`, служащий оболочкой для создаваемого класса примеси. И хотя этот шаг совсем не обязателен, он помогает гарантировать, что новый экземпляр класса примеси будет применяться для каждого объекта, снабженного советом. Ниже приведен исходный код класса `IsModifiedAdvisor`.

```
package com.apress.prospring5.ch5.introduction;

import org.springframework.aop.support.DefaultIntroductionAdvisor;

public class IsModifiedAdvisor extends DefaultIntroductionAdvisor {
  public IsModifiedAdvisor() {
    super(new IsModifiedMixin());
  }
}
```

Обратите внимание на то, что для создания класса `IsModifiedAdvisor` был расширен класс `DefaultIntroductionAdvisor`. Реализация совета довольно проста и не требует дополнительных пояснений.

Собирая все вместе

А теперь, когда имеются классы примеси и советника, можно протестировать каркас проверки изменений в объекте. Воспользуемся для этой цели упоминавшимся ранее классом `Contact`, входящим в состав пакета `common`. Этот класс нередко служит в качестве зависимости для примеров проектов, рассматриваемых в данной книге, по причинам повторного использования. Ниже приведено определение данного класса.

```
package com.apress.prospring5.ch2.common;

public class Contact {
  private String name;
  private String phoneNumber;
  private String email;

  public String getName() {
    return name;
  }

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

  // методы получения и установки значений в других полях
  ...
}
```

У этого класса имеется целый ряд свойств, но для тестирования примеси, проверяющей изменения в объекте, применяется лишь свойство `name`. В следующем фрагменте кода показано, каким образом собирается снабженный советом заместитель, а затем тестируется код проверки изменений в объекте:

```
package com.apress.prospring5.ch5;

import com.apress.prospring5.ch2.common.Contact;
import com.apress.prospring5.ch5.introduction.IsModified;
import com.apress.prospring5.ch5.introduction
          .IsModifiedAdvisor;
import org.springframework.aop.IntroductionAdvisor;
import org.springframework.aop.framework.ProxyFactory;

public class IntroductionDemo {
  public static void main(String... args) {
    Contact target = new Contact();
```

```
    target.setName("John Mayer");

    IntroductionAdvisor advisor = new IsModifiedAdvisor();

    ProxyFactory pf = new ProxyFactory();
    pf.setTarget(target);
    pf.addAdvisor(advisor);
    pf.setOptimize(true);

    Contact proxy = (Contact) pf.getProxy();
    IsModified proxyInterface = (IsModified)proxy;

    System.out.println("Is Contact?: "
                    + (proxy instanceof Contact));
    System.out.println("Is IsModified?: "
                    + (proxy instanceof IsModified));
    System.out.println("Has been modified?: "
                     + proxyInterface.isModified());

    proxy.setName("John Mayer");

    System.out.println("Has been modified?: "
                     + proxyInterface.isModified());

    proxy.setName("Eric Clapton");

    System.out.println("Has been modified?: "
                     + proxyInterface.isModified());
  }
}
```

При создании заместителя в признаке оптимизации устанавливается логическое значение `true`, чтобы обеспечить применение заместителя из библиотеки CGLIB. Дело в том, что если выбрать заместитель из комплекта JDK для введения примеси, то результирующий заместитель не будет экземпляром класса (в данном случае `Contact`). Ведь такой заместитель реализует только интерфейсы примеси, но не исходный класс. А если выбрать заместитель из библиотеки CGLIB, то исходный класс реализуется заместителем наряду с интерфейсами примеси.

В приведенном выше коде сначала проверяется, является ли заместитель экземпляром класса `Contact`, а затем является ли он экземпляром класса `IsModified`. В результате обеих проверок возвращается логическое значение `true`, если используется заместитель из библиотеки CGLIB, но для заместителя из комплекта JDK логическое значение `true` дает только проверка экземпляра класса `IsModified`. И, наконец, тестируется код проверки изменений в объекте. С этой целью устанавливается сначала текущее, а затем новое значение в свойстве `name`, и каждый раз проверяется значение признака `isModified`. Выполнение исходного кода из данного примера дает следующий результат:

```
Is Contact?: true
Is IsModified?: true
Has been modified?: false
Has been modified?: false
Has been modified?: true
```

Как и следовало ожидать, обе проверки с помощью операции `instanceof` возвращают логическое значение `true`. В результате первого вызова метода `isModified()` еще до всех изменений возвращается логическое значение `false`. И в результате следующего вызова этого метода после установки того же самого значения в свойстве `name` также возвращается логическое значение `false`. Но в результате последнего вызова метода `isModified()` после установки нового значения в свойстве `name` возвращается логическое значение `true`, указывая на то, что объект действительно был изменен.

Краткие итоги по введениям

Введения относятся к числу самых эффективных средств АОП в Spring, поскольку позволяют расширять не только функциональные возможности существующих методов, но и набор интерфейсов и реализаций объектов в динамическом режиме. Применение введений — идеальный способ реализовать сквозную логику, с которой приложение взаимодействует через ясно определенные интерфейсы. В общем, это такая разновидность логики, которую желательно применять декларативно, а не программно. Используя класс `IsModifiedMixin`, представленный в рассмотренном выше примере, и каркасные службы, обсуждаемые в следующем разделе, можно декларативно определить объекты, способные проверять наличие изменений, не внося коррективы в реализацию этих объектов.

Очевидно, что введения влекут за собой определенные издержки, так как они действуют через заместителей. Все методы заместителей считаются снабженными советом, поскольку применять срезы в сочетании с введениями нельзя. Но, учитывая многообразие служб, которые можно реализовать с помощью введений, включая проверку изменений в объекте, издержки, связанные с производительностью, являются лишь незначительной платой за сокращение объема кода, требующегося для реализации служб, а также за повышенную устойчивость и сопровождаемость, достигаемую благодаря полной централизации логики служб.

Каркасные службы для АОП

В приведенных до сих пор примерах нам пришлось написать немало кода, снабжающего объекты советами и генерирующего для них заместители. И хотя это не вызывает особых затруднений, тем не менее, означает, что вся конфигурация советов жестко закодирована в приложении, устраняя некоторые преимущества, которых можно добиться, прозрачно снабжая советом реализации методов. Правда, в Spring

предоставляются дополнительные каркасные службы, позволяющие создать снабженный советом заместитель в конфигурации приложения и затем внедрить его в целевой компонент Spring Bean подобно любым другим зависимостям.

Декларативный подход к конфигурированию АОП предпочтительнее ручного программного подхода. Применение декларативного механизма позволяет не только вынести наружу конфигурацию совета, но и уменьшает вероятность совершения ошибок при программировании. Можно также извлечь выгоды из сочетания АОП с внедрением зависимостей, чтобы применять АОП в полностью прозрачной среде.

Декларативное конфигурирование АОП

Для декларативного конфигурирования АОП в Spring доступны следующие возможности.

- **Применение класса `ProxyFactoryBean`.** Для реализации АОП в Spring класс `ProxyFactoryBean` предоставляет декларативный способ конфигурирования интерфейса `ApplicationContext`, а следовательно, и базового интерфейса `BeanFactory`, при создании заместителя АОП на основе уже определенных компонентов Spring Beans.
- **Применение пространства имен `aop` в Spring.** Появившееся в версии Spring 2.0 пространство имен `aop` предоставляет упрощенный (по сравнению с классом `ProxyFactoryBean`) способ определения аспектов и их требований к внедрению зависимостей в приложениях Spring, хотя для этого может потребоваться подключение ряда библиотек AspectJ. Тем не менее класс `ProxyFactoryBean` подспудно применяется в пространстве имен `aop`.
- **Применение аннотации в стиле `@AspectJ`.** Помимо пространства имен `aop`, ориентированного на формат XML, для конфигурирования реализации АОП в Spring можно также применять аннотации в стиле `@AspectJ` непосредственно в классах. И хотя используемый в этом случае синтаксис основан на языке AspectJ и требует подключения ряда библиотек AspectJ, в Spring по-прежнему применяется механизм заместителей (т.е. создаются замещаемые объекты для целевых объектов) во время начальной загрузки контекста типа `ApplicationContext`.

Применение класса `ProxyFactoryBean`

Класс `ProxyFactoryBean` является реализацией интерфейса `FactoryBean`, позволяющей указать целевой компонент Spring Bean и предоставляющей для него ряд советов и аспектов, которые впоследствии объединяются в заместитель АОП. Класс `ProxyFactoryBean` служит для применения логики перехвата в уже существующем целевом компоненте Spring Bean таким образом, чтобы перехватчики выполнялись как до, так и после методов, вызываемых для данного компонента. А поскольку вместе с классом `ProxyFactoryBean` можно использовать как совет, так и советник, декларативное конфигурирование доступно не только для советов, но и для срезов.

Класс ProxyFactoryBean разделяет общий интерфейс (org.springframework.aop.framework.Advised) с классом ProxyFactory (оба класса косвенно расширяют класс org.springframework.aop.framework.AdvisedSupport, реализующий интерфейс Advised), и поэтому в нем доступно немало тех же самых признаков, в том числе frozen, optimize и exposeProxy. Значения для этих признаков передаются непосредственно базовому классу ProxyFactory, который также позволяет декларативно конфигурировать фабрику.

Демонстрация класса ProxyFactoryBean в действии

Пользоваться классом ProxyFactoryBean очень просто: достаточно определить сначала целевой компонент Spring Bean, а затем, используя класс ProxyFactoryBean, определить тот компонент, к которому приложение будет получать доступ через целевой компонент, служащий в качестве цели заместителя. Целевой компонент Spring Bean следует определять там, где это возможно, как анонимный компонент в самом объявлении компонента заместителя. Благодаря этому предотвращается случайный доступ из приложения к компоненту, не снабженному советом. Но в некоторых случаях, как в рассматриваемом здесь примере, может возникнуть потребность в создании нескольких заместителей для одного и того же компонента Spring Bean, и тогда следует воспользоваться обычным компонентом верхнего уровня.

Для рассматриваемого здесь примера допустим, что имеется певец, работающий с документалистом над созданием фильма о своем гастрольном туре. В данном случае у класса Documentarist имеется зависимость от реализации интерфейса Singer. Используемая здесь реализация интерфейса Singer была представлена ранее в классе GrammyGuitarist и еще раз приведена ниже.

```
package com.apress.prospring5.ch5;

import com.apress.prospring5.ch2.common.Guitar;
import com.apress.prospring5.ch2.common.Singer;

public class GrammyGuitarist implements Singer {
  @Override public void sing() {
    System.out.println("sing: Gravity is working "
              + "against me\n"
              + "And gravity wants to bring me down");
  }

  public void sing(Guitar guitar) {
    System.out.println("play: " + guitar.play());
  }

  public void rest() {
    System.out.println("zzz");
  }
```

```
  public void talk(){
    System.out.println("talk");
  }
}
```

Ниже приведен класс `Documentarist`, в котором, по существу, сообщается, что именно певец должен делать во время съемок документального фильма.

```
package com.apress.prospring5.ch5;

public class Documentarist {
  private GrammyGuitarist guitarist;

  public void execute() {
    guitarist.sing();
    guitarist.talk();
  }

  public void setDep(GrammyGuitarist guitarist) {
    this.guitarist = guitarist;
  }
}
```

Создадим для рассматриваемого здесь примера два заместителя одного экземпляра класса `GrammyGuitarist`, причем оба с одним и тем же простым советом:

```
package com.apress.prospring5.ch5;

import org.springframework.aop.MethodBeforeAdvice;
import java.lang.reflect.Method;

public class AuditAdvice implements MethodBeforeAdvice {
  @Override
  public void before(Method method, Object[] args,
                     Object target) {
    System.out.println("Executing: " + method);
  }
}
```

Первый заместитель служит лишь для того, чтобы снабдить советом целевой объект, а следовательно, и все его методы, используя совет непосредственно. А для второго заместителя сконфигурируем срез типа `AspectJExpressionPointcut` и советник типа `DefaultPointcutAdvisor`, чтобы снабдить советом только метод `sing()` из класса `GrammySinger`. Чтобы проверить совет, определим два компонента Spring Beans типа `Documentarist`, каждый из которых будет внедрен с помощью отдельного заместителя. Затем для каждого из этих компонентов будет вызван метод `execute()`, чтобы посмотреть, что происходит при вызове снабженных советом методов для зависимости.

Конфигурация из файла `app-context-xml.xml` для данного примера приведена на рис. 5.10. Она представлена здесь в виде иллюстрации потому, что выглядит немного запутанно, а нам хотелось ясно показать, где именно внедряется каждый компонент Spring Bean. В данном примере мы просто устанавливаем свойства непосредственно в коде, используя возможности внедрения зависимостей в Spring. Единственный интерес представляет применение анонимного компонента Spring Bean для среза, а также класса `ProxyFactoryBean`. Мы предпочитаем пользоваться анонимными компонентами Spring Beans для срезов, если они не являются общими, поскольку это дает возможность сохранить непосредственно доступные компоненты компактными и как можно более уместными для приложения. Применяя класс `ProxyFactoryBean`, очень важно уяснить, что объявление этого класса доступно как для

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/util
       http://www.springframework.org/schema/util/spring-util.xsd">

       <bean id="johnMayer"class="com.apress.prospring5.ch5.GrammyGuitarist"/>
       <bean id="advice" class="com.apress.prospring5.ch5.AuditAdvice"/>

       <bean id="documentaristOne" class="com.apress.prospring5.ch5.Documentarist"
             p:guitarist-ref="proxyOne"/>

       <bean id="proxyOne" class="org.springframework.aop.framework.ProxyFactoryBean"
             p:target-ref="johnMayer"
             p:interceptorNames-ref="interceptorAdviceNames"/>

       <util:list id="interceptorAdviceNames">
              <value>advice</value>
       </util:list>


       <bean id="documentaristTwo" class="com.apress.prospring5.ch5.Documentarist"
             p:guitarist-ref="proxyTwo"/>

       <bean id="proxyTwo" class="org.springframework.aop.framework.ProxyFactoryBean"
             p:target-ref="johnMayer"
             p:interceptorNames-ref="interceptorAdvisorNames"/>

       <util:list id="interceptorAdvisorNames">
              <value>advisor</value>
       </util:list>

       <bean id="advisor" class="org.springframework.aop.support.DefaultPointcutAdvisor"
              p:advice-ref="advice">
              <property name="pointcut">
                     <bean class="org.springframework.aop.aspectj.AspectJExpressionPointcut"
                           p:expression="execution(* sing*(..))"/>
              </property>
       </bean>

</beans>
```

Рис. 5.10. Декларативное конфигурирование АОП

приложения, так и для воплощения зависимостей. Базовый целевой компонент Spring Bean не снабжается советом, и поэтому его следует применять только в том случае, если требуется обойти каркас АОП, хотя в приложении, как правило, не должно быть ничего известно о каркасе АОП, а значит, и обходить его не нужно. Именно поэтому следует пользоваться анонимными компонентами Spring Beans везде, где это только возможно, чтобы исключить случайный доступ к ним из приложения.

В следующем фрагменте кода демонстрируется простой класс, в котором сначала получаются два экземпляра типа `Documentarist` из контекста типа `Application Context`, а затем для каждого из них выполняется метод `execute()`:

```
package com.apress.prospring5.ch5;

import org.springframework.context.support
          .GenericXmlApplicationContext;

public class ProxyFactoryBeanDemo {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("spring/app-context-xml.xml");
    ctx.refresh();

    Documentarist documentaristOne = ctx.getBean(
                  "documentaristOne", Documentarist.class);
    Documentarist documentaristTwo = ctx.getBean(
                  "documentaristTwo", Documentarist.class);

    System.out.println("Documentarist One >>");
    documentaristOne.execute();

    System.out.println("\nDocumentarist Two >> ");
    documentaristTwo.execute();
  }
}
```

После выполнения исходного кода из данного примера на консоль будет выведен следующий результат:

```
Documentarist One >>
Executing: public void com.apress.prospring5
           .ch5.GrammyGuitarist.sing()
sing: Gravity is working against me
And gravity wants to bring me down
Executing: public void com.apress.prospring5
          .ch5.GrammyGuitarist.talk()
talk

Documentarist Two >>
Executing: public void com.apress.prospring5
```

```
            .ch5.GrammyGuitarist.sing()
sing: Gravity is working against me
And gravity wants to bring me down
talk
```

Как и следовало ожидать, оба метода, `sing()` и `talk()`, из первого заместителя снабжаются советом, поскольку для его конфигурирования *не* был использован срез. Но во втором заместителе советом был снабжен только метод `sing()`, поскольку для его конфигурирования был использован срез.

Применение класса `ProxyFactoryBean` для введений

Применение класса ProxyFactoryBean не ограничивается одним лишь снабжением объекта советом. Его можно использовать и для введения примесей в объекты. Как упоминалось ранее при обсуждении введений, внедрить введение можно только через интерфейс `IntroductionAdvisor`, поскольку сделать это напрямую нельзя. Это же правило распространяется и на применение класса `ProxyFactoryBean` для введений. Применяя класс `ProxyFactoryBean`, намного проще сконфигурировать заместители, если для примеси создана специальная реализация интерфейса `Advisor`. Ниже приведен пример конфигурации из файла `app-context-xml.xml` для настройки введения `IsModifiedMixin`, представленного ранее в этой главе.

```
<beans ...>

  <bean id="guitarist"
    class="com.apress.prospring5.ch2.common.Contact"
    p:name="John Mayer"/>
  <bean id="advisor"
    class="com.apress.prospring5.ch5.introduction
              .IsModifiedAdvisor"/>

  <util:list id="interceptorAdvisorNames">
    <value>advisor</value>
  </util:list>

  <bean id="bean"
    class="org.springframework.aop
              .framework.ProxyFactoryBean"
    p:target-ref="guitarist"
    p:interceptorNames-ref="interceptorAdvisorNames"
    p:proxyTargetClass="true">
  </bean>

</beans>
```

Как следует из приведенной выше конфигурации, класс `IsModifiedAdvisor` служит в качестве советника для класса `ProxyFactoryBean`, а поскольку создавать другой заместитель для того же самого целевого объекта не нужно, то используется

анонимное объявление компонента Spring Bean. Ниже приведен видоизмененный вариант предыдущего примера введения, в котором заместитель получается из контекста типа `ApplicationContext`.

```
package com.apress.prospring5.ch5;

import com.apress.prospring5.ch2.common.Contact;
import com.apress.prospring5.ch5.introduction.IsModified;
import org.springframework.context.support
          .GenericXmlApplicationContext;

public class IntroductionConfigDemo {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();

    Contact bean = (Contact) ctx.getBean("bean");
    IsModified mod = (IsModified) bean;

    System.out.println("Is Contact?: "
                    + (bean instanceof Contact));
    System.out.println("Is IsModified?: "
                    + (bean instanceof IsModified));
    System.out.println("Has been modified?: "
                     + mod.isModified());
    bean.setName("John Mayer");

    System.out.println("Has been modified?: "
                     + mod.isModified());
    bean.setName("Eric Clapton");

    System.out.println("Has been modified?: "
                     + mod.isModified());
  }
}
```

Выполнение исходного кода из данного примера дает точно такой же вывод, как и в предыдущем примере введения, но на этот раз заместитель получается из контекста типа `ApplicationContext`, а в прикладном коде отсутствует всякая конфигурация. Но поскольку мы уже рассматривали порядок конфигурирования на Java, то представленную выше конфигурацию в формате XML можно заменить конфигурационным классом, как показано ниже.

```
package com.apress.prospring5.ch5.config;

import com.apress.prospring5.ch2.common.Contact;
import com.apress.prospring5.ch5
          .introduction.IsModifiedAdvisor;
```

```
import org.springframework.aop.Advisor;
import org.springframework.aop.framework.ProxyFactoryBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context
         .annotation.Configuration;

@Configuration
public class AppConfig {
  @Bean
  public Contact guitarist() {
    Contact guitarist = new Contact();
    guitarist.setName("John Mayer");
    return guitarist;
  }

  @Bean
  public Advisor advisor() {
    return new IsModifiedAdvisor();
  }

  @Bean ProxyFactoryBean bean() {
    ProxyFactoryBean proxyFactoryBean =
                     new ProxyFactoryBean();
    proxyFactoryBean.setTarget(guitarist());
    proxyFactoryBean.setProxyTargetClass(true);
    proxyFactoryBean.addAdvisor(advisor());
    return proxyFactoryBean;
  }
}
```

Чтобы проверить работоспособность приведенного выше класса, достаточно заменить в методе `main()` из класса `IntroductionConfigDemo` строки кода, в которых инициализируется контекст, следующими строками кода:

```
GenericApplicationContext ctx =
       new AnnotationConfigApplicationContext(
           AppConfig.class);
```

Конфигурационный класс отличается тем, что в нем не нужно ссылаться на компонент `advisor` по имени или вводить его в список, чтобы предоставить в качестве аргумента конструктору класса `ProxyFactoryBean`, поскольку метод `addAdvisor(..)` можно вызвать непосредственно, предоставив ему компонент `advisor` в качестве аргумента. Очевидно, что это простейшая конфигурация.

Краткие итоги по классу `ProxyFactoryBean`

Применяя класс `ProxyFactoryBean`, можно конфигурировать заместители АОП и, таким образом, обеспечить гибкость программного способа, не привязывая прикладной код к конфигурированию АОП. Но если на стадии выполнения не требуется

принимать решение о порядке создания заместителей, то лучше придерживаться декларативного, а не программного способа конфигурирования заместителей. А теперь перейдем к рассмотрению еще двух вариантов декларативного конфигурирования АОП в Spring. Оба варианта рекомендуются для разработки приложений, основанных на каркасе Spring, начиная с версии 2.0, а также на комплекте JDK, начиная с версии 5.

Применение пространства имен `aop`

Пространство имен `aop` предоставляет существенно упрощенный синтаксис для декларативного конфигурирования АОП в Spring. Чтобы продемонстрировать его в действии, воспользуемся предыдущим примером применения класса `ProxyFactoryBean`, внеся в него незначительные коррективы. В данном примере по-прежнему применяется класс `GrammyGuitarist`. Тем не менее класс `Documentarist` будет расширен для вызова метода `sing()` с аргументом типа `Guitar`, как показано ниже.

```
package com.apress.prospring5.ch5;

import com.apress.prospring5.ch2.common.Guitar;

public class NewDocumentarist extends Documentarist {
  @Override
  public void execute() {
    guitarist.sing();
    guitarist.sing(new Guitar());
    guitarist.talk();
  }
}
```

Коррективы в класс совета внесены следующим образом:

```
package com.apress.prospring5.ch5;

import org.aspectj.lang.JoinPoint;

public class SimpleAdvice {
  public void simpleBeforeAdvice(JoinPoint joinPoint) {
    System.out.println("Executing: "
          + joinPoint.getSignature().getDeclaringTypeName()
          + " " + joinPoint.getSignature().getName());
  }
}
```

Как видите, в классе совета больше не требуется реализовывать интерфейс `MethodBeforeAdvice`. Кроме того, метод, реализующий предшествующий совет, принимает в качестве аргумента срез, но не метод, объект и аргументы. На самом деле этот аргумент не является обязательным, и поэтому вызов данного метода можно сделать из без аргументов. Но если в совете требуется доступ к подробным сведениям о сре-

зе (в данном случае для того, чтобы вывести сведения о вызывающем типе и имени метода), то следует указать принимаемый аргумент. Если аргумент для данного метода определен, то Spring автоматически передаст ему срез для последующей обработки. Ниже приведена конфигурация АОП с пространством имен `aop` в Spring, выполненная в формате XML и сохраненная в файле `app-context-xml.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation=
          "http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans
                 /spring-beans.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop
                /spring-aop.xsd">

  <aop:config>
    <aop:pointcut id="singExecution"
      expression="execution(
           * com.apress.prospring5.ch5..sing*(
                com.apress.prospring5.ch2.common.Guitar)
     )"/>

    <aop:aspect ref="advice">
      <aop:before pointcut-ref="singExecution"
        method="simpleBeforeAdvice"/>
    </aop:aspect>
  </aop:config>

  <bean id="advice"
    class="com.apress.prospring5.ch5.SimpleAdvice"/>
  <bean id="johnMayer"
    class="com.apress.prospring5.ch5.GrammyGuitarist"/>
  <bean id="documentarist"
    class="com.apress.prospring5.ch5.NewDocumentarist"
    p:guitarist-ref="johnMayer"/>
</beans>
```

Во-первых, необходимо объявить пространство имен `aop` в дескрипторе разметки `<beans>`. Во-вторых, вся конфигурация АОП в Spring размещается в дескрипторе разметки `<aop:config>`. В этом дескрипторе можно определить срез, аспекты, советники, советы и прочие элементы АОП, а также ссылаться на другие компоненты Spring Beans обычным образом.

В приведенной выше конфигурации определен срез с идентификатором `singExecution`. В частности, выражение

```
"execution(*
  com.apress.prospring5.ch5..sing*(
      com.apress.prospring5.ch2.common.Guitar)
)"
```

означает, что советом требуется снабдить все методы с префиксом **`sing`** в классах, определенных в пакете `com.apress.prospring5.ch5` (включая все его подпакеты). Кроме того, метод `sing()` должен принимать один аргумент типа `Guitar`. Далее в дескрипторе разметки `<aop:aspect>` объявлен аспект со ссылкой на класс совета как на компонент Spring Bean с идентификатором `advice`, который представлен классом `SimpleAdvice`.

В атрибуте `pointcut-ref` указана ссылка на определенный срез с идентификатором `singExecution`, а предшествующий совет, объявленный с помощью дескриптора `<aop:before>`, реализован в методе `simpleBeforeAdvice()` из компонента совета. Для проверки приведенной выше конфигурации можно воспользоваться следующим классом:

```
package com.apress.prospring5.ch5;

import org.springframework.context.support
          .GenericXmlApplicationContext;

public class AopNamespaceDemo {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml-01.xml");
    ctx.refresh();
    NewDocumentarist documentarist = ctx.getBean(
              "documentarist", NewDocumentarist.class);
    documentarist.execute();

    ctx.close();
  }
}
```

В данном примере просто инициализируется контекст типа `ApplicationContext`, компонент Spring Bean извлекается обычным образом и вызывается его метод `execute()`. Выполнение исходного кода этой тестовой программы приведет к выводу на консоль следующего результата:

```
sing: Gravity is working against me
And gravity wants to bring me down
Executing: com.apress.prospring5.ch5.GrammyGuitarist sing
play: G C G C Am D7
talk
```

Как видите, советом был снабжен только вызов метода `sing()` с аргументом типа `Guitar`, но не его вызов без аргументов и не вызов метода `talk()`. Рассмотренный здесь пример действует именно так, как и предполагалось. Обратите внимание на то, что конфигурация в данном случае значительно упростилась по сравнению с конфигурацией, составленной с помощью класса `ProxyFactoryBean`.

А теперь переделаем предыдущий пример для более сложного случая. Допустим, требуется снабдить советом только те методы из компонентов Spring Beans, идентификаторы которых начинаются на **`john`**, а параметр относится к типу `Guitar`, имеющему свойство `brand` с установленным значением `Gibson`.

С этой целью придется внести коррективы в класс `Guitar`, чтобы добавить в него свойство `brand`. Сделаем его необязательным и установим в нем значение по умолчанию лишь для того, чтобы сохранить работоспособность предыдущих примеров.

```
package com.apress.prospring5.ch2.common;

public class Guitar {
  private String brand =" Martin";

  public String play() {
    return "G C G C Am D7";
  }

  public String getBrand() {
    return brand;
  }

  public void setBrand(String brand) {
    this.brand = brand;
  }
}
```

Затем необходимо сделать вызов метода `sing()` из класса `NewDocumentarist` со специально указанной маркой гитары.

```
package com.apress.prospring5.ch5;

import com.apress.prospring5.ch2.common.Guitar;

public class NewDocumentarist extends Documentarist {
  @Override
  public void execute() {
    guitarist.sing();
    Guitar guitar = new Guitar();
    guitar.setBrand("Gibson");
    guitarist.sing(guitar);
    guitarist.talk();
  }
}
```

А теперь нам потребуется новый и более сложный тип совета. С этой целью в сигнатуру метода, реализующего предшествующий совет, вводится аргумент `guitar`. Кроме того, в данном методе организуется проверка и выполняется соответствующая логика лишь в том случае, если свойство, передаваемое в качестве аргумента `brand`, содержит значение `Gibson`.

```
package com.apress.prospring5.ch5;

import com.apress.prospring5.ch2.common.Guitar;
import org.aspectj.lang.JoinPoint;

public class ComplexAdvice {
  public void simpleBeforeAdvice(JoinPoint joinPoint,
                                 Guitar value) {
    if(value.getBrand().equals("Gibson")) {
      System.out.println("Executing: "
          + joinPoint.getSignature().getDeclaringTypeName()
          + " " + joinPoint.getSignature().getName());
    }
  }
}
```

Необходимо также пересмотреть конфигурацию в формате XML, поскольку в данном случае требуется новый тип совета и обновление выражения со срезом. (Полная конфигурация для данного примера находится в файле `app-context-xml-02.xml`. Она похожа на конфигурацию из файла `app-context-xml-01.xml`, за исключением нескольких строк кода, и поэтому ниже приведены только они.)

```
<beans ..>
...
  <bean id="advice"
    class="com.apress.prospring5.ch5.ComplexAdvice"/>
  <aop:config>
    <aop:pointcut id="singExecution"
      expression="execution(* sing*(
                com.apress.prospring5.ch2.common.Guitar))
                and args(value) and bean(john*)"/>
</beans>
```

В выражение со срезом введены две дополнительные директивы. В первой из них, `args(Value)`, сообщается о необходимости передать аргумент `Value` предшествующему совету. А во второй директиве, `bean(john*)`, каркасу Spring предписывается снабдить советом только компоненты Spring Beans с идентификатором, имеющим префикс **`john`**. Это довольно эффективное средство. При наличии четко определенной структуры именования компонентов Spring Beans можно очень легко снабдить советом только требуемые объекты. Например, используя директиву `bean(*DAO*)`, можно применить совет ко всем компонентам DAO, а с помощью директивы `bean(`

Service) — ко всем компонентам Spring Beans, относящимся к уровням всех служб, не указывая полностью уточненные имена классов для сопоставления. Выполнение исходного кода той же самой тестовой программы с новым файлом конфигурации app-context-xml02.xml даст приведенный ниже результат. Как видите, советом снабжен только метод sing(), вызываемый с аргументом типа Guitar, где значение свойства brand равно Gibson.

```
sing: Gravity is working against me
And gravity wants to bring me down
Executing: com.apress.prospring5.ch5.GrammyGuitarist sing
play: G C G C Am D7
talk
```

Рассмотрим еще один пример применения пространства имен aop для окружающего совета. Вместо создания очередного класса для реализации интерфейса Method Interceptor достаточно ввести новый метод в класс ComplexAdvice. В следующем примере кода демонстрируется новый метод simpleAroundAdvice() в переделанном классе ComplexAdvice:

```
// Файл ComplexAdvice.java
public Object simpleAroundAdvice(ProceedingJoinPoint pjp,
              Guitar value) throws Throwable {
  System.out.println("Before execution: "
           + pjp.getSignature().getDeclaringTypeName()
           + " " + pjp.getSignature().getName()
           + " argument: " + value.getBrand());
  Object retVal = pjp.proceed();
  System.out.println("After execution: "
           + pjp.getSignature().getDeclaringTypeName()
           + " " + pjp.getSignature().getName()
           + " argument: " + value.getBrand());
  return retVal;
}
```

Вновь введенный метод simpleAroundAdvice() должен получить по крайней мере один аргумент типа ProceedingJoinPoint, чтобы продолжить свое выполнение, обратившись к целевому объекту. Кроме того, его сигнатура дополнена аргументом value с целью отобразить его значение в самом совете.

Конфигурацию в формате XML с помощью дескриптора разметки <aop:aspect> необходимо адаптировать, чтобы ввести новый совет. (Полная конфигурация для данного примера находится в файле app-context-xml-03.xml. Она похожа на конфигурацию из файла app-context-xml-02.xml, за исключением нескольких строк кода, и поэтому ниже приведены только они.)

```
<beans ..>
...
  <aop:config>
    <aop:pointcut id="singExecution"
```

```
      expression="execution(
          * sing*(com.apress.prospring5.ch2.common.Guitar))
          and args(value) and bean(john*)"
    />

    <aop:aspect ref="advice">
      <aop:before pointcut-ref="singExecution"
         method="simpleBeforeAdvice"/>
      <aop:around pointcut-ref="singExecution"
         method="simpleAroundAdvice"/>
    </aop:aspect>
  </aop:config>
</beans>
```

Здесь был введен лишь новый дескриптор `<aop:around>` для объявления окружающего совета и ссылки на тот же самый срез. А теперь внесем очередные коррективы в метод `NewDocumentarist.execute()`, чтобы включить в него вызов метода `sing()` с получаемым по умолчанию экземпляром типа `Guitar` и добиться в итоге поведения, которое требуется проанализировать.

```
package com.apress.prospring5.ch5;

import com.apress.prospring5.ch2.common.Guitar;

public class NewDocumentarist extends Documentarist {
  @Override
  public void execute() {
    guitarist.sing();
    Guitar guitar = new Guitar();
    guitar.setBrand("Gibson");
    guitarist.sing(guitar);
    guitarist.sing(new Guitar());
    guitarist.talk();
  }
}
```

Если выполнить тестовую программу снова, то на консоль будет выведен следующий результат:

```
sing: Gravity is working against me
And gravity wants to bring me down

Executing:[13] com.apress.prospring5.ch5.GrammyGuitarist sing
Before execution:[14] com.apress.prospring5.ch5.GrammyGuitarist
sing argument: Gibson
play: G C G C Am D7
```

[13] Выполнение:

[14] До выполнения:

```
After execution: com.apressprospring5.ch5.GrammyGuitarist
sing argument: Gibson

Before execution: com.apress.prospring5.ch5.GrammyGuitarist
sing argument: Martin
play: G C G C Am D7
After execution:[15] com.apress.prospring5.ch5.GrammyGuitarist
sing argument: Martin
talk
```

В приведенном выше результате необходимо отметить два интересных момента. Во-первых, окружающий совет применен к обоим вызовам метода `sing(..)` с аргументом типа `Guitar`, поскольку в нем проверяется аргумент. И во-вторых, для вызова метода `sing()` с передачей значения `"Gibson"` в качестве аргумента типа `Guitar` были выполнены предшествующий и окружающий советы, причем преимущество по умолчанию было отдано предшествующему совету.

На заметку Если применяется пространство имен **`aop`** или аннотации в стиле **`@AspectJ`**, то для этой цели имеются два типа последующего совета. В частности, *послевозвратный совет* (определяемый в дескрипторе разметки **`<aop:after-returning>`**) применяется только в том случае, если целевой метод завершается обычным образом. А *обычный последующий совет* (определяемый в дескрипторе разметки **`<aop:after>`**) выполняется как после завершения метода обычным образом, так и при возникновении в методе ошибки и генерирования соответствующего исключения. Если же требуется, чтобы совет применялся независимо от результата выполнения целевого метода, следует использовать обычный последующий совет.

Применение аннотаций в стиле `@AspectJ`

Если реализация АОП в Spring применяется вместе с комплектом JDK, начиная с версии 5, то для объявления совета можно также воспользоваться аннотациями в стиле `@AspectJ`. Но, как было указано ранее, для снабжения советами целевых методов в Spring по-прежнему применяется собственный механизм замещения, а не механизм связывания на языке AspectJ.

В этом разделе будет показано, как реализовать такие же аспекты, как и построенные с помощью пространства имен `aop`, но с помощью аннотаций в стиле `@AspectJ`. Язык AspectJ является универсальным аспектно-ориентированным расширением языка Java, предназначенным для решения вопросов или задач, недостаточно охваченных традиционными методиками программирования и называемых иначе сквозной функциональностью. В примерах из данного раздела аннотации будут применяться также и для других компонентов Spring Beans. А кроме того, в них будут использоваться конфигурационные классы Java.

---

[15] `После выполнения:`

В следующем примере приведен класс `GrammyGuitarist`, где компонент Spring Bean объявляется с помощью аннотаций:

```
package com.apress.prospring5.ch5;

import com.apress.prospring5.ch2.common.Guitar;
import com.apress.prospring5.ch2.common.Singer;
import org.springframework.stereotype.Component;

@Component("johnMayer")
public class GrammyGuitarist implements Singer {
  @Override public void sing() {
    System.out.println("sing: Gravity is working "
            + "against me\n"
            + "And gravity wants to bring me down");
  }

  public void sing(Guitar guitar) {
    System.out.println("play: " + guitar.play());
  }

  public void rest() {
    System.out.println("zzz");
  }

  public void talk(){
    System.out.println("talk");
  }
}
```

Необходимо также адаптировать класс `NewDocumentarist`, как показано ниже.

```
package com.apress.prospring5.ch5;

import com.apress.prospring5.ch2.common.Guitar;
import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.beans.factory
          .annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component("documentarist")
public class NewDocumentarist {
  protected GrammyGuitarist guitarist;

  public void execute() {
    guitarist.sing();
    Guitar guitar = new Guitar();
    guitar.setBrand("Gibson");
    guitarist.sing(guitar);
```

```
    guitarist.talk();
  }

  @Autowired
  @Qualifier("johnMayer")
  public void setGuitarist(GrammyGuitarist guitarist) {
    this.guitarist = guitarist;
  }
}
```

Оба приведенных выше класса снабжены аннотацией `@Component`, где им присвоены соответствующие имена. В классе `NewDocumentarist` метод установки значений в свойстве `guitarist` снабжен аннотацией `@Autowired` для автоматического внедрения средствами Spring.

А теперь рассмотрим класс `AnnotationAdvice`, в котором применяются аннотации в стиле `@AspectJ`. С этой целью реализуем срезы вместе с предшествующим и окружающим советами, как показано в приведенном ниже исходном коде класса `AnnotationAdvice`.

```
package com.apress.prospring5.ch5;

import com.apress.prospring5.ch2.common.Guitar;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class AnnotatedAdvice {
  @Pointcut("execution(*com.apress.prospring5.ch5.sing*(
            com.apress.prospring5.ch2.common.Guitar))
      && args(value)")
  public void singExecution(Guitar value) {
  }

  @Pointcut("bean(john*)")
  public void isJohn() {
  }

  @Before("singExecution(value) && isJohn()")
  public void simpleBeforeAdvice(JoinPoint joinPoint,
                                 Guitar value) {
    if(value.getBrand().equals("Gibson")) {
      System.out.println("Executing: "
```

```
            + joinPoint.getSignature().getDeclaringTypeName()
            + " " + joinPoint.getSignature().getName()
            + " argument: " + value.getBrand());
    }
  }

  @Around("singExecution(value) && isJohn()")
  public Object simpleAroundAdvice(ProceedingJoinPoint pjp,
                Guitar value) throws Throwable {
    System.out.println("Before execution: "
              + pjp.getSignature().getDeclaringTypeName()
              + " " + pjp.getSignature().getName()
              + " argument: " + value.getBrand());

    Object retVal = pjp.proceed();

    System.out.println("After execution: "
              + pjp.getSignature().getDeclaringTypeName()
              + " " + pjp.getSignature().getName()
              + " argument: " + value.getBrand());

    return retVal;
  }
}
```

Обратите внимание на то, что структура приведенного выше кода очень похожа на ту, что использовалась в примере с пространством имен `aop`, только в данном случае применяются аннотации. Тем не менее здесь важно отметить следующие моменты.

- Для аннотирования класса `AnnotatedAdvice` использованы аннотации `@Component` и `@Aspect`. Так, в аннотации `@Aspect` данный класс объявляется как аспект. Данный класс необходимо также снабдить аннотацией `@Component`, чтобы дать каркасу Spring возможность просматривать компонент, когда в конфигурации, составленной в формате XML, применяется дескриптор разметки `<context:component-scan>`.
- Срезы определены как методы, ничего не возвращающие (тип `void`). В данном классе определены два среза, снабженных аннотацией `@Pointcut`. Выражение со срезами было намеренно разделено на две части в примере с пространством имен `aop`. В первой части данного выражения, обозначаемой как вызов метода `singExecution(Guitar value))`, определяется срез для выполнения методов `sing*()` во всех классах из пакета `com.apress.prospring5.ch5` с аргументом `value` типа `Guitar`, который будет также передан совету. Во второй части данного выражения выражение, обозначаемой как вызов метода `isJohn()`, определяется другой срез, предназначенный для выполнения всех методов из компонентов Spring Beans, имена которых начинаются с префикса **`john`**. Обратите также внимание на то, что для обозначения условия объедине-

ния по "И" в рассматриваемом здесь выражении со срезами применяется логическая операция `&&`, тогда как в примере с пространством имен `aop` для этой цели использовалась логическая операция `and`.

- Метод, реализующий предшествующий совет, снабжен аннотацией `@Before`, а метод, реализующий окружающий совет, — аннотацией `@Around`. Для обоих советов передается значение, в котором указаны оба среза, определенных в данном классе. Так, значение `singExecution(value) && isJohn()` означает, что для применения совета должны совпасть оба среза, аналогично операции пересечения в классе `ComposablePointcut`.
- Логика предшествующего и окружающего советов такая же, как и в примере с пространством имен `aop`.

При наличии всех упомянутых выше аннотаций конфигурация в формате XML заметно упрощается, как показано ниже.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context=
            "http://www.springframework.org/schema/context"
       xsi:schemaLocation=
          "http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop
                 /spring-aop.xsd
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans
                 /spring-beans.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context
                /spring-context.xsd">

       <aop:aspectj-autoproxy/>

       <context:component-scan
          base-package="com.apress.prospring5.ch5"/>
</beans>
```

Здесь объявлены только два дескриптора разметки. В частности, дескриптор `<aop:aspect-autorpoxy>` служит для извещения каркаса Spring о необходимости просмотреть аннотации в стиле `@AspectJ`, а дескриптор `<context:component-scan>` по-прежнему требуется в Spring для просмотра компонентов Spring Beans в том пакете, где находится совет. Необходимо также снабдить аннотацией `@Component` класс, реализующий совет, указав на то, что он является компонентом Spring Bean.

В следующем фрагменте кода приведен класс, предназначенный для тестирования упомянутой выше конфигурации:

```
package com.apress.prospring5.ch5;

import org.springframework.context.support
          .GenericXmlApplicationContext;

public class AspectJAnnotationDemo {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();

    NewDocumentarist documentarist = ctx.getBean(
              "documentarist", NewDocumentarist.class);
    documentarist.execute();
  }
}
```

Если вы выполните исходный код из данного примера как есть, то вас несколько удивит результат, выведенный на консоль и показанный ниже.

```
Exception in thread "main"
org.springframework.beans.factory
   .UnsatisfiedDependencyException:
Error creating bean with name 'documentarist':
  Unsatisfied dependency expressed through method
  'setGuitarist' parameter 0; nested exception is
  org.springframework.beans.factory
     .BeanNotOfRequiredTypeException:
  Bean named 'johnMayer' is expected to be of type
  'com.apress.prospring5.ch5.GrammyGuitarist' but
  was actually of type 'com.sun.proxy.$Proxy18'[16]
...
```

[16] Исключение в потоке "главный"
org.springframework.beans.factory
   .UnsatisfiedDependencyException:
Ошибка создания компонента Spring Bean
под именем 'documentarist':
Неудовлетворенная зависимость, выражаемая через
параметр 0 метода 'setGuitarist';
вложенное исключение: org.springframework
    .beans.factory.BeanNotOfRequiredTypeException:
Ожидалось, что компонент Spring Bean под именем 'johnMayer'
относится к типу 'com.apress.prospring5.ch5.GrammyGuitarist',
а фактически – к типу 'com.sun.proxy.$Proxy18'

Что же здесь происходит? В классе GrammyGuitarist реализуется интерфейс Singer и по умолчанию создаются основанные на интерфейсах динамические заместители из комплекта JDK. Но в классе NewDocumentarist явно требуется зависимость от класса GrammyGuitarist или его расширения, и поэтому генерируется указанное выше исключение. Как же исправить эту ошибку? Это можно сделать двумя способами. Во-первых, внести коррективы в класс NewDocumentarist, чтобы внедрить в него зависимость от интерфейса Singer, хотя это неприемлемо для данного примера, поскольку в нем требуется доступ к тем методам из класса GrammyGuitarist, которые не являются реализациями методов, определенных в интерфейсе Singer. И во-вторых, можно запросить каркас Spring сформировать основанные на классах заместители из библиотеки CGLIB. Для этого достаточно откорректировать конфигурацию в формате XML, установив логическое значение true в атрибуте proxy-target-class дескриптора <aop:aspectj-autoproxy/>.

Еще проще внести следующие коррективы в конфигурационный класс Java:

```
@Configuration
@ComponentScan(basePackages = {"com.apress.prospring5.ch5"})
@EnableAspectJAutoProxy(proxyTargetClass = true)
public class AppConfig {
}
```

Обратите внимание на аннотацию @EnableAspectJAutoProxy, которая равнозначна дескриптору разметки <aop:aspectj-autoproxy/> и обладает атрибутом proxyTargetClass, аналогичным атрибуту proxy-target-class этого дескриптора. Эта аннотация позволяет поддерживать обращение с компонентами, помеченными аннотацией @Aspect, и предназначена для применения в классах, снабженных аннотацией @Configuration.

Ниже приведена тестовая программа, служащая в качестве контрольного примера JUnit для модульного тестирования обоих вариантов конфигурирования в формате XML и на языке Java, которые могут быть размещены в одном и том же классе. В таких IDE, как IntelliJ IDEA, предоставляется возможность выполнить каждый тестовый метод в отдельности.

```
package com.apress.prospring5.ch5;

import org.junit.Test;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;
import org.springframework.context.support
          .GenericXmlApplicationContext;

public class AspectJAnnotationTest {
  @Test
  public void xmlTest() {
```

```
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();

    NewDocumentarist documentarist = ctx.getBean(
              "documentarist", NewDocumentarist.class);
    documentarist.execute();

    ctx.close();
}

  @Test
  public void configTest() {
    GenericApplicationContext ctx = new
       AnnotationConfigApplicationContext(AppConfig.class);
  .
    NewDocumentarist documentarist = ctx.getBean(
              "documentarist", NewDocumentarist.class);
    documentarist.execute();
    ctx.close();
  }
}
```

Если тест проходит при выполнении любого из приведенных выше тестовых методов, то на консоль выводится следующий результат:

```
sing: Gravity is working against me
And gravity wants to bring me down
Before execution: com.apress.prospring5.ch5
                    .GrammyGuitarist sing argument: Gibson
Executing: com.apress.prospring5.ch5
                   .GrammyGuitarist sing argument: Gibson
play: G C G C Am D7
After execution: com.apress.prospring5.ch5
                   .GrammyGuitarist sing argument: Gibson
talk
```

В модуле Spring Boot предоставляется специальная стартовая библиотека, немного упрощающая конфигурирование АОП. Эта библиотека конфигурируется, как обычно, в файле свойств `pro-spring-15/build.properties` и внедряется как зависимость по ее заданному имени в файл конфигурации `aspectj-boot/build.gradle` порожденного проекта.

```
// файл свойств pro-spring-15/build.properties
ext {
    bootVersion = '2.0.0.BUILD-SNAPSHOT'

    ...
```

```
    boot = [
        springBootPlugin:"org.springframework.boot:
              spring-boot-gradle-plugin:$bootVersion",
        ...
        starterAop:"org.springframework.boot:
               spring-boot-starter-aop:$bootVersion"
    ]
}
// файл конфигурации aspectj-boot/build.gradle
buildscript {
        ...
    dependencies {
        classpath boot.springBootPlugin
    }
}

apply plugin: 'org.springframework.boot'

dependencies {
    compile boot.starterAop
}
```

На рис. 5.11 приведен ряд библиотек, внедренных как зависимости в модуль Spring Boot. Если внедрить одну из этих библиотек как зависимость в свое приложение, то аннотация `@EnableAspectJAutoProxy(proxyTargetClass = true)` больше не понадобится, поскольку поддержка реализации АОП в Spring уже активизирована по умолчанию. Устанавливать атрибут `proxyTargetClass` нигде больше не придется, потому что модуль Spring Boot автоматически обнаруживает тип требующихся заместителей. Если снова обратиться к предыдущему примеру, то из его исходного кода можно удалить класс `AppConfig`, заменив его типичным классом из приложения Spring Boot, как показано ниже.

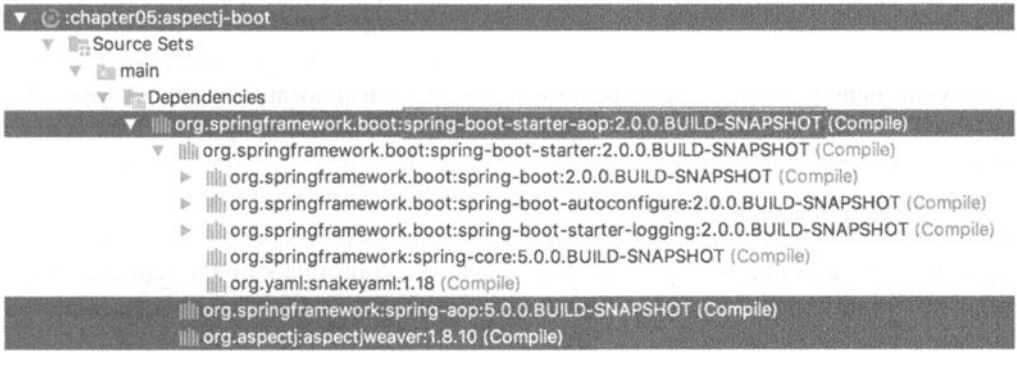


Рис. 5.11. Транзитивные зависимости от стартовых библиотек АОП из модуля Spring Boot в IDE IntelliJ IDEA

```
package com.apress.prospring5.ch5;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

@SpringBootApplication
public class Application {
  private static Logger logger =
          LoggerFactory.getLogger(Application.class);

  public static void main(String args) throws Exception {
    ConfigurableApplicationContext ctx =
            SpringApplication.run(Application.class, args);
    assert (ctx != null);

    NewDocumentarist documentarist = ctx.getBean(
              "documentarist", NewDocumentarist.class);
    documentarist.execute();

    System.in.read();
    ctx.close();
  }
}
```

Соображения по поводу декларативного конфигурирования АОП в Spring

До сих пор мы обсудили три способа определения конфигурирования реализации АОП в Spring, включая класс `ProxyFactoryBean`, пространство имен `aop` и аннотации в стиле `@AspectJ`. Согласитесь, что применять пространство имен `aop` намного проще, чем класс `ProxyFactoryBean`. Итак, остается ответить на главный вопрос: что лучше использовать — пространство имен `aop` или аннотации в стиле `@AspectJ`?

Если приложение Spring основано на конфигурировании в формате XML, то применение пространства имен `aop` является естественным выбором, поскольку оно обеспечивает согласованность стилей конфигурирования АОП и внедрения зависимостей. А если приложение основывается, главным образом, на аннотациях, то следует выбрать аннотации `@AspectJ`. Как обычно, требования к приложению должны определять выбор способа конфигурирования, наилучшим образом соответствующего разрабатываемому проекту.

Кроме того, существуют и другие отличия в подходах к конфигурированию АОП с помощью пространства имен `aop` и аннотаций `@AspectJ`. Они перечислены ниже.

- Синтаксис для выражений со срезами имеет небольшие отличия (например, как упоминалось ранее, при конфигурировании АОП с помощью пространства имен `aop` следует применять логическую операцию `and`, а при конфигурировании АОП с помощью аннотаций `@AspectJ` — логическую операцию `&&`).
- При конфигурировании АОП с помощью пространства имен `aop` поддерживается только модель создания одиночных экземпляров аспектов.
- При конфигурировании АОП с помощью пространства имен `aop` нельзя объединять несколько выражений со срезами. Например, в рассмотренном ранее примере применения аннотаций `@AspectJ` можно было объединить два определения срезов (т.е. `singExecution(value) && isJohn()`) в предшествующих и окружающих советах. Но поступить таким же образом, применяя пространство имен `aop`, не удастся, поэтому придется создать новое выражение со срезами, объединяющее условия сопоставления, воспользовавшись классом `ComposablePointcut`.

Интеграция AspectJ

АОП предлагает эффективное решение многих общих задач, которые возникают при объектно-ориентированной разработке приложений. Применяя реализацию АОП в Spring, можно выбрать подмножество функциональных средств АОП, позволяющее, как правило, решить задачи, возникающие в процессе разработки приложений. Но иногда могут понадобиться функциональные средства АОП, которые выходят за рамки доступных возможностей реализации АОП в Spring.

На уровне точек соединения доступная в Spring реализация АОП поддерживает только срезы, соответствующие выполнению открытых нестатических методов. Но иногда совет необходимо применять к защищенным или закрытым методам при создании объекта или доступе к его полям и т.д.

В подобных случаях следует обратиться к реализации АОП с более полным набором функциональных возможностей. Мы предпочитаем пользоваться языком AspectJ, а поскольку теперь можно конфигурировать аспекты AspectJ с помощью Spring, то язык AspectJ становится идеальным дополнением реализации АОП в Spring.

Общее представление о AspectJ

Язык AspectJ — это полнофункциональная реализация АОП, в которой процесс связывания используется (на стадии компиляции или выполнения) для введения аспектов в прикладной код. В языке AspectJ аспекты и срезы строятся с применением Java-подобного синтаксиса, что упрощает его изучение программирующими на Java. Мы не собираемся уделять слишком много времени исследованию принципа действия AspectJ, потому что этот вопрос выходит за рамки рассмотрения данной книги. Вместо этого мы представим ряд простых примеров применения AspectJ и покажем, как их сконфигурировать средствами Spring. Подробнее о языке AspectJ можно уз-

нать из книги *AspectJ in Action: Enterprise AOP with Spring Applications* Рамниваса Ладдада (Ramnivas Laddad; издательство Manning, 2009 г.).

На заметку Мы не будем описывать, каким образом связывать аспекты AspectJ с приложением. За дополнительными сведениями обращайтесь к документации на AspectJ или же к исходному коду примеров из главы 5, где демонстрируется построение проекта **`aspectj-aspects`** в Gradle.

Применение одиночных экземпляров аспектов

По умолчанию аспекты AspectJ являются одиночными экземплярами в том смысле, что на каждый загрузчик класса получается единственный экземпляр. Затруднение, которое возникает в каркасе Spring в связи с каждым аспектом AspectJ, состоит в том, что в этом каркасе нельзя получить экземпляр аспекта, поскольку это делается непосредственно в AspectJ. Но в каждом аспекте доступен метод `org.aspectj.lang.Aspects.aspectOf()`, который можно использовать для доступа к экземпляру аспекта. Применяя метод `aspectOf()` и специальное средство конфигурирования в Spring, можно поручить конфигурирование аспекта каркасу Spring.

Благодаря такой поддержке можно извлечь наибольшую пользу из эффективного набора функциональных средств АОП, предлагаемого в AspectJ, не теряя превосходных возможностей внедрения зависимостей и конфигурирования в Spring. Это также означает, что в приложении не понадобятся два отдельных способа конфигурирования, поскольку один и тот же подход к конфигурированию контекста типа `ApplicationContext` можно использовать как для всех компонентов Spring Beans, так и для аспектов AspectJ.

Для поддержки аспектов AspectJ в приложении Spring необходимо ввести в конфигурацию модуль, подключаемый к Gradle. Исходный код и инструкции по применению этого подключаемого модуля в приложении Gradle можно найти по адресу `https://github.com/eveoh/gradle-aspectj`, а ниже приведено содержимое файла конфигурации `chapter05/aspectj-aspects/build.gradle`.

```
buildscript {
  repositories {
    mavenLocal()
    mavenCentral()
    maven { url "http://repo.spring.io/release" }
    maven { url "http://repo.spring.io/milestone" }
    maven { url "http://repo.spring.io/snapshot" }
    maven { url "https://repo.spring.io/libs-snapshot" }
    maven { url "https://maven.eveoh.nl/content
          /repositories/releases" }
  }
  dependencies {
    classpath "nl.eveoh:gradle-aspectj:1.6"
```

```
  }
}

apply plugin: 'aspectj'

jar {
  manifest {
    attributes(
        'Main-Class': 'com.apress.prospring5
                       .ch5.AspectJDemo',
         "Class-Path": configurations.compile
                     .collect { it.getName() }.join(' '))
  }
}
```

Ниже приведен исходный код простого класса `MessageWriter`, который будет снабжен советом средствами AspectJ.

```
package com.apress.prospring5.ch5;

public class MessageWriter {
  public void writeMessage() {
    System.out.println("foobar!");
  }

  public void foo() {
    System.out.println("foo");
  }
}
```

В данном примере мы воспользуемся средствами AspectJ, чтобы снабдить метод `writeMessage()` советом и вывести сообщение до и после вызова метода. Выводимые сообщения будут сконфигурированы средствами Spring. Ниже приведен исходный код аспекта `MessageWrapper` (его исходный файл называется `MessageWrapper.aj`, т.е. это файл аспекта AspectJ, а не стандартный файл класса Java).

```
package com.apress.prospring5.ch5;

public aspect MessageWrapper {
  private String prefix;
  private String suffix;

  public void setPrefix(String prefix) {
    this.prefix = prefix;
  }

  public String getPrefix() {
    return this.prefix;
  }
```

```
  public void setSuffix(String suffix) {
    this.suffix = suffix;
  }

  public String getSuffix() {
    return this.suffix;
  }

  pointcut doWriting() :
    execution(*com.apress.prospring5.ch5
                  .MessageWriter.writeMessage());
  before() : doWriting() {
    System.out.println(prefix);
  }

  after() : doWriting() {
    System.out.println(suffix);
  }
}
```

Сначала в приведенном выше примере кода создается аспект `MessageWrapper`, которому предоставляются, как и в обычном классе Java, два свойства, `suffix` и `prefix`, применяемые при снабжении метода `writeMessage()` советом. Затем определяется именованный срез `doWriting()` для единственной точки соединения (в данном случае — для выполнения метода `writeMessage()`). В языке AspectJ поддерживается немало точек соединения, но здесь недостаточно места, чтобы изложить их полностью. И, наконец, в рассматриваемом здесь примере кода определяются два типа советов: один из них выполняется до среза `doWriting()`, а другой — после него. В приведенном ниже фрагменте кода из файла конфигурации `app-config-xml.xml` показано, каким образом данный аспект конфигурируется в Spring.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation=
          "http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans
                 /spring-beans.xsd">

       <bean id="aspect"
          class="com.apress.prospring5.ch5.MessageWrapper"
          factory-method="aspectOf" p:prefix="The Prefix"
                p:suffix="The Suffix"/>
</beans>
```

Как видите, бо́льшая часть конфигурации компонента Spring Bean данного аспекта очень похожа на конфигурацию стандартного компонента Spring Bean. Единственное отличие заключается в использовании атрибута `factory-method` непосредственно в дескрипторе разметки `<bean>`. Этот атрибут позволяет интегрировать в Spring классы, построенные по проектному шаблону "Фабрика" (Factory). Так, если имеется класс `Foo` с закрытым конструктором и статическим фабричным методом `getInstance()`, то с помощью атрибута `factory-method` можно управлять компонентом этого класса в Spring. Метод `aspectOf()`, присутствующий в каждом аспекте AspectJ, предоставляет доступ к экземпляру аспекта, позволяя устанавливать свойства этого аспекта в Spring. Ниже приведен исходный код простого тестового приложения для данного примера.

```
package com.apress.prospring5.ch5;

import org.springframework.context
          .support.GenericXmlApplicationContext;

public class AspectJDemo {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context-xml.xml");
    ctx.refresh();

    MessageWriter writer = new MessageWriter();
    writer.writeMessage();
    writer.foo();
  }
}
```

Обратите внимание на то, что в данном тестовом приложении сначала загружается контекст типа `ApplicationContext`, чтобы дать возможность Spring сконфигурировать аспект. Затем получается экземпляр типа `MessageWriter` и вызываются методы `writeMessage()` и `foo()`. Выполнение исходного кода из данного примера даст следующий результат:

```
The Prefix
foobar!
The Suffix
foo
```

Как видите, совет в аспекте `MessageWrapper` был применен к методу `write Message()`, а значения свойств `prefix` и `suffix`, указанные в конфигурации контекста типа `ApplicationContext`, использовались в совете при выводе сообщений до и после вызова данного метода.

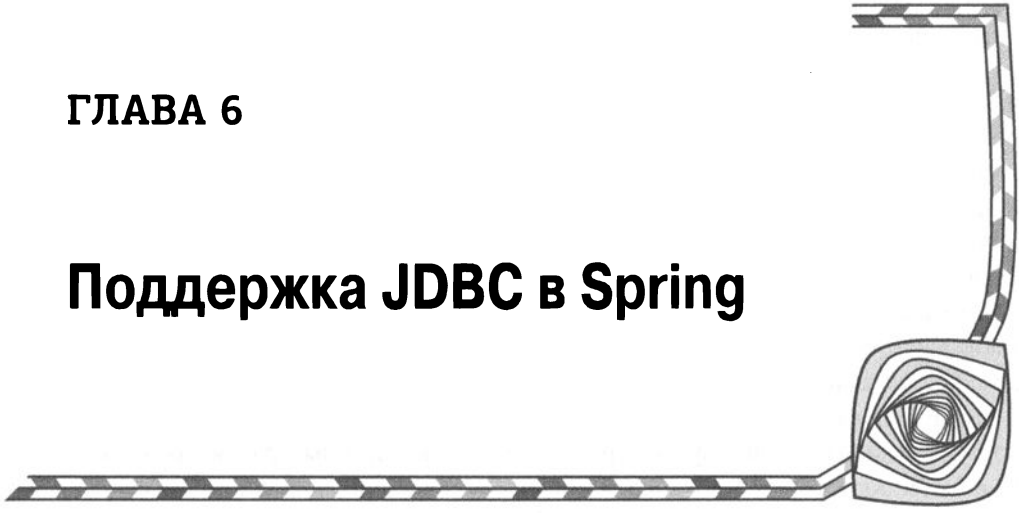
Резюме

В этой главе было представлено немало основных понятий АОП и показано, каким образом они реализованы в Spring. В ней были рассмотрены функциональные средства АОП, реализованные (и не реализованные) в Spring, а также было указано на AspectJ как на решение с возможностями, которыми не обладает реализация АОП в Spring. Немного внимания было уделено разъяснению особенностей разных типов советов, доступных в Spring, а также разбору примеров, демонстрирующих четыре разновидности советов в действии. Было также показано, как с помощью срезов ограничить методы, к которым применяется совет. В частности, были представлены шесть простых реализаций срезов в Spring и разъяснены особенности конструирования заместителей АОП, различные их варианты и отличия. Путем сравнения производительности трех типов заместителей были подчеркнуты основные отличия и ограничения при выборе между заместителями из комплекта JDK и библиотеки CGLIB. Далее в главе были рассмотрены дополнительные возможности для создания срезов и показано, как расширить набор интерфейсов, реализуемых объектом, с помощью введений. Кроме того, были исследованы каркасные службы Spring Framework, предназначенные для декларативного конфигурирования АОП и позволяющие избежать жесткого кодирования логики для конструирования заместителей АОП. И в завершение были вкратце рассмотрены особенности интеграции Spring и AspectJ, чтобы воспользоваться дополнительными возможностями AspectJ, не теряя преимуществ гибкости Spring. Безусловно, материал этой главы, посвященной АОП, оказался довольно внушительным!

В следующей главе мы перейдем к совсем другой теме: использованию поддержки интерфейса JDBC в Spring для радикального упрощения процесса программирования доступа к данным через интерфейс JDBC.

ГЛАВА 6

Поддержка JDBC в Spring



В предыдущих главах было наглядно показано, насколько просто строить приложение, полностью управляемое средствами Spring. В них было дано ясное представление о конфигурировании компонентов Spring Beans и об аспектно-ориентированном программировании (АОП). Тем не менее отсутствует еще одна часть загадки: как получить данные, управляющие приложением?

Помимо простейших утилит, однократно выполняемых из командной строки, почти каждое приложение нуждается в сохранении данных в каком-нибудь информационном хранилище. Наиболее употребительным и удобным информационным хранилищем является реляционная база данных.

Ниже перечислены самые распространенные в настоящее время реляционные базы данных масштаба предприятия.

- Oracle Database
- Microsoft SQL Server
- IBM DB2
- SAP Sybase ASE
- PostgreSQL
- MariaDB Enterprise
- MySQL

Если вы не работаете в крупной компании, которая может себе позволить приобрести лицензии на первые четыре из перечисленных выше баз данных, то, скорее всего, пользуетесь одной из трех последних из них. К числу наиболее употребительных реляционных баз данных с открытым исходным кодом относятся MySQL (`http://mysql.com`) и PostgreSQL (`https://www.postgresql.org/`). База данных MySQL обычно употребляется при разработке веб-приложений, особенно на

платформе Linux[1]. А база данных PostgreSQL удобнее всего для разработчиков приложений на платформе Oracle благодаря ее процедурному языку PLpgSQL, очень похожему на язык PL/SQL от компании Oracle.

Даже если выбрать самую быстродействующую и надежную базу данных, приложение может потерять гибкость в быстродействии, если в нем присутствует неудачно спроектированный и реализованный уровень доступа к данным. Приложения, как правило, очень часто пользуются уровнем доступа к данным, и поэтому любые узкие места в коде доступа к данным оказывают отрицательное влияние на все приложение, независимо от того, насколько удачно оно спроектировано.

В этой главе будет показано, как с помощью Spring упростить реализацию кода доступа к данным, используя интерфейс JDBC. И начнем мы с рассмотрения большого объема повторяющегося кода, который пришлось бы написать, не имея в своем распоряжении Spring, а затем сравним его с классом доступа к данным, реализованным средствами Spring. Результат действительно впечатляет, поскольку каркас Spring позволяет использовать весь потенциал составленных вручную запросов SQL, сведя к минимуму объем поддерживающего кода, который придется написать. В этой главе будут, в частности, рассмотрены следующие вопросы.

- **Сравнение традиционного кода JDBC и поддержки JDBC в Spring.** В этой части мы исследуем, каким образом каркас Spring упрощает код JDBC, написанный в старом стиле, обеспечивая в то же время аналогичные функциональные возможности. Здесь будет также показано, каким образом каркас Spring обращается к низкоуровневому прикладному интерфейсу JDBC API и как этот прикладной интерфейс отображается на удобные классы вроде `JdbcTemplate`.
- **Подключение к базе данных.** Несмотря на то что мы не собираемся вникать в мельчайшие подробности управления подключением к базе данных, мы продемонстрируем в этой части основные отличия соединения типа `Connection` от источника данных типа `DataSource`. Естественно, мы обсудим, каким образом в Spring организовано управление источниками данных и какие источники данных можно употреблять в приложениях.
- **Извлечение и преобразование данных в объекты Java.** В этой части мы покажем, как извлекать данные и затем эффективно преобразовывать их в объекты Java. Здесь также поясняется, что реализация интерфейса JDBC в Spring является жизнеспособной альтернативой инструментальным средствам объектно-реляционного преобразования (ORM).
- **Ввод, обновление и удаление данных.** Наконец, мы обсудим, как реализуются операции ввода, обновления и удаления данных с помощью каркаса Spring для выполнения соответствующих запросов к базе данных.

---

[1] Для хранения информации на платформе WordPress, широко употребляемой для ведения блогов, применяется база данных MySQL или MariaDB.

Введение в лямбда-выражения

В версии Java 8 наряду со многими другими функциональными возможностями появилась поддержка лямбда-выражений. Лямбда-выражения служат отличной заменой внутренним анонимным классам, а также идеально подходят для работы с реализацией интерфейса JDBC, поддерживаемой в Spring. Для применения лямбда-выражений *обязательно* наличие версии Java 8. Данная книга была написана еще до выпуска окончательной версии Java 8, и поэтому мы отдаем себе отчет, что не все разработчики работают с версией Java 8. Принимая во внимание данное обстоятельство, в примерах из этой главы приводятся оба варианта исходного кода там, где это уместно. Лямбда-выражения пригодны в большинстве тех мест прикладного интерфейса Spring API, где используются шаблоны или обратные вызовы, так что их применение совсем не ограничивается интерфейсом JDBC. В этой главе сами лямбда-выражения не раскрываются, поскольку они являются языковым средством Java, и поэтому вы должны быть знакомы с принципами и синтаксисом лямбда-выражений. За дополнительными сведениями о лямбда-выражениях обращайтесь к соответствующему руководству, доступному по ссылке `http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`.

Модель выборочных данных для исходного кода примеров

Прежде чем продолжить дальше, необходимо представить простую модель данных, которая применяется во всех примерах исходного кода из этой главы, а также в ряде последующих глав, где обсуждаются другие технологии доступа к данным (мы будем соответствующим образом расширять эту модель для удовлетворения потребностей в раскрытии каждой новой темы).

Рассматриваемая здесь модель включает в себя простую базу музыкальных данных с двумя таблицами. Первая таблица называется `SINGER` и хранит сведения о певцах, вторая таблица — `ALBUM` — содержит перечень выпущенных ими альбомов. У каждого певца может быть от нуля и больше альбомов. Иными словами, между таблицами `SINGER` и `ALBUM` существует отношение "один ко многим". Сведения о певце включают в себя имя, фамилию и дату его рождения. На рис. 6.1 приведена диаграмма "сущность–отношение" (ER) для рассматриваемой здесь базы данных.

Как видите, в обеих упомянутых выше таблицах присутствует столбец `ID`, значение которого автоматически устанавливается базой данных во время ввода данных. У таблицы `ALBUM` имеется отношение по внешнему ключу с таблицей `SINGER`, которое поддерживает связь по столбцу `SINGER_ID` с первичным ключом из таблицы `ALBUM` (т.е. со столбцом `ID`).

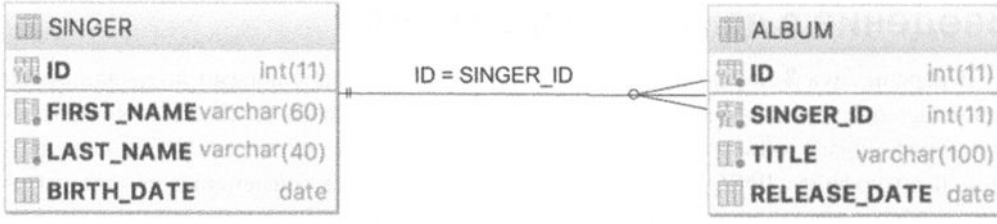


Рис. 6.1. Простая модель выборочных данных для исходного кода примеров

На заметку В ряде примеров из этой главы для демонстрации взаимодействия с реальной базой данных применяется база данных MySQL с открытым кодом. Поэтому для проработки этих примеров потребуется экземпляр базы данных MySQL. Здесь не рассматривается процесс установки MySQL. При желании вы можете воспользоваться другой базой данных, но тогда вам придется внести соответствующие коррективы в определения схемы и функций. Здесь будет также показано, как пользоваться встроенной в Spring базой данных, не прибегая к необходимости устанавливать экземпляр базы данных MySQL.

На тот случай, если потребуется база данных MySQL, на ее официальном сайте можно найти очень хорошие учебные материалы по установке и конфигурированию MySQL. Загрузив и установив базу данных MySQL[2], можно получить доступ к ней, используя учетную запись `root`. Как правило, при разработке приложений требуются новая схема и пользователь базы данных. Для примеров из этой главы выбрана схема `MUSICDB` базы данных и получающий к ней доступ пользователь по имени `prospring5`. Ниже приведен код SQL, исполняемый для их создания и находящийся в файле `ddl.sql`, который содержится в каталоге `resources` проекта `plain-jdbc`. В этот код включено также исправление программной ошибки в версии MySQL Community Server 5.17.18, которая была текущей на момент написания данной книги.

```
CREATE USER 'prospring5'@'localhost'
       IDENTIFIED BY 'prospring5';

CREATE SCHEMA MUSICDB;
GRANT ALL PRIVILEGES ON MUSICDB . *
      TO 'prospring5'@'localhost';
FLUSH PRIVILEGES;

/* На тот случай, если возникнет исключение
   типа java.sql.SQLException: значение 'UTC'
   часового пояса не распознано или представляет
   не один часовой пояс. */
SET GLOBAL time_zone = '+3:00';
```

[2] Загрузить сервер MySQL Community Server можно по адресу `https://dev.mysql.com/downloads/`.

В приведенном ниже фрагменте кода демонстрируется код SQL, требующийся для создания двух упомянутых выше таблиц. Этот код находится в файле `schema.sql`, который находится в каталоге `resources` проекта `plain-jdbc` из этой главы.

```
CREATE TABLE SINGER (
       ID INT NOT NULL AUTO_INCREMENT
     , FIRST_NAME VARCHAR(60) NOT NULL
     , LAST_NAME VARCHAR(40) NOT NULL
     , BIRTH_DATE DATE
     , UNIQUE UQ_SINGER_1 (FIRST_NAME, LAST_NAME)
     , PRIMARY KEY (ID)
);

CREATE TABLE ALBUM (
       ID INT NOT NULL AUTO_INCREMENT
     , SINGER_ID INT NOT NULL
     , TITLE VARCHAR(100) NOT NULL
    , RELEASE_DATE DATE
    , UNIQUE UQ_SINGER_ALBUM_1 (SINGER_ID, TITLE)
    , PRIMARY KEY (ID)
    , CONSTRAINT FK_ALBUM FOREIGN KEY (SINGER_ID)
                REFERENCES SINGER (ID)
);
```

Если воспользоваться IDE вроде IntelliJ IDEA, то с помощью представления **Database** (База данных) можно исследовать схему и таблицы базы данных. На рис. 6.2 приведено содержимое схемы `MUSICDB`, представленное в IDE IntelliJ IDEA.

Чтобы проверить применение интерфейса JDBC, потребуются соответствующие данные. Для этой цели предоставляется файл `test-data.sql`, содержащий ряд операторов `INSERT`, предназначенных для заполнения обеих упомянутых выше таблиц:

```
insert into singer (first_name, last_name, birth_date)
       values ('John', 'Mayer', '1977-10-16');
insert into singer (first_name, last_name, birth_date)
       values ('Eric', 'Clapton', '1945-03-30');
insert into singer (first_name, last_name, birth_date)
       values ('John', 'Butler', '1975-04-01');

insert into album (singer_id, title, release_date)
     values (1, 'The Search For Everything', '2017-01-20');
insert into album (singer_id, title, release_date)
       values (1, 'Battle Studies', '2009-11-17');
insert into album (singer_id, title, release_date)
       values (2, ' From The Cradle ', '1994-09-13');
```

В последующих разделах этой главы будут приведены примеры извлечения информации из базы данных через JDBC и преобразования результирующего набора непосредственно в объекты Java (т.е. простые объекты POJO). Объекты, преобразуе-

мые в записи таблицы базы данных, называются *сущностями*. Так, для таблицы SINGER определяется приведенный ниже класс Singer, экземпляр которого служит для создания объектов Java, преобразуемых в записи о певцах.

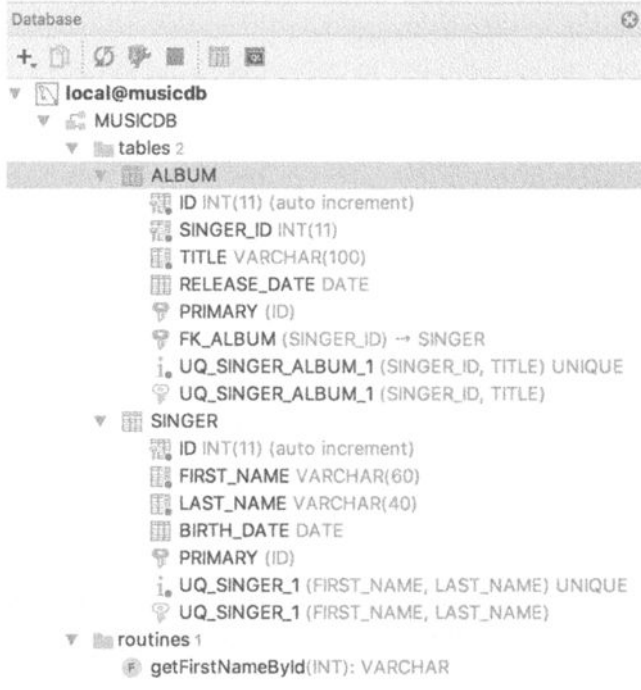


Рис. 6.2. Содержимое схемы **MUSICDB**

```
package com.apress.prospring5.ch6.entities;

import java.io.Serializable;
import java.sql.Date;
import java.util.List;

public class Singer implements Serializable {

  private Long id;
  private String firstName;
  private String lastName;
  private Date birthDate;
  private List<Album> albums;

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

  public Long getId() {
```

```
  return this.id;
}

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

public String getFirstName() {
  return this.firstName;
}

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

public String getLastName() {
  return this.lastName;
}

public boolean addAbum(Album album) {
  if (albums == null) {
    albums = new ArrayList<>();
    albums.add(album);
    return true;
  } else {
    if (albums.contains(album)) {
      return false;
    }
  }
  albums.add(album);
  return true;
}

public void setAlbums(List<Album> albums) {
  this.albums = albums;
}

public List<Album> getAlbums() {
  return albums;
}

public void setBirthDate(Date birthDate) {
  this.birthDate = birthDate;
}

public Date getBirthDate() {
  return birthDate;
}
```

```
  public String toString() {
    return "Singer - Id: " + id + ", First name: "
         + firstName + ", Last name: " + lastName
         + ", Birthday: " + birthDate;
  }
}
```

Аналогичным образом определяется класс `Album`:

```
package com.apress.prospring5.ch6.entities;

import java.io.Serializable;
import java.sql.Date;

public class Album implements Serializable {
  private Long id;
  private Long singerId;
  private String title;
  private Date releaseDate;

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

  public Long getId() {
    return this.id;
  }

  public void setSingerId(Long singerId) {
    this.singerId = singerId;
  }

  public Long getSingerId() {
    return this.singerId;
  }

  public void setTitle(String title) {
    this.title = title;
  }

  public String getTitle() {
    return this.title;
  }

  public void setReleaseDate(Date releaseDate) {
    this.releaseDate = releaseDate;
  }

  public Date getReleaseDate() {
    return this.releaseDate;
```

```
  }

  @Override
  public String toString() {
    return "Album - Id: " + id + ", Singer id: "
         + singerId + ", Title: " + title
         + ", Release Date: " + releaseDate;
 }
}
```

Начнем с простого интерфейса `SingerDao`, инкапсулирующего все услуги доступа к данным для получения сведений о певце. Исходный код этого интерфейса приведен ниже. В интерфейсе `SingerDao` определен ряд методов поиска, а также методы ввода, обновления и удаления данных, которые совместно обозначаются термином CRUD (create, read, update, delete — создание, чтение, обновление и удаление).

```
package com.apress.prospring5.ch6.dao;

import com.apress.prospring5.ch6.entities.Singer;
import java.util.List;

public interface SingerDao {
  List<Singer> findAll();
  List<Singer> findByFirstName(String firstName);
  String findLastNameById(Long id);
  String findFirstNameById(Long id);
  void insert(Singer singer);
  void update(Singer singer);
  void delete(Long singerId);
  List<Singer> findAllWithDetail();
  void insertWithDetail(Singer singer);
}
```

И, наконец, чтобы упростить тестирование, внесем коррективы в файл конфигурации `logback.xml`, указав уровень протоколирования `DEBUG` для всех классов. На уровне протоколирования `DEBUG` модуль Spring JDBC будет выводить все основные операторы SQL, выполняемые в базе данных, чтобы стало понятно, что именно происходит. Это особенно удобно во время поиска и устранения синтаксических ошибок в операторах SQL. Ниже приведено содержимое файла конфигурации `logback.xml`, находящегося в каталоге `plain-jdbc/src/main/resources` с исходными файлами первого проекта из главы 6, где установлен уровень протоколирования `DEBUG`.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

  <contextListener class=
        "ch.qos.logback.classic.jul.LevelChangePropagator">
    <resetJUL>true</resetJUL>
```

```
  </contextListener>

  <appender name="console"
          class="ch.qos.logback.core.ConsoleAppender">
    <encoder>

      <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level
               %logger{5} - %msg%n</pattern>
    </encoder>

  </appender>
  <logger name="com.apress.prospring5.ch5" level="debug"/>

  <logger name="org.springframework" level="off"/>

  <root level="debug">
    <appender-ref ref="console" />
  </root>
</configuration>
```

Исследование инфраструктуры JDBC

Интерфейс JDBC предоставляет приложениям на Java стандартный способ доступа к информации, хранящейся в базе данных. В основу JDBC положен драйвер, характерный для каждой базы данных, который и обеспечивает доступ к базе данных из кода, написанного на Java.

После загрузки драйвер регистрируется с помощью класса `java.sql.DriverManager`. Этот класс управляет списком драйверов и предоставляет статические методы для установления соединений с базой данных. В частности, метод `getConnection()` из класса `DriverManager` возвращает реализуемый драйвером интерфейс `java.sql.Connection`, позволяющий выполнять операторы SQL в базе данных.

Инфраструктура JDBC довольно сложна и хорошо проверена, но именно ее сложность и становится препятствием, затрудняющим разработку приложений. Первый уровень сложности связан с тем, что прикладной код необходимо снабдить средствами управления соединением с базой данных. Соединение — дефицитный ресурс, требующий немалых затрат на установление. Как правило, база данных создает поток исполнения или порождает дочерний процесс для каждого соединения. Но количество параллельных соединений, как правило, ограничено, поскольку большое число установленных соединений замедляет работу базы данных.

Далее будет показано, каким образом Spring помогает справиться с подобной сложностью. Но сначала необходимо выяснить, как выбираются, удаляются и обновляются данные непосредственно в интерфейсе JDBC.

Во всех проектах, представленных в этой главе, требуются в виде зависимостей специальные библиотеки для баз данных, в том числе `mysql-connector`, `spring-`

jdbc, dbcp и пр. Версии применяемых библиотек можно выяснить из содержимого файлов конфигурации build.gradle каждого проекта, а также из файла конфигурации pro-spring-15/build.gradle.

Итак, создадим простую форму реализации интерфейса SingerDao для взаимодействия с базой данных исключительно через интерфейс JDBC. Памятуя о том, что уже известно о соединениях с базой данных, выберем осмотрительный, хотя и дорогостоящий (в смысле производительности) подход к установлению соединения с базой данных для выполнения каждого оператора SQL. Это приведет к значительному снижению производительности прикладного кода Java и увеличит нагрузку на базу данных, поскольку соединение должно устанавливаться по каждому ее запросу. Но если оставить подключение установленным, то сервер базы данных может даже остановиться. Ниже приведен исходный код, требующийся для управления подключением через интерфейс JDBC, где в качестве примера используется база данных MySQL.

```
package com.apress.prospring5.ch6.dao;
...
public class PlainSingerDao implements SingerDao {

  private static Logger logger =
          LoggerFactory.getLogger(PlainSingerDao.class);

  static {
    try {
      Class.forName("com.mysql.cj.jdbc.Driver");
    } catch (ClassNotFoundException ex) {
      logger.error("Prblem loadng DB dDiver!", ex);
    }
  }

  private Connection getConnection() throws SQLException {
    return DriverManager.getConnection(
         "jdbc:mysql://localhost:3306/musicdb?useSSL=true",
         "prospring5", "prospring5");
    }

  private void closeConnection(Connection connection) {
    if (connection == null) {
      return;
    }

    try {
      connection.close();
    } catch (SQLException ex) {
      logger.error(
        "Problem closing connection to the database!", ex);
    }
  }
...
```

Несмотря на то что приведенный выше код далек от завершения, он дает ясное представление о том, какие именно действия должны быть предприняты для управления соединением с базой данных через интерфейс JDBC. В данном коде даже не организован пул соединений — весьма распространенный способ более эффективного управления соединением с базой данных. Мы пока что не будем обсуждать организацию пула соединений, поскольку это будет сделано в разделе "Соединения с базой данных и источники данных", а вместо этого приведем реализацию методов `findAll()`, `insert()` и `delete()` из интерфейса `SingerDao` простыми средствами JDBC.

```
package com.apress.prospring5.ch6.dao;
...
public class PlainSingerDao implements SingerDao {
  @Override
  public List<Singer> findAll() {
    List<Singer> result = new ArrayList<>();
    Connection connection = null;
    try {
      connection = getConnection();
      PreparedStatement statement = connection
              .prepareStatement("select * from singer");
      ResultSet resultSet = statement.executeQuery();
      while (resultSet.next()) {
        Singer singer = new Singer();
        singer.setId(resultSet.getLong("id"));
        singer.setFirstName(resultSet
                            .getString("first_name"));
        singer.setLastName(resultSet
                           .getString("last_name"));
        singer.setBirthDate(resultSet
                            .getDate("birth_date"));
        result.add(singer);
      }
      statement.close();
    } catch (SQLException ex) {
      logger.error("Problem when executing SELECT!",ex);
    } finally {
      closeConnection(connection);
    }
      return result;
  }

  @Override
  public void insert(Singer singer) {
    Connection connection = null;
    try {
      connection = getConnection();
      PreparedStatement statement =
```

```
            connection.prepareStatement("insert into Singer "
                    + "(first_name, last_name, birth_date)
                    + values (?, ?, ?)"
                    , Statement.RETURN_GENERATED_KEYS);
        statement.setString(1, singer.getFirstName());
        statement.setString(2, singer.getLastName());
        statement.setDate(3, singer.getBirthDate());
        statement.execute();
        ResultSet generatedKeys =
                  statement.getGeneratedKeys();
        if (generatedKeys.next()) {
          singer.setId(generatedKeys.getLong(1));
        }
        statement.close();
      } catch (SQLException ex) {
        logger.error("Prblem executing INSERT", ex);
      } finally {
        closeConnection(connection);
      }
    }

    @Override
    public void delete(Long singerId) {
      Connection connection = null;
      try {
        connection = getConnection();
        PreparedStatement statement =
             connection.prepareStatement(
                       "delete from singer where id=?");
        statement.setLong(1, singerId);
        statement.execute();
        statement.close();
      } catch (SQLException ex) {
        logger.error("Prblem executing DELETE", ex);
      } finally {
        closeConnection(connection);
      }
    }
    ...
}
```

Для проверки класса `PlainSingerDao` можно воспользоваться следующим классом:

```
package com.apress.prospring5.ch6;

import com.apress.prospring5.ch6.dao.PlainSingerDao;
import com.apress.prospring5.ch6.dao.SingerDao;
import com.apress.prospring5.ch6.entities.Singer;
```

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.sql.Date;
import java.util.GregorianCalendar;
import java.util.List;

public class PlainJdbcDemo {
  private static SingerDao singerDao = new PlainSingerDao();
  private static Logger logger =
          LoggerFactory.getLogger(PlainJdbcDemo.class);

  public static void main(String... args) {
    logger.info("Listing initial singer data:");
    listAllSingers();
    logger.info("-------------");
    logger.info("Insert a new singer");

   Singer singer = new Singer();
   singer.setFirstName("Ed");
   singer.setLastName("Sheeran");
   singer.setBirthDate(new Date
          ((new GregorianCalendar(1991, 2, 1991))
            .getTime().getTime()));
    singerDao.insert(singer);

    logger.info("Listing singer data "
              + "after new singer created:");

    listAllSingers();

    logger.info("-------------");
    logger.info("Deleting the previous created singer");
    singerDao.delete(singer.getId());
    logger.info("Listing singer data "
              + "after new singer deleted:");
    listAllSingers();
  }

  private static void listAllSingers() {
    List<Singer> singers = singerDao.findAll();
    for (Singer singer: singers) {
      logger.info(singer.toString());
    }
  }
}
```

Как видите, в данном примере применяется регистратор для вывода протокольных сообщений на консоль. Выполнение исходного кода из данного примера дает приве-

денный ниже результат, если установить локально базу данных MUSICDB типа MySQL, задать имя пользователя и пароль prospring5 для доступа к ней и предварительно загрузить выборочные данные, как пояснялось в начале этой главы.

```
INFO  c.a.p.c.PlainJdbcDemo - Listing initial singer data:
INFO  c.a.p.c.PlainJdbcDemo - Singer - Id: 1,
                              First name: John,
                              Last name: Mayer,
                              Birthday: 1977-10-15
INFO  c.a.p.c.PlainJdbcDemo - Singer - Id: 2,
                              First name: Eric,
                              Last name: Clapton,
                              Birthday: 1945-03-29
INFO  c.a.p.c.PlainJdbcDemo - Singer - Id: 3,
                              First name: John,
                              Last name: Butler,
                              Birthday: 1975-03-31
INFO  c.a.p.c.PlainJdbcDemo - -------------
INFO  c.a.p.c.PlainJdbcDemo - Insert a new singer
INFO  c.a.p.c.PlainJdbcDemo - Listing singer data after
                              new singer created:
INFO  c.a.p.c.PlainJdbcDemo - Singer - Id: 1,
                              First name: John,
                              Last name: Mayer,
                              Birthday: 1977-10-15
INFO  c.a.p.c.PlainJdbcDemo - Singer - Id: 2,
                              First name: Eric,
                              Last name: Clapton,
                              Birthday: 1945-03-29
INFO  c.a.p.c.PlainJdbcDemo - Singer - Id: 3,
                              First name: John,
                              Last name: Butler,
                              Birthday: 1975-03-31
INFO  c.a.p.c.PlainJdbcDemo - Singer - Id: 5,
                              First name: Ed,
                              Last name: Sheeran,
                              Birthday: 1996-08-10
INFO  c.a.p.c.PlainJdbcDemo - -------------
INFO  c.a.p.c.PlainJdbcDemo - Deleting the previous
                              created singer
INFO  c.a.p.c.PlainJdbcDemo - Listing singer data after
                              new singer deleted:
INFO  c.a.p.c.PlainJdbcDemo - Singer - Id: 1,
                              First name: John,
                              Last name: Mayer,
                              Birthday: 1977-10-15
INFO  c.a.p.c.PlainJdbcDemo - Singer - Id: 2,
                              First name: Eric,
                              Last name: Clapton,
```

```
                                Birthday: 1945-03-29
INFO c.a.p.c.PlainJdbcDemo - Singer - Id: 3,
                             First name: John,
                             Last name: Butler,
                             Birthday: 1975-03-31
```

В первом блоке строк выведенного выше результата отображаются первоначальные данные, во втором блоке строк показано, что была введена новая запись. И в завершающем блоке строк отражается факт удаления ранее созданной записи сведений о певце (`Ed Sheeran`).

Как следует из приведенных выше примеров кода, большой объем кода нуждается в перемещении во вспомогательный класс или, что еще хуже, дублируется в каждом классе DAO. Это главный недостаток интерфейса JDBC с точки зрения разработчика приложений, поскольку ему просто не хватит времени и терпения повторять один и тот же код в каждом классе DAO. Вместо этого лучше сосредоточиться на написании кода, который действительно делает то, что должен делать класс DAO: выборку, обновление и удаление данных. Чем больше вспомогательного кода требуется написать, тем больше проверяемых исключений придется проверить и тем больше ошибок может быть внесено в код.

Именно здесь вступают в действие объекты DAO в каркасе Spring. В частности, каркас Spring исключает код, не выполняющий никакой специальной логики, избавляя разработчиков приложений от обязанности помнить о любых служебных операциях, которые необходимо произвести, работая с базой данных. Кроме того, обширная поддержка JDBC в Spring значительно упрощает задачу разработчиков приложений.

Инфраструктура JDBC в Spring

Код, обсуждавшийся в первой части этой главы, не был особенно сложным, но его не только утомительно писать, но и трудно набрать без ошибок. Поэтому самое время выяснить, каким образом Spring упрощает реализацию JDBC и делает ее более изящной.

Краткий обзор применяемых пакетов

Поддержка интерфейса JDBC в Spring разделена на пять пакетов (табл. 6.1), каждый из которых реализует определенные аспекты доступа к базе данных через интерфейс JDBC.

Начнем обсуждение поддержки интерфейса JDBC в Spring с рассмотрения функциональных возможностей самого низкого уровня. Прежде чем выполнять запросы SQL, необходимо установить соединение с базой данных.

Таблица 6.1. Пакеты JDBC в Spring

Пакет	Описание
`org.springframework.jdbc.core`	Содержит ядро для классов JDBC в Spring. Включает базовый класс JDBC, называемый **`JdbcTemplate`** и упрощающий программирование операций в базе данных через интерфейс JDBC. Многочисленные подпакеты обеспечивают поддержку доступа к данным через интерфейс JDBC с более конкретными целями (например, в классе **`JdbcTemplate`** поддерживаются именованные параметры), а также соответствующие классы для такой поддержки
`org.springframework.jdbc.datasource`	Содержит вспомогательные классы и реализации интерфейса **`DataSource`**, которые можно использовать для выполнения кода JDBC за пределами контейнера JEE. Многочисленные подпакеты обеспечивают поддержку встроенных баз данных, инициализацию баз данных и различные механизмы поиска информации в источниках данных
`org.springframework.jdbc.object`	Содержит классы, помогающие преобразовывать данные, возвращаемые из базы, в объекты или списки объектов. А поскольку это простые объекты Java, то они отключены от базы данных
`org.springframework.jdbc.support`	Наиболее важным средством в этом пакете является поддержка преобразования исключений типа **`SQLException`**. Это дает возможность каркасу Spring распознавать коды ошибок, применяемые в базе данных, а также преобразовывать их в исключения более высокого уровня
`org.springframework.jdbc.config`	Содержит классы, поддерживающие конфигурацию JDBC в контексте типа **`ApplicationContext`**. Например, в этот пакет включается класс обработчика для пространства имен **`jdbc`** (например, для дескрипторов разметки **`<jdbc:embedded-database>`**)

Соединения с базой данных и источники данных

Для управления автоматическим подключением к базе данных можно воспользоваться каркасом Spring, предоставив компонент Spring Bean, реализующий интерфейс `javax.sql.DataSource`. Отличие между интерфейсами `DataSource` и `Connection` заключается в том, что интерфейс `DataSource` предоставляет реализации `Connection` и управляет ими.

Простейшей реализацией интерфейса `DataSource` служит класс `DriverManager DataSource` (из пакета `org.springframework.jdbc.datasource`). Судя по имени данного класса, он просто обращается к классу `DriverManager` для получения соединения. Вследствие того, что в классе `DriverManagerDataSource` не поддерживается пул соединений с базой данных, он непригоден ни для каких других целей, кроме тестирования. Сконфигурировать класс `DriverManagerDataSource` совсем не трудно, как показано ниже. Для этого достаточно указать имя класса драйвера, URL соединения, имя пользователя и пароль (см. файл конфигурации `datasource-cfg-01.xml`).

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context=
            "http://www.springframework.org/schema/context"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation=
               "http://www.springframework.org/schema/beans
                http://www.springframework.org/schema
                     /beans/spring-beans.xsd
                http://www.springframework.org
                     /schema/context
                http://www.springframework.org/schema
                      /context/spring-context.xsd">

    <bean id="dataSource"
      class="org.springframework.jdbc
                .datasource.DriverManagerDataSource"
       p:driverClassName="${jdbc.driverClassName}"
       p:url="${jdbc.url}" p:username="${jdbc.username}"
       p:password="${jdbc.password}"/>

    <context:property-placeholder
            location="classpath:db/jdbc.properties"/>
</beans>
```

В приведенной выше конфигурации нетрудно распознать свойства. Они предоставляют значения, которые обычно передаются интерфейсу JDBC для получения реализации интерфейса `Connection`. Информация о подключении к базе данных обычно хранится в файле свойств, чтобы упростить сопровождение и замену в разных средах развертывания. Ниже приведено содержимое файла `jdbc.properties`, из которого заполнитель свойств Spring будет загружать сведения о соединении.

```
jdbc.driverClassName=com.mysql.cj.jdbc.Driver
jdbc.url= jdbc:mysql://localhost:3306/musicdb?useSSL=true
jdbc.username=prospring5
jdbc.password=prospring5
```

Если добавить пространство имен `util`, то ту же самую конфигурацию можно переписать так, как показано ниже (см. файл конфигурации `drivermanager-cfg-02.xml`).

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi=
            "http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util=
            "http://www.springframework.org/schema/util"
        xmlns:p="http://www.springframework.org/schema/p"
```

```
        xsi:schemaLocation=
           "http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans
                  /spring-beans.xsd
            http://www.springframework.org/schema/util
            http://www.springframework.org/schema/util
                  /spring-util.xsd">

    <bean id="dataSource"
       class="org.springframework.jdbc.datasource
                 .DriverManagerDataSource"
       p:driverClassName="#{jdbc.driverClassName}"
       p:url="#{jdbc.url}"
       p:username="#{jdbc.username}"
       p:password="#{jdbc.password}"/>

    <util:properties id="jdbc"
          location="classpath:db/jdbc2.properties"/>
</beans>
```

Здесь требуется внести еще одно изменение. Свойства загружаются в компонент `jdbc` типа `java.util.Properties`, и поэтому необходимо изменить имена свойств в их файле, чтобы предоставить доступ к ним с префиксом **`jdbc`**. Ниже приведено содержимое файла свойств `jdbc2.properties`.

```
driverClassName=com.mysql.cj.jdbc.Driver
url=jdbc:mysql://localhost:3306/musicdb?useSSL=true
username=prospring5
password=prospring5
```

В данной книге основное внимание уделяется конфигурационным классам Java, поэтому ниже приведен также конфигурационный класс.

```
package com.apress.prospring5.ch6.config;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context
          .annotation.Configuration;
import org.springframework.context.annotation.Lazy;
import org.springframework.context
          .annotation.PropertySource;
import org.springframework.context.support
          .PropertySourcesPlaceholderConfigurer;
import org.springframework.jdbc.datasource
          .SimpleDriverDataSource;
import org.springframework.jdbc.datasource
          .init.DatabasePopulatorUtils;
import javax.sql.DataSource;
import java.sql.Driver;
```

```
@Configuration
@PropertySource("classpath:db/jdbc2.properties")
public class DbConfig {

  @Value("${driverClassName}")
  private String driverClassName;
  @Value("${url}")
  private String url;
  @Value("${username}")
  private String username;
  @Value("${password}")
  private String password;

  @Bean
  public static PropertySourcesPlaceholderConfigurer
         propertySourcesPlaceholderConfigurer() {
    return new PropertySourcesPlaceholderConfigurer();
  }

  @Lazy
  @Bean
  public DataSource dataSource() {
    try {
      SimpleDriverDataSource dataSource =
                  new SimpleDriverDataSource();
      Class<? extends Driver> driver =
             (Class<? extends Driver>)
              Class.forName(driverClassName);
      dataSource.setDriverClass(driver);
      dataSource.setUrl(url);
      dataSource.setUsername(username);
      dataSource.setPassword(password);
      return dataSource;
    } catch (Exception e) {
      return null;
    }
  }
}
```

Чтобы проверить любой из приведенных выше классов, можно воспользоваться следующим тестовым классом:

```
package com.apress.prospring5.ch6;

import com.apress.prospring5.ch6.config.DbConfig;
import org.junit.Test;
...
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;
```

```
public class DbConfigTest {

  private static Logger logger =
          LoggerFactory.getLogger(DbConfigTest.class);

  @Test
  public void testOne() throws SQLException {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/drivermanager-cfg-01.xml");
    ctx.refresh();

    DataSource dataSource = ctx.getBean(
              "dataSource", DataSource.class);
    assertNotNull(dataSource);
    testDataSource(dataSource);

    ctx.close();
  }

  @Test
  public void testTwo() throws SQLException {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               DbConfig.class);

    DataSource dataSource = ctx.getBean(
              "dataSource", DataSource.class);
    assertNotNull(dataSource);
    testDataSource(dataSource);

    ctx.close();
  }

  private void testDataSource(DataSource dataSource)
          throws SQLException {
    Connection connection = null;
    try {
      connection = dataSource.getConnection();
      PreparedStatement statement =
      connection.prepareStatement("SELECT 1");
      ResultSet resultSet = statement.executeQuery();
      while (resultSet.next()) {
        int mockVal = resultSet.getInt("1");
        assertTrue(mockVal== 1);
      }
      statement.close();
    } catch (Exception e) {
      logger.debug("Something unexpected happened.", e);
```

```
    } finally {
      if (connection != null) {
        connection.close();
      }
    }
  }
}
```

И в данном случае применяется тестовый класс, поскольку намного практичнее повторно использовать какой-нибудь код и заодно показать, как работать с платформой JUnit для написания модульных тестов, предназначенных для проверки работоспособности любого разрабатываемого кода. Первым в приведенном выше тестовом классе объявляется метод `testOne()`, предназначенный для тестирования конфигураций в формате XML, а второй метод служит для тестирования конфигурационного класса `DbConfig`. Как только компонент `dataSource` будет определен любым из двух способов конфигурирования, по имитируемому запросу `SELECT 1` проверяется подключение к базе данных MySQL.

В реальных приложениях можно применять доступный в рамках проекта Apache Commons[3] класс `BasicDataSource` (`http://commons.apache.org/dbcp/`) или класс, реализующий интерфейс `DataSource` на сервере приложений JEE (например, `JBoss`, `WebSphere`, `WebLogic` или `GlassFish`), что позволит дополнительно увеличить производительность приложения. С одной стороны, источник данных можно было бы использовать в простом коде JDBC и получить те же преимущества организации пула соединений, но зачастую центрального места для конфигурирования источника данных будет по-прежнему недоставать. А с другой стороны, в Spring можно объявить компонент `dataSource` и установить свойства соединения в файлах определений интерфейса `ApplicationContext`. В качестве примера ниже приведен соответствующий фрагмент содержимого файла конфигурации `datasource-dbcp.xml`.

```
<beans ...>

  <bean id="dataSource"
     class="org.apache.commons.dbcp2.BasicDataSource"
     destroy-method="close"
     p:driverClassName="#{jdbc.driverClassName}"
     p:url="#{jdbc.url}"
     p:username="#{jdbc.username}"
     p:password="#{jdbc.password}"/>
  <util:properties id="jdbc"
        location="classpath:db/jdbc2.properties"/>

</beans>
```

[3] Официальный сайт данного проекта находится по адресу `http://commons.apache.org/dbcp`.

Этот конкретный источник данных, управляемый средствами Spring, реализуется в классе `org.apache.commons.dbcp.BasicDataSource`. Но самое главное, что в компоненте `dataSource` реализуется интерфейс `javax.sql.DataSource`, и его можно непосредственно применять в своих классах для доступа к данным.

Другой способ конфигурирования компонента `dataSource` предусматривает применение интерфейса JNDI. Если разрабатываемое приложение должно выполняться в контейнере JEE, то можно выгодно воспользоваться пулом соединений, организуемым под управлением контейнера. Чтобы воспользоваться источником данных, ориентированным на интерфейс JNDI, придется изменить объявление компонента `dataSource`, как показано в следующем примере из файла конфигурации `datasource-jndi.xml`:

```
<beans ...>
  <bean id="dataSource"
     class="org.springframework.jndi.JndiObjectFactoryBean"
     p:jndiName="java:comp/env/jdbc/musicdb"/>
</beans>
```

В приведенном выше примере конфигурации компонент Spring Bean типа `JndiObjectFactoryBean` применяется для получения источника данных в результате поиска средствами JNDI. Начиная с версии 2.5 в Spring предоставляется пространство имен `jee`, еще больше упрощающее конфигурирование. Ниже приведена та же самая конфигурация источника данных JNDI, в которой применяется пространство имен `jee` (см. файл конфигурации `datasource-jee.xml`).

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jee="http://www.springframework.org/schema/jee"
       xsi:schemaLocation=
          "http://www.springframework.org/schema/beans
           http://www.springframework.org/schema
                 /beans/spring-beans.xsd
           http://www.springframework.org/schema/jee
           http://www.springframework.org/schema/jee
                 /spring-jee.xsd">

    <jee:jndi-lookup jndi-name=
        "java:comp/env/jdbc/prospring5ch6"/>
</beans>
```

В приведенном выше примере конфигурации пространство имен `jee` объявляется в дескрипторе разметки `<beans>`, а источник данных — в дескрипторе разметки `<jee:jndi-lookup>`. Если выбрать применение интерфейса JNDI, то следует ввести ссылку на ресурс (`resource-ref`) в файл дескриптора приложения, как показано ниже.

```
<root-node>
  <resource-ref>
    <res-ref-name>jdbc/musicdb</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
</root-node>
```

Дескриптор разметки `<root-node>` служит в качестве заполнителя. Его следует заменить в зависимости от того, как упакован конкретный модуль. Например, он заменяется дескриптором `<web-app>` в файле дескриптора развертываемого веб-приложения (`WEB-INF/web.xml`), если оно упаковано в модуль. Скорее всего, потребуется настроить и элемент разметки `resource-ref` в файле дескриптора для конкретного сервера приложений. Но обратите внимание на то, что в элементе разметки `resource-ref` указано имя ссылки `jdbc/musicdb`, а в свойстве `jndiName` компонента `dataSource` установлено значение `java:comp/env/jdbc/musicdb`.

Как видите, каркас Spring позволяет конфигурировать источник данных практически любым требующимся образом, скрывая конкретную реализацию или местоположение источника данных от остальной части прикладного кода. Иными словами, классам DAO приложения неизвестно, да и не нужно знать, на что указывает источник данных. Управление соединениями с базой данных также поручено компоненту `dataSource`, который, в свою очередь, сам управляет подключением к базе данных или использует для этой цели контейнер JEE.

Поддержка встроенной базы данных

Начиная с версии 3.0, в Spring поддерживается также встроенная база данных, которая автоматически запускается и становится доступной приложению в виде источника данных типа `DataSource`. Ниже приведена конфигурация встроенной базы данных из файла `embedded-h2-cfg.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jdbc=
            "http://www.springframework.org/schema/jdbc"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:c="http://www.springframework.org/schema/c"
       xsi:schemaLocation=
          "http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans
                 /spring-beans.xsd
           http://www.springframework.org/schema/jdbc
           http://www.springframework.org/schema/jdbc
                /spring-jdbc.xsd">
```

```
    <jdbc:embedded-database id="dataSource" type="H2">
       <jdbc:script location="classpath:db/h2/schema.sql"/>
       <jdbc:script
             location="classpath:db/h2/test-data.sql"/>

  </jdbc:embedded-database>

</beans>
```

В приведенной выше конфигурации сначала объявляется пространство имен `jdbc` в дескрипторе разметки `<beans>`. Затем в дескрипторе разметки `<jdbc:embedded database>` объявляется встроенная база данных, которой присваивается идентификатор `dataSource`. В этом дескрипторе каркасу Spring также предписывается выполнить сценарии, указанные для создания схемы базы данных и ее заполнения тестовыми данными. Здесь очень важен порядок следования сценариев. В частности, файл с командами DDL (Data Definition Language — язык определения данных) должен всегда следовать первым, а за ним — файл с командами DML (Data Manipulation Language — язык манипулирования данными). В атрибуте `type` задается тип используемой встроенной базы данных. Начиная с версии Spring 4.0 поддерживаются типы HSQL (стандартный), H2 и DERBY.

Встроенные базы данных можно сконфигурировать и с помощью конфигурационных классов Java. Для этой цели, в частности, служит класс `EmbeddedDatabase Builder`, в котором сценарии создания и загрузки базы данных указываются в качестве аргументов при получении экземпляра класса `EmbeddedDatabase`, реализующего интерфейс `DataSource`, как показано ниже.

```
package com.apress.prospring5.ch6.config;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context
          .annotation.Configuration;
import org.springframework.jdbc.datasource
          .embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource
          .embedded.EmbeddedDatabaseType;
import javax.sql.DataSource;

@Configuration
public class EmbeddedJdbcConfig {

  private static Logger logger =
       LoggerFactory.getLogger(EmbeddedJdbcConfig.class);

@Bean
  public DataSource dataSource() {
```

```
    try {
      EmbeddedDatabaseBuilder dbBuilder =
                    new EmbeddedDatabaseBuilder();
      return dbBuilder.setType(EmbeddedDatabaseType.H2)
            .addScripts("classpath:db/h2/schema.sql",
            "classpath:db/h2/test-data.sql").build();
    } catch (Exception e) {
      logger.error("Embedded DataSource bean
                + "cannot be created!", e);
      return null;
    }
  }
  ...
}
```

Поддержка встроенной базы данных особенно полезна при локальной разработке или модульном тестировании. В оставшейся части этой главы мы будем пользоваться встроенной базой данных для выполнения исходного кода примеров, чтобы избавить вас от необходимости устанавливать какую-то конкретную базу данных на своей машине.

Можно не только задействовать поддержку встроенной базы данных через пространство имен `jdbc`, но и инициализировать экземпляр конкретной базы данных (например, MySQL, Oracle и т.д), действующий в каком-нибудь другом месте. Вместо атрибутов `type` и `embedded-database` в конфигурации базы данных достаточно указать атрибут `initialize-database`, чтобы соответствующие сценарии выполнялись относительно заданного источника данных `dataSource` так, как будто это встроенная база данных.

Применение источников данных в классах DAO

Проектный шаблон DAO (Data Access Object — объект доступа к данным) предназначен для того, чтобы отделить низкоуровневые прикладные интерфейсы API или операции доступа к данным от высокоуровневых услуг бизнес-логики. Для реализации проектного шаблона DAO требуются следующие компоненты.

- **Интерфейс DAO.** Определяет стандартные операции, выполняемые над отдельным объектом или несколькими объектами модели.
- **Реализация DAO.** Это класс, в котором предоставляется конкретная реализация интерфейса DAO. Как правило, в нем применяется соединение через JDBC или источник данных для манипулирования отдельным объектом или несколькими объектами модели.
- **Объекты модели, иначе называемые объектами данных или сущностями.** Это результат простого преобразования объекта POJO в запись таблицы базы данных.

В качестве конкретного примера реализации проектного шаблона DAO определим интерфейс `SingerDao`:

```
package com.apress.prospring5.ch6.dao;

public interface SingerDao {
  String findNameById(Long id);
}
```

А в качестве простой реализации этого интерфейса введем сначала свойство `dataSource` в класс `JdbcSingerDao`. Причина, по которой требуется ввести свойство `dataSource` в класс реализации, а не в интерфейс, должна быть достаточно очевидной: в интерфейсе совсем не обязательно знать, каким образом данные будут извлекаться и обновляться. Вводя в интерфейс методы, модифицирующие источник данных, мы в лучшем случае вынуждаем его реализации объявлять заглушки для методов получения и установки. Очевидно, что такое проектное решение нельзя считать удачным. Рассмотрим следующий простой класс `JdbcSingerDao`:

```
import com.apress.prospring5.ch6.dao.SingerDao;
import com.apress.prospring5.ch6.entities.Singer;
import org.apache.commons.lang3.NotImplementedException;
import org.springframework.beans
          .factory.BeanCreationException;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.jdbc.core.JdbcTemplate;

import javax.sql.DataSource;
import java.util.List;

public class JdbcSingerDao
       implements SingerDao, InitializingBean {

  private DataSource dataSource;

  public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
  }

  public void afterPropertiesSet() throws Exception {
    if (dataSource == null) {
      throw new BeanCreationException(
            "Must set dataSource on SingerDao");
    }
  }
  ...
}
```

Теперь каркасу Spring можно дать команду сконфигурировать компонент `singer Dao`, воспользовавшись классом `JdbcSingerDao` и установив свойство `dataSource`,

как демонстрируется в следующем конфигурационном классе EmbeddedJdbcConfig:

```
package com.apress.prospring5.ch6.config;
...
@Configuration
public class EmbeddedJdbcConfig {

  private static Logger logger =
        LoggerFactory.getLogger(EmbeddedJdbcConfig.class);
  @Bean
  public DataSource dataSource() {
    try {
      EmbeddedDatabaseBuilder dbBuilder =
                    new EmbeddedDatabaseBuilder();
      return dbBuilder.setType(EmbeddedDatabaseType.H2)
            .addScripts("classpath:db/h2/schema.sql",
            "classpath:db/h2/test-data.sql").build();
    } catch (Exception e) {
      logger.error("Embedded DataSource bean cannot "
                + "be created!", e);
      return null;
    }
  }

  @Bean
  public SingerDao singerDao() {
    JdbcSingerDao dao = new JdbcSingerDao();
    dao.setDataSource(dataSource());
    return dao;
  }
}
```

В каркасе Spring поддерживается немало встроенных баз данных, но их приходится внедрять в свои проекты как зависимости. Ниже показано, как ряд библиотек для работы с базой данных настраивается в файле конфигурации pro-spring-15\gradle.build.

```
ext {
  derbyVersion = '10.13.1.1'
  dbcpVersion = '2.1'
  mysqlVersion = '6.0.6'
  h2Version = '1.4.194'
  ...

  db = [
    mysql: "mysql:mysql-connector-java:$mysqlVersion",
    derby: "org.apache.derby:derby:$derbyVersion",
    dbcp : "org.apache.commons:commons-dbcp2:$dbcpVersion",
```

```
    h2 : "com.h2database:h2:$h2Version"
  ]
}
```

Таким образом, компонент Spring Bean под названием `singerDao` создается благодаря получению экземпляра класса `JdbcSingerDao` со свойством `dataSource`, в котором установлен компонент `dataSource`. В качестве надлежащей нормы практики рекомендуется устанавливать все обязательные свойства компонента Spring Bean. И это проще всего сделать, реализовав интерфейс `InitializingBean` вместе с методом `afterPropertiesSet()`. Подобным образом можно гарантировать, что все обязательные свойства класса `JdbcSingerDao` будут правильно установлены. Подробнее об инициализации компонентов Spring Beans см. в главе 4.

В исходном коде приведенного выше примера для управления источником данных применялся каркас Spring, а также интерфейс `SingerDao` и его реализация средствами JDBC. Кроме того, в данном примере свойство `dataSource` из класса `JdbcSingerDao` было установлено в контексте типа `ApplicationContext`. Попробуем далее расширить код данного примера, внедрив в интерфейс и его реализации конкретные операции над объектами DAO.

Обработка исключений

В каркасе Spring рекомендуется использовать исключения времени выполнения, а не проверяемые исключения, и поэтому требуется какой-то механизм преобразования проверяемого исключения типа `SQLException` в исключение, возникающее во время выполнения в модуле Spring JDBC. А поскольку исключения, возникающие в Spring при обработке запросов SQL, относятся к категории исключений времени выполнения, то они могут быть намного более детализированными, чем проверяемые исключения. И хотя это не является по определению характеристикой исключений времени выполнения, тем не менее, объявлять длинный список проверяемых исключений в операторе `throws` не совсем удобно. Таким образом, проверяемые исключения, как правило, менее детализированы, чем их аналоги времени выполнения.

В каркасе Spring предоставляется стандартная реализация интерфейса `SQLExceptionTranslator`, которая берет на себя обязанности по преобразованию обобщенных кодов ошибок, появляющихся при обработке запросов SQL, в исключения, возникающие в модуле Spring JDBC. Как правило, этой стандартной реализации в Spring оказывается достаточно, но ее можно расширить, указав новую реализацию интерфейса `SQLExceptionTranslator` для применения в классе `JdbcTemplate`, как показано в приведенном ниже примере.

```
package com.apress.prospring5.ch6;

import java.sql.SQLException;
import org.springframework.dao.DataAccessException;
import org.springframework.dao.DeadlockLoserDataAccessException;
```

```
import org.springframework.jdbc.support
          .SQLErrorCodeSQLExceptionTranslator;

public class MySQLErrorCodesTranslator
       extends SQLErrorCodeSQLExceptionTranslator {
  @Override
  protected DataAccessException customTranslate(String task,
            String sql, SQLException sqlex) {
    if (sqlex.getErrorCode() == -12345) {
      return new DeadlockLoserDataAccessException(task, sqlex);
    }
    return null;
  }
}
```

В то же время необходимо внедрить зависимость от библиотеки `spring-jdbc` в данный проект. Ниже показано, как можно настроить библиотеку `spring-jdbc` на применение в файле конфигурации `pro-spring-15/gradle.build`.

```
ext {
    springVersion = '5.0.0.M4'
    ...

    spring = [
        jdbc : "org.springframework:
                spring-jdbc:$springVersion",
        ...
    ]
}
```

Данная библиотека внедряется как зависимость во все проекты JDBC, представленные в примерах из этой главы, следующим образом:

```
// Файл конфигурации
// chapter06/spring-jdbc-embedded/build.gradle
dependencies {
  compile spring.jdbc
  compile db.h2, db.derby
}
```

Чтобы воспользоваться специальным преобразователем исключений SQL, его необходимо передать объекту типа `JdbcTemplate` в классах DAO. В качестве примера, демонстрирующего применение такого преобразователя, ниже приведен фрагмент кода из расширенного метода `JdbcSingerDao.setDataSource()`.

```
...
public class JdbcSingerDao
       implements SingerDao, InitializingBean {
  private DataSource dataSource;
  private JdbcTemplate jdbcTemplate;
```

```
  public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
    JdbcTemplate jdbcTemplate = new JdbcTemplate();
    jdbcTemplate.setDataSource(dataSource);
    MySQLErrorCodesTranslator errorTranslator =
              new MySQLErrorCodesTranslator();
    errorTranslator.setDataSource(dataSource);
    jdbcTemplate.setExceptionTranslator(errorTranslator);
    this.jdbcTemplate = jdbcTemplate;
  }
...
}
```

Этот специально установленный преобразователь исключений SQL будет вызываться в Spring при обнаружении исключений в операторах SQL при обработке запросов SQL в базе данных, а само преобразование произойдет, когда код ошибки окажется равным `-12345`. А для других ошибок в каркасе Spring будет использован собственный стандартный механизм преобразования исключений. Очевидно, что ничто не мешает создать преобразователь типа `SQLExceptionTranslator` в виде управляемого компонента Spring Bean и воспользоваться компонентом типа `JdbcTemplate` в своих классах DAO. Если вам еще не до конца понятно, как пользоваться классом `JdbcTemplate`, не отчаивайтесь — мы обсудим его более подробно в следующем разделе.

Описание класса `JdbcTemplate`

Этот класс представляет ядро поддержки интерфейса JDBC в Spring. Он способен выполнять все типы операторов SQL. Проще говоря, операторы могут быть разделены на определяющие данные и манипулирующие данными. Операторы определения данных отвечают за создание разнообразных объектов базы данных (таблиц, представлений, хранимых процедур и т.п.). А операторы манипулирования данными взаимодействуют с данными и могут быть подразделены на операторы выборки и операторы обновления. Как правило, оператор выборки возвращает из таблицы базы данных ряд строк, каждая из которых содержит один и тот же ряд столбцов. А оператор обновления модифицирует данные в базе, но никаких результатов не возвращает.

Класс `JdbcTemplate` позволяет выдать базе данных оператор SQL любого типа, а также возвратить результат любого типа. В этом разделе мы рассмотрим ряд типичных случаев программируемой поддержки интерфейса JDBC в Spring с помощью класса `JdbcTemplate`.

Инициализация объекта типа `JdbcTemplate` в классе DAO

Прежде чем обсуждать применение класса `JdbcTemplate`, выясним, как подготовить его объект к применению в классе DAO. Делается это очень просто. Для этого,

как правило, достаточно сконструировать объект класса `JdbcTemplate`, передав его конструктору объект источника данных, который должен быть внедрен каркасом Spring в класс DAO. Ниже приведен фрагмент кода, в котором инициализируется объект типа `JdbcTemplate`.

```
public class JdbcSingerDao
       implements SingerDao, InitializingBean {
  private DataSource dataSource;
  private JdbcTemplate jdbcTemplate;

  public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
    JdbcTemplate jdbcTemplate = new JdbcTemplate();
    jdbcTemplate.setDataSource(dataSource);
  }
  ...
}
```

Как правило, инициализация объекта типа `JdbcTemplate` происходит в методе `setDataSource()`. И как только источник данных будет внедрен средствами Spring, объект типа `JdbcTemplate` будет также инициализирован и готов к применению. Сконфигурированный таким образом объект типа `JdbcTemplate` оказывается потокобезопасным. Это означает, что можно также инициализировать одиночный экземпляр класса `JdbcTemplate` в XML-файле конфигурации Spring и внедрить его во все классы DAO, реализующие компоненты Spring Beans. Пример такой конфигурации приведен ниже.

```
package com.apress.prospring5.ch6.config;
...
@Configuration
public class EmbeddedJdbcConfig {

  private static Logger logger =
       LoggerFactory.getLogger(EmbeddedJdbcConfig.class);

  @Bean
  public DataSource dataSource() {
    try {
      EmbeddedDatabaseBuilder dbBuilder =
              new EmbeddedDatabaseBuilder();
      return dbBuilder.setType(EmbeddedDatabaseType.H2)
             .addScripts("classpath:db/h2/schema.sql",
             "classpath:db/h2/test-data.sql").build();
    } catch (Exception e) {
      logger.error("Embedded DataSource bean cannot "
                + "be created!", e);
      return null;
    }
```

```
  }

  @Bean public JdbcTemplate jdbcTemplate() {
    JdbcTemplate jdbcTemplate = new JdbcTemplate();
    jdbcTemplate.setDataSource(dataSource());
    return jdbcTemplate;
  }

  @Bean
  public SingerDao singerDao() {
    JdbcSingerDao dao = new JdbcSingerDao();
    dao.setJdbcTemplate(jdbcTemplate());
    return dao;
  }
}
```

Извлечение одиночного значения средствами класса JdbcTemplate

Начнем с простого запроса, по которому возвращается одиночное значение. Допустим, требуется реализовать возможность извлекать имя певца по его идентификатору. Извлечь одиночное значение с помощью класса `JdbcTemplate` нетрудно. Ниже приведена реализация метода `findNameById()` в классе `JdbcSingerDao`. А для остальных методов предоставлены пустые реализации.

```
...
public class JdbcSingerDao
       implements SingerDao, InitializingBean {

  private JdbcTemplate jdbcTemplate;

  public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
   this.jdbcTemplate = jdbcTemplate;
  }

  @Override public String findNameById(Long id) {
    return jdbcTemplate.queryForObject(
           "select first_name || ' ' || "
         + "last_name from singer where id = ?",
           new Object{id}, String.class);
    }

  @Override public void insert(Singer singer) {
    throw new NotImplementedException("insert");
  }
  ...
}
```

В приведенном выше примере кода для извлечения имени певца используется метод `queryForObject()` из класса `JdbcTemplate`. В качестве первого аргумента указывается строка с оператором SQL, а в качестве второго аргумента — параметры, передаваемые оператору SQL для привязки параметров в формате массива объектов. Последний аргумент обозначает возвращаемый тип (в данном случае — `String`). Кроме типа `Object`, можно также запрашивать другие типы, в том числе `Long` и `Integer`. А теперь попробуем получить конкретный результат. Ниже приведен исходный код тестовой программы.

```
package com.apress.prospring5.ch6;
...
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;

public class JdbcCfgTest {

  @Test
  public void testH2DB() {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/embedded-h2-cfg.xml");
    ctx.refresh();
    testDao(ctx.getBean(SingerDao.class));
    ctx.close();
  }

  private void testDao(SingerDao singerDao) {
    assertNotNull(singerDao);
    String singerName = singerDao.findNameById(1l);
     assertTrue("John Mayer".equals(singerName));
  }
}
```

В результате выполнения тестового метода `testH2DB()` предполагается получить символьную строку `"John Mayer"`, возвращаемую из вызова метода `singerDao.findNameById(1l)`. И это предположение проверяется с помощью метода `asertTrue()`. Если при инициализации базы данных возникнут какие-нибудь осложнения, этот тест не пройдет проверку.

Применение именованных параметров запроса с помощью класса `NamedParameterJdbcTemplate`

В предыдущем примере в качестве параметра запроса был использован обычный заполнитель (знак ?), а значение этого параметра пришлось передавать в виде массива типа `Object`. Если применяется обычный заполнитель, то очень важен порядок указания параметров запроса. В частности, порядок размещения параметров в массиве должен быть таким же, как и порядок их указания в запросе.

Некоторые разработчики предпочитают пользоваться именованными параметрами запроса, чтобы гарантировать точную привязку каждого параметра. Соответствующую поддержку в Spring обеспечивает расширение класса `JdbcTemplate` под названием `NamedParameterJdbcTemplate` (из пакета `org.springframework.jdbc.core.namedparam`).

Инициализация объекта типа `NamedParameterJdbcTemplate` выполняется таким же образом, как и инициализация объекта типа `JdbcTemplate`. Поэтому достаточно объявить компонент Spring Bean типа `NamedParameterJdbcTemplate` и внедрить его в класс DAO. В качестве примера ниже приведен новый, усовершенствованный вариант класса `JdbcSingerDao`.

```
package com.apress.prospring5.ch6;

public class JdbcSingerDao
       implements SingerDao, InitializingBean {
  private NamedParameterJdbcTemplate
          namedParameterJdbcTemplate;

  @Override
  public String findNameById(Long id) {
    String sql = "SELECT first_name ||' '|| last_name "
               + "FROM singer WHERE id = :singerId";
    Map<String, Object> namedParameters = new HashMap<>();
    namedParameters.put("singerId", id);
    return namedParameterJdbcTemplate.queryForObject(sql,
                namedParameters, String.class);
  }

  public void setNamedParameterJdbcTemplate
         (NamedParameterJdbcTemplate
          namedParameterJdbcTemplate) {
    this.namedParameterJdbcTemplate =
         namedParameterJdbcTemplate;
  }

  @Override
  public void afterPropertiesSet() throws Exception {
    if (namedParameterJdbcTemplate == null) {
      throw new BeanCreationException("Null "
            + "NamedParameterJdbcTemplate on SingerDao");
    }
  }
  ...
}
```

Как видите, вместо заполнителя ? в данном случае применяется именованный параметр, предваряемый двоеточием. Ниже приведен соответствующий код для тестирования нового варианта класса `JdbcSingerDao`.

```
public class NamedJdbcCfgTest {

  @Test
  public void testCfg() {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               NamedJdbcCfg.class);
    SingerDao singerDao = ctx.getBean(SingerDao.class);
    assertNotNull(singerDao);
    String singerName = singerDao.findNameById(1l);
    assertTrue("John Mayer".equals(singerName));
    ctx.close();
  }
}
```

В результате выполнения тестового метода `testCfg()` предполагается получить символьную строку `"John Mayer"`, возвращаемую из вызова метода `singerDao.findNameById(1l)`. И это предположение проверяется с помощью метода `asertTrue()`. Если при инициализации базы данных возникнут какие-нибудь осложнения, этот тест не пройдет проверку.

Извлечение объектов предметной области с помощью интерфейса RowMapper<T>

Вместо извлечения одиночного значения обычно требуется запрашивать одну или несколько строк из таблицы базы данных, а затем преобразовать каждую строку в соответствующий объект предметной области. В каркасе Spring предоставляется интерфейс `RowMapper<T>` (из пакета `org.springframework.jdbc.core`), предоставляющий простой способ преобразования результирующего набора, получаемого через интерфейс JDBC, в объекты POJO. Продемонстрируем его в действии, реализовав метод `findAll()` из интерфейса `SingerDao` с помощью интерфейса `RowMapper<T>`:

```
package com.apress.prospring5.ch6;
...
public class JdbcSingerDao
       implements SingerDao, InitializingBean {

  private NamedParameterJdbcTemplate
          namedParameterJdbcTemplate;

  public void setNamedParameterJdbcTemplate(
              NamedParameterJdbcTemplate
              namedParameterJdbcTemplate) {
    this.namedParameterJdbcTemplate =
         namedParameterJdbcTemplate;
  }
```

```
  @Override
  public List<Singer> findAll() {
    String sql = "select id, first_name, last_name, "
               + "birth_date from singer";
    return namedParameterJdbcTemplate.query(sql,
           new SingerMapper());
  }

  private static final class SingerMapper
          implements RowMapper<Singer> {

    @Override
    public Singer mapRow(ResultSet rs, int rowNum)
                  throws SQLException {
      Singer singer = new Singer();
      singer.setId(rs.getLong("id"));
      singer.setFirstName(rs.getString("first_name"));
      singer.setLastName(rs.getString("last_name"));
      singer.setBirthDate(rs.getDate("birth_date"));
      return singer;
    }
  }

  @Override
  public void afterPropertiesSet() throws Exception {
    if (namedParameterJdbcTemplate == null) {
      throw new BeanCreationException("Null "
          + "NamedParameterJdbcTemplate on SingerDao");
    }
  }
}
```

В приведенном выше коде определяется статический внутренний класс `SingerMapper`, реализующий интерфейс `RowMapper<Singer>`. Этот класс должен предоставить реализацию метода `mapRow()`, в которой значения из конкретной записи из результирующего набора типа `ResultSet` преобразуются в требуемый объект предметной области. Благодаря тому что он объявлен как статический внутренний класс, интерфейс `RowMapper<Singer>` можно сделать общим для многих методов поиска.

Если воспользоваться лямбда-выражениями, внедренными в версии Java 8, то можно вообще обойтись без явной реализации класса `SingerMapper`. Таким образом, исходный код метода `findAll()` можно реорганизовать следующим образом:

```
public List<Singer> findAll() {
  String sql = "select id, first_name, last_name, "
             + "birth_date from singer";
  return namedParameterJdbcTemplate.query(
         sql, (rs, rowNum) -> {
    Singer singer = new Singer();
    singer.setId(rs.getLong("id"));
```

```
    singer.setFirstName(rs.getString("first_name"));
    singer.setLastName(rs.getString("last_name"));
    singer.setBirthDate(rs.getDate("birth_date"));
    return singer;
  });
}
```

После этого в методе `findAll()` достаточно вызвать метод запроса `query()`, передав ему строку запроса и преобразователь строк из таблицы базы данных. На тот случай, когда для запроса требуются параметры, при вызове метода `query()` предоставляется его перегружаемая версия, принимающая параметры запроса. Ниже приведен тестовый класс, содержащий метод для проверки метода `findAll()`.

```
public class RowMapperTest {
  @Test
  public void testRowMapper() {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               NamedJdbcCfg.class);
    SingerDao singerDao = ctx.getBean(SingerDao.class);
    assertNotNull(singerDao);
    List<Singer> singers = singerDao.findAll();
    assertTrue(singers.size() == 3);

    singers.forEach(singer -> {
      System.out.println(singer);
      if (singer.getAlbums() != null) {
        for (Album album : singer.getAlbums()) {
          System.out.println("---" + album);
        }
      }
    });

    ctx.close();
  }
}
```

Если выполнить исходный код метода `testRowMapper()`, то упомянутый выше тест должен пройти, а на консоль будет выведен приведенный ниже результат. Названия альбомов здесь не выводятся, поскольку в приведенной выше реализации интерфейса `RowMapper<Singer>` они не задаются в возвращаемых экземплярах типа `Singer`.

```
Singer - Id: 1, First name: John, Last name: Mayer,
   Birthday: 1977-10-16
Singer - Id: 2, First name: Eric, Last name: Clapton,
   Birthday: 1945-03-30
Singer - Id: 3, First name: John, Last name: Butler,
   Birthday: 1975-04-01
```

Извлечение вложенных объектов предметной области с помощью интерфейса ResultSetExtractor

А теперь рассмотрим более сложный пример, в котором требуется извлечь данные из родительской (SINGER) и порожденной (ALBUM) таблиц базы данных с помощью операции соединения, а затем преобразовать данные обратно во вложенный объект (типа Album, соответственно вложенный в объект типа Singer). Упомянутый ранее интерфейс RowMapper<T> пригоден только для преобразования строки из таблицы базы данных в одиночный объект предметной области. Для более сложной структуры объектов следует применять интерфейс ResultSetExtractor. Чтобы продемонстрировать его применение, введем в интерфейс SingerDao еще один метод, findAllWithAlbums(), который должен заполнить список певцов со сведениями об их альбомах. Ниже показано, как метод findAllWithAlbums() вводится в интерфейс SingerDao и реализуется с помощью интерфейса ResultSetExtractor.

```
package com.apress.prospring5.ch6;
...
public class JdbcSingerDao
       implements SingerDao, InitializingBean {

  private NamedParameterJdbcTemplate
          namedParameterJdbcTemplate;

  public void setNamedParameterJdbcTemplate(
         NamedParameterJdbcTemplate
         namedParameterJdbcTemplate) {
    this.namedParameterJdbcTemplate =
         namedParameterJdbcTemplate;
  }

  @Override
  public List<Singer> findAllWithAlbums() {
    String sql = "select s.id, s.first_name, s.last_name, "
               + "s.birth_date, a.id as a.album_id, "
               + "a.title, a.release_date from singer s "
               + "left join album a on s.id = a.singer_id";
    return namedParameterJdbcTemplate.query(sql,
           new SingerWithDetailExtractor());
  }

  private static final class SingerWithDetailExtractor
          implements ResultSetExtractor<List<Singer>> {

    @Override
    public List<Singer> extractData(ResultSet rs)
```

```
            throws SQLException, DataAccessException {
        Map<Long, Singer> map = new HashMap<>();
        Singer singer;
        while (rs.next()) {
          Long id = rs.getLong("id");
          singer = map.get(id);
          if (singer == null) {
            singer = new Singer();
            singer.setId(id);
            singer.setFirstName(rs.getString("first_name"));
            singer.setLastName(rs.getString("last_name"));
            singer.setBirthDate(rs.getDate("birth_date"));
            singer.setAlbums(new ArrayList<>());
            map.put(id, singer);
          }
          Long albumId = rs.getLong("singer_tel_id");
          if (albumId > 0) {
            Album album = new Album();
            album.setId(albumId);
            album.setSingerId(id);
            album.setTitle(rs.getString("title"));
            album.setReleaseDate(rs.getDate("release_date"));
            singer.addAbum(album);
          }
        }
        return new ArrayList<>(map.values());
      }
    }

    @Override
    public void afterPropertiesSet() throws Exception {
      if (namedParameterJdbcTemplate == null) {
        throw new BeanCreationException("Null "
                + "NamedParameterJdbcTemplate on SingerDao");
      }
    }
}
```

Приведенный выше код похож на исходный код из примера реализации интерфейса `RowMapper`, но на этот раз объявляется внутренний класс, реализующий интерфейс `ResultSetExtractor`. Затем в этом коде реализуется метод `extractData()` для преобразования результирующего набора в список объектов типа `Singer`. В методе `findAllWithAlbums()` составляется запрос, в котором используется левое соединение двух таблиц базы данных, чтобы извлечь из нее и сведения о певцах без альбомов. В итоге получается декартово произведение двух таблиц. И, наконец, в рассматриваемом здесь коде вызывается метод `JdbcTemplate.query()`, которому передается строка запроса и экземпляр внутреннего класса `SingerWithDetail`

Extractor, реализующего средство извлечения результирующего набора из базы данных.

Безусловно, вместо внутреннего класса SingerWithDetailExtractor можно воспользоваться лямбда-выражениями. В качестве примера ниже приведен вариант метода findAllWithAlbums(), в котором применяются лямбда-выражения, внедренные в версии Java 8.

```
public List<Singer> findAllWithAlbums() {
  String sql = "select s.id, s.first_name, s.last_name, "
             + "s.birth_date, a.id as a.album_id, "
             + "a.title, a.release_date from singer s "
             + "left join album a on s.id = a.singer_id";
  return namedParameterJdbcTemplate.query(sql, rs -> {
    Map<Long, Singer> map = new HashMap<>();
    Singer singer;
    while (rs.next()) {
      Long id = rs.getLong("id");
      singer = map.get(id);
      if (singer == null) {
        singer = new Singer();
        singer.setId(id);
        singer.setFirstName(rs.getString("first_name"));
        singer.setLastName(rs.getString("last_name"));
        singer.setBirthDate(rs.getDate("birth_date"));
        singer.setAlbums(new ArrayList<>());
        map.put(id, singer);
      }
      Long albumId = rs.getLong("singer_tel_id");
      if (albumId > 0) {
        Album album = new Album();
        album.setId(albumId);
        album.setSingerId(id);
        album.setTitle(rs.getString("title"));
        album.setReleaseDate(rs.getDate("release_date"));
        singer.addAbum(album);
      }
    }
    return new ArrayList<>(map.values());
  });
}
```

Ниже приведен тестовый класс, содержащий метод, предназначенный для проверки метода findAllWithAlbums().

```
public class ResultSetExtractorTest {

  @Test
  public void testResultSetExtractor() {
    GenericApplicationContext ctx =
```

```
            new AnnotationConfigApplicationContext(
                NamedJdbcCfg.class);
    SingerDao singerDao = ctx.getBean(SingerDao.class);
    assertNotNull(singerDao);
    List<Singer> singers = findAllWithAlbums();
    assertTrue(singers.size() == 3);

    singers.forEach(singer -> {
      System.out.println(singer);
      if (singer.getAlbums() != null) {
        for (Album album : singer.getAlbums()) {
          System.out.println("\t--> " + album);
        }
      }
    });

    ctx.close();
  }
}
```

Если выполнить исходный код метода `testResultSetExtractor()`, то упомянутый выше тест должен пройти, а на консоль будет выведен следующий результат:

```
Singer - Id: 1, First name: John, Last name: Mayer,
   Birthday: 1977-10-16
   --> Album - Id: 2, Singer id: 1, Title: Battle Studies,
       Release Date: 2009-11-17
   --> Album - Id: 1, Singer id: 1,
       Title: The Search For Everything,
       Release Date: 2017-01-20
Singer - Id: 2, First name: Eric, Last name: Clapton,
   Birthday: 1945-03-30
   --> Album - Id: 3, Singer id: 2, Title: From The Cradle,
       Release Date: 1994-09-13
Singer - Id: 3, First name: John, Last name: Butler,
   Birthday: 1975-04-01
```

Как видите, в приведенном выше результате перечислены сведения о певцах и их альбомах. Эти сведения основываются на сценарии из файла `resources/db/test-data.sql`, предназначенного для заполнения базы данных через интерфейс JDBC для каждого примера проекта из этой главы.

До сих пор было показано, как применять класс `JdbcTemplate` для выполнения ряда общих операций обработки запросов. В классе `JdbcTemplate` (а также в классе `NamedParameterJdbcTemplate`) предоставляется несколько перегружаемых вариантов метода `update()`, поддерживающих операции обновления данных, включая ввод, обновление, удаление и т.д. Но поскольку метод `update()` не требует особых пояснений, мы оставляем его вам для самостоятельного изучения. С другой стороны,

для выполнения операций обновления в примерах из последующих разделов будет использоваться класс `SqlUpdate`, предоставляемый в Spring.

Классы Spring, моделирующие операции в JDBC

В предыдущем разделе было показано, что класс `JdbcTemplate` и связанные с ним служебные классы для преобразования данных значительно упрощают программную модель при разработке логики доступа к данным через интерфейс JDBC. На основании класса `JdbcTemplate` в каркасе Spring предоставляется также целый ряд полезных классов, моделирующих операции над данными через интерфейс JDBC и дающих разработчикам возможность реализовывать логику запроса и преобразований результирующего набора типа `ResultSet` в объекты предметной области способом, более похожим на объектно-ориентированный. В этом разделе представлены следующие классы.

- **Класс `MappingSqlQuery<T>`.** Позволяет заключить строку запроса вместе с методом `mapRow()` в оболочку единственного класса.
- **Класс `SqlUpdate`.** Позволяет заключить в свою оболочку любой SQL-оператор обновления. В нем также предоставляется немало полезных методов для привязки параметров SQL, извлечения ключа, сгенерированного в системе управления реляционной базой данных (СУРБД) после ввода новой записи, и т.д.
- **Класс `BatchSqlUpdate`.** Позволяет выполнять пакетные операции обновления. Например, можно перебрать в цикле содержимое объекта Java типа `List` и с помощью класса `BatchSqlUpdate` ввести в очередь записи, для которых затем будут автоматически выполнены операторы обновления в пакетном режиме. Размер пакета можно задать и сбросить операцию в любой удобный момент.
- **Класс `SqlFunction<T>`.** Позволяет вызывать хранимые в базе данных функции с аргументами и возвращаемым типом. Имеется также класс `StoredProcedure`, помогающий вызывать хранимые процедуры.
- **Классы JDBC DAO.** Представляют объекты DAO и устанавливаются с помощью аннотаций.

Прежде всего выясним, как установить класс реализации DAO, используя аннотации. В следующем примере кода интерфейс `SingerDao` реализуется постепенно, метод за методом, до тех пор, пока не будет получена полная его реализация. Ниже приведено определение интерфейса `SingerDao` с полным перечнем предоставляемых им услуг доступа к данным.

```
package com.apress.prospring5.ch6.dao;

import com.apress.prospring5.ch6.entities.Singer;
import java.util.List;

public interface SingerDao {
```

```
  List<Singer> findAll();
  List<Singer> findByFirstName(String firstName);
  String findNameById(Long id);
  String findLastNameById(Long id);
  String findFirstNameById(Long id);
  List<Singer> findAllWithAlbums();

  void insert(Singer singer);
  void update(Singer singer);
  void delete(Long singerId);
  void insertWithAlbum(Singer singer);
}
```

В начале данной книги были представлены стереотипные аннотации, а в качестве особой разновидности аннотации `@Component` — аннотация `@Repository`, предназначенная для определения компонентов Spring Beans, выполняющих операции над базой данных[4]. В приведенном ниже примере кода демонстрируется первоначальное объявление и внедрение свойства источника данных в класс DAO, снабженный аннотацией `@Repository`, с помощью аннотации по спецификации JSR-250.

```
package com.apress.prospring5.ch6.dao;
...
@Repository("singerDao")
public class JdbcSingerDao implements SingerDao {

  private static final Log logger =
          LogFactory.getLog(JdbcSingerDao.class);
  private DataSource dataSource;

  @Resource(name = "dataSource")
  public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
  }

  public DataSource getDataSource() {
    return dataSource;
  }
  ...
}
```

В приведенном выше примере кода аннотация `@Repository` применяется для объявления компонента Spring Bean под названием `singerDao`. А поскольку реали-

[4] Эта аннотация указывает на то, что снабженный ею класс представляет информационное хранилище, первоначально определенное как "механизм инкапсуляции поведения при сохранении, извлечении и поиске информации, эмулирующий коллекцию объектов", как поясняется в книге *Domain-Driven Design* Эрика Эванса (Eric Evans, издательство Addison Wesley, 2003 г.; в русском переводе книга вышла под названием *Предметно-ориентированное проектирование* в ИД "Вильямс", 2011 г.).

зующий его класс содержит код доступа к данным, то аннотация `@Repository` предписывает каркасу Spring преобразовать исключения, возникающие при обработке запросов SQL в базе данных, в более удобную для приложения иерархию исключений типа `DataAccessException`, поддерживаемую в Spring. В данном примере кода объявляется также переменная `logger` для вывода протокольных сообщений на консоль с помощью библиотеки SL4J. В определении свойства `dataSource` применяется аннотация `@Resource` по спецификации JSR-250, чтобы предоставить каркасу Spring возможность внедрить источник данных под названием `dataSource`.

В следующем примере кода демонстрируется конфигурационный класс Java, предназначенный на тот случай, когда аннотации применяются для объявления компонентов Spring Beans, обрабатывающих объекты DAO:

```
package com.apress.prospring5.ch6.config;

import org.apache.commons.dbcp2.BasicDataSource;
...
@Configuration
@PropertySource("classpath:db/jdbc2.properties")
@ComponentScan(basePackages = "com.apress.prospring5.ch6")
public class AppConfig {

  private static Logger logger =
          LoggerFactory.getLogger(AppConfig.class);

  @Value("${driverClassName}")
  private String driverClassName;
  @Value("${url}")
  private String url;
  @Value("${username}")
  private String username;
  @Value("${password}")
  private String password;

  @Bean
  public static PropertySourcesPlaceholderConfigurer
         propertySourcesPlaceholderConfigurer() {
    return new PropertySourcesPlaceholderConfigurer();
  }

  @Bean(destroyMethod = "close")
  public DataSource dataSource() {
    try {
      BasicDataSource dataSource = new BasicDataSource();
      dataSource.setDriverClassName(driverClassName);
      dataSource.setUrl(url);
      dataSource.setUsername(username);
      dataSource.setPassword(password);
```

```
      return dataSource;
    } catch (Exception e) {
      logger.error("DBCP DataSource bean "
                + "cannot be created!", e);
      return null;
    }
  }
}
```

В приведенном выше конфигурационном классе объявляется база данных MySQL, доступная через объединяемый источник данных типа `BasicDataSource`, а также выполняется просмотр компонентов Spring Beans для их автоматического обнаружения. В начале этой главы пояснялось, как устанавливать и настраивать базу данных MySQL и создавать ее схему (в данном случае — `musicdb`). Как только требующаяся инфраструктура будет организована, можно перейти к реализации операций в JDBC.

Выборка данных с помощью класса `MappingSqlQuery<T>`

Для моделирования операций обработки запросов в Spring предоставляется класс `MappingSqlQuery<T>`. По существу, экземпляр класса `MappingSqlQuery<T>` конструируется с помощью источника данных и строки запроса, а затем реализуется метод `mapRow()` для преобразования каждой записи из результирующего набора в соответствующий объект предметной области.

Итак, начнем с создания класса `SelectAllSingers`, представляющего операцию обработки запроса на выборку сведений обо всех певцах и расширяющего абстрактный класс `MappingSqlQuery<T>`. Ниже приведен исходный код класса `SelectAllSingers`.

```
package com.apress.prospring5.ch6;

import java.sql.ResultSet;
import java.sql.SQLException;
import javax.sql.DataSource;
import com.apress.prospring5.ch6.entities.Singer;
import org.springframework.jdbc.object.MappingSqlQuery;

public class SelectAllSingers
       extends MappingSqlQuery<Singer> {
  private static final String SQL_SELECT_ALL_SINGER =
          "select id, first_name, last_name, "
        + "birth_date from singer";

  public SelectAllSingers(DataSource dataSource) {
    super(dataSource, SQL_SELECT_ALL_SINGER);
  }
```

```
  protected Singer mapRow(ResultSet rs, int rowNum)
            throws SQLException {
    Singer singer = new Singer();
    singer.setId(rs.getLong("id"));
    singer.setFirstName(rs.getString("first_name"));
    singer.setLastName(rs.getString("last_name"));
    singer.setBirthDate(rs.getDate("birth_date"));
    return singer;
  }
}
```

В классе `SelectAllSingers` объявляется оператор SQL для выборки всех певцов. В конструкторе этого класса вызывается метод `super()` для конструирования экземпляра класса `MappingSqlQuery<Singer>` с помощью заданного источника данных и составленного оператора SQL. Кроме того, в классе `SelectAllSingers` реализуется метод `MappingSqlQuery<T>.mapRow()`, предназначенный для преобразования результирующего набора в объект предметной области типа `Singer`.

Создав класс `SelectAllSingers`, можно реализовать метод `findAll()` в классе `JdbcSingerDao`, как показано ниже.

```
package com.apress.prospring5.ch6;

@Repository("singerDao")
public class JdbcSingerDao implements SingerDao {
  private DataSource dataSource;
  private SelectAllSingers selectAllSingers;

  @Resource(name = "dataSource")
  public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
    this.selectAllSingers = new SelectAllSingers(dataSource);
  }

  @Override
  public List<Singer> findAll() {
    return selectAllSingers.execute();
  }
  ...
}
```

После внедрения источника данных в методе `setDataSource()` получается экземпляр класса `SelectAllSingers`. А в методе `findAll()` просто вызывается метод `execute()`, который косвенно наследуется из абстрактного класса `SqlQuery<T>`. И это все, что нужно сделать. Ниже приведен тестовый класс, содержащий метод, предназначенный для проверки метода `findAll()`.

```
com.apress.prospring5.ch6;
public class AnnotationJdbcTest {
```

```
  private GenericApplicationContext ctx;
  private SingerDao singerDao;

  @Before
  public void setUp() {
    ctx = new AnnotationConfigApplicationContext(
          AppConfig.class);
    singerDao = ctx.getBean(SingerDao.class);
    assertNotNull(singerDao);
  }

  @Test
  public void testFindAll() {
    List<Singer> singers = singerDao.findAll();
    assertTrue(singers.size() == 3);
    singers.forEach(singer -> {
      System.out.println(singer);
      if (singer.getAlbums() != null) {
        for (Album album : singer.getAlbums()) {
          System.out.println("\t--> " + album);
        }
      }
    });
    ctx.close();
  }

  @After
  public void tearDown() {
    ctx.close();
  }
}
```

Если выполнить исходный код метода `testFindAll()`, то упомянутый выше тест должен пройти, а на консоль будет выведен такой результат:

```
Singer - Id: 1, First name: John, Last name: Mayer,
   Birthday: 1977-10-15
Singer - Id: 2, First name: Eric, Last name: Clapton,
   Birthday: 1945-03-29
Singer - Id: 3, First name: John, Last name: Butler,
   Birthday: 1975-03-31
```

Если установить уровень протоколирования `DEBUG` для пакета `org.springframework.jdbc`, отредактировав файл конфигурации `logback-test.xml` и введя в него следующий элемент разметки:

```
<logger name="org.springframework.jdbc" level="debug"/>
```

то на консоль будет также выведен запрос, передаваемый каркасом Spring на обработку:

```
DEBUG o.s.j.c.JdbcTemplate - Executing prepared SQL query
DEBUG o.s.j.c.JdbcTemplate - Executing prepared SQL statement
[select id, first_name, last_name, birth_date from singer]
```

А теперь перейдем к реализации метода `findByFirstName()`, принимающего один именованный параметр. Аналогично предыдущему примеру, создадим для моделирования операции поиска певцов по имени отдельный класс `SelectSingerByFirstName`, исходный код которого приведен ниже.

```
package com.apress.prospring5.ch6;
...
import org.springframework.jdbc.core.SqlParameter;

public class SelectSingerByFirstName
       extends MappingSqlQuery<Singer> {
  private static final String SQL_FIND_BY_FIRST_NAME =
          "select id, first_name, last_name, birth_date "
        + "from singer where first_name = :first_name";
  public SelectSingerByFirstName(DataSource dataSource) {
    super(dataSource, SQL_FIND_BY_FIRST_NAME);
    super.declareParameter(new SqlParameter("first_name",
                           Types.VARCHAR));
    }

  protected Singer mapRow(ResultSet rs, int rowNum)
            throws SQLException {
    Singer singer = new Singer();

    singer.setId(rs.getLong("id"));
    singer.setFirstName(rs.getString("first_name"));
    singer.setLastName(rs.getString("last_name"));
    singer.setBirthDate(rs.getDate("birth_date"));

    return singer;
  }
}
```

Класс `SelectSingerByFirstName` похож на класс `SelectAllSingers`. Прежде всего, в нем используется другой оператор SQL с именованным параметром `first_name`. В конструкторе данного класса вызывается метод `declareParameter()`, косвенно наследуемый из абстрактного класса `org.springframework.jdbc.object.RdbmsOperation`). Перейдем далее к реализации метода `findByFirstName()` в классе `JdbcSingerDao`. Ниже приведен обновленный исходный код данного класса.

```
package com.apress.prospring5.ch6.dao;
...
@Repository("singerDao")
public class JdbcSingerDao implements SingerDao {
```

```
  private static Logger logger =
          LoggerFactory.getLogger(JdbcSingerDao.class);
  private DataSource dataSource;
  private SelectSingerByFirstName selectSingerByFirstName;
  @Resource(name = "dataSource")
  public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
    this.selectSingerByFirstName =
         new SelectSingerByFirstName(dataSource);
  }

  @Override
  public List<Singer> findByFirstName(String firstName) {
    Map<String, Object> paramMap = new HashMap<>();
    paramMap.put("first_name", firstName);
    return selectSingerByFirstName
           .executeByNamedParam(paramMap);
  }
  ...
}
```

После внедрения источника данных в приведенном выше коде получается экземпляр класса `SelectSingerByFirstName`. А затем в методе `findByFirstName()` конструируется экземпляр класса `HashMap` с именованными параметрами и их значениями. И, наконец, в данном коде вызывается метод `executeByNamedParam()`, косвенно унаследованный из абстрактного класса `SqlQuery<T>`. Проверим этот метод, выполнив приведенный ниже метод `testFindByFirstName()`.

```
package com.apress.prospring5.ch6;
...
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;

public class AnnotationJdbcTest {
  private GenericApplicationContext ctx;
  private SingerDao singerDao;

  @Before
  public void setUp() {
    ctx = new AnnotationConfigApplicationContext(
              AppConfig.class);
    singerDao = ctx.getBean(SingerDao.class);
    assertNotNull(singerDao);
  }
```

```
  @Test
  public void testFindByFirstName() {
    List<Singer> singers =
         singerDao.findByFirstName("John");
    assertTrue(singers.size() == 1);
    listSingers(singers);
    ctx.close();
  }

  private void listSingers(List<Singer> singers) {
    singers.forEach(singer -> {
      System.out.println(singer);
      if (singer.getAlbums() != null) {
        for (Album album : singer.getAlbums()) {
          System.out.println("\t--> " + album);
        }
      }
    });
  }

  @After
  public void tearDown() {
    ctx.close();
  }
}
```

Если выполнить исходный код метода `testFindByFirstName()`, то упомянутый выше тест должен пройти, а на консоль будет выведен приведенный ниже результат.

```
Singer - Id: 1, First name: John, Last name: Mayer,
   Birthday: 1977-10-15
```

Здесь важно отметить, что класс `MappingSqlQuery<T>` подходит только для преобразования одной строки из таблицы базы данных в объект предметной области. Для вложенного объекта по-прежнему приходится применять класс `JdbcTemplate` вместе с интерфейсом `ResultSetExtractor`, как это было сделано в примере метода `findAllWithAlbums()`, представленного ранее в разделе, посвященном классу `JdbcTemplate`.

Обновление данных с помощью класса `SqlUpdate`

Для обновления данных в Spring предоставляется класс `SqlUpdate`. Ниже приведен исходный код класса `UpdateSinger`, расширяющего класс `SqlUpdate` для реализации операций обновления.

```
package com.apress.prospring5.ch6;

import java.sql.Types;
import javax.sql.DataSource;
```

```
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.SqlUpdate;

public class UpdateSinger extends SqlUpdate {
  private static final String SQL_UPDATE_SINGER =
          "update singer set first_name=:first_name, "
        + "last_name=:last_name, birth_date=:birth_date "
        + "where id=:id";

  public UpdateSinger(DataSource dataSource) {
    super(dataSource, SQL_UPDATE_SINGER);
    super.declareParameter(new SqlParameter(
                           "first_name", Types.VARCHAR));
    super.declareParameter(new SqlParameter("last_name",
                           Types.VARCHAR));
    super.declareParameter(new SqlParameter("birth_date",
                           Types.DATE));
    super.declareParameter(new SqlParameter("id",
                           Types.INTEGER));
  }
}
```

Приведенный выше код должен быть уже знаком вам. В нем получается экземпляр класса `SqlUpdate` с запросом, а также объявляются именованные параметры. Ниже представлена реализация метода `update()` в классе `JdbcSingerDao`.

```
package com.apress.prospring5.ch6.dao;
...
@Repository("singerDao")
public class JdbcSingerDao implements SingerDao {

  private static Logger logger =
          LoggerFactory.getLogger(JdbcSingerDao.class);
  private DataSource dataSource;
  private UpdateSinger updateSinger;

  @Override
  public void update(Singer singer) {
    Map<String, Object> paramMap =
        new HashMap<String, Object>();
    paramMap.put("first_name", singer.getFirstName());
    paramMap.put("last_name", singer.getLastName());
    paramMap.put("birth_date", singer.getBirthDate());
    paramMap.put("id", singer.getId());
    updateSinger.updateByNamedParam(paramMap);
    logger.info("Existing singer updated with id: "
              + singer.getId());
  }
```

```
  @Resource(name = "dataSource")
  public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
    this.updateSinger = new UpdateSinger(dataSource);
  }
  ...
}
```

После внедрения источника данных в приведенном выше коде конструируется экземпляр класса `UpdateSinger`. В методе `update()` на основе переданного ему объекта типа `Singer` создается хеш-отображение именованных параметров типа `HashMap`, после чего вызывается метод `updateByNamedParam()` для обновления записи о певце. Чтобы проверить операцию обновления, введем новый метод в тестовый класс `AnnotationJdbcTest`, как показано ниже.

```
package com.apress.prospring5.ch6;
...
public class AnnotationJdbcTest {

  private GenericApplicationContext ctx;
  private SingerDao singerDao;

  public void setUp() {
    ctx = new AnnotationConfigApplicationContext(
              AppConfig.class);
    singerDao = ctx.getBean(SingerDao.class);
    assertNotNull(singerDao);
  }

  @Test
  public void testSingerUpdate() {
    Singer singer = new Singer();
    singer.setId(1L);
    singer.setFirstName("John Clayton");
    singer.setLastName("Mayer");
    singer.setBirthDate(new Date(
          (new GregorianCalendar(1977, 9, 16))
          .getTime().getTime()));
    singerDao.update(singer);
    List<Singer> singers = singerDao.findAll();
    listSingers(singers);
  }

  private void listSingers(List<Singer> singers) {
    singers.forEach(singer -> {
      System.out.println(singer);
      if (singer.getAlbums() != null) {
        for (Album album : singer.getAlbums()) {
          System.out.println("\t--> " + album);
```

```
        }
      }
    });
  }

  @After
  public void tearDown() {
    ctx.close();
  }
}
```

Сначала в приведенном выше коде конструируется объект типа `Singer`, а затем вызывается метод `update()`. Если выполнить этот тестовый код, то на консоль из последнего в нем метода `listSingers()` будет выведен приведенный ниже результат, при условии, что тест пройдет.

```
Singer - Id: 1, First name: John Clayton, Last name: Mayer,
   Birthday: 1977-10-16
Singer - Id: 2, First name: Eric, Last name: Clapton,
   Birthday: 1945-03-29
Singer - Id: 3, First name: Jimi, Last name: Hendrix,
   Birthday: 1942-11-26
```

Ввод данных и извлечение сгенерированного ключа

Для ввода данных можно также воспользоваться классом `SqlUpdate`. Любопытно отметить, каким образом генерируется первичный ключ, который обычно хранится в столбце идентификатора `id`. Значение этого ключа доступно только после завершения оператора ввода, когда СУРБД сгенерирует значение, идентифицирующее вставляемую запись. Столбец идентификатора `id` объявлен с атрибутом `AUTO_INCREMENT` и содержит первичный ключ, поэтому СУРБД присваивает ему значение во время выполнения операции ввода. Так, если вы пользуетесь СУРБД Oracle, то, скорее всего, получите сначала однозначный идентификатор из последовательности операций в Oracle, а затем выполните оператор вставки с результатом запроса.

В прежних версиях JDBC сделать подобное было не так-то просто. Так, в СУРБД MySQL для этого приходилось выполнять оператор `select last_insert_id()`, а в СУРБД Microsoft SQL Server — оператор `select @@IDENTITY`.

Правда, в версии JDBC 3.0 появилась новая функциональная возможность, позволяющая единообразно извлекать ключи, сгенерированные в СУРБД. Ниже приведена реализация метода `insert()`, где сгенерированный ключ извлекается для вставляемой записи сведений о певце. Эта реализация пригодна для большинства, если не всех баз данных. Нужно лишь удостовериться в наличии драйвера, совместимого с интерфейсом JDBC, начиная с версии 3.0.

Начнем с создания класса `InsertSinger`, предназначенного для операции вставки и расширяющего класс `SqlUpdate`. Исходный код класса `InsertSinger` приведен ниже.

```
package com.apress.prospring5.ch6;

import java.sql.Types;
import javax.sql.DataSource;
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.SqlUpdate;

public class InsertSinger extends SqlUpdate {
  private static final String SQL_INSERT_SINGER =
          "insert into singer (first_name, "
        + "last_name, birth_date)
          values (:first_name, :last_name, :birth_date)";

  public InsertSinger(DataSource dataSource) {
    super(dataSource, SQL_INSERT_SINGER);
    super.declareParameter(new SqlParameter("first_name",
                           Types.VARCHAR));
    super.declareParameter(new SqlParameter("last_name",
                           Types.VARCHAR));
    super.declareParameter(new SqlParameter("birth_date",
                           Types.DATE));
    super.setGeneratedKeysColumnNames(new String {"id"});
    super.setReturnGeneratedKeys(true);
  }
}
```

Класс `InsertSinger` очень похож на класс `UpdateSinger`, поэтому в него нужно внести лишь два дополнения. Во-первых, вызвать метод `SqlUpdate.setGeneratedKeysColumnNames()` при конструировании экземпляра класса `InsertSinger`, чтобы объявить имя столбца идентификатора `id`. И во-вторых, вызвать метод `SqlUpdate.setReturnGeneratedKeys()`, чтобы дать исходному драйверу JDBC команду извлечь сгенерированный ключ. Ниже приведена реализация метода `insert()` в классе `JdbcSingerDao`.

```
package com.apress.prospring5.ch6.dao;
...
@Repository("singerDao")
public class JdbcSingerDao implements SingerDao {
  private static Logger logger =
          LoggerFactory.getLogger(JdbcSingerDao.class);
  private DataSource dataSource;
  private InsertSinger insertSinger;

  @Override
```

```
  public void insert(Singer singer) {
    Map<String, Object> paramMap = new HashMap<>();
    paramMap.put("first_name", singer.getFirstName());
    paramMap.put("last_name", singer.getLastName());
    paramMap.put("birth_date", singer.getBirthDate());
    KeyHolder keyHolder = new GeneratedKeyHolder();
    insertSinger.updateByNamedParam(paramMap, keyHolder);
    singer.setId(keyHolder.getKey().longValue());
    logger.info("New singer inserted with id: "
              + singer.getId());
  }

  @Resource(name = "dataSource")
  public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
    this.insertSinger = new InsertSinger(dataSource);
  }
  ...
}
```

После внедрения источника данных в приведенном выше коде получается экземпляр класса `InsertSinger`, а в теле метода `insert()` вызывается метод `SqlUpdate.updateByNamedParam()`, которому передается экземпляр класса `KeyHolder`, где будет храниться сгенерированный идентификатор. После вставки данных сгенерированный ключ можно будет извлечь из объекта типа `KeyHolder`. Ниже приведен тестовый код для проверки метода `insert()`.

```
package com.apress.prospring5.ch6;
...
public class AnnotationJdbcTest {
  private GenericApplicationContext ctx;
  private SingerDao singerDao;

  @Before
  public void setUp() {
    ctx = new AnnotationConfigApplicationContext(
              AppConfig.class);
    singerDao = ctx.getBean(SingerDao.class);
    assertNotNull(singerDao);
  }

  @Test
  public void testSingerInsert() {
    Singer singer = new Singer();
    singer.setFirstName("Ed");
    singer.setLastName("Sheeran");
    singer.setBirthDate(new Date(
          (new GregorianCalendar(1991, 1, 17))
          .getTime().getTime()));
```

```
    singerDao.insert(singer);
    List<Singer> singers = singerDao.findAll();
    listSingers(singers);
  }

  private void listSingers(List<Singer> singers) {
    singers.forEach(singer -> {
      System.out.println(singer);
      if (singer.getAlbums() != null) {
        for (Album album : singer.getAlbums()) {
          System.out.println("\t--> " + album);
        }
      }
    });
  }

  @After
  public void tearDown() {
    ctx.close();
  }
}
```

Если тест проходит при выполнении приведенного выше тестового кода, то на консоль выводится следующий результат:

```
Singer - Id: 1, First name: John Clayton, Last name: Mayer,
   Birthday: 1977-10-16
Singer - Id: 2, First name: Eric, Last name: Clapton,
   Birthday: 1945-03-29
Singer - Id: 3, First name: Jimi, Last name: Hendrix,
   Birthday: 1942-11-26
Singer - Id: 6, First name: Ed, Last name: Sheeran,
   Birthday: 1991-02-17
```

Группирование операций с помощью класса `BatchSqlUpdate`

Для выполнения групповых операций служит класс `BatchSqlUpdate`, а новый метод `insertWithAlbum()` — для ввода в базу данных сведений о певце и выпущенном им альбоме. Кроме того, для вставки записи со сведениями об альбоме в таблицу базы данных потребуется класс `InsertSingerAlbum`, исходный код которого приведен ниже.

```
package com.apress.prospring5.ch6;

import java.sql.Types;
import javax.sql.DataSource;
```

```
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.BatchSqlUpdate;

public class InsertSingerAlbum extends BatchSqlUpdate {
  private static final String SQL_INSERT_SINGER_ALBUM =
       "insert into album (singer_id, title, release_date)
  values (:singer_id, :title, :release_date)";

  private static final int BATCH_SIZE = 10;

  public InsertSingerAlbum(DataSource dataSource) {
    super(dataSource, SQL_INSERT_SINGER_ALBUM);

    declareParameter(new SqlParameter(
                     "singer_id", Types.INTEGER));
    declareParameter(new SqlParameter("title",
                     Types.VARCHAR));
    declareParameter(new SqlParameter("release_date",
                     Types.DATE));
    setBatchSize(BATCH_SIZE);
  }
}
```

Обратите внимание на то, что в конструкторе данного класса вызывается метод `BatchSqlUpdate.setBatchSize()`, чтобы установить размер пакета для операции вставки, выполняемой через интерфейс JDBC. Ниже приведена реализация метода `insertWithAlbum()` в классе `JdbcSingerDao`.

```
package com.apress.prospring5.ch6.dao;
...
@Repository("singerDao")
public class JdbcSingerDao implements SingerDao {

  private static Logger logger =
          LoggerFactory.getLogger(JdbcSingerDao.class);

  private DataSource dataSource;
  private InsertSingerAlbum insertSingerAlbum;

  @Override
  public void insertWithAlbum(Singer singer) {
    insertSingerAlbum = new InsertSingerAlbum(dataSource);
    Map<String, Object> paramMap = new HashMap<>();
    paramMap.put("first_name", singer.getFirstName());
    paramMap.put("last_name", singer.getLastName());
    paramMap.put("birth_date", singer.getBirthDate());
    KeyHolder keyHolder = new GeneratedKeyHolder();
    insertSinger.updateByNamedParam(paramMap, keyHolder);
    singer.setId(keyHolder.getKey().longValue());
```

```
    logger.info("New singer inserted with id: "
              + singer.getId());
    List<Album> albums = singer.getAlbums();
    if (albums != null) {
      for (Album album : albums) {
        paramMap = new HashMap<>();
        paramMap.put("singer_id", singer.getId());
        paramMap.put("title", album.getTitle());
        paramMap.put("release_date",
                     album.getReleaseDate());
        insertSingerAlbum.updateByNamedParam(paramMap);
      }
    }
    insertSingerAlbum.flush();
  }

  @Resource(name = "dataSource")
  public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
  }

  @Override
  public List<Singer> findAllWithAlbums() {
    JdbcTemplate jdbcTemplate =
                 new JdbcTemplate(getDataSource());
    String sql = "SELECT s.id, s.first_name, "
               + "s.last_name, s.birth_date "
               + "a.title, a.release_date FROM "
               + "singer s, a.id AS album_id, "
               + "LEFT JOIN album a ON s.id = a.singer_id";
    return jdbcTemplate.query(sql,
           new SingerWithAlbumExtractor());
  }

  private static final class SingerWithAlbumExtractor
          implements ResultSetExtractor<List<Singer>> {
    public List<Singer> extractData(ResultSet rs)
           throws SQLException, DataAccessException {
      Map<Long, Singer> map = new HashMap<>();
      Singer singer;
      while (rs.next()) {
        Long id = rs.getLong("id");
        singer = map.get(id);
        if (singer == null) {
          singer = new Singer();
          singer.setId(id);
          singer.setFirstName(rs.getString("first_name"));
          singer.setLastName(rs.getString("last_name"));
          singer.setBirthDate(rs.getDate("birth_date"));
```

```
            singer.setAlbums(new ArrayList<>());
            map.put(id, singer);
          }
          Long albumId = rs.getLong("album_id");
          if (albumId > 0) {
            Album album = new Album();
            album.setId(albumId);
            album.setSingerId(id);
            album.setTitle(rs.getString("title"));
            album.setReleaseDate(rs.getDate("release_date"));
            singer.getAlbums().add(album);
          }
        }
        return new ArrayList<>(map.values());
      }
    }
    ...
}
```

Всякий раз, когда вызывается метод `insertWithAlbum()`, получается новый экземпляр класса `InsertSingerAlbum`, поскольку класс `BatchSqlUpdate` не является потокобезопасным. Далее он используется подобно классу `SqlUpdate`, за исключением того, что класс `BatchSqlUpdate` сначала разместит операции вставки в очереди, а затем передаст их базе данных пакетом. И как только количество записей окажется равным размеру пакета, Spring выполнит в базе данных операцию групповой вставки записей, ожидающих своей очереди. С другой стороны, по завершении данной операции вызывается метод `BatchSqlUpdate.flush()`, чтобы предписать каркасу Spring сбросить все ожидающие операции (т.е. находящиеся в очереди операции вставки записей, пока еще не достигших размера пакета). И, наконец, объекты типа `Album` циклически перебираются в объекте типа `Singer` и вызывается метод `BatchSqlUpdate.updateByNamedParam()`. Чтобы упростить тестирование, реализуется также метод `insertWithAlbum()`. Рассмотренную здесь реализацию можно сделать более компактной, воспользовавшись лямбда-выражениями, внедренными в версии Java 8, как показано ниже.

```
public List<Singer> findAllWithAlbums() {
  JdbcTemplate jdbcTemplate =
               new JdbcTemplate(getDataSource());
  String sql = "SELECT s.id, s.first_name, s.last_name, "
             + "s.birth_date, a.id AS album_id, a.title, "
             + "a.release_date FROM singer s "
             + "LEFT JOIN album a ON s.id = a.singer_id";
  return jdbcTemplate.query(sql, rs -> {
    Map<Long, Singer> map = new HashMap<>();
    Singer singer;
    while (rs.next()) {
      Long id = rs.getLong("id");
```

```
      singer = map.get(id);
      if (singer == null) {
        singer = new Singer();
        singer.setId(id);
        singer.setFirstName(rs.getString("first_name"));
        singer.setLastName(rs.getString("last_name"));
        singer.setBirthDate(rs.getDate("birth_date"));
        singer.setAlbums(new ArrayList<>());
        map.put(id, singer);
      }
      Long albumId = rs.getLong("album_id");
      if (albumId > 0) {
        Album album = new Album();
        album.setId(albumId);
        album.setSingerId(id);
        album.setTitle(rs.getString("title"));
        album.setReleaseDate(rs.getDate("release_date"));
        singer.getAlbums().add(album);
      }
    }
    return new ArrayList<>(map.values());
  });
}
```

Ниже приведен тестовый код для проверки метода `insertWithAlbum()`.

```
package com.apress.prospring5.ch6;
...
public class AnnotationJdbcTest {
  private GenericApplicationContext ctx;
  private SingerDao singerDao;

  @Before
  public void setUp() {
    ctx = new AnnotationConfigApplicationContext(
              AppConfig.class);
    singerDao = ctx.getBean(SingerDao.class);
    assertNotNull(singerDao);
  }

  @Test
  public void testSingerInsertWithAlbum() {
    Singer singer = new Singer();
    singer.setFirstName("BB");
    singer.setLastName("King");
    singer.setBirthDate(new Date(
          (new GregorianCalendar(1940, 8, 16))
          .getTime().getTime()));
    Album album = new Album();
    album.setTitle("My Kind of Blues");
```

```
    album.setReleaseDate(new Date(
         (new GregorianCalendar(1961, 7, 18))
         .getTime().getTime()));
    singer.addAbum(album);
    album = new Album();
    album.setTitle("A Heart Full of Blues");
    album.setReleaseDate(new Date(
         (new GregorianCalendar(1962, 3, 20))
         .getTime().getTime()));
    singer.addAbum(album);

    singerDao.insertWithAlbum(singer);

    List<Singer> singers = singerDao.findAllWithAlbums();
    listSingers(singers);
  }

  private void listSingers(List<Singer> singers) {
    singers.forEach(singer -> {
      System.out.println(singer);
      if (singer.getAlbums() != null) {
        for (Album album : singer.getAlbums()) {
          System.out.println("\t--> " + album);
        }
      }
    });
  }

  @After
  public void tearDown() {
    ctx.close();
  }
}
```

Если тест проходит при выполнении приведенного выше тестового кода, то на консоль выводится следующий результат:

```
Singer - Id: 1, First name: John, Last name: Mayer,
   Birthday: 1977-10-15
   --> Album - Id: 1, Singer id: 1,
      Title: The Search For Everything,
                Release Date: 2017-01-19
   --> Album - Id: 2, Singer id: 1, Title: Paradise Valley,
                Release Date: 2013-08-19
   --> Album - Id: 3, Singer id: 1, Title: Born and Raised,
                Release Date: 2012-05-22
   --> Album - Id: 4, Singer id: 1, Title: Battle Studies,
                Release Date: 2009-11-16
Singer - Id: 2, First name: Eric, Last name: Clapton,
```

```
   Birthday: 1945-03-29
  --> Album - Id: 5, Singer id: 2, Title: From The Cradle,
              Release Date: 1994-09-13
Singer - Id: 3, First name: Jimi, Last name: Hendrix,
   Birthday: 1942-11-26
Singer - Id: 4, First name: BB, Last name: King,
   Birthday: 1940-09-15
  --> Album - Id: 6, Singer id: 4, Title: My Kind of Blues,
              Release Date: 1961-08-17
  --> Album - Id: 7, Singer id: 4,
           Title: A Heart Full of Blues,
              Release Date: 1962-04-19
```

Вызов хранимых функций с помощью класса `SqlFunction`

В каркасе Spring предоставляются также классы, упрощающие выполнение хранимых процедур или функций средствами интерфейса JDBC. В этом разделе будет показано, как выполнить простую функцию с помощью класса `SqlFunction<T>`. С этой целью мы воспользуемся базой данных MySQL, создадим хранимую функцию и вызовем ее с помощью данного класса.

Здесь предполагается, что в нашем распоряжении имеется база данных MySQL со схемой `musicdb`, а также заданным одинаковым значением `prospring5` для имени пользователя и пароля (это та же самая база данных, которая применялась ранее в примере из раздела "Исследование инфраструктуры JDBC"). Итак, создадим хранимую функцию `getFirstNameById()`, принимающую идентификатор певца и возвращающую его имя. Ниже приведен сценарий для создания этой хранимой функции в MySQL (см. файл `stored-function.sql`). Выполните этот сценарий в указанной выше базе данных MySQL.

```
DELIMITER //
CREATE FUNCTION getFirstNameById(in_id INT)
    RETURNS VARCHAR(60)
    BEGIN
        RETURN (SELECT first_name FROM singer
                WHERE id = in_id);
    END //
DELIMITER;
```

Созданная таким образом хранимая функция принимает идентификатор и возвращает имя из записи певца по этому идентификатору. Создадим далее класс `StoredFunctionFirstNameById`, представляющий операцию вызова хранимой функции и расширяющий класс `SqlFunction<T>`. Исходный код этого класса приведен ниже.

```
package com.apress.prospring5.ch6;

import java.sql.Types;
import javax.sql.DataSource;
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.SqlFunction;

public class StoredFunctionFirstNameById
       extends SqlFunction<String> {
  private static final String SQL =
                "select getfirstnamebyid(?)";

  public StoredFunctionFirstNameById
         (DataSource dataSource) {
    super(dataSource, SQL);
    declareParameter(new SqlParameter(Types.INTEGER));
    compile();
  }
}
```

Приведенный выше класс расширяет класс `SqlFunction<T>` и передает тип `String`, обозначающий тип, возвращаемый хранимой функцией. Затем в данном классе объявляется строка запроса SQL для вызова хранимой функции из базы данных MySQL. После этого в конструкторе данного класса объявляется параметр и составляется операция. Теперь данный класс готов к применению в классе реализации. Ниже приведен обновленный вариант класса `JdbcSingerDao`, предназначенный для применения хранимой функции.

```
package com.apress.prospring5.ch6.dao;
...
@Repository("singerDao")
public class JdbcSingerDao implements SingerDao {
  private static Logger logger =
          LoggerFactory.getLogger(JdbcSingerDao.class);

  private DataSource dataSource;
  private StoredFunctionFirstNameById
          storedFunctionFirstNameById;

  @Override
  public String findFirstNameById(Long id) {
    List<String> result =
                storedFunctionFirstNameById.execute(id);
    return result.get(0);
  }
  ...
}
```

После внедрения источника данных в приведенном выше коде получается экземпляр класса `StoredFunctionFirstNameById`, а затем в теле метода `findFirstNameById()` вызывается метод `execute()`, которому передается идентификатор певца. Этот метод возвращает список объектов типа `String`, в котором требуется лишь первый элемент, поскольку результирующий набор должен содержать только одну запись. Проверить работоспособность данного кода совсем не трудно, как показано ниже.

```
package com.apress.prospring5.ch6;
...
public class AnnotationJdbcTest {

  private GenericApplicationContext ctx;
  private SingerDao singerDao;

  @Before
  public void setUp() {
    ctx = new AnnotationConfigApplicationContext(
          AppConfig.class);
    singerDao = ctx.getBean(SingerDao.class);
    assertNotNull(singerDao);
  }

  @Test
  public void testFindFirstNameById() {
    String firstName = singerDao.findFirstNameById(2L);
    assertEquals("Eric", firstName);
    System.out.println("Retrieved value: " + firstName);
  }

  @After
  public void tearDown() {
    ctx.close();
  }
}
```

В приведенном выше тестовом коде хранимой функции передается идентификатор, равный 2. В итоге эта функция возвратит имя `Eric` по идентификатору 2 записи в таблице базе данных, если выполнить сценарий из файла `test-data.sql` по отношению к базе данных MySQL. Выполнение тестового кода дает следующий результат:

```
o.s.j.c.JdbcTemplate - Executing prepared SQL query
o.s.j.c.JdbcTemplate - Executing prepared SQL statement
    [select getfirstnamebyid?]
o.s.j.d.DataSourceUtils - Fetching JDBC Connection
        from DataSource
o.s.j.d.DataSourceUtils - Returning JDBC Connection
```

```
        to DataSource
Retrieved value: Eric[5]
```

Как видите, имя было извлечено правильно. Здесь был представлен очень простой пример, демонстрирующий функциональные возможности модуля Spring JDBC. В каркасе Spring предоставляются и другие классы (например, `StoredProcedure`), предназначенные для вызова сложных хранимых процедур, возвращающие составные типы данных. Тем, кого интересует доступ к хранимым процедурам через интерфейс JDBC, рекомендуется обратиться за справкой к руководству по Spring.

Проект Spring Data: расширения JDBC

За последние годы технологии баз данных развивались настолько быстрыми темпами, порождая огромное количество специализированных баз данных, что ныне СУРБД не является единственным выбором при разработке приложений. В ответ на эту эволюцию технологий баз данных, а также с целью удовлетворить потребности сообщества разработчиков был организован проект Spring Data (`http://projects.spring.io/spring-data/`). Главная цель этого проекта — предоставить удобные расширения основных функциональных средств доступа к данным в Spring для взаимодействия с базами данных, отличающимися от традиционных СУРБД.

В проекте Spring Data предоставляются различные расширения. Мы упомянем здесь одно из них: JDBC Extensions (`http://projects.spring.io/spring-data-jdbc-ext/`). Нетрудно догадаться, что в этом расширении предоставляется ряд усовершенствованных средств для упрощения разработки приложений JDBC в Spring. Ниже перечислены основные функциональные возможности расширения JDBC Extensions.

- **Поддержка QueryDSL.** QueryDSL (`www.querydsl.com`) — это характерный для предметной области язык, предоставляющий каркас для разработки типизированных запросов. В расширении JDBC Extensions из проекта Spring Data предоставляется класс `QueryDslJdbcTemplate`, упрощающий написание приложений JDBC благодаря применению языковых средств QueryDSL вместо операторов SQL.
- **Расширенная поддержка базы данных Oracle Database.** Расширение JDBC Extensions делает доступными многочисленные развитые средства для пользователей базы данных Oracle Database. Со стороны подключения к базе данных

---

[5] `o.s.j.c.JdbcTemplate - Выполнение подготовленного запроса SQL`
`o.s.j.c.JdbcTemplate - Выполнение подготовленного`
`        оператора SQL [select getfirstnamebyid?]`
`o.s.j.d.DataSourceUtils - Извлечение соединения через`
`        интерфейс JDBC из источника данных`
`o.s.j.d.DataSourceUtils - Возврат соединения через`
`        интерфейс JDBC из источника данных`
`Возвращенное значение: Eric`

оно поддерживает параметры сеансов связи, характерные для базы данных Oracle, а также технологию Fast Connection Failover (Восстановление быстрого соединения после отказа) в работе с дополнением Oracle RAC базы данных Oracle Database. В этом расширении предоставляются также классы, которые интегрируются с расширенной поддержкой очередей в базе данных Oracle (Oracle Advanced Queueing). С точки зрения типов данных обеспечивается собственная поддержка типов XML, `STRUCT` и `ARRAY` и прочих средств базы данных Oracle.

Если вы занимаетесь разработкой приложений JDBC, используя каркас Spring и базу данных Oracle Database, вам определенно стоит обратить внимание на расширение JDBC Extensions.

Соображения по поводу применения JDBC

Благодаря столь богатому набору функциональных средств каркас Spring позволяет существенно упростить разработку, когда для взаимодействия с исходной СУРБД применяется интерфейс JDBC. Но по-прежнему приходится писать довольно много кода, особенно для преобразования результирующих наборов в соответствующие объекты предметной области.

На основе интерфейса JDBC создано немало библиотек с открытым кодом, призванных устранить разрыв между структурой реляционных данных и объектно-ориентированной моделью языка Java. Например, iBATIS является распространенным каркасом для преобразования данных, также основанным на преобразовании в SQL. Каркас iBatis дает возможность преобразовывать объекты с хранимыми процедурами или запросами в XML-файл дескриптора. Аналогично Spring, в iBATIS поддерживается декларативный способ преобразования объектов из запросов, значительно сокращающий время на сопровождение запросов SQL, которые могут быть разбросаны по различным классам DAO.

Имеется также немало других каркасов объектно-реляционного преобразования, ориентированных на объектную модель, а не на запрос. К числу распространенных каркасов данной категории относятся Hibernate, EclipseLink (иначе называемый TopLink) и OpenJPA. Все они соответствуют спецификации JPA, составленной в организации JCP.

За последние годы инструментальные средства и каркасы объектно-реляционного преобразования стали намного более зрелыми, поэтому большинство разработчиков будут пользоваться одним из таких инструментальных средств, а не напрямую через интерфейс JDBC. Но в тех случаях, когда для обеспечения максимальной производительности требуется полный контроль над запросом, отправляемым в базу данных (например, в том случае, когда делается иерархический запрос к базе данных Oracle), интерфейс Spring JDBC оказывается по-настоящему жизнеспособным вариантом. Кроме того, применение Spring дает заметное преимущество в том, что можно сочетать разные технологии доступа к данным. Например, Hibernate можно применять в

качестве главного инструментального средства объектно-реляционного преобразования, а JDBC — как дополнение к логике сложных запросов или пакетных операций. Обе эти технологии допускается сочетать в пределах одной операции в предметной области, а затем заключать их в оболочку одной и той же транзакции в базе данных. Каркас Spring помогает легко справляться с подобного рода ситуациями.

Стартовая библиотека Spring Boot для JDBC

Ранее уже был представлен проект Spring Boot и одноименный модуль для разработки простых консольных и веб-приложений, и поэтому было бы вполне логично рассмотреть здесь стартовую библиотеку Spring Boot для интерфейса JDBC. Это поможет избавиться от стереотипных конфигураций и перейти непосредственно к реализации.

Когда стартовая библиотека `spring-boot-starter-jdbc` внедряется в проект как зависимость, вместе с ней в проект вводится ряд дополнительных библиотек, но не драйвер базы данных, поскольку решение об его установке разработчик должен принимать самостоятельно. В качестве примера в этом разделе рассматривается проект `spring-boot-jdbc` как подчиненный модуль из проекта `chapter06`. Его зависимости от Gradle и версии указаны в файле конфигурации `build.gradle` родительского проекта следующим образом:

```
ext {
    h2Version = '1.4.194'
    bootVersion = '2.0.0.BUILD-SNAPSHOT'
    boot = [
        springBootPlugin: "org.springframework.boot:
                  spring-boot-gradle-plugin:$bootVersion",
        Starter: "org.springframework.boot:
                  spring-boot-starter:$bootVersion",
        starterWeb: "org.springframework.boot:
                    spring-boot-starter-web:$bootVersion",
         Actuator: "org.springframework.boot:
                 spring-boot-starter-actuator:$bootVersion",
        starterTest: "org.springframework.boot:
                    spring-boot-starter-test:$bootVersion",
        starterAop: "org.springframework.boot:
                    spring-boot-starter-aop:$bootVersion",
        starterJdbc: "org.springframework.boot:
                    spring-boot-starter-jdbc:$bootVersion"
    ]
    db = [
        h2 : "com.h2database:h2:$h2Version",
        ...
    ]
    ...
}
```

Они объявляются как зависимости в файле конфигурации `spring-boot-jdbc\build.gradle`, но только по именам их свойств, как показано ниже.

```
buildscript {
  repositories {
    mavenLocal() mavenCentral()
    maven { url "http://repo.spring.io/release" }
    maven { url "http://repo.spring.io/milestone" }
    maven { url "http://repo.spring.io/snapshot" }
    maven { url "https://repo.spring.io/libs-snapshot" }
  }
  dependencies {
    classpath boot.springBootPlugin
  }
}

apply plugin: 'org.springframework.boot'

dependencies {
  compile project(':chapter06:plain-jdbc')
  compile boot.starterJdbc, db.h2
}
```

Автоматически конфигурируемые библиотеки приведены на рис. 6.3 в представлении Gradle Projects, доступном в IDE IntelliJ IDEA. В частности, библиотека `tomcat-jdbc` применяется в библиотеке `spring-boot-starter-jdbc` для конфигурирования компонента Spring Bean типа `DataSource`. Так, если отсутствует сконфигурированный вручную компонент Spring Bean типа `DataSource`, а в пути к классам присутствует драйвер базы данных, модуль Spring Boot автоматически зарегистрирует компонент Spring Bean типа `DataSource`, используя доступные в оперативной памяти параметры настройки базы данных. Кроме того, модуль Spring Boot автоматически зарегистрирует следующие компоненты Spring Beans.

- Компонент Spring Bean типа `JdbcTemplate`.
- Компонент Spring Bean типа `NamedParameterJdbcTemplate`.
- Компонент Spring Bean типа `PlatformTransactionManager` (`DataSource TransactionManager`).

Ниже описаны вкратце другие любопытные возможности, сокращающие объем работы по установке рабочей среды.

- Модуль Spring Boot осуществляет поиск файлов инициализации встроенной базы данных в каталоге `src/main/resources`. В нем предполагается обнаружить файл `schema.sql`, содержащий операторы языка DDL в SQL (например, операторы `CREATE TABLE`), а также файл `data.sql`, содержащий операторы языка DML (например, операторы `INSERT`). Этот файл используется для инициализации базы данных на стадии начальной загрузки.

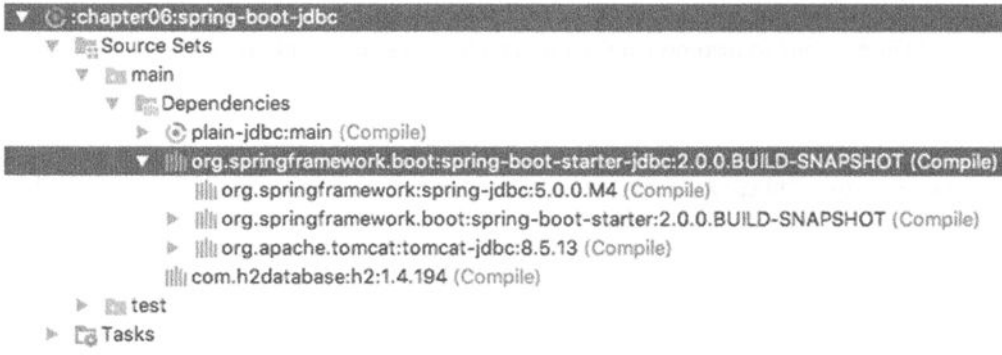


Рис. 6.3. Зависимости от стартовой библиотеки Spring Boot JDBC

- Местоположение и имена этих файлов могут быть сконфигурированы в файлах свойств `application.properties`, также находящихся в каталоге `src/main/resources`. Ниже в качестве примера приведено содержимое файла конфигурации, позволяющего использовать файлы запросов SQL в приложении Spring Boot.

```
spring.datasource.schema=db/schema.sql
spring.datasource.data=db/test-data.sql
```

- По умолчанию модуль Spring Boot инициализирует базу данных на стадии начальной загрузки, но и это положение можно изменить, введя свойство `spring.datasource.initialize=false` в файл свойств `application.properties`.

Помимо всего упомянутого выше, в модуле Spring Boot остается лишь предоставить ряд классов или сущностей предметной области и компонент DAO. Так, компонент Spring Bean типа `Singer` остается таким же, как и в остальных примерах из этой главы, а его реализация находится в проекте `chapter06/plain-jdbc`, внедряемом повсюду как зависимость. Ниже приведен исходный код применяемого здесь класса `JdbcSingerDao`.

```
package com.apress.prospring5.ch6;

import com.apress.prospring5.ch6.dao.SingerDao;
import com.apress.prospring5.ch6.entities.Singer;
import org.apache.commons.lang3.NotImplementedException;
import org.springframework.beans
          .factory.BeanCreationException;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;
import java.util.List;
```

```
@Component
public class JdbcSingerDao
       implements SingerDao, InitializingBean {
  private JdbcTemplate jdbcTemplate;

  @Autowired
  public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
  }

  @Override public String findNameById(Long id) {
    return jdbcTemplate.queryForObject(
           "SELECT first_name || ' ' || last_name "
         + "FROM singer WHERE id = ?",
           new Object{id}, String.class);
  }
  ...
}
```

В качестве примера ниже приведен класс, служащий входной точкой в модуль Spring Boot и снабженный аннотацией `@SpringBootApplication`. Обратите внимание, насколько он прост.

```
package com.apress.prospring5.ch6;

import com.apress.prospring5.ch6.dao.SingerDao;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot
          .autoconfigure.SpringBootApplication;
import org.springframework.context
          .ConfigurableApplicationContext;

@SpringBootApplication
public class Application {

  private static Logger logger =
          LoggerFactory.getLogger(Application.class);

  public static void main(String... args)
         throws Exception {
    ConfigurableApplicationContext ctx =
           SpringApplication.run(Application.class, args);
    assert (ctx != null);

    SingerDao singerDao = ctx.getBean(SingerDao.class);
    String singerName = singerDao.findNameById(1L);
    logger.info("Retrieved singer: " + singerName);
```

```
        System.in.read();
        ctx.close();
    }
}
```

Резюме

В этой главе было показано, как пользоваться каркасом Spring с целью упростить программирование доступа к данным через интерфейс JDBC. В ней было показано, как подключаться к базе данных и выполнять операции выборки, обновления, удаления и вставки, а также обращаться к хранимым функциям в базе данных. Кроме того, в этой главе подробно обсуждался базовый для модуля Spring JDBC класс `JdbcTemplate`, а также рассмотрены другие классы Spring, построенные на основе класса `JdbcTemplate` и помогающие моделировать различные операции в JDBC. А там, где это оказалось уместным, в этой главе было продемонстрировано применение лямбда-выражений, внедренных в версии Java 8. И в завершение этой главы была рассмотрена стартовая библиотека Spring Boot для интерфейса JDBC, поскольку она так или иначе помогает уделить больше внимания реализации бизнес-логики приложения и меньше внимания конфигурированию, а следовательно, разработчикам приложений следует непременно знать об этом замечательном инструментальном средстве. В двух последующих главах речь пойдет о применении Spring вместе с распространенными технологиями объектно-реляционного преобразования при разработке логики доступа к данным.

ГЛАВА 7

Применение Hibernate в Spring

В предыдущей главе было показано, как пользоваться интерфейсом JDBC в приложениях Spring. И хотя каркас Spring позволяет значительно упростить разработку приложений JDBC, приходится по-прежнему писать немало прикладного кода. В этой главе мы рассмотрим одну из библиотек объектно-реляционного преобразования (ORM) под названием Hibernate.

Если у вас имеется опыт разработки приложений с доступом к данным через компоненты сущностей EJB (до версии EJB 3.0), то вы, вероятно, помните, насколько мучительным оказывается этот процесс. Утомительное конфигурирование преобразований, установление границ транзакций и употребление в каждом компоненте большого объема стереотипного кода, предназначенного для управления его жизненным циклом, значительно снижало производительность труда при разработке корпоративных приложений на Java.

Каркас Spring был создан для того, чтобы охватить весь процесс разработки на основе простых объектов POJO и декларативного конфигурирования вместо трудной и нескладной настройки компонентов EJB, и поэтому разработчики стали осознавать, что более простой, облегченный и основанный на объектах POJO каркас может упростить разработку логики доступа к данным. С тех пор появилось немало библиотек, которые совместно называются *библиотеками ORM*. Главное назначение библиотеки ORM — устранить разрыв между структурой реляционных данных в СУРБД и объектно-ориентированной моделью в Java, чтобы разработчики могли сосредоточиться на программировании с помощью объектной модели и в то же время легко выполнять действия, связанные с сохраняемостью данных.

Среди всех библиотек ORM, доступных в сообществе разработчиков открытого кода, Hibernate является одной из самых удачных. Ее функциональные возможности, включая подход, основанный на простых объектах POJO, простоту разработки и поддержку определений сложных отношений, завоевали сердца многих участников сообщества ведущих направлений разработок на Java.

Распространенность Hibernate повлияла также на организацию JCP, в которой была разработана спецификация объектов данных Java (Java Data Object — JDO) в качестве одной из стандартных технологий ORM на платформе Java EE. В версии EJB 3.0 компонент сущности EJB был даже заменен прикладным интерфейсом Java Persistence API (JPA), на многие принципы организации которого оказали влияние такие распространенные библиотеки ORM, как Hibernate, TopLink и JDO.

Между Hibernate и JPA также сложились очень тесные отношения. Гэвин Кинг (Gavin King), основатель Hibernate, представил сервер приложений JBoss, будучи одним из членов экспертной группы JCP по определению спецификации JPA. Начиная с версии 3.2 в Hibernate предоставляется реализация прикладного интерфейса JPA. Это означает, что при разработке приложений с помощью библиотеки Hibernate в качестве поставщика услуг сохраняемости данных можно выбирать между собственным прикладным интерфейсом API библиотеки Hibernate и прикладным JPA API с поддержкой Hibernate.

После краткой истории развития Hibernate в этой главе будет показано, как применять Spring вместе с Hibernate при разработке логики доступа к данным. Hibernate является настолько обширной библиотекой ORM, что раскрыть все ее особенности в одной главе просто невозможно. Этой теме посвящены целые книги. Поэтому здесь рассматриваются основные принципы и примеры применения библиотеки Hibernate в Spring. В этой главе будут, в частности, рассмотрены следующие вопросы.

- **Конфигурирование фабрики сеансов Hibernate.** Основной принцип действия библиотеки Hibernate опирается на интерфейс `Session`, которым управляет компонент `SessionFactory`, представляющий фабрику сеансов в Hibernate. В этой части будет показано, как конфигурировать фабрику сеансов Hibernate для работы в приложении Spring.
- **Основные принципы объектно-реляционного преобразования средствами Hibernate.** В этой части рассмотрены основные принципы преобразования простых объектов POJO в структуру исходной реляционной базы данных с помощью библиотеки Hibernate. Здесь будут также описаны некоторые из наиболее употребительных отношений, в том числе "один ко многим" и "многие ко многим".
- **Операции над данными.** В этой части представлены примеры выполнения операций над данными (запроса, вставки, обновления, удаления) с помощью библиотеки Hibernate в среде Spring. В работе с библиотекой Hibernate взаимодействовать придется главным образом с ее основным интерфейсом `Session`.

На заметку При определении объектно-реляционных преобразований в библиотеке Hibernate поддерживаются два стиля конфигурирования. Один из них предусматривает размещение информации о преобразовании в XML-файлах, а другой — применение аннотаций Java в классах сущностей. В области ORM или JPA класс Java, объект которого преобразуется в структуру исходной реляционной базы данных, называется *классом сущности* (entity class). В этой

главе основное внимание будет уделено применению аннотаций для объектно-реляционного преобразования. Аннотирование преобразований будет опираться на стандарты JPA (например, в пакете **`javax.persistence`**), поскольку они совместимы с собственными аннотациями библиотеки Hibernate, что упростит впоследствии переход в среду JPA.

Модель выборочных данных для исходного кода примеров

Модель данных, применяемая в примерах из этой главы, приведена на рис. 7.1.

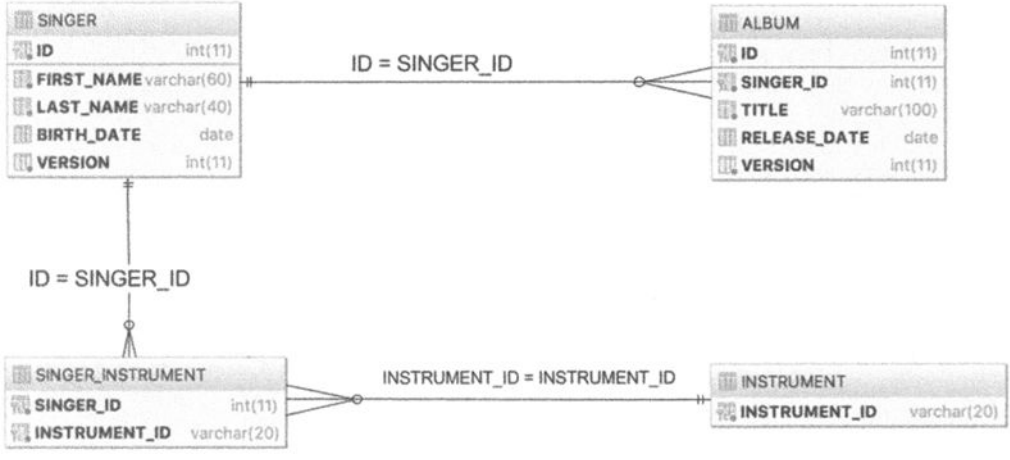


Рис. 7.1. Модель выборочных данных

Как следует из рис. 7.1, эта модель данных дополнена двумя таблицами: `INSTRUMENT` и `SINGER_INSTRUMENT`. В частности, таблица `SINGER_INSTRUMENT` служит для соединения и моделирует отношение "многие ко многим" между таблицами `SINGER` и `INSTRUMENT`. Кроме того, в таблицы `SINGER` и `ALBUM` введен столбец `VERSION` для оптимистической блокировки, подробно обсуждаемой далее в этой главе. В примерах из этой главы будет использоваться встроенная база данных H2, поэтому указывать имя базы данных не нужно. Ниже приведены сценарии для создания таблиц, применяемых в примерах из этой главы.

```
CREATE TABLE SINGER (
  ID INT NOT NULL AUTO_INCREMENT
  , FIRST_NAME VARCHAR(60) NOT NULL
  , LAST_NAME VARCHAR(40) NOT NULL
  , BIRTH_DATE DATE
  , VERSION INT NOT NULL DEFAULT 0
  , UNIQUE UQ_SINGER_1 (FIRST_NAME, LAST_NAME)
  , PRIMARY KEY (ID)
);
```

```
CREATE TABLE ALBUM (
  ID INT NOT NULL AUTO_INCREMENT
  , SINGER_ID INT NOT NULL
  , TITLE VARCHAR(100) NOT NULL
  , RELEASE_DATE DATE
  , VERSION INT NOT NULL DEFAULT 0
  , UNIQUE UQ_SINGER_ALBUM_1 (SINGER_ID, TITLE)
  , PRIMARY KEY (ID)
  , CONSTRAINT FK_ALBUM_SINGER FOREIGN KEY (SINGER_ID)
  REFERENCES SINGER (ID)
);

CREATE TABLE INSTRUMENT (
  INSTRUMENT_ID VARCHAR(20) NOT NULL
  , PRIMARY KEY (INSTRUMENT_ID)
);

CREATE TABLE SINGER_INSTRUMENT (
  SINGER_ID INT NOT NULL
  , INSTRUMENT_ID VARCHAR(20) NOT NULL
  , PRIMARY KEY (SINGER_ID, INSTRUMENT_ID)
  , CONSTRAINT FK_SINGER_INSTRUMENT_1
    FOREIGN KEY (SINGER_ID)
  REFERENCES SINGER (ID) ON DELETE CASCADE
  , CONSTRAINT FK_SINGER_INSTRUMENT_2
    FOREIGN KEY (INSTRUMENT_ID)
  REFERENCES INSTRUMENT (INSTRUMENT_ID)
);
```

Следующий сценарий SQL служит для заполнения таблиц выборочными данными:

```
insert into singer (first_name, last_name, birth_date)
  values ('John', 'Mayer', '1977-10-16');
insert into singer (first_name, last_name, birth_date)
  values ('Eric', 'Clapton', '1945-03-30');
insert into singer (first_name, last_name, birth_date)
  values ('John', 'Butler', '1975-04-01');

insert into album (singer_id, title, release_date)
  values (1, 'The Search For Everything', '2017-01-20');
insert into album (singer_id, title, release_date)
  values (1, 'Battle Studies', '2009-11-17');
insert into album (singer_id, title, release_date)
  values (2, 'From The Cradle ', '1994-09-13');

insert into instrument (instrument_id) values ('Guitar');
insert into instrument (instrument_id) values ('Piano');
insert into instrument (instrument_id) values ('Voice');
insert into instrument (instrument_id) values ('Drums');
insert into instrument (instrument_id)
```

```
  values ('Synthesizer');
insert into singer_instrument(singer_id, instrument_id)
  values (1, 'Guitar');
insert into singer_instrument(singer_id, instrument_id)
  values (1, 'Piano');
insert into singer_instrument(singer_id, instrument_id)
  values (2, 'Guitar');
```

Конфигурирование фабрики сеансов Hibernate

Как упоминалось ранее, основной принцип действия библиотеки Hibernate опирается на интерфейс `Session`, который получается из компонента `SessionFactory`, реализующего фабрику сеансов Hibernate. В каркасе Spring предоставляются классы, поддерживающие конфигурирование фабрики сеансов Hibernate в виде компонента Spring Bean с требующимися свойствами. Чтобы воспользоваться библиотекой Hibernate, необходимо внедрить ее как зависимость в свой проект. Ниже приведена конфигурация Gradle для сборки проектов, рассмотренных в этой главе.

```
// Файл конфигурации pro-spring-15/build.gradle
ext {
  hibernateVersion = '5.2.10.Final'
  ...
  hibernate = [
    validator: "org.hibernate:hibernate-validator:
                5.1.3.Final",
    ehcache : "org.hibernate:hibernate-ehcache:
               $hibernateVersion",
    [ em] : "org.hibernate:hibernate-entitymanager:
            $hibernateVersion"
  ]
  ...
}

// Файл конфигурации chapter07.gradle
dependencies {
  // Эти зависимости указываются для всех подчиненных
  // модулей, кроме модуля начальной загрузки, который
  // определяется отдельно
  if !project.name.contains"boot" {
    compile spring.contextSupport, spring.orm,
    misc.slf4jJcl, misc.logback, db.h2, misc.lang3,
    [hibernate.em]
  }
  testCompile testing.junit
}
```

В следующей конфигурации приведены элементы разметки в формате XML, требующиеся для конфигурирования примера приложения из этой главы:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context=
            "http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:jdbc=
            "http://www.springframework.org/schema/jdbc"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="
          http://www.springframework.org/schema/jdbc
          http://www.springframework.org/schema/jdbc
                /spring-jdbc.xsd
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans
                /spring-beans.xsd
          http://www.springframework.org/schema/tx
          http://www.springframework.org/schema/tx
                /spring-tx.xsd
          http://www.springframework.org/schema/util
          http://www.springframework.org/schema/util
                /spring-util.xsd
          http://www.springframework.org/schema/context
          http://www.springframework.org/schema/context
                /spring-context.xsd">

    <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:sql/schema.sql"/>
      <jdbc:script location="classpath:sql/test-data.sql"/>
    </jdbc:embedded-database>

    <bean id="transactionManager"
      class="org.springframework.orm.hibernate5
                .HibernateTransactionManager"
      p:sessionFactory-ref="sessionFactory"/>

    <tx:annotation-driven/>
    <context:component-scan base-package=
            "com.apress.prospring5.ch7"/>

  <bean id="sessionFactory"
    class="org.springframework.orm.hibernate5
              .LocalSessionFactoryBean"
    p:dataSource-ref="dataSource"
    p:packagesToScan="com.apress.prospring5.ch7.entities"
```

```
    p:hibernateProperties-ref="hibernateProperties"/>

    <util:properties id="hibernateProperties">
      <prop key="hibernate.dialect">org.hibernate
                        .dialect.H2Dialect</prop>
      <prop key="hibernate.max_fetch_depth">3</prop>
      <prop key="hibernate.jdbc.fetch_size">50</prop>
      <prop key="hibernate.jdbc.batch_size">10</prop>
      <prop key="hibernate.hbm2ddl.auto">create-drop</prop>
      <prop key="hibernate.format_sql">true</prop>
      <prop key="hibernate.use_sql_comments">true</prop>
    </util:properties>
</beans>
```

Ниже приведена равнозначная конфигурация, а компоненты этих двух конфигураций поясняются параллельно после фрагмента кода данной конфигурации.

```
package com.apress.prospring5.ch7.config;

import com.apress.prospring5.ch6.CleanUp;
import org.hibernate.SessionFactory;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseType;
import org.springframework.orm.hibernate5
          .HibernateTransactionManager;
import org.springframework.orm.hibernate5
          .LocalSessionFactoryBean;
import org.springframework.transaction
          .PlatformTransactionManager;
import org.springframework.transaction.annotation
          .EnableTransactionManagement;
import javax.sql.DataSource;
import java.io.IOException;
import java.util.Properties;

@Configuration
@ComponentScan(basePackages = "com.apress.prospring5.ch7")
@EnableTransactionManagement
public class AppConfig {

  private static Logger logger =
          LoggerFactory.getLogger(AppConfig.class);
```

```
  @Bean
  public DataSource dataSource() {
    try {
      EmbeddedDatabaseBuilder dbBuilder =
              new EmbeddedDatabaseBuilder();
      return dbBuilder.setType(EmbeddedDatabaseType.H2)
             .addScripts("classpath:sql/schema.sql",
             "classpath:sql/test-data.sql").build();
    } catch (Exception e) {
      logger.error("Embedded DataSource bean cannot "
                + "be created!", e);
      return null;
    }
  }

  private Properties hibernateProperties() {
    Properties hibernateProp = new Properties();
    hibernateProp.put("hibernate.dialect",
                      "org.hibernate.dialect.H2Dialect");
    hibernateProp.put("hibernate.format_sql", true);
    hibernateProp.put("hibernate.use_sql_comments", true);
    hibernateProp.put("hibernate.show_sql", true);
    hibernateProp.put("hibernate.max_fetch_depth", 3);
    hibernateProp.put("hibernate.jdbc.batch_size", 10);
    hibernateProp.put("hibernate.jdbc.fetch_size", 50);
    return hibernateProp;
  }

  @Bean public SessionFactory sessionFactory()
              throws IOException {
    LocalSessionFactoryBean sessionFactoryBean =
                new LocalSessionFactoryBean();
    sessionFactoryBean.setDataSource(dataSource());
    sessionFactoryBean.setPackagesToScan(
           "com.apress.prospring5.ch7.entities");
    sessionFactoryBean.setHibernateProperties(
           hibernateProperties());
    sessionFactoryBean.afterPropertiesSet();
    return sessionFactoryBean.getObject();
  }

  @Bean public PlatformTransactionManager
        transactionManager() throws IOException {
    return new HibernateTransactionManager(
              sessionFactory());
  }
}
```

В приведенных выше конфигурациях объявлено несколько компонентов Spring Beans для поддержки фабрики сеансов Hibernate. Ниже перечислены основные элементы конфигурации.

- **Компонент `dataSource`.** В данном случае применяется встроенная база данных H2, объявленная, как пояснялось в главе 6.
- **Компонент `transactionManager`.** Для транзакционного доступа к данным в фабрике сеансов Hibernate требуется диспетчер транзакций. В каркасе Spring предоставляется диспетчер транзакций специально для версии Hibernate 5 (`org.springframework.orm.hibernate5.HibernateTransactionManager`). Этот компонент объявлен с присвоенным ему идентификатором `transactionManager`. Всякий раз, когда потребуется диспетчер транзакций, каркас Spring будет по умолчанию искать в контексте типа `ApplicationContext` компонент `transactionManager`. Транзакции подробно рассматриваются в главе 9. Кроме того, в дескрипторе `<tx:annotation-driven>` объявляется поддержка требований к установлению границ транзакций с помощью аннотации. Эквивалентное конфигурирование на языке Java осуществляется с помощью аннотации `@EnableTransactionManagement`.
- **Просмотр компонентов.** Дескриптор `<context:component-scan>` и аннотация `@ComponentScan` уже не раз упоминались в этой книге. В данном случае они предписывают каркасу Spring просмотреть компоненты в пакете `com.apress.prospring5.ch7`, чтобы обнаружить в нем те компоненты Spring Beans, которые снабжены аннотацией `@Repository`.
- **Компонент `sessionFactory`.** Этот компонент Spring Bean является самой важной частью конфигурации, поскольку в нем определено немало свойств. Во-первых, необходимо внедрить компонент источника данных в фабрику сеансов. Во-вторых, библиотеке Hibernate предписывается просмотреть объекты предметной области в пакете `com.apress.prospring5.ch.entities`. И, в-третьих, свойство `hibernateProperties` предоставляет подробности конфигурирования Hibernate. Имеется немало конфигурационных параметров, но в данном случае определяются лишь несколько важных свойств, которые должны предоставляться для каждого приложения. В табл. 7.1 перечислены главные конфигурационные параметры и соответствующие свойства для фабрики сеансов Hibernate.

Полный перечень свойств, поддерживаемых в библиотеке Hibernate, можно найти в руководстве пользователя ORM в Hibernate и, в частности, в разделе 23, доступном по адресу `https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html#configurations`.

Таблица 7.1. Свойства Hibernate

Свойство	Описание
`hibernate.dialect`	Обозначает диалект базы данных для обработки запросов, который должен использоваться в Hibernate. В библиотеке Hibernate поддерживаются диалекты SQL для многих баз данных. Эти диалекты являются подклассами, производными от класса `org.hibernate.dialect.Dialect`. К числу основных диалектов Hibernate относятся `H2Dialect`, `Oracle10gDialect`, `PostgreSQLDialect`, `MySQLDialect`, `SQLServerDialect` и т.д.
`hibernate.max_fetch_depth`	Объявляет "глубину" для внешних соединений, когда одни преобразующие объекты связаны с другими преобразуемыми объектами. Позволяет предотвратить выборку средствами Hibernate слишком большого объема данных при наличии многих вложенных ассоциаций. Обычно принимает значение 3
`hibernate.jdbc.fetch_size`	Обозначает количество записей из базового результирующего набора JDBC типа `ResultSet`, который должен использоваться в Hibernate для извлечения записей из базы данных в каждой выборке. Допустим, что по запросу, направленному базе данных, получен результирующий набор из 500 записей. Если размер выборки равен 50, то для получения всех требующихся данных библиотеке Hibernate придется произвести 10 выборок
`hibernate.jdbc.batch_size`	Указывает библиотеке Hibernate количество операций обновления, которые должны быть сгруппированы в пакет. Это очень удобно для выполнения пакетных заданий в Hibernate. Очевидно, что когда выполняется пакетное задание, обновляющее сотни записей, было бы желательно, чтобы библиотека Hibernate сгруппировала запросы в пакеты, а не отправляла запросы на обновление по одному
`hibernate.show_sql`	Указывает, должна ли библиотека Hibernate направлять запросы SQL в файл регистрации или на консоль. Этот режим имеет смысл включать в среде разработки, потому что он оказывает помощь в тестировании и устранении ошибок
`hibernate.format_sql`	Указывает, следует ли форматировать вывод запросов SQL на консоль или в файл регистрации
`hibernate.use_sql_comments`	Если в этом свойстве установлено логическое значение `true`, библиотека Hibernate сформирует комментарии в запросах SQL, чтобы упростить их отладку

Объектно-реляционное преобразование с помощью аннотаций Hibernate

Имея в своем распоряжении приведенную ранее конфигурацию, можно далее смоделировать классы Java для сущностей POJO и их преобразования в исходную структуру реляционных данных. Преобразование можно реализовать двумя способа-

ми. Первым способом сначала проектируется объектная модель, а затем на ее основе формируются сценарии для базы данных. Например, для конфигурирования фабрики сеансов библиотеке Hibernate можно передать свойство `hibernate.hbm2ddl.auto`, чтобы автоматически экспортировать схему DDL в базу данных. Второй способ состоит в том, чтобы начать с модели данных, а затем построить простые объекты POJO для требующегося преобразования. Мы отдаем предпочтение второму способу, поскольку он обеспечивает больший контроль над моделью данных, что очень удобно для оптимизации производительности при доступе к данным. Тем не менее первый способ будет рассмотрен далее в этой главе, чтобы продемонстрировать другой подход к конфигурированию приложения Spring вместе с библиотекой Hibernate. На основании упомянутой ранее модели выборочных данных можно построить объектно-ориентированную модель с диаграммой классов, приведенной на рис. 7.2. Как видите, между объектами типа `Singer` и `Album` имеется отношение "один ко многим", тогда как между объектами типа `Singer` и `Instrument` — отношение "многие ко многим".

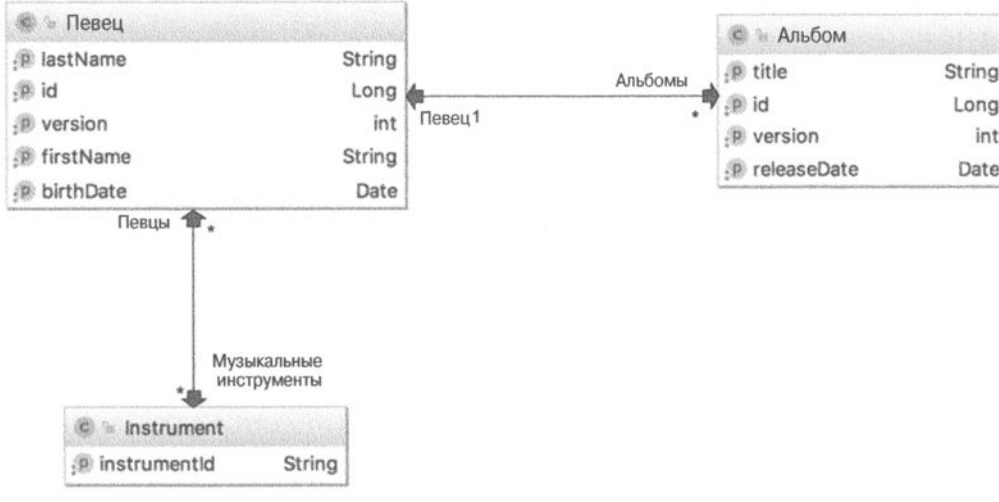


Рис. 7.2. Диаграмма классов для модели выборочных данных

Простые преобразования

Начнем с преобразования простых атрибутов класса. Ниже приведен исходный код класса `Singer` с аннотациями преобразования.

```
package com.apress.prospring5.ch7.entities;

import javax.persistence.*;
import java.io.Serializable;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
```

```
import static javax.persistence.GenerationType.IDENTITY;

@Entity
@Table(name = "singer")
public class Singer implements Serializable {

  private Long id;
  private String firstName;
  private String lastName;
  private Date birthDate;
  private int version;

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

  @Id
  @GeneratedValue(strategy = IDENTITY)
  @Column(name = "ID")
  public Long getId() {
    return this.id;
  }

  @Version
  @Column(name = "VERSION")
  public int getVersion() {
    return version;
  }

  @Column(name = "FIRST_NAME")
  public String getFirstName() {
    return this.firstName;
  }

  @Column(name = "LAST_NAME")
  public String getLastName() {
    return this.lastName;
  }

  @Temporal(TemporalType.DATE)
  @Column(name = "BIRTH_DATE")
  public Date getBirthDate() {
    return birthDate;
  }

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

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

  public void setBirthDate(Date birthDate) {
    this.birthDate = birthDate;
  }

  public void setVersion(int version) {
    this.version = version;
  }

  public String toString() {
    return "Singer - Id: " + id + ", First name: "
         + firstName + ", Last name: " + lastName
         + ", Birthday: " + birthDate;
  }
}
```

Сначала в приведенном выше коде тип данных снабжается аннотацией `@Entity`, где указывается на то, что это преобразуемый класс сущности. Аннотация `@Table` определяет имя таблицы в базе данных, в которую преобразуется эта сущность. Каждый преобразуемый атрибут снабжается аннотацией `@Column` с указанием имени столбца.

На заметку Имена таблиц и столбцов могут быть опущены, если имена типов данных и атрибутов совпадают с именами таблиц и столбцов.

В отношении рассматриваемых здесь преобразований необходимо отметить следующее.

- Атрибут `birthDate` снабжается аннотацией `@Temporal` со значением `TemporalType.DATE` в качестве аргумента. Это означает, что тип данных Java (`java.util.Date`) желательно преобразовать в тип данных SQL (`java.sql.Date`). Это дает возможность получить в приложении доступ к свойству `birthDate` из объекта типа `Singer`, используя, как обычно, тип данных `java.util.Date`.
- Атрибут `id` снабжается аннотацией `@Id`. Это означает, что он является первичным ключом объекта. В библиотеке Hibernate он будет применяться как однозначный идентификатор для управлении экземплярами сущностей певцов в пределах сеанса. Кроме того, аннотация `@GeneratedValue` сообщает библиотеке Hibernate, каким образом было сгенерировано значение идентификатора `id`. Значение `IDENTITY` атрибута `strategy` в этой аннотации означает, что идентификатор сгенерирован СУРБД во время вставки данных.

- Атрибут `version` снабжен аннотацией `@Version`. Тем самым библиотеке Hibernate сообщается, что в данном случае требуется применить механизм оптимистичной блокировки, а для его управления — атрибут `version`. Всякий раз, когда библиотека Hibernate обновляет запись, она сравнивает версию экземпляра сущности с версией записи в базе данных. Если версии совпадают, значит, данные раньше не обновлялись, и поэтому Hibernate обновит данные и увеличит значение в столбце версии. Но если версии отличаются, то это означает, что кто-то уже обновил запись, и тогда Hibernate сгенерирует исключение типа `StaleObjectStateException`, которое Spring преобразует в исключение типа `HibernateOptimisticLockingFailureException`. В данном примере для контроля версий используется целочисленное значение. Помимо целых чисел, в Hibernate поддерживаются отметки времени. Тем не менее для контроля версий рекомендуется применять именно целочисленное значение, поскольку в этом случае Hibernate будет всегда увеличивать номер версии на 1 после каждого обновления. Когда же используется отметка времени, после каждого обновления Hibernate будет заменять значение этой отметки текущим показанием времени. Отметки времени менее безопасны, поскольку две параллельные транзакции могут загрузить и обновить один и тот же элемент данных практически одновременно в пределах миллисекунды.

Еще один преобразуемый объект представлен ниже в виде класса `Album`.

```
package com.apress.prospring5.ch7.entities;

import javax.persistence.*;
import java.io.Serializable;
import java.util.Date;
import static javax.persistence.GenerationType.IDENTITY;

@Entity
@Table(name = "album")
public class Album implements Serializable {

  private Long id;
  private String title;
  private Date releaseDate;
  private int version;

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

  @Id
  @GeneratedValue(strategy = IDENTITY)
  @Column(name = "ID")
  public Long getId() {
    return this.id;
```

```
    }

    @Version
    @Column(name = "VERSION")
    public int getVersion() {
        return version;
    }

    @Column
    public String getTitle() {
        return this.title;
    }

    @Temporal(TemporalType.DATE)
    @Column(name = "RELEASE_DATE")
    public Date getReleaseDate() {
        return this.releaseDate;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public void setReleaseDate(Date releaseDate) {
        this.releaseDate = releaseDate;
    }

    public void setVersion(int version) {
        this.version = version;
    }

    @Override
    public String toString() {
        return "Album - Id: " + id + ", Title: "
                + title + ", Release Date: " + releaseDate;
    }
}
```

А вот и третий класс сущности, применяемый в примерах из этой главы:

```
package com.apress.prospring5.ch7.entities;

import javax.persistence.*;
import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;

@Entity
@Table(name = "instrument")
public class Instrument implements Serializable {
```

```
  private String instrumentId;

  @Id
  @Column(name = "INSTRUMENT_ID")
  public String getInstrumentId() {
    return this.instrumentId;
  }

  public void setInstrumentId(String instrumentId) {
    this.instrumentId = instrumentId;
  }

  @Override
  public String toString() {
    return "Instrument :" + getInstrumentId();
  }
}
```

Преобразование связей "один ко многим"

Библиотека Hibernate обладает способностью моделировать самые разные виды связей. Наиболее распространенными являются связи "один ко многим" и "многие ко многим". У каждого певца может быть нуль или больше альбомов, из чего получается связь "один ко многим" (в терминологии ORM связь "один ко многим" служит для моделирования отношений "нуль ко многим" и "один ко многим" в структуре данных). В следующем фрагменте кода представлены свойства и методы, требующиеся для определения конкретного отношения "один ко многим" между сущностями типа `Singer` и `Album`:

```
package com.apress.prospring5.ch7.entities;

import javax.persistence.*;
import java.io.Serializable;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import static javax.persistence.GenerationType.IDENTITY;

@Entity
@Table(name = "singer")
public class Singer implements Serializable {

  private Long id;
  private String firstName;
  private String lastName;
  private Date birthDate;
  private int version;
```

```
  private Set<Album> albums = new HashSet<>();
  ...
  @OneToMany(mappedBy = "singer", cascade=CascadeType.ALL,
             orphanRemoval=true)

  public Set<Album> getAlbums() {
    return albums;
  }

  public boolean addAbum(Album album) {
    album.setSinger(this);
    return getAlbums().add(album);
  }

  public void removeAlbum(Album album) {
    getAlbums().remove(album);
  }

  public void setAlbums(Set<Album> albums) {
    this.albums = albums;
  }
  ...
}
```

Метод получения значения из свойства `albums` снабжен аннотацией `@OneToMany`, которая указывает на наличие отношения "один ко многим" с классом `Album`. Этой аннотации передается несколько атрибутов. Так, в атрибуте `mappedBy` задается свойство `singer` из класса `Album`, обеспечивающее связь (по определению внешнего ключа из таблицы `FK_ALBUM_SINGER`). Атрибут `cascade` означает, что операция обновления должна распространяться "каскадом" на порожденные записи. И, наконец, атрибут `orphanRemoval` указывает, что после обновления сведений об альбомах записи, которые больше не существуют в наборе, должны быть удалены из базы данных. Ниже приведен исходный код класса `Album`, обновленный для преобразования связей.

```
package com.apress.prospring5.ch7.entities;

import javax.persistence.*;
import java.io.Serializable;
import java.util.Date;
import static javax.persistence.GenerationType.IDENTITY;

@Entity
@Table(name = "album")
public class Album implements Serializable {

  private Long id;
  private String title;
```

```
  private Date releaseDate;
  private int version;

  private Singer singer;

  @ManyToOne
  @JoinColumn(name = "SINGER_ID")
  public Singer getSinger() {
    return this.singer;
  }

  public void setSinger(Singer singer) {
    this.singer = singer;
  }
  ...
}
```

В приведенном выше коде метод получения свойства `singer` снабжен аннотацией `@ManyToOne`, где задается другая сторона связи с классом `Singer`. В этом коде была также указана аннотация `@JoinColumn` для столбца с именем внешнего ключа. И, наконец, здесь был переопределен метод `toString()` с целью упростить последующее тестирование кода из данного примера благодаря выводу полученного в этом методе результата на консоль.

Преобразование связей "многие ко многим"

Каждый певец может играть на нескольких музыкальных инструментах или ни на одном из них, а каждый инструмент может быть связан с несколькими певцами или же ни с одним из них, что, по существу, означает связь "многие ко многим". Для преобразования связей "многие ко многим" требуется промежуточная таблица, через которую осуществляется соединение. В данном случае это таблица `SINGER_INSTRUMENT`. Ниже приведен код, который требуется ввести в класс `Singer`, чтобы реализовать подобное отношение между таблицами базы данных.

```
package com.apress.prospring5.ch7.entities;

import javax.persistence.*;
import java.io.Serializable;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import static javax.persistence.GenerationType.IDENTITY;

@Entity
@Table(name = "singer")
public class Singer implements Serializable {

  private Long id;
```

```
  private String firstName;
  private String lastName;
  private Date birthDate;
  private int version;

  private Set<Instrument> instruments = new HashSet<>();

  @ManyToMany
  @JoinTable(name = "singer_instrument",
             joinColumns = @JoinColumn(name = "SINGER_ID"),
             inverseJoinColumns = @JoinColumn(
                                  name = "INSTRUMENT_ID"))
  public Set<Instrument> getInstruments() {
    return instruments;
  }

  public void setInstruments(Set<Instrument> instruments) {
    this.instruments = instruments;
  }
  ...
}
```

В приведенном выше коде метод получения свойства `instruments` из класса `Singer` снабжен аннотацией `@ManyToMany`. В данном коде предоставляется аннотация `@JoinTable` для указания промежуточной таблицы для соединения, которую должна искать библиотека Hibernate. В атрибуте `name` задается имя промежуточной таблицы для соединения, в атрибуте `joinColumns` определяется столбец с внешним ключом для таблицы `SINGER`, а в атрибуте `inverseJoinColumns` указывается столбец с внешним ключом для таблицы `INSTRUMENT` на другой стороне устанавливаемой связи. Ниже приведен исходный код класса `Instrument`, где дополнительно реализована другая сторона рассматриваемого здесь отношения. Данное преобразование связей напоминает приведенное ранее преобразование для класса `Singer`, но здесь атрибуты `joinColumns` и `inverseJoinColumns` поменялись местами, отражая отношение "многие ко многим".

```
package com.apress.prospring5.ch7.entities;

import javax.persistence.*;
import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;

@Entity
@Table(name = "instrument")
public class Instrument implements Serializable {
  private String instrumentId;
  private Set<Singer> singers = new HashSet<>();
```

```
  @Id
  @Column(name = "INSTRUMENT_ID")
  public String getInstrumentId() {
    return this.instrumentId;
  }

  @ManyToMany
  @JoinTable(name = "singer_instrument",
      joinColumns = @JoinColumn(name = "INSTRUMENT_ID"),
      inverseJoinColumns = @JoinColumn(name = "SINGER_ID"))
  public Set<Singer> getSingers() {
    return this.singers;
  }

  public void setSingers(Set<Singer> singers) {
    this.singers = singers;
  }

  public void setInstrumentId(String instrumentId) {
    this.instrumentId = instrumentId;
  }

  @Override
  public String toString() {
    return "Instrument :" + getInstrumentId();
  }
}
```

Интерфейс `Session` из библиотеки `Hibernate`

При взаимодействии с базой данных в библиотеке Hibernate приходится иметь дело с интерфейсом `Session`, который получается из фабрики сеансов, реализуемой в компоненте `SessionFactory`. Ниже приведен исходный код класса `SingerDao Impl`, который используется в примерах из этой главы и в который внедрена сконфигурированная фабрика сеансов Hibernate.

```
package com.apress.prospring5.ch7.dao;

import com.apress.prospring5.ch7.entities.Singer;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.hibernate.SessionFactory;
import org.springframework.stereotype.Repository;
import org.springframework.transaction
          .annotation.Transactional;
import javax.annotation.Resource;
import java.util.List;
```

```
@Transactional
@Repository("singerDao")
public class SingerDaoImpl implements SingerDao {

  private SessionFactory sessionFactory;

  public SessionFactory getSessionFactory() {
    return sessionFactory;
  }

  @Resource(name = "sessionFactory")
  public void setSessionFactory(
              SessionFactory sessionFactory) {
    this.sessionFactory = sessionFactory;
  }
  ...
}
```

Как обычно, в данном коде класс DAO объявляется как компонент Spring Bean с помощью аннотации `@Repository`. В аннотации `@Transactional` определяются требования к транзакциям, которые будут рассматриваться в главе 9. А с помощью атрибута `sessionFactory` в аннотации `@Resource` внедряется фабрика сеансов Hibernate. Ниже приведено определение интерфейса `SingerDao`.

```
package com.apress.prospring5.ch7.dao;

import com.apress.prospring5.ch7.entities.Singer;
import java.util.List;

public interface SingerDao {
  List<Singer> findAll();
  List<Singer> findAllWithAlbum();
  Singer findById(Long id);
  Singer save(Singer contact);
  void delete(Singer contact);
}
```

Приведенный выше интерфейс довольно прост и содержит лишь три метода поиска, один метод сохранения и еще один метод удаления. В частности, метод `save()` служит как для операций вставки, так и для операций обновления.

Выборка данных на языке запросов Hibernate

Библиотека Hibernate, как и другие инструментальные средства ORM вроде JDO и JPA, спроектирована на основе объектной модели. Это означает, что после определения всех преобразований составлять операторы SQL для взаимодействия с базой данных не придется. Вместо этого для составления запросов в Hibernate служит язык

запросов HQL (Hibernate Query Language). При взаимодействии с базой данных библиотека Hibernate преобразует запросы HQL в операторы SQL.

Синтаксис языка HQL очень похож на синтаксис SQL. Но при этом необходимо мыслить категориями объектов, а не базы данных. В последующих разделах будет приведено несколько тому примеров.

Простой запрос с отложенной выборкой

Итак, начнем с реализации метода `findAll()`, где из базы данных просто извлекаются все записи о певцах. Ниже приведен исходный код из предыдущего примера, видоизмененный с целью внедрить эти функциональные возможности.

```
package com.apress.prospring5.ch7.dao;

import com.apress.prospring5.ch7.entities.Singer;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.hibernate.SessionFactory;
import org.springframework.stereotype.Repository;
import org.springframework.transaction
          .annotation.Transactional;
import javax.annotation.Resource;
import java.util.List;

@Transactional
@Repository("singerDao")
public class SingerDaoImpl implements SingerDao {

  private static final Log logger =
          LogFactory.getLog(SingerDaoImpl.class);
  private SessionFactory sessionFactory;

  @Transactional(readOnly = true)
  public List<Singer> findAll() {
    return sessionFactory.getCurrentSession()
            .createQuery("from Singer s").list();
  }
  ...
}
```

Метод `SessionFactory.getCurrentSession()` получает реализацию интерфейса `Session` из Hibernate. Затем вызывается метод `Session.createQuery()`, которому передается оператор HQL. Оператор `from Singer s` извлекает все записи о певцах из базы данных. Альтернативный синтаксис этого оператора выглядит следующим образом: `select s from Singer s`. Аннотация `@Transactional(readOnly=true)` означает, что транзакция должна быть установлена как доступная только для чтения. Установка этого атрибута для методов, выполняющих только чтение, позволяет повысить производительность.

Ниже приведена простая программа для тестирования класса `SingerDaoImpl`.

```
package com.apress.prospring5.ch7;

import com.apress.prospring5.ch7.config.AppConfig;
import com.apress.prospring5.ch7.dao.SingerDao;
import com.apress.prospring5.ch7.entities.Singer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;
import java.util.List;

public class SpringHibernateDemo {

  private static Logger logger =
       LoggerFactory.getLogger(SpringHibernateDemo.class);

  public static void main(String... args) {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               AppConfig.class);

    SingerDao singerDao = ctx.getBean(SingerDao.class);
    singerDao.delete(singer);
    listSingers(singerDao.findAll());

    ctx.close();
  }

  private static void listSingers(List<Singer> singers) {
    logger.info(" ---- Listing singers:");
    for (Singer singer : singers) {
      logger.info(singer.toString());
    }
  }
}
```

Выполнение исходного кода приведенной выше тестовой программы дает следующий результат:

```
---- Listing singers:[1]
Singer - Id: 1, First name: John, Last name: Mayer,
   Birthday: 1977-10-16
Singer - Id: 3, First name: John, Last name: Butler,
   Birthday: 1975-04-01
```

[1] ---- Список певцов:

```
Singer - Id: 2, First name: Eric, Last name: Clapton,
   Birthday: 1945-03-30
```

На первый взгляд, записи о певцах были извлечены, но где же сведения об альбомах и музыкальных инструментах? Внесем коррективы в тестовую программу, чтобы вывести и эти сведения. С этой целью заменим метод `listSingers()` на метод `listSingersWithAlbum()`, как показано ниже.

```
package com.apress.prospring5.ch7;

import com.apress.prospring5.ch7.config.AppConfig;
import com.apress.prospring5.ch7.dao.SingerDao;
import com.apress.prospring5.ch7.entities.Album;
import com.apress.prospring5.ch7.entities.Instrument;
import com.apress.prospring5.ch7.entities.Singer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;
import java.util.List;

public class SpringHibernateDemo {

  private static Logger logger = LoggerFactory.getLogger(
                        SpringHibernateDemo.class);

  public static void main(String... args) {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               AppConfig.class);

    SingerDao singerDao = ctx.getBean(SingerDao.class);
    Singer singer = singerDao.findById(2l);
    singerDao.delete(singer);
    listSingersWithAlbum(singerDao.findAllWithAlbum());

    ctx.close();
  }

  private static void listSingersWithAlbum(
                List<Singer> singers) {
    logger.info(" ---- Listing singers with instruments:");
    for (Singer singer : singers) {
      logger.info(singer.toString());
      if (singer.getAlbums() != null) {
        for (Album album : singer.getAlbums()) {
          logger.info("\t" + album.toString());
```

```
        }
      }
      if (singer.getInstruments() != null) {
        for (Instrument instrument :
             singer.getInstruments()) {
          logger.info("\t" + instrument.getInstrumentId());
        }
      }
    }
  }
}
```

Если снова выполнить исходный код приведенной выше тестовой программы, то на консоль будет выведен следующий результат:

```
---- Listing singers with instruments:[2]
Singer - Id: 1, First name: John, Last name: Mayer,
   Birthday: 1977-10-16
org.hibernate.LazyInitializationException:
   failed to lazily initialize a collection of role:[3]
   com.apress.prospring5.ch7.entities.Singer.albums,
   could not initialize proxy - no Session[4]
```

При попытке получить доступ к связям библиотека Hibernate сгенерирует исключение типа `LazyInitializationException`. Дело в том, что по умолчанию библиотека Hibernate выбирает связь отложенно (или по требованию), а это означает, что таблицы этой связи (т.е. `ALBUM`) не будут соединены для заполнения записями. Веским основанием для такого подхода служит производительность. Ведь если по запросу извлекаются тысячи записей и все связи, то большой объем передаваемых данных приведет к снижению производительности.

Запрос с выборкой связей

Извлечь данные из связей средствами Hibernate можно одним из двух способов. Первым способом связь можно определить с режимом выборки `EAGER`, например: `@ManyToMany(fetch=FetchType.EAGER)`. Тем самым библиотеке Hibernate предписывается выбирать связанные записи по каждому запросу. Но, как пояснялось ранее, такой способ оказывает отрицательное влияние на производительность при извлечении данных.

Второй способ состоит в том, чтобы предписать библиотеке Hibernate выбирать связанные данные только по требованию в запросе. Так, если делается запрос типа `Criteria`, можно вызвать метод `Criteria.setFetchMode()`, чтобы библиотека Hibernate немедленно произвела выборку связи. А если делается запрос типа `Named`

---

[2] `---- Список певцов с инструментами:`

[3] `Не удалось инициализировать коллекцию ролей по требованию:`

[4] `Не удалось инициализировать заместитель из-за отсутствия сеанса`

Query, то можно воспользоваться операцией fetch, чтобы предписать библиотеке Hibernate немедленно произвести выборку связи.

Рассмотрим в качестве примера реализацию метода findAllWithAlbum(), в котором будут извлекаться сведения обо всех певцах вместе с их альбомами. В данном случае мы воспользуемся именованным запросом типа NamedQuery. Именованный запрос может быть вынесен в XML-файл или объявлен с помощью аннотации в классе сущности. Ниже приведен фрагмент исходного кода из переделанного варианта класса Singer, представляющего объект предметной области с именованным запросом, определяемым с помощью аннотаций.

```
package com.apress.prospring5.ch7.entities;

import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
...
@Entity
@Table(name = "singer")
@NamedQueries({
      @NamedQuery(name="Singer.findAllWithAlbum",
                  query="select distinct s from Singer s "
                      + "left join fetch s.albums a "
                      + "left join fetch s.instruments i")
})
public class Singer implements Serializable {
  ...
}
```

Сначала в приведенном выше коде определяется экземпляр Singer.findAll WithAlbum именованного запроса типа NamedQuery, а затем тело запроса на языке HQL. Обратите внимание на предложение left join fetch, в котором библиотеке Hibernate предписывается немедленно произвести выборку связи. В этом запросе применяется также оператор select distinct, иначе Hibernate возвратит дублированные объекты (например, два объекта со сведениями об одном и том же певце, если с ним связаны два альбома).

Ниже приведена реализация метода findAllWithAlbum().

```
package com.apress.prospring5.ch7.dao;
...
@Transactional
@Repository("singerDao")
public class SingerDaoImpl implements SingerDao {
  @Transactional(readOnly = true)
  public List<Singer> findAllWithAlbum() {
    return sessionFactory.getCurrentSession()
```

```
            .getNamedQuery("Singer.findAllWithAlbum").list();
    }
}
```

На этот раз вызывается метод `Session.getNamedQuery()`, которому передается имя экземпляра типа `NamedQuery`. Внеся коррективы в тестовую программу (т.е. в класс `SpringHibernateDemo`) с целью вызвать метод `singerDao.findAllWithAlbum()` и запустив ее на выполнение, можно получить следующий результат:

```
---- Listing singers with instruments:
Singer - Id: 1, First name: John, Last name: Mayer,
   Birthday: 1977-10-16
Album - Id: 2, Singer id: 1, Title:
   Battle Studies,Release Date: 2009-11-17
Album - Id: 1, Singer id: 1, Title:
   The Search For Everything, Release Date: 2017-01-20
   Instrument: Guitar
   Instrument: Piano
Singer - Id: 3, First name: John, Last name: Butler,
   Birthday: 1975-04-01
Singer - Id: 2, First name: Eric, Last name: Clapton,
   Birthday: 1945-03-30
Album - Id: 3, Singer id: 2, Title:
   From The Cradle , Release Date: 1994-09-13
   Instrument: Guitar
```

Теперь подробные сведения обо всех певцах извлекаются правильно. Рассмотрим еще один пример именованного запроса типа `NamedQuery` с параметрами. На этот раз реализуем метод `findById()`, который также должен производить выборку связей. Ниже приведен фрагмент исходного кода класса `Singer`, в который введен новый именованный запрос.

```
package com.apress.prospring5.ch7.entities;

import javax.persistence.*;
import java.io.Serializable;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import static javax.persistence.GenerationType.IDENTITY;

@Entity
@Table(name = "singer")
@NamedQueries({
      @NamedQuery(name="Singer.findById",
                  query="select distinct s from Singer s "
                      + "left join fetch s.albums a "
                      + "left join fetch s.instruments i "
                      + "where s.id = :id"),
```

```
    @NamedQuery(name="Singer.findAllWithAlbum",
                query="select distinct s from Singer s "
                    + "left join fetch s.albums a "
                    + "left join fetch s.instruments i")
})
public class Singer implements Serializable {
  ...
}
```

В именованном запросе `Singer.findById` объявляется именованный параметр `id`. Ниже приведена реализация метода `findById()` в классе `SingerDaoImpl`.

```
package com.apress.prospring5.ch7.dao;

import com.apress.prospring5.ch7.entities.Singer;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.hibernate.SessionFactory;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
import javax.annotation.Resource;
import java.util.List;

@Transactional
@Repository("singerDao")
public class SingerDaoImpl implements SingerDao {

  private static final Log logger =
          LogFactory.getLog(SingerDaoImpl.class);
  private SessionFactory sessionFactory;

  @Transactional(readOnly = true)
  public Singer findById(Long id) {
    return (Singer) sessionFactory.getCurrentSession()
                    .getNamedQuery("Singer.findById")
                    .setParameter("id", id).uniqueResult();
  }
  ...
}
```

В приведенном выше коде используется тот же самый метод `Session.getNamedQuery()`. Кроме того, в нем вызывается метод `setParameter()`, которому передается устанавливаемое значение параметра именованного запроса. Если же таких параметров несколько, то можно вызвать метод `setParameterList()` или `setParameters()` из интерфейса `Query`.

Имеются и более развитые способы составления запросов, в том числе собственных запросов или запросов с критериями поиска, которые будут обсуждаться в следующей главе при рассмотрении прикладного интерфейса JPA. Чтобы протестировать

метод `findById()`, необходимо внести соответствующие изменения в исходный код класса `SpringHibernateDemo`, как показано ниже.

```
package com.apress.prospring5.ch7;
...
public class SpringHibernateDemo {

  private static Logger logger = LoggerFactory.getLogger(
                        SpringHibernateDemo.class);

  public static void main(String... args) {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               AppConfig.class);
    SingerDao singerDao = ctx.getBean(SingerDao.class);
    Singer singer = singerDao.findById(2l);
    logger.info(singer.toString());

    ctx.close();
  }
}
```

Выполнение этой тестовой программы дает следующий результат:

```
Singer - Id: 1, First name: John, Last name: Mayer,
   Birthday: 1977-10-16
```

Вставка данных

Вставка данных в Hibernate осуществляется очень просто. Единственная особенность состоит в извлечении первичного ключа, сгенерированного базой данных. Как было показано в предыдущей главе, в интерфейсе JDBC приходилось явно заявлять, что требуется извлечь сгенерированный ключ, передав экземпляр класса `KeyHolder` и получив из него ключ после выполнения оператора вставки. А в Hibernate все эти действия не требуются. Библиотека Hibernate извлечет сгенерированный ключ и заполнит объект предметной области после вставки. В качестве примера ниже приведена реализация метода `save()` в классе `SingerDaoImpl`.

```
package com.apress.prospring5.ch7.dao;

import com.apress.prospring5.ch7.entities.Singer;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.hibernate.SessionFactory;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
import javax.annotation.Resource;
import java.util.List;
```

```
@Transactional
@Repository("singerDao")
public class SingerDaoImpl implements SingerDao {
  private static final Log logger =
          LogFactory.getLog(SingerDaoImpl.class);
  private SessionFactory sessionFactory;

  public Singer save(Singer singer) {
    sessionFactory.getCurrentSession().saveOrUpdate(singer);
    logger.info("Singer saved with id: " + singer.getId());
    return singer;
  }
  ...
}
```

В данном случае достаточно вызвать метод `Session.saveOrUpdate()`, который может использоваться для выполнения операций вставки и обновления. Кроме того, здесь протоколируется идентификатор сохраненного объекта, который представляет певца и будет заполнен средствами Hibernate после сохранения в базе данных. Ниже приведен фрагмент кода для вставки новой записи со сведениями о певце в таблицу `SINGER` и еще двух порожденных записей в таблицу `ALBUM`, а также для проверки удачного исхода операции вставки. А поскольку в данном случае изменяется содержимое обеих таблиц, то для проверки каждой операции в отдельности в большей степени подходит класс JUnit.

```
package com.apress.prospring5.ch7;

import com.apress.prospring5.ch7.config.AppConfig;
import com.apress.prospring5.ch7.dao.SingerDao;
import com.apress.prospring5.ch7.entities.Album;
import com.apress.prospring5.ch7.entities.Instrument;
import com.apress.prospring5.ch7.entities.Singer;
import org.junit.Before;
import org.junit.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.List;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

public class SingerDaoTest {
  private static Logger logger =
```

```
        LoggerFactory.getLogger(SingerDaoTest.class);

private GenericApplicationContext ctx;
private SingerDao singerDao;

@Before
public void setUp() {
  ctx = new AnnotationConfigApplicationContext(
            AppConfig.class);
  singerDao = ctx.getBean(SingerDao.class);
  assertNotNull(singerDao);
}

@Test
public void testInsert() {
  Singer singer = new Singer();
  singer.setFirstName("BB");
  singer.setLastName("King");
  singer.setBirthDate(new Date(
         (new GregorianCalendar(1940, 8, 16))
         .getTime().getTime()));

  Album album = new Album();
  album.setTitle("My Kind of Blues");
  album.setReleaseDate(new java.sql.Date(
        (new GregorianCalendar(1961, 7, 18))
        .getTime().getTime()));
  singer.addAbum(album);
  album = new Album();
  album.setTitle("A Heart Full of Blues");
  album.setReleaseDate(new java.sql.Date(
        (new GregorianCalendar(1962, 3, 20))
        .getTime().getTime()));
  singer.addAbum(album);

  singerDao.save(singer);
  assertNotNull(singer.getId());

  List<Singer> singers = singerDao.findAllWithAlbum();
  assertEquals(4, singers.size());
  listSingersWithAlbum(singers);
}

@Test
public void testFindAll() {
  List<Singer> singers = singerDao.findAll();
  assertEquals(3, singers.size());
  listSingers(singers);
}
```

```
  @Test
  public void testFindAllWithAlbum() {
    List<Singer> singers = singerDao.findAllWithAlbum();
    assertEquals(3, singers.size());
    listSingersWithAlbum(singers);
  }

  @Test
  public void testFindByID() {
    Singer singer = singerDao.findById(1L);
    assertNotNull(singer);
    logger.info(singer.toString());
  }

  private static void listSingers(List<Singer> singers) {
    logger.info(" ---- Listing singers:");
    for (Singer singer : singers) {
      logger.info(singer.toString());
    }
  }

  private static void listSingersWithAlbum(
          List<Singer> singers) {
    logger.info(" ---- Listing singers with instruments:");
    for (Singer singer : singers) {
      logger.info(singer.toString());
      if (singer.getAlbums() != null) {
        for (Album album : singer.getAlbums()) {
          logger.info("\t" + album.toString());
        }
      }
      if (singer.getInstruments() != null) {
        for (Instrument instrument :
             singer.getInstruments()) {
          logger.info("\tInstrument: "
                    + instrument.getInstrumentId());
        }
      }
    }
  }

  @After
  public void tearDown() {
    ctx.close();
  }
}
```

Как следует из приведенного выше кода, в методе `testInsert()` вводятся два альбома и сохраняется объект, представляющий певца. После этого вызывается ме-

тод `listSingersWithAlbum()`, чтобы снова перечислить всех певцов. Если выполнить тестовый метод `testInsert()`, то на консоль будет выведен следующий результат:

```
...
INFO o.h.d.Dialect - HHH000400:
  Using dialect: org.hibernate.dialect.H2Dialect
INFO o.h.h.i.QueryTranslatorFactoryInitiator - HHH000397:
  Using ASTQueryTranslatorFactory
Hibernate:
  /* insert com.apress.prospring5.ch7.entities.Singer
   */ insert
   into
     singer
     (ID, BIRTH_DATE, FIRST_NAME, LAST_NAME, VERSION)
   values
     (null, ?, ?, ?, ?)
Hibernate:
  /* insert com.apress.prospring5.ch7.entities.Album
   */ insert
   into
     album
     (ID, RELEASE_DATE, SINGER_ID, title, VERSION)
   values
     (null, ?, ?, ?, ?)
Hibernate:
  /* insert com.apress.prospring5.ch7.entities.Album
   */ insert
   into
     album
     (ID, RELEASE_DATE, SINGER_ID, title, VERSION)
   values
     (null, ?, ?, ?, ?)
INFO c.a.p.c.d.SingerDaoImpl - Singer saved with id: 4
...
INFO - ---- Listing singers with instruments:
INFO - Singer - Id: 4, First name: BB, Last name: King,
       Birthday: 1940-09-16
INFO - Album - Id: 5, Singer id: 4, Title:
       A Heart Full of Blues, Release Date: 1962-04-20
INFO - Album - Id: 4, Singer id: 4, Title: My Kind of Blues,
       Release Date: 1961-08-18
INFO - Singer - Id: 1, First name: John, Last name: Mayer,
       Birthday: 1977-10-16
INFO - Album - Id: 2, Singer id: 1, Title: Battle Studies,
       Release Date: 2009-11-17
INFO - Album - Id: 1, Singer id: 1, Title:
       The Search For Everything, Release Date: 2017-01-20
INFO - Instrument: Piano
```

```
INFO - Instrument: Guitar
INFO - Singer - Id: 3, First name: John, Last name: Butler,
Birthday: 1975-04-01
INFO - Singer - Id: 2, First name: Eric, Last name: Clapton,
       Birthday: 1945-03-30
INFO - Album - Id: 3, Singer id: 2, Title: From The Cradle,
       Release Date: 1994-09-13
INFO - Instrument: Guitar
```

Конфигурация протоколирования была видоизменена таким образом, чтобы вывести на консоль более подробные сведения из библиотеки Hibernate. Как следует из протокольной записи `INFO`, вновь сохраненный объект, представляющий певца, правильно заполнен по его идентификатору. Кроме того, библиотека Hibernate отображает все операторы SQL, выполняемые над базой данных, а это дает возможность ясно видеть происходящее в ней подспудно.

Обновление данных

Обновить запись в базе данных так же просто, как и вставить в нее данные. Допустим, требуется обновить имя певца с идентификатором 1 и удалить один его альбом из базы данных. Для проверки подобной операции обновления ниже приведен фрагмент исходного кода из класса `SingerDaoTest`, в который введен метод `testUpdate()`.

```
package com.apress.prospring5.ch7;
...
public class SingerDaoTest {

  private GenericApplicationContext ctx;
  private SingerDao singerDao;
  ...

  @Test
  public void testUpdate() {
    Singer singer = singerDao.findById(1L);
    // убедиться, что такой певец существует
    assertNotNull(singer);

    // убедиться, что получен ожидаемый певец
    assertEquals("Mayer", singer.getLastName());

    // извлечь альбом
    Album album = singer.getAlbums().stream().filter(
      a -> a.getTitle().equals("Battle Studies"))
            .findFirst().get();

    singer.setFirstName("John Clayton");
    singer.removeAlbum(album);
```

```
    singerDao.save(singer);

    // проверить обновление
    listSingersWithAlbum(singerDao.findAllWithAlbum());
  }
  ...
}
```

Как следует из приведенного выше примера кода, сначала извлекается запись с идентификатором 1, а затем изменяется имя певца. После этого циклически перебираются объекты, представляющие альбомы, а также извлекается альбом *Battle Studies* (Боевые учения), который удаляется из свойства `albums` объекта, представляющего данного певца. И, наконец, в данном примере кода снова вызывается метод `singerDao.save()`. Если выполнить обновленную таким образом тестовую программу, то на консоль будет выведен следующий результат:

```
INFO o.h.h.i.QueryTranslatorFactoryInitiator - HHH000397:
   Using ASTQueryTranslatorFactory
...
INFO - Singer saved with id: 1
Hibernate:
   /* update
      com.apress.prospring5.ch7.entities.Album */
          update album
      set
         RELEASE_DATE=?,
         SINGER_ID=?,
         title=?,
         VERSION=?
      where
         ID=?
         and VERSION=?
Hibernate:
   /* delete com.apress.prospring5.ch7.entities.Album */
    delete from
      album
    where
       ID=?
    and VERSION=?
INFO ----- Listing singers with instruments:
INFO - Singer - Id: 1, First name: John Clayton,
       Last name: Mayer, Birthday: 1977-10-16
INFO - Album - Id: 1, Singer id: 1, Title:
       The Search For Everything, Release Date: 2017-01-20
INFO - Instrument: Guitar
INFO - Instrument: Piano
INFO - Singer - Id: 2, First name: Eric,
       Last name: Clapton, Birthday: 1945-03-30
INFO - Album - Id: 3, Singer id: 2, Title: From The Cradle,
```

```
       Release Date: 1994-09-13
INFO - Instrument: Guitar
INFO - Singer - Id: 3, First name: John, Last name: Butler,
       Birthday: 1975-04-01
```

Как видите, имя певца было обновлено, тогда как альбом *Battle Studies* удален. Альбом может быть удален, поскольку связи "один ко многим" можно передать атрибут `orphanRemoval=true`, который предписывает библиотеке Hibernate, что все висячие (т.е. не связанные с данным певцом) записи в базе данных должны быть удалены.

Удаление данных

Удаление данных выполняется также просто. Для этого достаточно вызвать метод `Session.delete()`, передав ему удаляемый объект. Ниже приведен исходный код реализации данного метода.

```
package com.apress.prospring5.ch7.dao;
...
@Transactional
@Repository("singerDao")
public class SingerDaoImpl implements SingerDao {

  private static final Log logger =
          LogFactory.getLog(SingerDaoImpl.class);
  private SessionFactory sessionFactory;

  public void delete(Singer singer) {
    sessionFactory.getCurrentSession().delete(singer);
    logger.info("Singer deleted with id: "
              + singer.getId());
  }
  ...
}
```

В данном случае из базы данных будет удалена запись о певце, а также вся связанная с ним информация, включая альбомы и музыкальные инструменты, поскольку в преобразовании было определено условие `cascade=CascadeType.ALL`. Ниже приведен исходный код тестового метода `testDelete()`, предназначенного для проверки метода удаления.

```
package com.apress.prospring5.ch7;
...
public class SingerDaoTest {
  private static Logger logger =
          LoggerFactory.getLogger(SingerDaoTest.class);
  private GenericApplicationContext ctx;
  private SingerDao singerDao;
```

```
  @Test
  public void testDelete() {
    Singer singer = singerDao.findById(2l);
    // убедиться, что певец существует
    assertNotNull(singer);
    singerDao.delete(singer);

    listSingersWithAlbum(singerDao.findAllWithAlbum());
  }
}
```

В приведенном выше примере кода сначала извлекается запись о певце с идентификатором 2, а затем вызывается метод `delete()` для удаления сведений о певце. Выполнение этой тестовой программы даст приведенный ниже результат. Как видите, запись о певце с идентификатором 2 удалена вместе с порожденными записями из таблиц `ALBUM` и `SINGER_INSTRUMENT`.

```
INFO o.h.h.i.QueryTranslatorFactoryInitiator - HHH000397:
   Using ASTQueryTranslatorFactory
...
INFO c.a.p.c.d.SingerDaoImpl - Singer deleted with id: 2
Hibernate:
   /* delete collection com.apress.prospring5.ch7.entities
      .Singer.instruments */ delete
    from
       singer_instrument
    where
       SINGER_ID=?
Hibernate:
   /* delete com.apress.prospring5.ch7.entities.Album */
      delete
         from
            album
         where
            ID=?
            and VERSION=?
Hibernate:
   /* delete com.apress.prospring5.ch7.entities.Singer */
      delete
          from
             singer
          where
             ID=?
             and VERSION=?
INFO - ---- Listing singers with instruments:
INFO - Singer - Id: 1, First name: John, Last name: Mayer,
       Birthday: 1977-10-16
INFO - Album - Id: 1, Singer id: 1, Title:
       The Search For Everything, Release Date: 2017-01-20
```

```
INFO - Album - Id: 2, Singer id: 1, Title: Battle Studies,
       Release Date: 2009-11-17
INFO - Instrument: Piano
INFO - Instrument: Guitar
INFO - Singer - Id: 3, First name: John, Last name: Butler,
       Birthday: 1975-04-01
```

Конфигурирование Hibernate для формирования таблиц из сущностей

При разработке приложений автозапуска с применением библиотеки Hibernate зачастую наблюдается стремление написать сначала классы сущностей, а затем сформировать таблицы базы данных, исходя из их содержимого. И это делается с помощью свойства `hibernate.hbm2ddl.auto` из библиотеки Hibernate. Когда приложение запускается на выполнение в первый раз, в данном свойстве устанавливается значение `create`. В итоге библиотека Hibernate просмотрит все сущности, сформирует таблицы и (первичные, внешние, однозначные) ключи по отношениям, определенным с помощью аннотаций JPA и Hibernate.

Если сущности сконфигурированы правильно и в базе данных находятся предполагаемые объекты, то в упомянутом выше свойстве следует установить значение `update`. Тем самым библиотеке Hibernate предписывается обновить существующую базу данных, внеся в дальнейшем любые изменения в сущности и в то же время сохранив исходную базу данных и любую введенную в нее информацию.

При разработке производственных приложений целесообразно писать модульные и комплексные тесты, выполняемые над имитируемой базой данных, которая отвергается после выполнения всех контрольных примеров. Тестовая база данных обычно находится в оперативной памяти, поэтому библиотеке Hibernate предписывается создать такую базу данных и отвергнуть ее после выполнения тестов, установив значение `create-drop` в свойстве `hibernate.hbm2ddl.auto`. Полный перечень значений свойства `hibernate.hbm2ddl.auto` можно найти в официальной документации на Hibernate.[5]

В приведенном ниже фрагменте кода демонстрируется конфигурационный класс Java под названием `AdvancedConfig`. Как видите, в этом классе внедрено свойство `hibernate.hbm2ddl.auto`, а также применяется объединенный в пул источник данных DBCP.

```
package com.apress.prospring5.ch7.config;

import com.apress.prospring5.ch6.CleanUp;
import org.apache.commons.dbcp2.BasicDataSource;
import org.hibernate.SessionFactory;
```

[5] См. табл. 3.7 по ссылке на странице, доступной по адресу `https://docs.jboss.org/hibernate/orm/5.0/manual/en-US/html/ch03.html`.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context
          .annotation.ComponentScan;
import org.springframework.context
          .annotation.Configuration;
import org.springframework.context
          .annotation.PropertySource;
import org.springframework.context.support
          .PropertySourcesPlaceholderConfigurer;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.orm.hibernate5
          .HibernateTransactionManager;
import org.springframework.orm.hibernate5
          .LocalSessionFactoryBuilder;
import org.springframework.transaction
          .PlatformTransactionManager;
import org.springframework.transaction.annotation
          .EnableTransactionManagement;

import javax.sql.DataSource;
import java.io.IOException;
import java.util.Properties;

@Configuration
@ComponentScan(basePackages = "com.apress.prospring5.ch7")
@EnableTransactionManagement
@PropertySource("classpath:db/jdbc.properties")
public class AdvancedConfig {
  private static Logger logger =
          LoggerFactory.getLogger(AdvancedConfig.class);

  @Value("${driverClassName}")
  private String driverClassName;
  @Value("${url}")
  private String url;
  @Value("${username}")
  private String username;
  @Value("${password}")
  private String password;

  @Bean
  public static PropertySourcesPlaceholderConfigurer
         propertySourcesPlaceholderConfigurer() {
    return new PropertySourcesPlaceholderConfigurer();
  }
```

```
  @Bean(destroyMethod = "close")
  public DataSource dataSource() {
    try {
      BasicDataSource dataSource = new BasicDataSource();
      dataSource.setDriverClassName(driverClassName);
      dataSource.setUrl(url);
      dataSource.setUsername(username);
      dataSource.setPassword(password);
      return dataSource;
    } catch (Exception e) {
      logger.error("DBCP DataSource bean cannot "
                + "be created!", e);
      return null;
    }
  }

  private Properties hibernateProperties() {
    Properties hibernateProp = new Properties();
    hibernateProp.put("hibernate.dialect",
                      "org.hibernate.dialect.H2Dialect");
    hibernateProp.put("hibernate.hbm2ddl.auto",
                      "create-drop");
    hibernateProp.put("hibernate.format_sql", true);
    hibernateProp.put("hibernate.use_sql_comments", true);
    hibernateProp.put("hibernate.show_sql", true);
    hibernateProp.put("hibernate.max_fetch_depth", 3);
    hibernateProp.put("hibernate.jdbc.batch_size", 10);
    hibernateProp.put("hibernate.jdbc.fetch_size", 50);
    return hibernateProp;
  }

  @Bean
  public SessionFactory sessionFactory() {
    return new LocalSessionFactoryBuilder(dataSource())
        .scanPackages("com.apress.prospring5.ch7.entities")
        .addProperties(hibernateProperties())
        .buildSessionFactory();
  }

  @Bean public PlatformTransactionManager
              transactionManager()
    throws IOException {
    return new HibernateTransactionManager(
          sessionFactory());
  }
}
```

В файле свойств `jdbc.properties` содержатся следующие свойства, необходимые для доступа к базе данных в оперативной памяти:

```
driverClassName=org.h2.Driver
url=jdbc:h2:musicdb
username=prospring5
password=prospring5
```

Но как в таком случае заполнить базу данных первоначальной информацией? Для этого можно воспользоваться экземпляром класса `DatabasePopulator`, библиотекой вроде DbUnit[6] или специальным компонентом Spring Bean для заполнения базы данных, аналогичным компоненту типа `DbIntializer`:

```
package com.apress.prospring5.ch7.config;

import com.apress.prospring5.ch7.dao.InstrumentDao;
import com.apress.prospring5.ch7.dao.SingerDao;
import com.apress.prospring5.ch7.entities.Album;
import com.apress.prospring5.ch7.entities.Instrument;
import com.apress.prospring5.ch7.entities.Singer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import javax.annotation.PostConstruct;
import java.util.Date;
import java.util.GregorianCalendar;

@Service
public class DBInitializer {
  private Logger logger =
          LoggerFactory.getLogger(DBInitializer.class);

  @Autowired SingerDao singerDao;
  @Autowired InstrumentDao instrumentDao;

  @PostConstruct
  public void initDB() {
    logger.info("Starting database initialization...");
    Instrument guitar = new Instrument();
    guitar.setInstrumentId("Guitar");
    instrumentDao.save(guitar);
    ...
    Singer singer = new Singer();
    singer.setFirstName("John");
    singer.setLastName("Mayer");
    singer.setBirthDate(new Date(
          (new GregorianCalendar(1977, 9, 16))
```

[6] Официальный веб-сайт библиотеки DbUnit доступен по адресу `http://dbunit.sourceforge.net/`.

```
            .getTime().getTime()));
        singer.addInstrument(guitar);
        singer.addInstrument(piano);

        Album album1 = new Album();
        album1.setTitle("The Search For Everything");
        album1.setReleaseDate(new java.sql.Date(
              (new GregorianCalendar(2017, 0, 20))
              .getTime().getTime()));
        singer.addAbum(album1);

        Album album2 = new Album();
        album2.setTitle("Battle Studies");
        album2.setReleaseDate(new java.sql.Date(
               (new GregorianCalendar(2009, 10, 17))
               .getTime().getTime()));
        singer.addAbum(album2);

        singerDao.save(singer);
        ...
        logger.info("Database initialization finished.");
    }
}
```

Класс `DbIntializer` представляет собой простой компонент Spring Bean, в котором информационные хранилища внедряются как зависимости, а с помощью аннотации `@PostConstruct` определяется метод инициализации, где объекты создаются и сохраняются в базе данных. Сам же класс `DbIntializer` снабжен аннотацией `@Service` как компонент Spring Bean, предоставляющий услуги по инициализации содержимого базы данных. Экземпляр этого класса получается при создании контекста типа `ApplicationContext`, после чего выполняется метод инициализации, чтобы гарантировать заполнение базы данных прежде, чем будет использован данный контекст. Если воспользоваться конфигурационным классом `AdvancedConfig`, то те же самые тесты пройдут, как и прежде.

Аннотировать ли методы или поля

В приведенном выше примере классы сущностей содержат методы доступа, снабженные аннотациями JPA. Но ведь аннотациями JPA можно снабдить непосредственно и поля, что дает следующие преимущества.

- Конфигурация класса сущности становится более понятной и сосредоточена в разделе полей, а не разбросана по всему содержимому классу. Очевидно, что это справедливо лишь в том случае, если исходный код был написан в соответствии с четкими рекомендациями по программированию, предписывающими хранить объявления всех полей в классе в одном непрерывном разделе.

- Аннотирование полей в классе сущности совсем не требует предоставлять для них методы установки и получения. Это удобно, например, в том случае, когда требуется снабдить аннотацией `@Version` поле, которое вообще не должно быть модифицировано вручную. При этом оно должно быть доступно только из библиотеки Hibernate.
- Аннотирование полей позволяет выполнять дополнительную обработку в методах установки (например, шифрование или вычисление значения после загрузки из базы данных). Но трудности доступа к свойствам состоят в том, что методы установки вызываются и при загрузке объекта.

В Интернете ведется горячая полемика, что лучше аннотировать: методы или поля? С точки зрения производительности особых отличий в обоих способах аннотирования не существует. А на самом деле полемика ведется между разработчиками потому, что иногда более целесообразно аннотировать методы доступа. Следует, однако, иметь в виду, что в базе данных, по существу, сохраняется состояние объектов, которое определяется значениями его полей, а не значениями, возвращаемыми методами доступа. Это также означает, что объект может быть воссоздан из базы данных таким же точно образом, каким он был сохранен. В какой-то степени установка аннотаций в методах установки можно рассматривать как нарушение инкапсуляции.

Ниже приведен класс сущности `Singer`, переделанный с целью аннотировать поля и расширяющий абстрактный класс `AbstractEntity`, содержащий два поля, общие для всех классов сущностей Hibernate в приложении.

```
// Исходный файл AbstractEntity.java
package com.apress.prospring5.ch7.entities;

import javax.persistence.*;
import java.io.Serializable;

@MappedSuperclass
public abstract class AbstractEntity
        implements Serializable {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  @Column(updatable = false)
  protected Long id;

  @Version
  @Column(name = "VERSION")
  private int version;

  public Long getId() {
    return id;
  }

  public void setId(Long id) {
```

```
    this.id = id;
  }
}

// Исходный файл Singer.java
@Entity
@Table(name = "singer")
@NamedQueries({
      @NamedQuery(name=Singer.FIND_SINGER_BY_ID,
                  query="select distinct s from Singer s "
                      + "left join fetch s.albums a "
                      + "left join fetch s.instruments i "
                      + "where s.id = :id"),
      @NamedQuery(name=Singer.FIND_ALL_WITH_ALBUM,
                  query="select distinct s from Singer s "
                      + "left join fetch s.albums a "
                      + "left join fetch s.instruments i")
})
public class Singer extends AbstractEntity {

  public static final String FIND_SINGER_BY_ID =
                "Singer.findById";
  public static final String FIND_ALL_WITH_ALBUM =
                "Singer.findAllWithAlbum";

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

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

  @Temporal(TemporalType.DATE)
  @Column(name = "BIRTH_DATE")
  private Date birthDate;

  @OneToMany(mappedBy = "singer", cascade=CascadeType.ALL,
                       orphanRemoval=true)
  private Set<Album> albums = new HashSet<>();

  @ManyToMany
  @JoinTable(name = "singer_instrument",
             joinColumns = @JoinColumn(name = "SINGER_ID"),
             inverseJoinColumns =
                    @JoinColumn(name = "INSTRUMENT_ID"))
  private Set<Instrument> instruments = new HashSet<>();
  ...
}
```

Соображения по поводу применения Hibernate

Как уже было продемонстрировано в примерах, при правильном определении объектно-реляционного преобразования, связей и запросов библиотека Hibernate может обеспечить среду, которая позволит сосредоточить основное внимание на программировании с помощью объектной модели, а не на построении операторов SQL для каждой операции. В последние годы библиотека Hibernate быстро развивалась и была принята широким сообществом разработчиков приложений на Java (как с открытым кодом, так и масштаба предприятия) в качестве библиотеки на уровне доступа к данным.

Однако не следует забывать о нескольких важных моментах. Во-первых, в отсутствие всякого контроля над генерируемыми операторами SQL необходимо очень тщательно определять преобразования, особенно связи и связанную с ними стратегию выборки информации из базы данных. И во-вторых, необходимо внимательно следить за операторами SQL, которые генерирует библиотека Hibernate, чтобы удостовериться в их правильном поведении.

Понимание внутреннего механизма управления сеансами Hibernate также играет очень важную роль, особенно в пакетных операциях. Библиотека Hibernate хранит управляемые объекты в своем сеансе, регулярно сбрасывая и очищая их. Неудачно спроектированная логика доступа к данным может привести к тому, что библиотека Hibernate будет очищать свой сеанс слишком часто, из-за чего значительно снизится производительность. Если же требуется полный контроль над запросом, можно воспользоваться собственным запросом, рассматриваемым в следующей главе.

И, наконец, важную роль в настройке производительности Hibernate играют также параметры, в том числе размер пакета, объем выборки и т.д. Эти параметры следует определить в фабрике сеансов и корректировать на стадии нагрузочного тестирования разрабатываемого приложения с целью обнаружить их оптимальные значения.

В конечном счете библиотека Hibernate и отличная поддержка в ней прикладного интерфейса JPA, как поясняется в следующей главе, служат естественным решением для разработчиков приложений на Java, нуждающихся в объектно-ориентированном способе реализации логики доступа к данным.

Резюме

В этой главе были рассмотрены основные принципы действия библиотеки Hibernate и показано, как она конфигурируется в приложении Spring. Были также представлены общие методики определения объектно-реляционных преобразований (ORM), раскрыто понятие связей и продемонстрировано применение класса `HibernateTemplate` для выполнения различных операций в базе данных.

В этой главе удалось охватить лишь небольшую часть функциональных возможностей и средств Hibernate. Поэтому настоятельно рекомендуется изучить стандартную документацию на Hibernate, если вы собираетесь пользоваться библиотекой

Hibernate вместе с каркасом Spring. Имеется также обширная литература на эту тему, включая книгу *Beginning Hibernate: For Hibernate 5* Джозефа Оттингера, Джеффа Оттингера, Джеффа Линвуда и Дэвида Минтера (Joseph Ottinger, Jeff Linwood, and Dave Minter; издательство Apress, 2016 г.), а также книгу *Pro JPA 2* Майка Кита и Меррика Шинкариола (Mike Keith, Merrick Schincariol; издательство Apress, 2013 г.).

В следующей главе будет рассмотрен прикладной интерфейс JPA и особенности его применения в Spring. В библиотеке Hibernate предоставляется отличная поддержка прикладного интерфейса JPA, поэтому в примерах из следующей главы применение библиотеки Hibernate будет продолжено в качестве поставщика услуг сохраняемости. В отношении операций выборки и обновления прикладной интерфейс JPA действует подобно библиотеке Hibernate. Кроме того, в следующей главе будет рассмотрен ряд дополнительных вопросов, включая собственный запрос и запрос с критериями поиска, а также применение библиотеки Hibernate и поддержки в ней прикладного интерфейса JPA.

ГЛАВА 8

Доступ к данным в Spring через интерфейс JPA 2

В предыдущей главе было показано, как пользоваться библиотекой Hibernate в Spring при реализации логики доступа к данным с объектно-ориентированным преобразованием (ORM). В ней были продемонстрированы возможности настройки фабрики сеансов Hibernate при конфигурировании каркаса Spring и пояснялось, как применять интерфейс `Session` для выполнения различных операций доступа к данным. Но это только один способ применения Hibernate в приложении Spring. Другой способ предусматривает применение Hibernate в качестве поставщика услуг сохраняемости, соответствующего стандартному прикладному интерфейсу сохраняемости Java API (Java Persistence API — JPA).

Библиотека Hibernate и эффективный язык запросов HQL добились немалых успехов в преобразовании простых объектов POJO, а также повлияли на разработку стандартов, определяющих технологии доступа к данным в области Java. После Hibernate в организации JCP был подготовлен стандарт JDO (Java Data Objects — объекты данных Java), а затем и прикладной интерфейс JPA.

На момент написания этой книги прикладной интерфейс JPA достиг версии 2.1 и поддерживает такие стандартизованные понятия, как контекст сохраняемости (аннотация `@PersistenceContext`), диспетчер сущностей (интерфейс `EntityManager`) и язык JPQL (Java Persistence Query Language — язык запросов сохраняемости в Java). Такая стандартизация предоставляет разработчикам возможность сменять поставщиков услуг сохраняемости JPA, в том числе Hibernate, EclipseLink, Oracle TopLink и Apache OpenJPA. В итоге прикладной интерфейс JPA применяется в большинстве новых приложений на платформе JEE на уровне доступа к данным.

В каркасе Spring также предоставляется отличная поддержка прикладного интерфейса JPA. Например, для начальной загрузки диспетчера сущностей JPA имеется несколько реализаций компонента Spring Bean типа `EntityManagerFactory` с под-

держкой всех упомянутых ранее поставщиков услуг JPA. В рамках проекта Spring Data ведется также подпроект Spring Data JPA, ориентированный на расширенную поддержку применения JPA в приложениях Spring. К числу основных возможностей проекта Spring Data JPA относятся понятия хранилища и спецификации, а также поддержка предметно-ориентированного языка запросов QueryDSL (Query Domain Specific Language).

В этой главе поясняется, как пользоваться прикладным интерфейсом JPA 2.1 в Spring, а также библиотекой Hibernate в качестве базового поставщика услуг сохраняемости. Сначала в ней показывается, как реализовывать различные операции в базе данных с помощью интерфейса `EntityManager` из JPA и языка JPQL, а затем поясняется, каким образом проект Spring Data JPA помогает еще больше упростить разработку приложений с помощью прикладного интерфейса JPA. И, наконец, будут рассмотрены дополнительные вопросы, касающиеся преобразований ORM, в том числе собственные запросы, а также запросы с критериями поиска.

В этой главе будут, в частности, рассмотрены следующие вопросы.

- **Основные понятия JPA.** В этой части поясняется ряд основных понятий прикладного интерфейса JPA.
- **Конфигурирование диспетчера сущностей JPA.** В этой части обсуждаются реализации компонента типа `EntityManagerFactory`, поддерживаемые в Spring, а также способы конфигурирования наиболее употребительной из них — класса `LocalContainerEntityManagerFactoryBean` — при конфигурировании Spring в формате XML.
- **Операции с данными.** В этой части показано, как в JPA реализуются элементарные операции над базой данных, которые в принципе мало чем отличаются от аналогичных операций, реализуемых с помощью Hibernate.
- **Расширенные операции обработки запросов.** В этой части рассмотрено применение собственных запросов в JPA, а также строго типизированный прикладной интерфейс API критериев поиска для реализации более гибких операций обработки запросов в JPA.
- **Введение в Spring Data JPA.** В этой части описан проект Spring Data JPA и пояснено, каким образом он позволяет упростить разработку логики доступа к данным.
- **Отслеживание изменений в сущностях и аудит.** Общим требованием к операциям обновления базы данных является отслеживание дат создания и обновления сущностей, а также того, кто производил их изменение. Кроме того, требуется такая критически важная информация, как сведения о клиенте и данные из таблицы предыстории, где обычно хранятся все версии сущности. Здесь будет показано, каким образом Spring Data JPA и Hibernate Envers (Hibernate Entity Versioning System — система контроля версий сущностей Hibernate) могут упростить разработку подобного рода логики.

На заметку Аналогично Hibernate, в прикладном интерфейсе JPA поддерживается определение преобразований в формате XML или в аннотациях Java. В этой главе основное внимание уделено определению преобразований с помощью аннотаций, поскольку такой стиль конфигурирования более распространен, чем в формате XML.

Введение в JPA 2.1

Подобно другим запросам спецификации Java (Java Specification Request — JSR), цель спецификации JPA 2.1 (JSR-338) заключается в стандартизации программной модели преобразований ORM в средах JSE и JEE. В ней определен общий набор понятий, аннотаций, интерфейсов и других служб, которые должен реализовывать поставщик услуг сохраняемости в JPA. Программируя по стандарту JPA, разработчики имеют возможность сменять базового поставщика услуг сохраняемости по своему усмотрению, что очень похоже на переключение на другой совместимый с платформой JEE сервер для приложений, построенных по стандартам JEE.

В основу JPA положен интерфейс `EntityManager`, который происходит от фабрик типа `EntityManagerFactory`. Главное назначение интерфейса `EntityManager` — поддержка контекста сохраняемости, в котором будут храниться все экземпляры сущностей, управляемые этим контекстом. Конфигурация интерфейса `EntityManager` определяется как единица сохраняемости, и в приложении может существовать не одна такая единица. Если применяется библиотека Hibernate, то контекст сохраняемости можно рассматривать таким же образом, как и интерфейс `Session`, а компонент типа `EntityManagerFactory` — как и компонент `SessionFactory`. В библиотеке Hibernate управляемые сущности сохраняются в сеансе, с которым можно взаимодействовать напрямую через компонент `SessionFactory` или интерфейс `Session`. Но в JPA нельзя непосредственно взаимодействовать с контекстом сохраняемости. Вместо этого для выполнения всей необходимой работы приходится полагаться на интерфейс `EntityManager`.

Язык JPQL очень похож на язык HQL, поэтому, если у вас имеется опыт работы с HQL, перейти на JPQL вам не составит большого труда. Тем не менее в JPA 2 появился строго типизированный прикладной интерфейс API Criteria для критериев поиска, который полагается на метаданные преобразуемых сущностей при составлении запроса. Учитывая это обстоятельство, любые ошибки будут обнаруживаться во время компиляции, а не во время выполнения.

За более подробные сведениями о спецификации JPA 2 рекомендуем обратиться к книге *Pro JPA 2* (издательство Apress, 2013 г.). В этом разделе мы обсудим основные понятия JPA, пример модели выборочных данных, который будет применяться в примерах из этой главы, а также способы конфигурирования контекста типа `ApplicationContext` для поддержки JPA в Spring.

Модель выборочных данных для исходного кода примеров

В этой главе мы будем пользоваться той же моделью данных, что и в главе 7. Но при обсуждении реализации средств аудита в целях демонстрации введем в нее несколько столбцов и таблицу предыстории. Итак, начнем с тех же самых сценариев создания базы данных, которые приводились в предыдущей главе. Если вы по каким-то причинам пропустили главу 7, просмотрите модель данных, приведенную в ее разделе "Модель выборочных данных для исходного кода примеров", что поможет вам лучше понять примеры кода, приведенные далее в этой главе.

Конфигурирование компонента типа `EntityManagerFactory` из интерфейса JPA

Ранее в этой главе упоминалось, что для применения прикладного интерфейса JPA в Spring необходимо сконфигурировать компонент типа `EntityManagerFactory`, как это делалось ранее для компонента типа `SessionFactory` в Hibernate. В каркасе Spring поддерживаются три способа конфигурирования компонента типа `EntityManagerFactory`.

В первом способе конфигурирования используется класс `LocalEntityManagerFactoryBean`. Это простейший случай, когда достаточно указать имя единицы сохраняемости. Но поскольку в данном случае не поддерживается внедрение источника данных, а следовательно, нельзя принимать участие в глобальных транзакциях, то такой способ конфигурирования подходит только для стадии разработки.

Второй способ конфигурирования применяется для совместимого с платформой JEE 6 контейнера, в который сервер приложений производит начальную загрузку единицы сохраняемости JPA на основе информации в дескрипторах развертывания. Это дает каркасу Spring возможность искать диспетчер сущностей через интерфейс JNDI. Ниже приведен фрагмент конфигурации, где демонстрируется элемент, необходимый для поиска диспетчера сущностей через JNDI.

```
<beans ...>
  <jee:jndi-lookup id="prospring5Emf"
    jndi-name="persistence/prospring5PersistenceUnit"/>
</beans>
```

В спецификации JPA единица сохраняемости должна быть определена в конфигурационном файле `META-INF/persistence.xml`. Но в версии Spring 3.1 появилась возможность, устраняющая потребность в этом. Далее в этой главе будет показано, как воспользоваться такой возможностью.

Третий способ конфигурирования, который является наиболее распространенным и применяется в этой главе, представлен в классе `LocalContainerEntityManagerFactoryBean`. В этом случае поддерживается внедрение источника данных и допускается участие как в локальных, так и в глобальных транзакциях. Ниже приведен

фрагмент из файла конфигурации `app-context-annotation.xml`, где демонстрируется данный способ конфигурирования в формате XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context=
            "http://www.springframework.org/schema/context"
       xmlns:jdbc=
            "http://www.springframework.org/schema/jdbc"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation=
          "http://www.springframework.org/schema/jdbc
          http://www.springframework.org/schema/jdbc
                /spring-jdbc.xsd
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans
                /spring-beans.xsd
          http://www.springframework.org/schema/tx
          http://www.springframework.org/schema/tx
                /spring-tx.xsd
          http://www.springframework.org/schema/context
          http://www.springframework.org/schema/context
                /spring-context.xsd">

    <jdbc:embedded-database id="dataSource" type="H2">
       <jdbc:script location="classpath:sql/schema.sql"/>
       <jdbc:script location="classpath:sql/test-data.sql"/>
    </jdbc:embedded-database>

    <bean id="transactionManager"
       class= "org.springframework.orm.jpa
                  .JpaTransactionManager">
       <property name="entityManagerFactory" ref="emf"/>
    </bean>

    <tx:annotation-driven
        transaction-manager="transactionManager" />

    <bean id="emf" class=
          "org.springframework.orm.jpa
              .LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="jpaVendorAdapter">
          <bean class=
                "org.springframework.orm.jpa.vendor
                    .HibernateJpaVendorAdapter" />
      </property>
      <property name="packagesToScan"
```

```
            value="com.apress.prospring5.ch8.entities"/>
    <property name="jpaProperties">
        <props>
    <prop key="hibernate.dialect">
        org.hibernate.dialect.H2Dialect
    </prop>
    <prop key="hibernate.max_fetch_depth">3</prop>
    <prop key="hibernate.jdbc.fetch_size">50</prop>
    <prop key="hibernate.jdbc.batch_size">10</prop>
    <prop key="hibernate.show_sql">true</prop>
        </props>
    </property>
  </bean>

  <context:component-scan
          base-package="com.apress.prospring5.ch8" />
</beans>
```

Для приведенной выше конфигурации вполне возможно создать равнозначную конфигурацию с помощью конфигурационных файлов Java:

```
package com.apress.prospring5.ch8.config;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context
        .annotation.ComponentScan;
import org.springframework.context
        .annotation.Configuration;
import org.springframework.jdbc.datasource
        .embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource
        .embedded.EmbeddedDatabaseType;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa
        .LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor
        .HibernateJpaVendorAdapter;
import org.springframework.transaction
        .PlatformTransactionManager;
import org.springframework.transaction.annotation
        .EnableTransactionManagement;
import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;
import java.util.Properties;

@Configuration
```

```
@EnableTransactionManagement
@ComponentScan(basePackages =
              {"com.apress.prospring5.ch8.service"})
public class JpaConfig {

  private static Logger logger =
          LoggerFactory.getLogger(JpaConfig.class);
  @Bean
  public DataSource dataSource() {
    try {
      EmbeddedDatabaseBuilder dbBuilder =
              new EmbeddedDatabaseBuilder();
      return dbBuilder.setType(EmbeddedDatabaseType.H2)
             .addScripts("classpath:db/schema.sql",
                         "classpath:db/test-data.sql")
             .build();
    } catch (Exception e) {
      logger.error("Embedded DataSource bean cannot "
                + "be created!", e);
      return null;
    }
  }

  @Bean
  public PlatformTransactionManager transactionManager() {
    return new JpaTransactionManager(entityManagerFactory());
  }

  @Bean
  public JpaVendorAdapter jpaVendorAdapter() {
    return new HibernateJpaVendorAdapter();
  }

  @Bean
  public Properties hibernateProperties() {
    Properties hibernateProp = new Properties();
    hibernateProp.put("hibernate.dialect",
                      "org.hibernate.dialect.H2Dialect");
    hibernateProp.put("hibernate.format_sql", true);
    hibernateProp.put("hibernate.use_sql_comments", true);
    hibernateProp.put("hibernate.show_sql", true);
    hibernateProp.put("hibernate.max_fetch_depth", 3);
    hibernateProp.put("hibernate.jdbc.batch_size", 10);
    hibernateProp.put("hibernate.jdbc.fetch_size", 50);
    return hibernateProp;
  }

  @Bean
  public EntityManagerFactory entityManagerFactory() {
```

```
    LocalContainerEntityManagerFactoryBean factoryBean =
         new LocalContainerEntityManagerFactoryBean();
    factoryBean.setPackagesToScan(
                "com.apress.prospring5.ch8.entities");
    factoryBean.setDataSource(dataSource());
    factoryBean.setJpaVendorAdapter(
                new HibernateJpaVendorAdapter());
    factoryBean.setJpaProperties(hibernateProperties());
    factoryBean.setJpaVendorAdapter(jpaVendorAdapter());
    factoryBean.afterPropertiesSet();
    return factoryBean.getNativeEntityManagerFactory();
  }
}
```

В приведенных выше конфигурациях объявлено несколько компонентов Spring Beans, предназначенных для поддержки конфигурации класса `LocalContainerEntityManagerFactoryBean` вместе с библиотекой Hibernate в качестве поставщика услуг сохраняемости. Ниже описаны основные компоненты такой конфигурации.

- **Компонент `dataSource`.** Источник данных объявлен со встроенной базой данных H2. А поскольку это встроенная база данных, указывать ее имя не требуется.
- **Компонент `transactionManager`.** Компоненту типа `EntityManagerFactory` требуется диспетчер транзакций для транзакционного доступа к данным. В каркасе Spring предоставляется диспетчер транзакций (`org.springframework.orm.jpa.JpaTransactionManager`) специально для прикладного интерфейса JPA. Этот компонент Spring Bean объявляется с идентификатором `transactionManager`. Более подробно транзакции рассматриваются в главе 9. А дескриптор `<tx:annotation-driven>` служит для поддержки объявления требований к разметке границ транзакций с помощью аннотаций. Разнозначная аннотация `@EnableTransactionManagement` должна быть размещена в классе, снабженном аннотацией `@Configuration`.
- **Дескриптор `<component-scan>`.** Он должен быть вам уже знаком. В нем каркасу Spring предписывается просмотреть компоненты в пакете `com.apress.prospring5.ch8`.
- **Компонент типа `EntityManagerFactory` прикладного интерфейса JPA.** Компонент `emf` является наиболее важным в данной конфигурации. Прежде всего, он объявляется для применения класса `LocalContainerEntityManagerFactoryBean`. В этом компоненте предоставляются различные свойства. Во-первых, как и следовало ожидать, необходимо внедрить компонент типа `DataSource`. Во-вторых, в свойстве `jpaVendorAdapter` указывается класс `HibernateJpaVendorAdapter`, поскольку применяется библиотека Hibernate. И, в-третьих, фабрике сущностей предписывается просмотреть объекты предметной области с аннотациями преобразований ORM в пакете

com.apress.prospring5.ch8 (указанном в дескрипторе <property name="packagesToScan">). Обратите внимание на то, что такая возможность доступна только с версии Spring 3.1, и благодаря поддержке просмотра классов предметной области можно опустить определение единицы сохраняемости в файле META-INF/persistence.xml. И, наконец, в свойстве jpaProperties задаются подробности конфигурации для поставщика услуг сохраняемости из библиотеки Hibernate. Параметры данной конфигурации такие же, как и в главе 7, поэтому они не поясняются здесь снова.

Применение аннотаций JPA для преобразований ORM

Библиотека Hibernate оказала большое влияние на структуру прикладного интерфейса JPA. Аннотации преобразований в JPA очень похожи на те, которые применялись в главе 7 для преобразования объектов предметной области в записи базы данных. Если проанализировать исходный код классов предметной области в главе 7, то можно заметить, что все аннотации преобразований находятся в пакете javax.persistence, а это означает, что они уже совместимы с JPA.

Как только компонент типа EntityManagerFactory будет сконфигурирован надлежащим образом, внедрить его в свои классы не составит большого труда. Ниже приведен исходный код класса SingerServiceImpl, который послужит в качестве примера для выполнения операций в базе данных через прикладной интерфейс JPA.

```
package com.apress.prospring5.ch8.service;

import com.apress.prospring5.ch8.entities.Singer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Service;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
import org.apache.commons.lang3.NotImplementedException;
import java.util.List;
import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;
import javax.persistence.TypedQuery;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

@Service("jpaSingerService")
@Repository
@Transactional
public class SingerServiceImpl implements SingerService {
  final static String ALL_SINGER_NATIVE_QUERY =
        "select id, first_name, last_name, birth_date, "
      + " version from singer";
```

```
  private static Logger logger =
          LoggerFactory.getLogger(SingerServiceImpl.class);

  @PersistenceContext
  private EntityManager em;
  @Transactional(readOnly=true)
  @Override
  public List<Singer> findAll() {
    throw new NotImplementedException("findAll");
  }

  @Transactional(readOnly=true)
  @Override
  public List<Singer> findAllWithAlbum() {
    throw new NotImplementedException("findAllWithAlbum");
  }

  @Transactional(readOnly=true)
  @Override
  public Singer findById(Long id) {
    throw new NotImplementedException("findById");
  }

  @Override
  public Singer save(Singer singer) {
   throw new NotImplementedException("save");
  }

  @Override
  public void delete(Singer singer) {
    throw new NotImplementedException("delete");
  }

  @Transactional(readOnly=true)
  @Override
  public List<Singer> findAllByNativeQuery() {
    throw new NotImplementedException(
              "findAllByNativeQuery");
  }
}
```

В данном классе применяется несколько аннотаций. В частности, аннотация `@Service` служит для обозначения данного класса как компонента Spring Bean, который предоставляет услуги бизнес-логики другому уровню. Этому компоненту Spring Bean присваивается имя `jpaContactService`. Аннотация `@Repository` указывает на то, что в данном классе содержится логика доступа к данным, а каркасу Spring предписывается преобразовывать характерные для поставщика услуг исключения в иерархию исключений типа `DataAccessException`, определенную в Spring. Как

вам должно быть уже известно, аннотация `@Transactional` служит для определения требований к транзакциям.

Для внедрения интерфейса `EntityManager` служит аннотация `@PersistenceContext`, которая является стандартной аннотацией JPA для внедрения диспетчера сущностей. У вас может возникнуть вопрос: почему для внедрения диспетчера сущностей применяется аннотация под именем `@PersistenceContext`? Но если принять во внимание, что сам контекст сохраняемости находится под управлением интерфейса `EntityManager`, то такое имя аннотации обретает истинный смысл. Если в приложении присутствует немало единиц сохраняемости, в аннотацию можно также ввести атрибут `unitName`, чтобы указать внедряемую единицу сохраняемости. Обычно единица сохраняемости представляет отдельный источник данных типа `DataSource` в СУРБД.

Выполнение операций в базе данных через прикладной интерфейс JPA

В этом разделе будет показано, как выполнять операции в базе данных через прикладной интерфейс JPA. Ниже приведен интерфейс `SingerService`, в котором определяются услуги по предоставлению сведений о певцах.

```
package com.apress.prospring5.ch8.service;

import com.apress.prospring5.ch8.entities.Singer;
import java.util.List;

public interface SingerService {
  List<Singer> findAll();
  List<Singer> findAllWithAlbum();
  Singer findById(Long id);
  Singer save(Singer singer);
  void delete(Singer singer);
  List<Singer> findAllByNativeQuery();
}
```

Этот интерфейс очень прост. В нем определяются три метода поиска, один метод сохранения и один метод удаления. В частности, метод сохранения служит для выполнения операций вставки и обновления.

Запрашивание данных на языке JPQL

Синтаксис языков JPQL и HQL очень похож, и на самом деле всеми запросами HQL, которые демонстрировались в примерах из главы 7, можно снова воспользоваться для реализации трех методов поиска в интерфейсе `SingerService`. Для применения прикладного интерфейса JPA и библиотеки Hibernate в данный проект необходимо внедрить следующие зависимости:

```
// Файл конфигурации pro-spring-15/build.gradle
ext {
    hibernateVersion = '5.2.10.Final'
    hibernateJpaVersion = '1.0.0.Final'
    ...
    hibernate = [
    em : "org.hibernate:hibernate-entitymanager:
          $hibernateVersion",
    jpaApi : "org.hibernate.javax.persistence:
              hibernate-jpa-2.1-api:$hibernateJpaVersion"
    ]
}

// Файл конфигурации chapter08.gradle
dependencies {
    // Эти зависимости указываются для всех подмодулей,
    // кроме модуля начальной загрузки, который
    // определяется отдельно
    if !project.name.contains"boot" {
      compile spring.contextSupport, spring.orm,
      spring.context, misc.slf4jJcl, misc.logback,
      db.h2, misc.lang3, hibernate.em, hibernate.jpaApi
    }
    testCompile testing.junit
}
```

Ниже еще раз приводится исходный код классов для реализации модели объектов предметной области типа `Singer`.

```
// Исходный файл Singer.java
package com.apress.prospring5.ch8.entities;

import static javax.persistence.GenerationType.IDENTITY;
import java.io.Serializable;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;
import javax.persistence.Version;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.persistence.OneToMany;
import javax.persistence.ManyToMany;
import javax.persistence.JoinTable;
import javax.persistence.JoinColumn;
```

```
import javax.persistence.CascadeType;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.SqlResultSetMapping;
import javax.persistence.EntityResult;

@Entity
@Table(name = "singer")
@NamedQueries({
      @NamedQuery(name=Singer.FIND_ALL,
                  query="select s from Singer s"),
      @NamedQuery(name=Singer.FIND_SINGER_BY_ID,
                  query="select distinct s from Singer s "
                      + "left join fetch s.albums a "
                      + "left join fetch s.instruments i "
                      + "where s.id = :id"),
      @NamedQuery(name=Singer.FIND_ALL_WITH_ALBUM,
                  query="select distinct s from Singer s "
                      + "left join fetch s.albums a "
                      + "left join fetch s.instruments i")
})
@SqlResultSetMapping(
    name="singerResult",
    entities=@EntityResult(entityClass=Singer.class)
)
public class Singer implements Serializable {

  public static final String FIND_ALL = "Singer.findAll";
  public static final String FIND_SINGER_BY_ID =
         "Singer.findById";
  public static final String FIND_ALL_WITH_ALBUM =
         "Singer.findAllWithAlbum";

  @Id
  @GeneratedValue(strategy = IDENTITY)
  @Column(name = "ID")
  private Long id;

  @Version
  @Column(name = "VERSION")
  private int version;

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

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

  @Temporal(TemporalType.DATE)
```

```
  @Column(name = "BIRTH_DATE")
  private Date birthDate;

  @OneToMany(mappedBy = "singer", cascade=CascadeType.ALL,
             orphanRemoval=true)
  private Set<Album> albums = new HashSet<>();

  @ManyToMany
  @JoinTable(name = "singer_instrument",
             joinColumns = @JoinColumn(name = "SINGER_ID"),
             inverseJoinColumns =
                    @JoinColumn(name = "INSTRUMENT_ID"))
  private Set<Instrument> instruments = new HashSet<>();

  // Методы установки и получения

  @Override
  public String toString() {
    return "Singer - Id: " + id + ", First name: "
         + firstName + ", Last name: " + lastName
         + ", Birthday: " + birthDate;
  }
}

// Исходный файл Album.java
package com.apress.prospring5.ch8.entities;
import static javax.persistence.GenerationType.IDENTITY;
import java.io.Serializable;
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.persistence.*;

@Entity
@Table(name = "album")
public class Album implements Serializable {
  @Id
  @GeneratedValue(strategy = IDENTITY)
  @Column(name = "ID")
  private Long id;

  @Version
  @Column(name = "VERSION")
  private int version;

  @Column
  private String title;

  @Temporal(TemporalType.DATE)
  @Column(name = "RELEASE_DATE")
  private Date releaseDate;
```

```
  @ManyToOne
  @JoinColumn(name = "SINGER_ID")
  private Singer singer;

  public Album() {
    // требуется в JPA
  }

  public Album(String title, Date releaseDate) {
    this.title = title;
    this.releaseDate = releaseDate;
  }

  // Методы установки и получения
}

// Исходный файл Instrument.java
package com.apress.prospring5.ch8.entities;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Column;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.JoinTable;
import javax.persistence.JoinColumn;
import java.util.Set;
import java.util.HashSet;

@Entity
@Table(name = "instrument")
public class Instrument implements Serializable {
  @Id
  @Column(name = "INSTRUMENT_ID")
  private String instrumentId;

  @ManyToMany
  @JoinTable(name = "singer_instrument",
      joinColumns = @JoinColumn(name = "INSTRUMENT_ID"),
      inverseJoinColumns = @JoinColumn(name = "SINGER_ID"))

  private Set<Singer> singers = new HashSet<>();

  // Методы установки и получения
}
```

Если проанализировать запросы, определенные с помощью аннотации `@NamedQuery`, то можно обнаружить отсутствие видимого отличия между запросами HQL и

JPQL. Итак, начнем с метода `findAll()`, в котором сведения обо всех певцах извлекаются из базы данных, как показано ниже.

```
package com.apress.prospring5.ch8.service;
...
@Service("jpaSingerService")
@Repository
@Transactional
public class SingerServiceImpl implements SingerService {
  final static String ALL_SINGER_NATIVE_QUERY =
        "select id, first_name, last_name, birth_date, "
      + "version from singer";

  private static Logger logger = LoggerFactory.getLogger(
                SingerServiceImpl.class);

  @PersistenceContext
  private EntityManager em;
  @Transactional(readOnly=true)
  @Override
  public List<Singer> findAll() {
    return em.createNamedQuery(Singer.FIND_ALL,
                               Singer.class)
             .getResultList();
  }
  ...
}
```

В приведенном выше коде вызывается метод `EntityManager.createNamedQuery()`, которому передается имя запроса и ожидаемый тип возвращаемого результата. В данном случае диспетчер сущностей типа `EntityManager` возвратит реализацию интерфейса `TypedQuery<X>`. Затем вызывается метод `TypedQuery.getResultList()` для извлечения сведений о певцах. Чтобы проверить реализацию данного метода, воспользуемся приведенным ниже тестовым классом, содержащим тестовый метод для проверки каждого реализуемого метода из прикладного интерфейса JPA.

```
package com.apress.prospring5.ch8;

import com.apress.prospring5.ch8.config.JpaConfig;
import com.apress.prospring5.ch8.entities.Singer;
import com.apress.prospring5.ch8.service.SingerService;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation
```

```
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;
import java.util.List;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

public class SingerJPATest {
  private static Logger logger =
          LoggerFactory.getLogger(SingerJPATest.class);

  private GenericApplicationContext ctx;
  private SingerService singerService;

  @Before
  public void setUp() {
    ctx = new AnnotationConfigApplicationContext(
              JpaConfig.class);
    singerService = ctx.getBean(SingerService.class);
    assertNotNull(singerService);
  }

  @Test
  public void testFindAll() {
    List<Singer> singers = singerService.findAll();
    assertEquals(3, singers.size());
    listSingers(singers);
  }

  private static void listSingers(List<Singer> singers) {
    logger.info(" ---- Listing singers:");
    for (Singer singer : singers) {
      logger.info(singer.toString());
    }
  }

  @After
  public void tearDown() {
    ctx.close();
  }
}
```

Если метод `assertEquals()` не генерирует исключение, когда тест не проходит, то выполнение тестового метода `testFindAll()` приведет к выводу на консоль следующего результата:

```
---- Listing singers:
Singer - Id: 1, First name: John, Last name: Mayer,
   Birthday: 1977-10-16
```

```
Singer - Id: 2, First name: Eric, Last name: Clapton,
   Birthday: 1945-03-30
Singer - Id: 3, First name: John, Last name: Butler,
   Birthday: 1975-04-01
```

На заметку В отношении связей в спецификации JPA утверждается, что по умолчанию поставщики услуг сохраняемости должны немедленно производить выборку связи. Но в реализации прикладного интерфейса JPA, предоставляемой в библиотеке Hibernate, по умолчанию по-прежнему применяется стратегия отложенной выборки. Поэтому, пользуясь реализацией JPA в Hibernate, совсем не обязательно определять вручную связь с отложенной выборкой. Стандартная стратегия выборки, принятая в Hibernate, отличается от той, что описана в спецификации JPA.

А теперь реализуем метод `findAllWithAlbum()`, который будет выбирать все связанные альбомы и музыкальные инструменты. Его реализация представлена в приведенном ниже коде.

```
package com.apress.prospring5.ch8.service;
...
@Service("jpaSingerService")
@Repository
@Transactional
public class SingerServiceImpl implements SingerService {
  final static String ALL_SINGER_NATIVE_QUERY =
        "select id, first_name, last_name, birth_date, "
      + "version from singer";

  private static Logger logger = LoggerFactory.getLogger(
                                 SingerServiceImpl.class);

  @PersistenceContext
  private EntityManager em;

  @Transactional(readOnly=true)
  @Override
  public List<Singer> findAllWithAlbum() {
    List<Singer> singers = em.createNamedQuery
                (Singer.FIND_ALL_WITH_ALBUM, Singer.class)
                .getResultList();
    return singers;
  }
  ...
}
```

Метод `findAllWithAlbum()` похож на метод `findAll()`, но в нем используется другой именованный запрос с активизированным предложением `left join fetch`.

Ниже показано, каким образом реализуется тестовый метод, проверяющий данный метод и выводящий сущности на консоль.

```
package com.apress.prospring5.ch8;
...
public class SingerJPATest {
  private static Logger logger =
          LoggerFactory.getLogger(SingerJPATest.class);

  private GenericApplicationContext ctx;
  private SingerService singerService;

  @Before
  public void setUp() {
    ctx = new AnnotationConfigApplicationContext(
              JpaConfig.class);
    singerService = ctx.getBean(SingerService.class);
    assertNotNull(singerService);
  }

  @Test
  public void testFindAllWithAlbum() {
    List<Singer> singers = singerService.findAllWithAlbum();
    assertEquals(3, singers.size());
    listSingersWithAlbum(singers);
  }

  private static void listSingersWithAlbum(
                      List<Singer> singers) {
    logger.info(" ---- Listing singers with instruments:");
    for (Singer singer : singers) {
      logger.info(singer.toString());
      if (singer.getAlbums() != null) {
        for (Album album : singer.getAlbums()) {
          logger.info("\t" + album.toString());
        }
      }
      if (singer.getInstruments() != null) {
        for (Instrument instrument :
             singer.getInstruments()) {
          logger.info("\tInstrument: "
                    + instrument.getInstrumentId());
        }
      }
    }
  }

  @After
  public void tearDown() {
```

```
    ctx.close();
  }
}
```

Если метод `assertEquals()` не генерирует исключение, когда тест не проходит, то после выполнения тестового метода `testFindAllWithAlbum()` на консоль будет выведен следующий результат:

```
INFO o.h.h.i.QueryTranslatorFactoryInitiator - HHH000397:
   Using ASTQueryTranslatorFactory
Hibernate:
   /* Singer.findAllWithAlbum */ select
      distinct singer0_.ID as ID1_2_0_,
      albums1_.ID as ID1_0_1_,
      instrument3_.INSTRUMENT_ID as INSTRUME1_1_2_,
      singer0_.BIRTH_DATE as BIRTH_DA2_2_0_,
      singer0_.FIRST_NAME as FIRST_NA3_2_0_,
      singer0_.LAST_NAME as LAST_NAM4_2_0_,
      singer0_.VERSION as VERSION5_2_0_,
      albums1_.RELEASE_DATE as RELEASE_2_0_1_,
      albums1_.SINGER_ID as SINGER_I5_0_1_,
      albums1_.title as title3_0_1_,
      albums1_.VERSION as VERSION4_0_1_,
      albums1_.SINGER_ID as SINGER_I5_0_0__,
      albums1_.ID as ID1_0_0 ,
      instrument2_.SINGER_ID as SINGER_I1_3_1__,
      instrument2_.INSTRUMENT_ID as INSTRUME2_3_1__
   from
      singer singer0_
   left outer join
      album albums1_
         on singer0_.ID=albums1_.SINGER_ID
   left outer join
      singer_instrument instrument2_
         on singer0_.ID=instrument2_.SINGER_ID
   left outer join
      instrument instrument3_
         on instrument2_.INSTRUMENT_ID=
            instrument3_.INSTRUMENT_ID
INFO ----- Listing singers with instruments:
INFO - Singer - Id: 1, First name: John, Last name: Mayer,
       Birthday: 1977-10-16
INFO - Album - id: 2, Singer id: 1, Title: Battle Studies,
       Release Date: 2009-11-17
INFO - Album - id: 1, Singer id: 1, Title:
       The Search For Everything, Release Date: 2017-01-20
INFO - Instrument: Guitar
INFO - Instrument: Piano
INFO - Singer - Id: 3, First name: John, Last name: Butler,
```

```
       Birthday: 1975-04-01
INFO - Singer - Id: 2, First name: Eric, Last name: Clapton,
       Birthday: 1945-03-30
INFO - Album - id: 3, Singer id: 2, Title: From The Cradle,
       Release Date: 1994-09-13
INFO - Instrument: Guitar
```

Если в Hibernate активизировано протоколирование, то можно обнаружить собственный запрос, сформированный для извлечения всей информации из базы данных. А теперь рассмотрим метод `findById()`, в котором демонстрируется применение именованного запроса с именованными параметрами в JPA, а также выборка связей. Ниже показано, как реализуется этот метод.

```
package com.apress.prospring5.ch8.service;
...
@Service("jpaSingerService")
@Repository
@Transactional
public class SingerServiceImpl implements SingerService {
  final static String ALL_SINGER_NATIVE_QUERY =
        "select id, first_name, last_name, birth_date, "
      + "version from singer";

  private static Logger logger =
          LoggerFactory.getLogger(SingerServiceImpl.class);

  @PersistenceContext
  private EntityManager em;

  @Transactional(readOnly=true)
  @Override
  public Singer findById(Long id) {
    TypedQuery<Singer> query = em.createNamedQuery
              (Singer.FIND_SINGER_BY_ID, Singer.class);

    query.setParameter("id", id);
    return query.getSingleResult();
  }
  ...
}
```

Метод `EntityManager.createNamedQuery(java.lang.String name, java.lang.Class<T> resultClass)` вызывается для получения экземпляра реализации интерфейса `TypedQuery<T>`, который гарантирует, что результат запроса должен относиться к типу `Singer`. Затем с помощью метода `TypedQuery<T>.setParameter()` устанавливаются значения именованных параметров в самом запросе, после чего вызывается метод `getSingleResult()`, поскольку результат должен

содержать только одиночный объект типа `Singer` с указанным идентификатором. Тестирование этого метода мы оставляем вам в качестве упражнения.

Запрос нетипизированных результатов

Зачастую возникает потребность отправить запрос в базу данных и обработать полученные результаты по собственному усмотрению, а не сохранять их в преобразованном классе сущности. Характерным тому примером может служить веб-ориентированный отчет, в котором перечислены только определенные столбцы из нескольких таблиц.

Допустим, имеется веб-страница, на которой отображаются сведения обо всех певцах и их альбомах. В итоговые сведения о каждом певце входит Ф.И.О. и название его последнего альбома. А сведения о певцах, не имеющих альбомы, не отображаются. Такой вариант использования можно реализовать с помощью запроса, чтобы затем манипулировать результирующим набором вручную.

С этой целью создадим новый класс `SingerSummaryUntypeImpl` с методом `displayAllSingerSummary()`. Ниже приведена типичная реализация этого метода.

```
package com.apress.prospring5.ch8.service;

import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.Iterator;
import java.util.List;

@Service("singerSummaryUntype")
@Repository
@Transactional
public class SingerSummaryUntypeImpl {

  @PersistenceContext
  private EntityManager em;
  @Transactional(readOnly = true)
  public void displayAllSingerSummary() {
    List result = em.createQuery(
         "select s.firstName, s.lastName, "
       + "a.title from Singer s "
       + "left join s.albums a "
       + "where a.releaseDate=(select max(a2.releaseDate) "
       + "from Album a2 where a2.singer.id = s.id)")
       .getResultList();
    int count = 0;
    for (Iterator i = result.iterator(); i.hasNext(); ) {
      Object[] values = (Object[]) i.next();
```

```
      System.out.println(++count + ": " + values[0] + ", "
                         + values[1] + ", " + values[2]);
    }
  }
}
```

Как следует из приведенного выше кода, для составления запроса вызывается метод `EntityManager.createQuery()`, которому передается оператор JPQL, а затем получается результирующий список. Если выбираемые столбцы указаны в операторе JPQL явно, то прикладной интерфейс JPA возвратит итератор, где каждый элемент представлен массивом объектов. И тогда с помощью итератора организуется цикл, где отображается значение каждого элемента из массива объектов. Каждый массив объектов соответствует записи в результирующем наборе, представленном объектом типа `ResultSet`. Ниже приведена тестовая программа для проверки кода, реализующего описанную здесь логику обработки извлекаемых данных.

```
package com.apress.prospring5.ch8;

import com.apress.prospring5.ch8.config.JpaConfig;
import com.apress.prospring5.ch8.service
          .SingerSummaryUntypeImpl;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;
import java.util.List;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

public class SingerSummaryJPATest {
  private static Logger logger = LoggerFactory.getLogger(
                SingerSummaryJPATest.class);
  private GenericApplicationContext ctx;
  private SingerSummaryUntypeImpl singerSummaryUntype;

  @Before
  public void setUp() {
    ctx = new AnnotationConfigApplicationContext(
              JpaConfig.class);
    singerSummaryUntype =
          ctx.getBean(SingerSummaryUntypeImpl.class);
    assertNotNull(singerSummaryUntype);
  }
```

```
  @Test
  public void testFindAllUntype() {
    singerSummaryUntype.displayAllSingerSummary();
  }

  @After
    public void tearDown() {
    ctx.close();
  }
}
```

Выполнение этой тестовой программы дает следующий результат:

```
1: John, Mayer, The Search For Everything
2: Eric, Clapton, From The Cradle
```

В прикладном интерфейсе JPA имеется более изящное решение, чем обработка массива объектов, возвращаемого по запросу. Именно это решение мы и обсудим в следующем разделе.

Запрашивание результатов специального типа с помощью выражения конструктора

При запрашивании результатов специального типа, подобного рассмотренному в предыдущем разделе, прикладному интерфейсу JPA можно предписать сконструировать простой объект POJO непосредственно из каждой записи. Продолжая пример из предыдущего раздела, создадим простой объект POJO под названием `SingerSummary`, в котором хранятся результаты запроса итоговых сведений о певце. Исходный код класса `SingerSummary`, реализующего подобные объекты POJO, приведен ниже.

```
package com.apress.prospring5.ch8.view;

import java.io.Serializable;

public class SingerSummary implements Serializable {
  private String firstName;
  private String lastName;
  private String latestAlbum;

  public SingerSummary(String firstName, String lastName,
                       String latestAlbum) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.latestAlbum = latestAlbum;
  }

  public String getFirstName() {
```

```
    return firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public String getLatestAlbum() {
    return latestAlbum;
  }

  public String toString() {
    return "First name: " + firstName
         + ", Last Name: " + lastName
         + ", Most Recent Album: " + latestAlbum;
  }
}
```

В классе `SingerSummary` имеются свойства для итоговых сведений о певце, а также конструктор, принимающий все эти свойства. Имея в своем распоряжении класс `SingerSummary`, можно переделать метод, воспользовавшись выражением конструктора в запросе, чтобы предписать поставщику услуг JPA преобразовать результирующий набор типа `ResultSet` в объект типа `SingerSummary`. Прежде всего создадим интерфейс для класса `SingerSummary`:

```
package com.apress.prospring5.ch8.service;

import com.apress.prospring5.ch8.view.SingerSummary;
import java.util.List;

public interface SingerSummaryService {
  List<SingerSummary> findAll();
}
```

Ниже приведена реализация метода `SingerSummaryImpl.findAll()`, в которой используется выражение конструктора для преобразования результирующего набора типа `ResultSet`.

```
package com.apress.prospring5.ch8.service;

import com.apress.prospring5.ch8.view.SingerSummary;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction
          .annotation.Transactional;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.List;
```

```
@Service("singerSummaryService")
@Repository
@Transactional
public class SingerSummaryServiceImpl
       implements SingerSummaryService {

  @PersistenceContext
  private EntityManager em;

  @Transactional(readOnly = true)
  @Override

  public List<SingerSummary> findAll() {
    List<SingerSummary> result = em.createQuery(
        "select new com.apress.prospring5.ch8
            .view.SingerSummary("s.firstName,"
              + " s.lastName, a.title) from Singer s "
              + "left join s.albums a where a.releaseDate="
              + "(select max(a2.releaseDate):"
              + "from Album a2 where a2.singer.id = s.id)",
        SingerSummary.class).getResultList();
    return result;
  }
}
```

В операторе JPQL указывается ключевое слово `new` вместе с полностью уточненным именем класса `SingerSummary`, представляющего простые объекты POJO и предназначенного для сохранения результатов. В качестве аргумента конструктору класса `SingerSummary` передаются выбираемые атрибуты. И, наконец, объект класса `SingerSummary` передается методу `createQuery()` для обозначения типа результата. Ниже приведена тестовая программа для проверки кода, реализующего описанную здесь логику запрашивания результата специального типа.

```
package com.apress.prospring5.ch8;

import com.apress.prospring5.ch8.config.JpaConfig;
import com.apress.prospring5.ch8.service
         .SingerSummaryService;
import com.apress.prospring5.ch8.view.SingerSummary;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation
         .AnnotationConfigApplicationContext;
import org.springframework.context.support
         .GenericApplicationContext;
import java.util.List;
```

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

public class SingerSummaryJPATest {

  private static Logger logger =
    LoggerFactory.getLogger(SingerSummaryJPATest.class);
  private GenericApplicationContext ctx;
  private SingerSummaryService singerSummaryService;

  @Before
  public void setUp() {
    ctx = new AnnotationConfigApplicationContext(
              JpaConfig.class);
    singerSummaryService =
          ctx.getBean(SingerSummaryService.class);
    assertNotNull(singerSummaryService);
  }

  @Test
  public void testFindAll() {
    List<SingerSummary> singers =
         singerSummaryService.findAll();
    listSingerSummary(singers);
    assertEquals(2, singers.size());
  }

  private static void listSingerSummary(
                List<SingerSummary> singers) {
    logger.info(" ---- Listing singers summary:");
    for (SingerSummary singer : singers) {
      logger.info(singer.toString());
    }
  }

  @After
  public void tearDown() {
    ctx.close();
  }
}
```

При выполнении этой тестовой программы на консоль будет выведен списком каждый объект типа `SingerSummary`, как показано ниже, где остальные результаты опущены. Как видите, выражение конструктора очень удобно для преобразования результата специального запроса в простые объекты POJO для дальнейшей обработки в приложении.

```
INFO ---- Listing singers summary:
INFO - First name: John, Last Name: Mayer,
```

```
       Most Recent Album: The Search For Everything
INFO - First name: Eric, Last Name: Clapton,
       Most Recent Album: From The Cradle
```

Вставка данных

Вставка данных средствами JPA осуществляется очень просто. Как и в библиотеке Hibernate, в прикладном интерфейсе JPA поддерживается извлечение первичного ключа, генерируемого базой данных. Ниже приведен исходный код, в котором реализуется метод `save()`.

```
package com.apress.prospring5.ch8.service;
...
@Service("jpaSingerService")
@Repository
@Transactional
public class SingerServiceImpl implements SingerService {
  final static String ALL_SINGER_NATIVE_QUERY =
        "select id, first_name, last_name, birth_date, "
      + "version from singer";

  private static Logger logger =
          LoggerFactory.getLogger(SingerServiceImpl.class);

  @PersistenceContext
  private EntityManager em;

  @Override
  public Singer save(Singer singer) {
    if (singer.getId() == null) {
      logger.info("Inserting new singer");
      em.persist(singer);
    } else {
      em.merge(singer);
      logger.info("Updating existing singer");
    }
    logger.info("Singer saved with id: " + singer.getId());
    return singer;
  }
  ...
}
```

Как следует из приведенного выше кода, в методе `save()` сначала по значению `id` проверяется, является ли объект новым экземпляром сущности. Если `id = null` (т.е. идентификатор еще не назначен), значит, это новый экземпляр сущности и будет вызван метод `EntityManager.persist()`. Когда вызывается метод `persist()`, диспетчер сущностей типа `EntityManager` сохраняет сущность и делает ее управляемым экземпляром в контексте сохраняемости. Если же значение `id` существует, зна-

чит, выполняется обновление и будет вызван метод `EntityManager.merge()`. При вызове метода `merge()` диспетчер сущностей типа `EntityManager` объединяет состояние сущности с текущим контекстом сохраняемости.

Ниже приведен код для вставки новой записи о певце. Все это делается в тестовом методе, поскольку в данном случае требуется проверить, была ли операция вставки удачной.

```
package com.apress.prospring5.ch8;
...
public class SingerJPATest {
  private static Logger logger =
          LoggerFactory.getLogger(SingerJPATest.class);

  private GenericApplicationContext ctx;
  private SingerService singerService;

  @Before
  public void setUp() {
    ctx = new AnnotationConfigApplicationContext(
              JpaConfig.class);
    singerService = ctx.getBean(SingerService.class);
    assertNotNull(singerService);
  }

  @Test
  public void testInsert() {
    Singer singer = new Singer();
    singer.setFirstName("BB");
    singer.setLastName("King");
    singer.setBirthDate(new Date(
           (new GregorianCalendar(1940, 8, 16))
           .getTime().getTime()));

    Album album = new Album();
    album.setTitle("My Kind of Blues");
    album.setReleaseDate(new java.sql.Date(
          (new GregorianCalendar(1961, 7, 18))
          .getTime().getTime()));
    singer.addAbum(album);

    album = new Album();
   album.setTitle("A Heart Full of Blues");
   album.setReleaseDate(new java.sql.Date(
         (new GregorianCalendar(1962, 3, 20))
         .getTime().getTime()));
    singer.addAbum(album);
    singerService.save(singer);
    assertNotNull(singer.getId());
```

```
    List<Singer> singers =
         singerService.findAllWithAlbum();
    assertEquals(4, singers.size());
    listSingersWithAlbum(singers);
  }
  ...

  @After
  public void tearDown(){
    ctx.close();
  }
}
```

В приведенном выше коде тестовой программы сначала создается новый объект, представляющий певца, затем в него вводятся два альбома, а далее этот объект сохраняется в базе данных. После этого снова выводится список всех певцов и проверяется правильность количества записей в таблице. Выполнение этой тестовой программы дает следующий результат:

```
INFO - ---- Listing singers with instruments:
INFO - Singer - Id: 4, First name: BB, Last name: King,
       Birthday: 1940-09-16
INFO - Album - id: 5, Singer id: 4,
       Title: A Heart Full of Blues, Release Date: 1962-04-20
INFO - Album - id: 4, Singer id: 4, Title: My Kind of Blues,
       Release Date: 1961-08-18
INFO - Singer - Id: 1, First name: John, Last name: Mayer,
       Birthday: 1977-10-16
INFO - Album - id: 1, Singer id: 1,
       Title: The Search For Everything,
       Release Date: 2017-01-20
INFO - Album - id: 2, Singer id: 1, Title: Battle Studies,
       Release Date: 2009-11-17
INFO - Instrument: Piano
INFO - Instrument: Guitar
INFO - Singer - Id: 3, First name: John, Last name: Butler,
       Birthday: 1975-04-01
INFO - Singer - Id: 2, First name: Eric, Last name: Clapton,
       Birthday: 1945-03-30
INFO - Album - id: 3, Singer id: 2, Title: From The Cradle,
       Release Date: 1994-09-13
INFO - Instrument: Guitar
```

Как следует из протокольной записи `INFO`, значение `id` вновь сохраненного контакта установлено правильно. Библиотека Hibernate также выводит операторы SQL, выполняемые в базе данных.

Обновление данных

Обновление данных реализуется так же просто, как и их вставка. Обратимся к конкретному примеру. Допустим, для певца с идентификатором 1 необходимо обновить имя и удалить альбом. Чтобы проверить операцию обновления, введем в исходный код класса SingerJPATest метод testUpdate(), как показано ниже.

```
package com.apress.prospring5.ch8;
...
public class SingerJPATest {
  private static Logger logger =
          LoggerFactory.getLogger(SingerJPATest.class);

  private GenericApplicationContext ctx;
  private SingerService singerService;

  @Before
  public void setUp() {
    ctx = new AnnotationConfigApplicationContext(
              JpaConfig.class);
    singerService = ctx.getBean(SingerService.class);
    assertNotNull(singerService);
  }

  @Test
  public void testUpdate() {
    Singer singer = singerService.findById(1L);
    // убедиться, что такой певец существует,
    // вызывав метод assertNotNull(singer);
    // убедиться, что получена предполагаемая запись,
    // вызывав метод assertEquals("Mayer",
    //                            singer.getLastName());
    // извлечь альбом
    Album album = singer.getAlbums().stream().filter(a ->
                  a.getTitle().equals("Battle Studies"))
                              .findFirst().get();
    singer.setFirstName("John Clayton");
    singer.removeAlbum(album);
    singerService.save(singer);
    listSingersWithAlbum(singerService.findAllWithAlbum());
  }
  ...

  @After
  public void tearDown() {
    ctx.close();
  }
}
```

Сначала в приведенном выше коде тестовой программы извлекается запись с идентификатором 1 и изменяется имя певца. Затем циклически перебираются объекты альбомов, чтобы извлечь объект с названием альбома *Battle Studies* и удалить его из свойства `albums` объекта данного певца. И, наконец, метод `SingerService.save()` вызывается снова. Если выполнить эту тестовую программу, то на консоль будет выведен следующий результат (остальные результаты здесь опущены):

```
---- Listing singers with instruments:
Singer - Id: 1, First name: John Clayton, Last name: Mayer,
         Birthday: 1977-10-16
Album - id: 1, Singer id: 1, Title: The Search For Everything,
        Release Date: 2017-01-20
Instrument: Piano
Instrument: Guitar
Singer - Id: 2, First name: Eric, Last name: Clapton,
         Birthday: 1945-03-30
Album - id: 3, Singer id: 2, Title: From The Cradle ,
        Release Date: 1994-09-13
Instrument: Guitar
Singer - Id: 3, First name: John, Last name: Butler,
         Birthday: 1975-04-01
```

Как видите, имя певца обновлено, а его альбом удален. Удаление альбома стало возможным потому, что в связи "один ко многим" определен атрибут `orphanRemoval=true`, предписывающий поставщику услуг JPA (Hibernate) удалить все висячие записи, существующие в базе данных, но больше не обнаруживаемые в объекте при сохранении.

```
@OneToMany(mappedBy = "singer", cascade=CascadeType.ALL,
           orphanRemoval=true)
```

Удаление данных

Удаление данных выполняется очень просто. Для этого достаточно вызвать метод `EntityManager.remove()`, передав ему объект, представляющий певца. Ниже приведен исходный код класса `SingerServiceImpl`, измененный для удаления записи о певце из базы данных.

```
package com.apress.prospring5.ch8.service;
...
@Service("jpaSingerService")
@Repository
@Transactional
public class SingerServiceImpl implements SingerService {
  final static String ALL_SINGER_NATIVE_QUERY =
        + "select id, first_name, last_name, birth_date, "
        + "version from singer";
```

```
  private static Logger logger =
          LoggerFactory.getLogger(SingerServiceImpl.class);

  @PersistenceContext
  private EntityManager em;

  @Override
  public void delete(Singer singer) {
    Singer mergedSinger = em.merge(singer);
    em.remove(mergedSinger);

    logger.info("Singer with id: " + singer.getId()
              + " deleted successfully");
  }
  ...
}
```

Прежде всего в приведенном выше коде вызывается метод `EntityManager.merge()` для объединения состояния сущности с текущим контекстом сохраняемости. Метод `merge()` возвращает управляемый экземпляр сущности. Затем вызывается метод `EntityManager.remove()`, которому передается управляемый экземпляр сущности певца. Метод `remove()` удаляет запись о певце, а также все связанные с ним сведения, включая альбомы и музыкальные инструменты, поскольку в преобразовании было задано условие `cascade=CascadeType.ALL`. Чтобы проверить операцию удаления, можно воспользоваться тестовым методом `testDelete()`, определяемым, как показано ниже.

```
package com.apress.prospring5.ch8;
...
public class SingerJPATest {
private static Logger logger =
  LoggerFactory.getLogger(SingerJPATest.class);
  private GenericApplicationContext ctx;
  private SingerService singerService;

  @Before
  public void setUp() {
    ctx = new AnnotationConfigApplicationContext(
              JpaConfig.class);
    singerService = ctx.getBean(SingerService.class);
    assertNotNull(singerService);
  }

  @Test
  public void testDelete() {
    Singer singer = singerService.findById(2l);
    // убедиться, что такой певец существует
    assertNotNull(singer);
```

```
    singerService.delete(singer);
    listSingersWithAlbum(singerService.findAllWithAlbum());
  }
  ...

  @After
  public void tearDown(){
    ctx.close();
  }
}
```

В исходном коде приведенной выше программы из базы данных извлекается запись о певце с идентификатором 1, после чего вызывается метод `delete()` для удаления сведений об этом певце. Выполнение этой программы дает приведенный ниже результат. Как видите, запись о певце с идентификатором 1 удалена из базы данных.

```
---- Listing singers with instruments:
Singer - Id: 1, First name: John, Last name: Mayer,
         Birthday: 1977-10-16
Album - id: 1, Singer id: 1, Title: The Search For Everything,
        Release Date: 2017-01-20
Album - id: 2, Singer id: 1, Title: Battle Studies,
        Release Date: 2009-11-17
Instrument: Piano
Instrument: Guitar
Singer - Id: 3, First name: John, Last name: Butler,
         Birthday: 1975-04-01
```

Применение собственного запроса

Обсудив выполнение элементарных операций в базе данных через прикладной интерфейс JPA, перейдем к дополнительным и более сложным вопросам. Иногда требуется полный контроль над запросом, который будет отправлен базе данных. Примером тому может служить иерархический запрос базы данных Oracle Database. Такая разновидность запроса является характерной для конкретной базы данных и называется *собственным запросом*.

В прикладном интерфейсе JPA поддерживается выполнение собственных запросов. В частности, диспетчер сущностей типа `EntityManager` отправит такой запрос базе данных в исходном виде, не производя никакого преобразования или видоизменений. Главное преимущество, которое дает применение собственных запросов в JPA, заключается в преобразовании результирующего набора типа `ResulSet` в классы сущностностей, преобразованных по принципу ORM. В двух последующих разделах будет показано, как пользоваться собственным запросом для извлечения сведений обо всех певцах и прямого преобразования результирующего набора типа `Resul Set` в объекты типа `Singer`.

Применение простого собственного запроса

Чтобы продемонстрировать, как пользоваться собственным запросом, реализуем новый метод для извлечения записей обо всех певцах из базы данных. Ниже показано, каким образом этот новый метод вводится в класс `SingerServiceImp`.

```
package com.apress.prospring5.ch8.service;
...
@Service("jpaSingerService")
@Repository
@Transactional
public class SingerServiceImpl implements SingerService {
  final static String ALL_SINGER_NATIVE_QUERY =
        "select id, first_name, last_name, birth_date, "
      + "version from singer";

  private static Logger logger =
          LoggerFactory.getLogger(SingerServiceImpl.class);

@Transactional(readOnly=true)
@Override
public List<Singer> findAllByNativeQuery() {
  return em.createNativeQuery(ALL_SINGER_NATIVE_QUERY,
            Singer.class).getResultList();
  }
  ...
}
```

Как видите, собственный запрос — это всего лишь простой оператор SQL для извлечения всех столбцов из таблицы `SINGER`. Чтобы составить и выполнить такой запрос, сначала в приведенном выше коде вызывается метод `EntityManager.createNativeQuery()`, которому передается строка запроса и тип результата. Типом результата должен быть класс преобразованной сущностный (в данном случае — класс `Singer`). Возвращаемым типом метода `createNativeQuery()` является интерфейс `Query`, где предоставляется метод `getResultList()`, реализующий операцию получения результирующего списка. Поставщик услуг JPA выполнит запрос и преобразует результирующий набор (объект типа `ResulSet`) в экземпляры сущностей, исходя из преобразований JPA, определенных в классе сущности. Выполнение упомянутого выше метода дает такой же результат, как и выполнение метода `findAll()`.

Собственный запрос с преобразованием результирующего набора SQL

Кроме преобразованного объекта предметной области, можно передавать строку, обозначающую наименование преобразования результирующего набора SQL. Преобразование результирующего набора SQL определяется на уровне класса сущности

с помощью аннотации `@SqlResultSetMapping`, как показано ниже. В свою очередь, преобразование результирующего набора SQL может состоять из одного или более преобразования сущностей и столбцов.

```
package com.apress.prospring5.ch8.entities;

import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.SqlResultSetMapping;
import javax.persistence.EntityResult;
...

@Entity
@Table(name = "singer")
@SqlResultSetMapping(
    name="singerResult",
    entities=@EntityResult(entityClass=Singer.class)
)
public class Singer implements Serializable {
  ...
}
```

Здесь для класса сущности определено преобразование результирующего набора SQL под названием `singerResult` с указанием в атрибуте `entityClass` самого класса `Singer`. В прикладном интерфейсе JPA поддерживается более сложное преобразование многих сущностей, а также преобразование вплоть до уровня отдельных столбцов.

После того как преобразование результирующего набора SQL будет определено, метод `findAllByNativeQuery()` может быть вызван с наименованием этого преобразования. Ниже приведен исходный код класса `SingerServiceImpl` с обновленным методом `findAllByNativeQuery()`. Как видите, в прикладном интерфейсе JPA поддерживается также выполнение собственных запросов с гибкими средствами преобразования результирующих наборов SQL.

```
package com.apress.prospring5.ch8.service;
...
@Service("jpaSingerService")
@Repository
@Transactional
public class SingerServiceImpl implements SingerService {
  final static String ALL_SINGER_NATIVE_QUERY =
        "select id, first_name, last_name, birth_date, "
      + "version from singer";

  private static Logger logger =
          LoggerFactory.getLogger(SingerServiceImpl.class);

@Transactional(readOnly=true)
```

```
@Override
public List<Singer> findAllByNativeQuery() {
  return em.createNativeQuery(ALL_SINGER_NATIVE_QUERY,
                        "singerResult").getResultList();
  }
  ...
}
```

Применение прикладного интерфейса JPA 2 Criteria API для запросов с критериями поиска

В большинстве приложений предоставляются пользовательские интерфейсы для поиска информации. Зачастую для этой цели в них отображается большое количество полей поиска, но пользователи вводят информацию только в некоторых из них, чтобы начать поиск. Но не так-то просто подготовить запросы для всех возможных сочетаний критериев, вводимых пользователями, из-за слишком большого их количества. И в этом случае на помощь приходит прикладной интерфейс API для составления запросов с критериями поиска.

Главным нововведением в версии JPA 2 стало появление нового строго типизированного прикладного интерфейса Criteria API для составления запросов с критериями поиска. В этом прикладном интерфейсе критерий, передаваемый запросу, основывается на метамодели преобразованных классов сущностей. В итоге каждый заданный критерий является строго типизированным, а ошибки обнаруживаются на стадии компиляции, но не на стадии выполнения.

В прикладном интерфейсе JPA Criteria API метамодель отдельного класса сущности обозначается именем этого класса с суффиксом в виде знака подчеркивания (_). Например, класс метамодели для класса сущности `Singer` называется `Singer_`. Ниже приведен исходный код класса `Singer_`.

```
package com.apress.prospring5.ch8;

import java.util.Date;
import javax.annotation.Generated;
import javax.persistence.metamodel.SetAttribute;
import javax.persistence.metamodel.SingularAttribute;
import javax.persistence.metamodel.StaticMetamodel;

@Generated(value = "org.hibernate.jpamodelgen
                    .JPAMetaModelEntityProcessor")
@StaticMetamodel(Singer.class)
public abstract class Singer_ {
  public static volatile SingularAttribute<Singer,
                String> firstName;
public static volatile SingularAttribute<Singer,
              String> lastName;
public static volatile SetAttribute<Singer, Album> albums;
```

```
public static volatile SetAttribute<Singer,
              Instrument> instruments;
public static volatile SingularAttribute<Singer, Long> id;
public static volatile SingularAttribute<Singer,
              Integer> version;
public static volatile SingularAttribute<Singer,
              Date> birthDate;
}
```

Класс метамодели снабжен аннотацией `@StaticMetamodel`, где указан атрибут преобразованного класса сущности. В этом классе объявлены все атрибуты и связанные с ними типы данных.

Программировать и сопровождать такие классы метамоделей было бы затруднительно. Правда, имеются инструментальные средства, генерирующие классы метамоделей автоматически на основании преобразований, выполняемых средствами JPA в классах сущностей. Такое инструментальное средство предоставляется в библиотеке Hibernate под названием Hibernate Metamodel Generator (Генератор метамоделей Hibernate; `http://hibernate.org/orm/tooling/`).

Порядок генерирования классов метамодели зависит от конкретного инструментального средства, применяемого для разработки и построения проекта. Поэтому настоятельно рекомендуется ознакомиться с разделом "Usage" (Применение) документации на библиотеку Hibernate (`http://docs.jboss.org/hibernate/jpamodelgen/1.3/reference/en-US/html_single/#chapter-usage`). В примерах кода из данной книги для генерирования классов метамодели применяется инструментальное средство Gradle, а также библиотека `hibernate-jpamodelgen`, внедряемая в качестве зависимости. Эта зависимость настраивается вместе со своей версией в файле конфигурации `pro-spring-15/build.gradle`, как показано ниже.

```
ext {
  ...

  // Библиотеки сохраняемости
  hibernateVersion = '5.2.10.Final'
  hibernateJpaVersion = '1.0.0.Final'
  hibernate = [
    ...
    jpaModelGen: "org.hibernate:
                hibernate-jpamodelgen:$hibernateVersion",
    jpaApi : "org.hibernate.javax.persistence:
             hibernate-jpa-2.1-api: $hibernateJpaVersion",
    querydslapt: "com.mysema.querydsl:querydsl-apt:2.7.1"
  ]
...
```

Это главная библиотека для генерирования классов метамодели. Она применяется из файла конфигурации `chapter08/jpa-criteria/build.gradle` при выполнении задачи `generateQueryDSL` в инструментальном средстве Gradle для генериро-

вания классов метамодели перед компиляцией модуля. Ниже приведено содержимое файла конфигурации `chapter08/jpa-criteria/build.gradle`.

```
sourceSets {
  generated
}

sourceSets.generated.java.srcDirs = ['src/main/generated']

configurations {
  querydslapt
}

dependencies {
  compile hibernate.querydslapt, hibernate.jpaModelGen
}

task generateQueryDSL(type: JavaCompile, group: 'build',
     description: 'Generates the QueryDSL query types') {
  source = sourceSets.main.java
  classpath = configurations.compile
            + configurations.querydslapt
  options.compilerArgs = [
    "-proc:only", "-processor",
    "org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor"
  ]
  destinationDir = sourceSets.generated
                              .java.srcDirs.iterator.next
}
compileJava.dependsOn generateQueryDSL
```

Установив порядок генерирования классов метамодели, определим запрос, принимающий Ф.И.О. для поиска певцов. Ниже приведено определение нового метода `findByCriteriaQuery()` в интерфейсе `SingerService`.

```
package com.apress.prospring5.ch8;

import java.util.List;
  public interface SingerService {
    List<Singer> findAll();
    List<Singer> findAllWithAlbum();
    Singer findById(Long id);
    Singer save(Singer singer);
    void delete(Singer singer);
    List<Singer> findAllByNativeQuery();
    List<Singer> findByCriteriaQuery(String firstName,
                                     String lastName);
}
```

В следующем фрагменте кода представлена реализация метода `findByCriteriaQuery()`, где запрос с критериями поиска составляется с помощью прикладного интерфейса JPA 2 Criteria API:

```
package com.apress.prospring5.ch8;

import org.springframework.stereotype.Service;
import org.springframework.stereotype.Repository;
import org.springframework.transaction
         .annotation.Transactional;
import java.util.List;
import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;
import javax.persistence.TypedQuery;
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Root;
import javax.persistence.criteria.JoinType;
import javax.persistence.criteria.Predicate;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

@Service("jpaSingerService")
@Repository
@Transactional
public class SingerServiceImpl implements SingerService {
  final static String ALL_SINGER_NATIVE_QUERY =
        "select id, first_name, last_name, birth_date, "
      + "version from singer";

  private Log log =
          LogFactory.getLog(SingerServiceImpl.class);

  @PersistenceContext
  private EntityManager em;
  ...

  @Transactional(readOnly=true)
  @Override
  public List<Singer> findByCriteriaQuery(String firstName,
                      String lastName) {
    log.info("Finding singer for firstName: " + firstName
           + " and lastName: " + lastName);
    CriteriaBuilder cb = em.getCriteriaBuilder();
    CriteriaQuery<Singer> criteriaQuery =
                  cb.createQuery(Singer.class);
    Root<Singer> singerRoot =
         criteriaQuery.from(Singer.class);
    singerRoot.fetch(Singer_.albums, JoinType.LEFT);
```

```
    singerRoot.fetch(Singer_.instruments, JoinType.LEFT);
    criteriaQuery.select(singerRoot).distinct(true);
    Predicate criteria = cb.conjunction();
    if (firstName != null) {
      Predicate p = cb.equal(singerRoot
                    .get(Singer_.firstName), firstName);
      criteria = cb.and(criteria, p);
  }

    if (lastName != null) {
      Predicate p = cb.equal(singerRoot
                    .get(Singer_.lastName), lastName);
      criteria = cb.and(criteria, p);
    }
    criteriaQuery.where(criteria);
    return em.createQuery(criteriaQuery).getResultList();
  }
}
```

Ниже анализируется по частям применение прикладного интерфейса JPA 2 Criteria API в приведенном выше фрагменте кода.

- Для извлечения экземпляра типа `CriteriaBuilder` вызывается метод `EntityManager.getCriteriaBuilder()`.
- С помощью метода `CriteriaBuilder.createQuery()`, которому объект класса `Singer` передается в качестве результирующего типа, создается типизированный запрос.
- Вызывается метод `CriteriaQuery.from()`, которому передается ссылка на класс сущности. В итоге получается корневой объект запроса, представляющий интерфейс `Root<Singer>`. Этот объект соответствует указанной сущности и формирует основу для путевых выражений в запросе.
- Два вызова метода `Root.fetch()` обеспечивают немедленную выборку связей с альбомами и музыкальными инструментами. Значение `JoinType.LEFT` второго аргумента обозначает внешнее соединение таблиц базы данных. Вызов метода `Root.fetch()` со значением `JoinType.LEFT` второго аргумента равнозначен указанию операции соединения `left join fetch` на языке JPQL.
- Вызывается метод `CriteriaQuery.select()`, которому передается корневой объект запроса в качестве результирующего типа. А вызов метода `distinct()` с логическим значением `true` означает исключение дублирующихся записей.
- В результате вызова метода `CriteriaBuilder.conjunction()` получается экземпляр типа `Predicate`. Это означает, что было объединено несколько ограничений. Экземпляр типа `Predicate` может быть как простым, так и сложным предикатом. В данном случае предикат накладывает ограничение, обозначающее критерий выборки, определяемый в выражении.

- Производится проверка аргументов, обозначающих Ф.И.О. Если значение аргумента не равно `null`, то получается новый экземпляр типа `Predicate` с помощью вызываемого метода `CriteriaBuilder.and()`. Метод `equal()` служит для указания ограничения на равенство. В этом методе вызывается метод `Root.get()`, которому передается соответствующее свойство метамодели класса сущности, на которое будет накладываться ограничение. Сконструированный предикат затем объединяется с существующим предикатом, хранящимся в переменной `criteria`, путем вызова метода `CriteriaBuilder.and()`.
- Экземпляр типа `Predicate` получается со всеми критериями и ограничениями, после чего он передается запросу в предложении `where` через вызов метода `CriteriaQuery.where()`.
- И, наконец, объект типа `CriteriaQuery` передается диспетчеру сущностей типа `EntityManager`, который составляет запрос на основе переданного объекта типа `CriteriaQuery`, выполняет его и возвращает результат.

Чтобы проверить составленный запрос с критериями поиска, необходимо внести коррективы в исходный код класса `SingerJPATest`:

```
package com.apress.prospring5.ch8;

import com.apress.prospring5.ch8.config.JpaConfig;
import com.apress.prospring5.ch8.Album;
import com.apress.prospring5.ch8.Instrument;
import com.apress.prospring5.ch8.Singer;
import com.apress.prospring5.ch8.SingerService;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;
import java.util.List;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

public class SingerJPATest {
  private static Logger logger =
          LoggerFactory.getLogger(SingerJPATest.class);

  private GenericApplicationContext ctx;
  private SingerService singerService;

  @Before
  public void setUp() {
```

```
    ctx = new AnnotationConfigApplicationContext(JpaConfig.class);
    singerService = ctx.getBean("jpaSingerService", SingerService.class);
    assertNotNull(singerService);
  }

  @Test
  public void tesFindByCriteriaQuery() {
    List<Singer> singers =
        singerService.findByCriteriaQuery("John", "Mayer");
    assertEquals(1, singers.size());
    listSingersWithAlbum(singers);
  }

  private static void listSingersWithAlbum(
                      List<Singer> singers) {
    logger.info(" ---- Listing singers with instruments:");
    singers.forEach(s -> {
      logger.info(s.toString());
      if (s.getAlbums() != null) {
        s.getAlbums().forEach(a -> logger.info("\t"
                          + a.toString()));
      }
      if (s.getInstruments() != null) {
        s.getInstruments().forEach(i -> logger.info
             ("\tInstrument: " + i.getInstrumentId()));
      }
    });
  }

  @After
  public void tearDown() {
    ctx.close();
  }
}
```

Выполнение представленной выше тестовой программы дает приведенный ниже результат. И хотя он показан не полностью, в нем содержится составленный запрос. Можете опробовать разные сочетания критериев поиска при составлении запроса или передать пустые значения `null` аргументов, чтобы понаблюдать за получаемыми в итоге результатами.

```
INFO o.h.h.i.QueryTranslatorFactoryInitiator -
     HHH000397: Using ASTQueryTranslatorFactory
INFO c.a.p.c.SingerServiceImpl -
     Finding singer for firstName: John and lastName: Mayer
Hibernate:
  select
    distinct singer0_.ID as ID1_2_0_,
    albums1_.ID as ID1_0_1_,
```

```
    instrument3_.INSTRUMENT_ID as INSTRUME1_1_2_,
    singer0_.BIRTH_DATE as BIRTH_DA2_2_0_,
    singer0_.FIRST_NAME as FIRST_NA3_2_0_,
    singer0_.LAST_NAME as LAST_NAM4_2_0_,
    singer0_.VERSION as VERSION5_2_0_,
    albums1_.RELEASE_DATE as RELEASE_2_0_1_,
    albums1_.SINGER_ID as SINGER_I5_0_1_,
    albums1_.title as title3_0_1_,
    albums1_.VERSION as VERSION4_0_1_,
    albums1_.SINGER_ID as SINGER_I5_0_0__,
    albums1_.ID as ID1_0_0__,
    instrument2_.SINGER_ID as SINGER_I1_3_1__,
    instrument2_.INSTRUMENT_ID as INSTRUME2_3_1__
  from
    singer singer0_
      left outer join
    album albums1_
      on singer0_.ID=albums1_.SINGER_ID
      left outer join
    singer_instrument instrument2_
      on singer0_.ID=instrument2_.SINGER_ID
      left outer join
    instrument instrument3_

      on instrument2_.INSTRUMENT_ID=
         instrument3_.INSTRUMENT_ID
    where
      1=1
      and singer0_.FIRST_NAME=?
      and singer0_.LAST_NAME=?
INFO c.a.p.c.SingerJPATest -
     ---- Listing singers with instruments:
INFO c.a.p.c.SingerJPATest - Singer - Id: 1,
     First name: John, Last name: Mayer,
       Birthday: 1977-10-16
INFO c.a.p.c.SingerJPATest - Album - id: 2, Singer id: 1,
     Title: Battle Studies, Release Date: 2009-11-17
INFO c.a.p.c.SingerJPATest - Album - id: 1, Singer id: 1,
     Title: The Search For Everything,
     Release Date: 2017-01-20
INFO c.a.p.c.SingerJPATest - Instrument: Guitar
INFO c.a.p.c.SingerJPATest - Instrument: Piano
```

Введение в проект Spring Data JPA

Spring Data JPA является подпроектом, разработанным в рамках проекта Spring Data. Главное назначение проекта Spring Data JPA — предоставить дополнительные

возможности для упрощения разработки приложений с помощью прикладного интерфейса JPA.

В проекте Spring Data JPA предоставляется немало основных средств, и два из них будут рассмотрены в этом разделе. Первым средством является абстракция информационного хранилища типа `Repository`, вторым — приемник сущностей, предназначенный для отслеживания в базе данных аудиторской информации по классам сущностей.

Внедрение зависимостей от библиотек Spring Data JPA

Чтобы воспользоваться средствами Spring Data JPA, в проект придется внедрить соответствующие зависимости. Ниже приведена конфигурация Gradle, требующаяся для применения проекта Spring Data JPA.

```
// Файл конфигурации pro-spring-15/build.gradle
ext {
  // Библиотеки Spring
  springVersion = '5.0.0.M5'
  bootVersion = '2.0.0.BUILD-SNAPSHOT'
  springDataVersion = '2.0.0.M2'
  ...
  spring = [
    data : "org.springframework.data:spring-data-jpa:
           $springDataVersion",
    ...
  ]
  ...
}

// Файл конфигурации chapter08/spring-data-jpa/build.gradle
dependencies {
  compile spring.aop, spring.data, misc.guava
}
```

Абстракция хранилища в Spring Data JPA для операций в базе данных

Одним из главных понятий в проекте Spring Data и всех его подпроектах является абстракция информационного хранилища типа `Repository`, относящаяся к проекту Spring Data Commons (`https://github.com/spring-projects/spring-data-commons`). На момент написания данной книги актуальной была версия 2.0.0. M2 данного проекта. В проекте Spring Data JPA абстракция информационного хранилища служит оболочкой, в которую заключается интерфейс `EntityManager` и предоставляется более простой интерфейс для доступа к данным через интерфейс JPA. Главным в Spring Data является маркерный интерфейс `org.springframework.`

data.repository.Repository<T, ID extends Serializable>, входящий в дистрибутив Spring Data Commons. В проекте Spring Data предоставляются различные расширения интерфейса Repository, одним из которых служит интерфейс org.springframework.data.repository.CrudRepository, также относящийся к проекту Spring Data Commons и рассматриваемый в этом разделе.

В интерфейсе CrudRepository предоставляется ряд часто употребляемых методов. Ниже приведено объявление этого интерфейса, взятое из исходного кода проекта Spring Data Commons.

```
package org.springframework.data.repository;

import java.io.Serializable;

@NoRepositoryBean
public interface CrudRepository<T, ID extends Serializable>
       extends Repository<T, ID> {
  long count();
  void delete(ID id);
  void delete(Iterable<? extends T> entities);
  void delete(T entity);
  void deleteAll();
  boolean exists(ID id);
  Iterable<T> findAll();
  T findOne(ID id);
  Iterable<T> save(Iterable<? extends T> entities);
  T save(T entity);
}
```

Несмотря на то что имена приведенных выше методов самоочевидны, принцип действия абстракции информационного хранилища типа Repository лучше исследовать на простом примере. С этой целью внесем некоторые коррективы в интерфейс SingerService, а точнее — в три метода поиска.

```
package com.apress.prospring5.ch8;

import com.apress.prospring5.ch8.entities.Singer;
import java.util.List;

public interface SingerService {
  List<Singer> findAll();
  List<Singer> findByFirstName(String firstName);
  List<Singer> findByFirstNameAndLastName(
               String firstName, String lastName);
}
```

Следующий шаг состоит в подготовке интерфейса SingerRepository, расширяющего интерфейс CrudRepository. Интерфейс SingerRepository объявляется следующим образом:

```
package com.apress.prospring5.ch8;

import java.util.List;
import com.apress.prospring5.ch8.entities.Singer;
import org.springframework.data.repository.CrudRepository;

public interface SingerRepository
       extends CrudRepository<Singer, Long> {
  List<Singer> findByFirstName(String firstName);
  List<Singer> findByFirstNameAndLastName(
               String firstName, String lastName);
}
```

В этом интерфейсе требуется объявить лишь два метода, поскольку реализация метода `findAll()` уже предоставляется в методе `CrudRepository.findAll()`. Как следует из приведенного выше кода, интерфейс `SingerRepository` расширяет интерфейс `CrudRepository`, передавая ему класс сущности (`Singer`) и тип идентификатора (`Long`). Одна из примечательных особенностей абстракции информационного хранилища типа `Repository` в Spring Data состоит в следующем: если соблюдать общее соглашение об именовании (например, `findByFirstName` и `findByFirstNameAndLastName`), то предоставлять именованный запрос прикладному интерфейсу Spring Data JPA не придется. Вместо этого Spring Data JPA самостоятельно выведет и составит запрос, исходя из имени метода. Например, для метода `findByFirstName()` будет автоматически подготовлен запрос `select s from Singer s where c.firstName = :firstName` и установлен именованный параметр `firstName` из его аргумента.

Чтобы воспользоваться абстракцией информационного хранилища, необходимо определить ее в конфигурации Spring. Ниже приведен соответствующий фрагмент из файла конфигурации `app-context-annotation.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context=
            "http://www.springframework.org/schema/context"
       xmlns:jdbc=
            "http://www.springframework.org/schema/jdbc"
       xmlns:jpa="http://www.springframework.org/schema
                        /data/jpa"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation=
          "http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans
                /spring-beans.xsd
          http://www.springframework.org/schema/context
          http://www.springframework.org/schema/context
                /spring-context.xsd
```

```
        http://www.springframework.org/schema/jdbc
        http://www.springframework.org/schema/jdbc
              /spring-jdbc.xsd
        http://www.springframework.org/schema/data/jpa
        http://www.springframework.org/schema/data/jpa
              /spring-jpa.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx
             /spring-tx.xsd">

<jdbc:embedded-database id="dataSource" type="H2">
  <jdbc:script location="classpath:db/schema.sql"/>
  <jdbc:script location="classpath:db/test-data.sql"/>
</jdbc:embedded-database>

<bean id="transactionManager"
   class="org.springframework.orm.jpa
             .JpaTransactionManager">
   <property name="entityManagerFactory" ref="emf"/>
</bean>

<tx:annotation-driven
    transaction-manager="transactionManager" />
<bean id="emf"
  class="org.springframework.orm.jpa
           .LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="jpaVendorAdapter">
  <bean class="org.springframework.orm.jpa.vendor
                .HibernateJpaVendorAdapter" />
  </property>
  <property name="packagesToScan"
    value="com.apress.prospring5.ch8.entities"/>
  <property name="jpaProperties">
    <props>
      <prop key="hibernate.dialect">
            org.hibernate.dialect.H2Dialect
      </prop>
      <prop key="hibernate.max_fetch_depth">3</prop>
      <prop key="hibernate.jdbc.fetch_size">50</prop>
      <prop key="hibernate.jdbc.batch_size">10</prop>
      <prop key="hibernate.show_sql">true</prop>
    </props>
  </property>
</bean>

<context:annotation-config/>
  <context:component-scan
          base-package="com.apress.prospring5.ch8" >
  <context:exclude-filter type="annotation"
```

```
        expression="org.springframework.context
                        .annotation.Configuration" />
    </context:component-scan>

  <jpa:repositories base-package="com.apress.prospring5.ch8"
      entity-manager-factory-ref="emf"
      transaction-manager-ref="transactionManager"/>
</beans>
```

Сначала в файл конфигурации вводится пространство имен `jpa`. Затем с помощью дескриптора `<jpa:repositories>` конфигурируется абстракция информационного хранилища типа `Repository` в Spring Data JPA. При этом каркасу Spring предписывается просмотреть в пакете `com.apress.prospring5.ch8` интерфейсы информационного хранилища, а также передать компонент типа `EntityManager Factory` и диспетчера транзакций соответственно.

Обратите внимание на элемент разметки `<context:exclude-filter>`, вложенный в дескриптор `<context:component-scan>`. Он введен для того, чтобы исключить из просмотра конфигурационный класс Java с аннотацией `@Configuration`, который может быть использован вместо приведенной выше конфигурации в формате XML. Исходный код этого конфигурационного класса приведен ниже.

```
package com.apress.prospring5.ch8.config;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository
          .config.EnableJpaRepositories;
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseBuilder;

import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseType;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa
          .LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor
          .HibernateJpaVendorAdapter;
import org.springframework.transaction
          .PlatformTransactionManager;
import org.springframework.transaction.annotation
          .EnableTransactionManagement;

import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;
import java.util.Properties;
```

```
@Configuration
@EnableTransactionManagement
@ComponentScan(basePackages =
            {"com.apress.prospring5.ch8"})
@EnableJpaRepositories(basePackages =
                    {"com.apress.prospring5.ch8"})
public class DataJpaConfig {

  private static Logger logger =
          LoggerFactory.getLogger(DataJpaConfig.class);

  @Bean
  public DataSource dataSource() {
    try {
      EmbeddedDatabaseBuilder dbBuilder =
              new EmbeddedDatabaseBuilder();
      return dbBuilder.setType(EmbeddedDatabaseType.H2)
             .addScripts("classpath:db/schema.sql",
             "classpath:db/test-data.sql").build();
    } catch (Exception e) {
      logger.error("Embedded DataSource bean cannot "
                + "be created!", e);
      return null;
    }
  }

  @Bean
  public PlatformTransactionManager transactionManager() {
    return new JpaTransactionManager(
              entityManagerFactory());
  }

  @Bean
  public JpaVendorAdapter jpaVendorAdapter() {
    return new HibernateJpaVendorAdapter();
  }

  @Bean
  public Properties hibernateProperties() {
    Properties hibernateProp = new Properties();
    hibernateProp.put("hibernate.dialect",
                      "org.hibernate.dialect.H2Dialect");
    hibernateProp.put("hibernate.format_sql", true);
    hibernateProp.put("hibernate.use_sql_comments", true);
    hibernateProp.put("hibernate.show_sql", true);
    hibernateProp.put("hibernate.max_fetch_depth", 3);
    hibernateProp.put("hibernate.jdbc.batch_size", 10);
    hibernateProp.put("hibernate.jdbc.fetch_size", 50);
    return hibernateProp;
```

```
  }

  @Bean
  public EntityManagerFactory entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean factoryBean =
         new LocalContainerEntityManagerFactoryBean();
    factoryBean.setPackagesToScan(
                "com.apress.prospring5.ch8.entities");
    factoryBean.setDataSource(dataSource());
    factoryBean.setJpaVendorAdapter(
            new HibernateJpaVendorAdapter());
    factoryBean.setJpaProperties(hibernateProperties());
    factoryBean.setJpaVendorAdapter(jpaVendorAdapter());
    factoryBean.afterPropertiesSet();
    return factoryBean.getNativeEntityManagerFactory();
  }
}
```

Единственным элементом конфигурации, использованным в приведенном выше коде с целью активизировать поддержку информационных хранилищ в Spring Data JPA, является аннотация `@EnableJpaRepositories`. В ее атрибуте `basePackages` указан пакет, в котором следует искать специальные расширения интерфейса `Repository` и создавать компоненты информационных хранилищ. А остальные зависимости (от компонентов `emf` и `transactionManager`) внедряются автоматически контейнером инверсии управления в Spring.

Ниже представлена реализация трех методов поиска из интерфейса `SingerService`.

```
package com.apress.prospring5.ch8;

import com.apress.prospring5.ch8.entities.Singer;
import org.springframework.stereotype.Service;
import org.springframework.transaction
          .annotation.Transactional;
import org.springframework.beans.factory
          .annotation.Autowired;
import java.util.List;
import com.google.common.collect.Lists;

@Service("springJpaSingerService")
@Transactional
public class SingerServiceImpl implements SingerService {
  @Autowired
  private SingerRepository singerRepository;

  @Transactional(readOnly=true)
  public List<Singer> findAll() {
    return Lists.newArrayList(singerRepository.findAll());
  }
```

```
  @Transactional(readOnly=true)
  public List<Singer> findByFirstName(String firstName) {
    return singerRepository.findByFirstName(firstName);
  }

  @Transactional(readOnly=true)
  public List<Singer> findByFirstNameAndLastName(
        String firstName, String lastName) {
    return singerRepository.findByFirstNameAndLastName(
                          firstName, lastName);
  }
}
```

Как видите, вместо интерфейса `EntityManager` достаточно внедрить компонент `singerRepository`, сформированный в Spring на основании интерфейса `Singer Repository`, в служебный класс, а Spring Data JPA автоматически выполнит все необходимые низкоуровневые операции. Ниже приведен исходный код тестового класса, который должен быть вам уже знаком. Если выполнить исходный код этого класса, все определенные в нем тесты должны пройти, а на консоль будет выведен вполне ожидаемый результат.

```
package com.apress.prospring5.ch8;

import com.apress.prospring5.ch8.config.DataJpaConfig;
import com.apress.prospring5.ch8.entities.Singer;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation
         .AnnotationConfigApplicationContext;
import org.springframework.context.support.GenericApplicationContext;
import java.util.List;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;

public class SingerDataJPATest {

  private static Logger logger =
          LoggerFactory.getLogger(SingerDataJPATest.class);

  private GenericApplicationContext ctx;
  private SingerService singerService;

  @Before
  public void setUp() {
    ctx = new AnnotationConfigApplicationContext(
              DataJpaConfig.class);
```

```
    singerService = ctx.getBean(SingerService.class);
    assertNotNull(singerService);
  }

  @Test
  public void testFindAll() {
    List<Singer> singers = singerService.findAll();
    assertTrue(singers.size() > 0);
    listSingers(singers);
  }

  @Test
  public void testFindByFirstName() {
    List<Singer> singers =
         singerService.findByFirstName("John");
    assertTrue(singers.size() > 0);
    assertEquals(2, singers.size());
    listSingers(singers);
  }

  @Test
  public void testFindByFirstNameAndLastName() {
    List<Singer> singers = singerService
        .findByFirstNameAndLastName("John", "Mayer");
    assertTrue(singers.size() > 0);
    assertEquals(1, singers.size());
    listSingers(singers);
  }

  private static void listSingers(List<Singer> singers) {
    logger.info(" ---- Listing singers:");
    for (Singer singer : singers) {
      logger.info(singer.toString());
    }
  }

  @After
  public void tearDown() {
    ctx.close();
  }
}
```

Применение интерфейса JpaRepository

Помимо интерфейса CrudRepository, в Spring имеется еще более развитый интерфейс JpaRepository, упрощающий создание информационных хранилищ. В этом интерфейсе предоставляются методы для выполнения операций пакетной обработки, страничного обмена и сортировки. На рис. 8.1 схематически представлено отношение между интерфейсами JpaRepository и CrudRepository. Выбор меж-

ду интерфейсами `JpaRepository` и `CrudRepository` зависит от сложности приложения. Как видите, интерфейс `JpaRepository` расширяет интерфейс `CrudRepository`, а следовательно, в первом из них предоставляются все функциональные возможности второго.

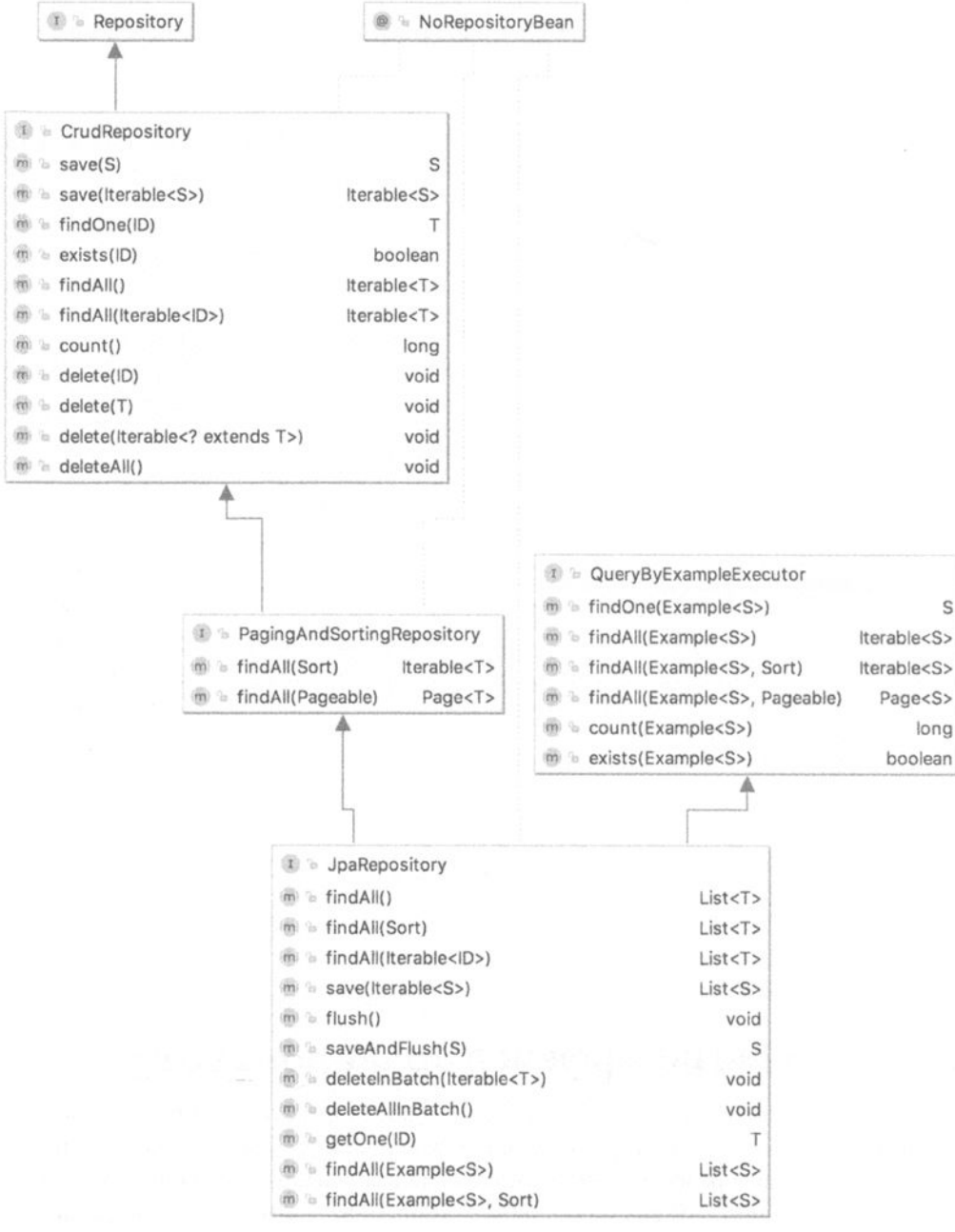


Рис. 8.1. Иерархия интерфейсов информационных хранилищ в Spring Data JPA

Специальные запросы Spring Data JPA

В сложных приложениях могут потребоваться специальные запросы, которые нельзя автоматически вывести средствами Spring. В таком случае запрос должен быть явно определен с помощью аннотации `@Query`. Воспользуемся этой аннотацией для поиска всех музыкальных альбомов, содержащих слово *The* в своем названии. С этой целью объявим интерфейс `AlbumRepository`:

```
package com.apress.prospring5.ch8.repos;

import com.apress.prospring5.ch8.entities.Album;
import com.apress.prospring5.ch8.entities.Singer;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import java.util.List;

public interface AlbumRepository
       extends JpaRepository<Album, Long> {
  List<Album> findBySinger(Singer singer);

  @Query("select a from Album a where a.title like %:title%")
  List<Album> findByTitle(@Param("title") String t);
}
```

В приведенном выше запросе присутствует параметр `title`. Если именованный параметр называется таким же образом, как и аргумент метода, снабженного аннотацией `@Query`, то аннотация `@Param` не требуется. Но если аргумент называется иначе, то аннотация `@Param` требуется для того, чтобы сообщить каркасу Spring, что значение данного параметра должно быть внедрено в именованный параметр запроса.

Ниже приведен довольно простой класс `AlbumServiceImpl`, в котором компонент `albumRepository` применяется только для вызова его методов.

```
// Исходный файл AlbumService.java

package com.apress.prospring5.ch8.services;

import com.apress.prospring5.ch8.entities.Album;
import com.apress.prospring5.ch8.entities.Singer;
import java.util.List;

public interface AlbumService {
  List<Album> findBySinger(Singer singer);
  List<Album> findByTitle(String title);
}

// Исходный файл AlbumServiceImpl.java
package com.apress.prospring5.ch8.services;
```

```
import com.apress.prospring5.ch8.entities.Album;
import com.apress.prospring5.ch8.entities.Singer;
import com.apress.prospring5.ch8.repos.AlbumRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.util.List;

@Service("springJpaAlbumService")
@Transactional
public class AlbumServiceImpl implements AlbumService {
  @Autowired
  private AlbumRepository albumRepository;

  @Transactional(readOnly=true)
  @Override public List<Album> findBySinger(Singer singer) {
    return albumRepository.findBySinger(singer);
  }

  @Override public List<Album> findByTitle(String title) {
    return albumRepository.findByTitle(title);
  }
}
```

Чтобы проверить метод `findByTitle()`, можно воспользоваться следующим тестовым классом:

```
package com.apress.prospring5.ch8;

import com.apress.prospring5.ch8.config.DataJpaConfig;
import com.apress.prospring5.ch8.entities.Album;
import com.apress.prospring5.ch8.services.AlbumService;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;

import java.util.List;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;

public class SingerDataJPATest {
  private static Logger logger =
```

```
        LoggerFactory.getLogger(SingerDataJPATest.class);

  private GenericApplicationContext ctx;
  private AlbumService albumService;

  @Before
  public void setUp() {
    ctx = new AnnotationConfigApplicationContext(
              DataJpaConfig.class);
    albumService = ctx.getBean(AlbumService.class);
    assertNotNull(albumService);
  }

  @Test
  public void testFindByTitle() {
    List<Album> albums = albumService.findByTitle("The");
    assertTrue(albums.size() > 0);
    assertEquals(2, albums.size());
    albums.forEach(a -> logger.info(a.toString()
                + ", Singer: "
                + a.getSinger().getFirstName() + " "
                + a.getSinger().getLastName()));
  }

  @After
  public void tearDown() {
    ctx.close();
  }
}
```

Если выполнить исходный код приведенного выше тестового класса, тест в методе `testFindByTitle()` пройдет и на консоль будут выведены сведения о двух обнаруженных альбомах:

```
INFO c.a.p.c.SingerDataJPATest - Album - id: 1,
     Singer id: 1,
Title: The Search For Everything, Release Date: 2017-01-20,
       Singer: John Mayer
INFO c.a.p.c.SingerDataJPATest - Album - id: 3,
     Singer id: 2,
Title: From The Cradle, Release Date: 1994-09-13,
     Singer: Eric Clapton
```

Отслеживание изменений в классе сущности

В большинстве приложений требуется отслеживать аудиторские операции над прикладными данными, которые поддерживаются пользователями. Аудиторская информация обычно включает в себя имя пользователя, сформировавшего прикладные

данные, дату их формирования, дату последней модификации, а также имя пользователя, выполнившего последнюю модификацию этих данных.

В проекте Spring Data JPA подобные функции выполняет приемник сущностей JPA, помогающий автоматически отслеживать аудиторскую информацию. Чтобы воспользоваться этим функциональным средством, в классе сущности следует реализовать интерфейс `org.springframework.data.domain.Auditable<U, ID extends Serializable>`, относящийся к проекту Spring Data Commons, или же расширить любой класс, реализующий данный интерфейс. Ниже приведено объявление интерфейса `Auditable`, взятое из документации на Spring Data.

```
package org.springframework.data.domain;

import java.io.Serializable;
import java.time.temporal.TemporalAccessor;
import java.util.Optional;

public interface Auditable<U, ID extends Serializable,
       T extends TemporalAccessor> extends Persistable<ID> {
  Optional<U> getCreatedBy();

  void setCreatedBy(U createdBy);

  Optional<T> getCreatedDate();

  void setCreatedDate(T creationDate);

  Optional<U> getLastModifiedBy();

  void setLastModifiedBy(U lastModifiedBy);

  Optional<T> getLastModifiedDate();

  void setLastModifiedDate(T lastModifiedDate);
}
```

Чтобы продемонстрировать, каким образом отслеживается аудиторская информация, создадим в схеме базы данных новую таблицу `SINGER_AUDIT` на основании таблицы `SINGER`, в которую добавлены четыре столбца, предназначенных для аудита. Ниже приведен сценарий создания этой таблицы из файла `schema.sql`.

```
CREATE TABLE SINGER_AUDIT (
    ID INT NOT NULL AUTO_INCREMENT
    , FIRST_NAME VARCHAR(60) NOT NULL
    , LAST_NAME VARCHAR(40) NOT NULL
    , BIRTH_DATE DATE
    , VERSION INT NOT NULL DEFAULT 0
    , CREATED_BY VARCHAR(20)
    , CREATED_DATE TIMESTAMP
```

```
    , LAST_MODIFIED_BY VARCHAR(20)
    , LAST_MODIFIED_DATE TIMESTAMP
    , UNIQUE UQ_SINGER_AUDIT_1 (FIRST_NAME, LAST_NAME)
    , PRIMARY KEY (ID)
);
```

Как следует из приведенного выше объявления интерфейса `Auditable`, столбцы данных ограничиваются типами, расширяющими интерфейс `java.time.temporal.TemporalAccessor`. Начиная с версии Spring 5 реализовывать интерфейс `Auditable <U, ID extends Serializable>` больше не требуется, поскольку его методы могут быть заменены аннотациями `@CreatedBy`, `@CreatedDate`, `@LastModifiedBy` и `@LastModifiedDate` соответственно. Используя эти аннотации, больше не придется накладывать ограничения на столбцы данных.

Далее необходимо создать класс сущности `SingerAudit`. Ниже приведен его исходный код.

```
package com.apress.prospring5.ch8.entities;

import org.springframework.data.annotation.CreatedBy;
import org.springframework.data.annotation.CreatedDate;
import org.springframework.data.annotation.LastModifiedBy;
import org.springframework.data.annotation.LastModifiedDate;
import org.springframework.data.jpa.domain.support
      .AuditingEntityListener;

import javax.persistence.*;
import java.io.Serializable;
import java.util.Date;
import java.util.Optional;
import static javax.persistence.GenerationType.IDENTITY;

@Entity
@EntityListeners(AuditingEntityListener.class)
@Table(name = "singer_audit")
public class SingerAudit implements Serializable {

  @Id
  @GeneratedValue(strategy = IDENTITY)
  @Column(name = "ID")
  private Long id;

  @Version
  @Column(name = "VERSION")
  private int version;

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

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

@Temporal(TemporalType.DATE)
@Column(name = "BIRTH_DATE")
private Date birthDate;

@CreatedDate
@Column(name = "CREATED_DATE")
@Temporal(TemporalType.TIMESTAMP)
private Date createdDate;

@CreatedBy
@Column(name = "CREATED_BY")
private String createdBy;

@LastModifiedBy
@Column(name = "LAST_MODIFIED_BY")
private String lastModifiedBy;

@LastModifiedDate
@Column(name = "LAST_MODIFIED_DATE")
@Temporal(TemporalType.TIMESTAMP)
private Date lastModifiedDate;

public Long getId() {
  return this.id;
}

public String getFirstName() {
  return this.firstName;
}

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

public String getLastName() {
  return this.lastName;
}

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

public Date getBirthDate() {
  return this.birthDate;
}
```

```
  public void setBirthDate(Date birthDate) {
    this.birthDate = birthDate;
  }

  public Optional<String> getCreatedBy() {
    return Optional.of(createdBy);
  }

  public void setCreatedBy(String createdBy) {
    this.createdBy = createdBy;
  }

  public Optional<Date> getCreatedDate() {
    return Optional.of(createdDate);
  }

  public void setCreatedDate(Date createdDate) {
    this.createdDate = createdDate;
  }

  public Optional<String> getLastModifiedBy() {
    return Optional.of(lastModifiedBy);
  }

  public void setLastModifiedBy(String lastModifiedBy) {
    this.lastModifiedBy = lastModifiedBy;
  }

  public Optional<Date> getLastModifiedDate() {
    return Optional.of(lastModifiedDate);
  }

  public void setLastModifiedDate(Date lastModifiedDate) {
    this.lastModifiedDate = lastModifiedDate;
  }

  public String toString() {
    return "Singer - Id: " + id + ", First name: "
         + firstName + ", Last name: " + lastName
         + ", Birthday: " + birthDate + ", Created by: "
         + createdBy + ", Create date: " + createdDate
         + ", Modified by: " + lastModifiedBy
         + ", Modified date: " + lastModifiedDate;
  }
}
```

Аннотации `@Column` применяются к аудиторским полям для преобразования в отдельные столбцы таблицы базы данных. Аннотация `@EntityListeners(AuditingEntityListener.class)` регистрирует приемник аудиторских сущностей типа

AuditingEntityListener, применяемый только к данной сущности в контексте сохраняемости. Если же в более сложных случаях требуется не один класс сущности, то функциональные средства аудита выделяются в отдельный класс, снабженный не только аннотацией @MappedSuperclass, но и аннотацией @EntityListeners (AuditingEntityListener.class). Если бы класс SingerAudit был частью подобной иерархии, его можно было бы расширить следующим классом:

```
package com.apress.prospring5.ch8.entities;

import org.springframework.data.annotation.CreatedBy;
import org.springframework.data.annotation.CreatedDate;
import org.springframework.data.annotation.LastModifiedBy;
import org.springframework.data.annotation.LastModifiedDate;
import org.springframework.data.jpa.domain.support
          .AuditingEntityListener;

import javax.persistence.*;
import java.io.Serializable;
import java.util.Date;
import java.util.Optional;

@MappedSuperclass
@EntityListeners(AuditingEntityListener.class)
public abstract class AuditableEntity<U>
       implements Serializable {
  @CreatedDate
  @Column(name = "CREATED_DATE")
  @Temporal(TemporalType.TIMESTAMP)
  protected Date createdDate;

  @CreatedBy
  @Column(name = "CREATED_BY")
  protected String createdBy;

  @LastModifiedBy
  @Column(name = "LAST_MODIFIED_BY")
  protected String lastModifiedBy;

  @LastModifiedDate
  @Column(name = "LAST_MODIFIED_DATE")
  @Temporal(TemporalType.TIMESTAMP)
  protected Date lastModifiedDate;

  public Optional<String> getCreatedBy() {
    return Optional.of(createdBy);
  }

  public void setCreatedBy(String createdBy) {
```

```
    this.createdBy = createdBy;
  }

  public Optional<Date> getCreatedDate() {
    return Optional.of(createdDate);
  }

  public void setCreatedDate(Date createdDate) {
    this.createdDate = createdDate;
  }

  public Optional<String> getLastModifiedBy() {
    return Optional.of(lastModifiedBy);
  }

  public void setLastModifiedBy(String lastModifiedBy) {
    this.lastModifiedBy = lastModifiedBy;
  }

  public Optional<Date> getLastModifiedDate() {
    return Optional.of(lastModifiedDate);
  }

  public void setLastModifiedDate(Date lastModifiedDate) {
    this.lastModifiedDate = lastModifiedDate;
  }
}
```

Благодаря этому значительно сокращается исходный код класса `SingerAudit`, как показано ниже.

```
package com.apress.prospring5.ch8.entities;

import javax.persistence.*;
import java.util.Date;
import static javax.persistence.GenerationType.IDENTITY;

@Entity
@Table(name = "singer_audit")
public class SingerAudit
       extends AuditableEntity<SingerAudit> {

  @Id
  @GeneratedValue(strategy = IDENTITY)
  @Column(name = "ID")
  private Long id;

  @Version
  @Column(name = "VERSION")
  private int version;
```

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

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

  @Temporal(TemporalType.DATE)
  @Column(name = "BIRTH_DATE")
  private Date birthDate;

  public Long getId() {
    return this.id;
  }

  public String getFirstName() {
    return this.firstName;
  }

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

  public String getLastName() {
    return this.lastName;
  }

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

  public Date getBirthDate() {
    return this.birthDate;
  }

  public void setBirthDate(Date birthDate) {
    this.birthDate = birthDate;
  }

  public String toString() {
    return "Singer - Id: " + id + ", First name: "
         + firstName + ", Last name: " + lastName
         + ", Birthday: " + birthDate + ", Created by: "
         + createdBy + ", Create date: " + createdDate
         + ", Modified by: " + lastModifiedBy
         + ", Modified date: " + lastModifiedDate;
  }
}
```

Ниже приведен интерфейс `SingerAuditService`, в котором определяются лишь те методы, которые требуются для демонстрации возможностей аудита.

```
package com.apress.prospring5.ch8.services;

import com.apress.prospring5.ch8.entities.SingerAudit;
import java.util.List;

public interface SingerAuditService {
  List<SingerAudit> findAll();
  SingerAudit findById(Long id);
  SingerAudit save(SingerAudit singer);
}
```

Интерфейс `SingerAuditRepository` просто расширяет интерфейс `CrudRepository`, где уже реализованы все методы, которыми нам предстоит воспользоваться для воплощения аудиторской службы типа `SingerAuditService`. Так, метод `findById()` реализуется в методе `CrudRepository.findOne()`. Ниже приведен служебный класс `SingerAuditServiceImpl`, реализующий данную службу.

```
package com.apress.prospring5.ch8.services;

import com.apress.prospring5.ch8.entities.SingerAudit;
import com.apress.prospring5.ch8
          .repos.SingerAuditRepository;
import org.springframework.stereotype.Service;
import org.springframework.transaction
          .annotation.Transactional;
import org.springframework.beans.factory
          .annotation.Autowired;

import java.util.List;
import com.google.common.collect.Lists;

Service("singerAuditService")
@Transactional
public class SingerAuditServiceImpl
       implements SingerAuditService {

  @Autowired
  private SingerAuditRepository singerAuditRepository;

  @Transactional(readOnly=true)
  public List<SingerAudit> findAll() {
    return Lists.newArrayList(
           singerAuditRepository.findAll());
  }

  public SingerAudit findById(Long id) {
    return singerAuditRepository.findOne(id).get();
  }
```

```
  public SingerAudit save(SingerAudit singer) {
    return singerAuditRepository.save(singer);
  }
}
```

Необходимо также немного потрудиться над конфигурацией. Так, в XML-файле конфигурации `/src/main/resources/META-INF/orm.xml` относительно корневого каталога текущего проекта следует определить приемник сущностей типа `AuditingEntityListener<T>`, предоставляющий аудиторскую службу, как показано ниже. Этот файл конфигурации должен быть обозначен именно таким именем, поскольку оно указано в спецификации JPA, хотя это справедливо только для конфигурирования до версии Spring 3.1.

```
<?xml version="1.0" encoding="UTF-8" ?>

<entity-mappings xmlns=
                "http://java.sun.com/xml/ns/persistence/orm"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation=
          "http://java.sun.com/xml/ns/persistence/orm
          http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
          version="2.0">
    <description>JPA</description>
    <persistence-unit-metadata>
      <persistence-unit-defaults>
        <entity-listeners>
          <entity-listener
             class="org.springframework.data.jpa.domain
                       .support.AuditingEntityListener" />
        </entity-listeners>
      </persistence-unit-defaults>
    </persistence-unit-metadata>
</entity-mappings>
```

Начиная с версии Spring 3.1 приведенный выше файл конфигурации можно заменить конфигурирующей аннотацией `@EntityListeners (AuditingEntityListener.class)`. Поставщик услуг JPA обнаружит указанный приемник сущностей при выполнении операций сохраняемости для обработки событий сохранения и обновления в аудиторских полях. Остальная часть конфигурации Spring аналогична представленной ранее, естественно, кроме новой конфигурирующей аннотации.

```
package com.apress.prospring5.ch8.com.apress.prospring5.ch8.config;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
```

```
import org.springframework.data.jpa.repository.config
          .EnableJpaAuditing;
import org.springframework.data.jpa.repository.config
          .EnableJpaRepositories;
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseType;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa
          .LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor
          .HibernateJpaVendorAdapter;
import org.springframework.transaction
          .PlatformTransactionManager;
import org.springframework.transaction.annotation
          .EnableTransactionManagement;

import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;
import java.util.Properties;

@Configuration
@EnableTransactionManagement
@ComponentScan(basePackages = {"com.apress.prospring5.ch8"})
@EnableJpaRepositories(basePackages =
                       {"com.apress.prospring5.ch8.repos"})
@EnableJpaAuditing(auditorAwareRef = "auditorAwareBean")

public class AuditConfig {
  // то же содержимое, что в конфигурации
  // из предыдущего раздела
  ...
}
```

Аннотация `@EnableJpaAuditing(auditorAwareRef = "auditorAwareBean")` равнозначна элементу разметки `<jpa:auditing auditor-aware-ref="auditorAwareBean"/>`, где средство аудита в JPA активизируется с помощью конфигурации в формате XML. Сведения о пользователе предоставляются в компоненте `auditorAwareBean`. Ниже приведен класс `AuditorAwareBean`, реализующий этот компонент Spring Bean.

```
package com.apress.prospring5.ch8;

import org.springframework.data.domain.AuditorAware;
import org.springframework.stereotype.Component;
import java.util.Optional;
```

```
@Component
public class AuditorAwareBean
       implements AuditorAware<String> {
  public Optional<String> getCurrentAuditor() {
    return Optional.of("prospring5");
  }
}
```

В классе `AuditorAwareBean` реализуетс интерфейс `AuditorAware<T>`, где обобщенный тип `T` заменяется конкретным типом `String`. На практике это должен быть экземпляр класса с пользовательской информацией (например, класса `User`), который представляет входящего в систему пользователя и выполняющего обновления данных. Тип `String` используется в данном случае лишь ради простоты. В классе `AuditorAwareBean` реализуется также метод `getCurrentAuditor()`, возвращающий жестко закодированное значение `prospring5`. На практике сведения о пользователе должны извлекаться из базовой инфраструктуры безопасности. Например, в Spring Security сведения о пользователе могут быть получены из класса `SecurityContextHolder`.

А теперь, когда все, что требовалось для реализации, сделано, перейдем к ее тестированию. Ниже приведен исходный код тестовой программы `SpringAuditJPADemo`.

```
package com.apress.prospring5.ch8;

import java.util.GregorianCalendar;
import java.util.List;
import java.util.Date;
import com.apress.prospring5.ch8.config.AuditConfig;
import com.apress.prospring5.ch8.entities.SingerAudit;
import com.apress.prospring5.ch8.services
          .SingerAuditService;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;

public class SpringAuditJPADemo {
  public static void main(String... args) {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               AuditConfig.class);

    SingerAuditService singerAuditService =
          ctx.getBean(SingerAuditService.class);

    List<SingerAudit> singers =
         singerAuditService.findAll();
    listSingers(singers);
```

```
    System.out.println("Add new singer");
    SingerAudit singer = new SingerAudit();
    singer.setFirstName("BB");
    singer.setLastName("King");
    singer.setBirthDate(new Date(
           (new GregorianCalendar(1940, 8, 16))
           .getTime().getTime()));
    singerAuditService.save(singer);
    singers = singerAuditService.findAll();
    listSingers(singers);

    singer = singerAuditService.findById(41);
    System.out.println("");
    System.out.println("Singer with id 4:" + singer);
    System.out.println("");

    System.out.println("Update singer");
    singer.setFirstName("John Clayton");
    singerAuditService.save(singer);
    singers = singerAuditService.findAll();
    listSingers(singers);

    ctx.close();
  }

  private static void
         listSingers(List<SingerAudit> singerAudits) {
    System.out.println("");
    System.out.println("Listing singers without details:");
    for (SingerAudit audit: singerAudits) {
      System.out.println(audit);
      System.out.println();
    }
  }
}
```

В методе `main()` перечисляется аудиторская информация как после ввода сведений о новом певце, так и после их обновления впоследствии. Выполнение данной тестовой программы дает следующий результат:

```
Add new singer
Listing singers without details:
// other singers ...[1]
Singer - Id: 4, First name: BB, Last name: King,
   Birthday: 1940-09-16,
```

[1] Ввести нового певца
Перечисление певцов без подробных сведений:
// другие певцы ...

```
  Created by: prospring5, Create date:
          2017-05-07 14:19:02.96,
  Modified by: prospring5, Modified date:
          2017-05-07 14:19:02.96
Update singer[2]
Listing singers without details:
// other singers ...
Singer - Id: 4, First name: Riley B., Last name: King,
   Birthday: 1940-09-16,
  Created by: prospring5, Create date:
          2017-05-07 14:33:15.645,
Modified by: prospring5, Modified date:
          2017-05-07 14:33:15.663
```

Как следует из приведенного выше результата, после создания записи о новом певце даты ее создания и последней модификации совпадают. Но после обновления дата последней модификации изменяется. Аудит — это еще одно удобное средство, предоставляемое в Spring Data JPA и избавляющее от необходимости реализовывать его логику самостоятельно.

Отслеживание версий сущностей средствами Hibernate Envers

В отношении критически важных данных из предметной области к корпоративным приложениям всегда предъявляется требование по сохранению *версий* каждой сущности. Например, всякий раз, когда запись о клиенте вводится, обновляется или удаляется в системе управления взаимоотношениями с клиентами (CRM), предыдущая версия этой записи должна быть сохранена в таблице предыстории или аудита, чтобы удовлетворять требованиям аудита или другим нормам, принятым в конкретной организации.

Достичь этих целей можно двумя способами. Первый из них предусматривает создание триггеров базы данных, которые будут сохранять копию записи в таблице предыстории до выполнения любой операции обновления, а второй — разработку соответствующей логики на уровне доступа к данным (например, с помощью АОП). Тем не менее обоим способам присущи свои недостатки. Так, триггерный способ привязан к платформе базы данных, тогда как реализация логики вручную довольно неудобна и сопряжена с ошибками.

Hibernate Envers (Entity Versioning System — система контроля версий сущностей) — это модуль Hibernate, предназначенный для автоматизации контроля версий сущностей. В этом разделе будет рассмотрено применение модуля Hibernate Envers для того, чтобы реализовать контроль версий сущностей типа `SingerAudit`.

---

[2] `Обновить певца`

На заметку Модуль Hibernate Envers не является функциональным средством JPA. Мы упоминаем его здесь, поскольку считаем, что его уместнее рассматривать после обсуждения ряда основных средств аудита, доступных в Spring Data JPA.

В модуле Hibernate Envers поддерживаются две стратегии аудита, вкратце описанные в табл. 8.1.

Таблица 8.1. Стратегии аудита Hibernate Envers

Стратегия аудита	Описание
Стандартная	В модуле Hibernate Envers поддерживается столбец для номера версии записи. Всякий раз, когда запись вводится или обновляется, в таблицу предыстории вводится новая запись с номером версии, извлекаемым из последовательности в базе данных или таблице
Аудит достоверности	При такой стратегии сохраняются начальная и конечная версии каждой записи предыстории. Когда запись вставляется или обновляется, в таблицу предыстории вводится новая запись с номером начальной версии. В то же время предыдущая запись обновляется номером конечной версии. Модуль Hibernate Envers можно также настроить на запись отметки времени, когда конечная версия была обновлена в предыдущей записи предыстории

В этом разделе будет продемонстрирована стратегия аудита достоверности. И хотя она приводит к большему числу обновлений базы данных, в конечном счете записи извлекаются из таблицы предыстории намного быстрее. А поскольку метка времени конечной версии записывается также в таблице предыстории, то при запросе данных упрощается идентификация моментального снимка записи в конкретный момент времени.

Ввод таблиц для контроля версий сущностей

Для поддержки контроля версий сущностей необходимо ввести ряд дополнительных таблиц. Прежде всего, для каждой таблицы, в которой будут контролироваться версии сущностей (в данном случае — класс сущности `SingerAudit`), необходимо создать соответствующую таблицу предыстории. В частности, для контроля версий записей в таблице `SINGER_AUDIT` создается таблица предыстории `SINGER_AUDIT_H`. Ниже приведен сценарий создания этой таблицы из файла `schema.sql`.

```
CREATE TABLE SINGER_AUDIT_H (
    ID INT NOT NULL AUTO_INCREMENT
    , FIRST_NAME VARCHAR(60) NOT NULL
    , LAST_NAME VARCHAR(40) NOT NULL
    , BIRTH_DATE DATE
    , VERSION INT NOT NULL DEFAULT 0
    , CREATED_BY VARCHAR(20)
    , CREATED_DATE TIMESTAMP
    , LAST_MODIFIED_BY VARCHAR(20)
```

```
    , LAST_MODIFIED_DATE TIMESTAMP
    , AUDIT_REVISION INT NOT NUL
    , ACTION_TYPE INT
    , AUDIT_REVISION_END INT
    , AUDIT_REVISION_END_TS TIMESTAMP
    , UNIQUE UQ_SINGER_AUDIT_H_1 (FIRST_NAME, LAST_NAME)
    , PRIMARY KEY (ID, AUDIT_REVISION)
);
```

Для поддержки стратегии аудита достоверности в каждую таблицу предыстории должны быть добавлены четыре столбца, выделенных выше полужирным. Назначение этих столбцов вкратце описано в табл. 8.2.

Таблица 8.2. Столбцы, требующиеся для поддержки стратегии аудита в таблице предыстории

Наименование столбца	Тип данных	Описание
AUDIT_REVISION	INT	Начальная версия записи предыстории
ACTION_TYPE	INT	Тип действия со следующими возможными значениями: 0 — ввод, 1 — модификация, 2 — удаление
AUDIT_REVISION_END	INT	Конечная версия записи предыстории
AUDIT_REVISION_END_TS	TIMESTAMP	Отметка времени, когда была обновлена конечная версия

Для отслеживания номера версии и отметки времени, когда она была создана, в модуле Hibernate Envers требуется еще одна таблица, которая должна называться REVINFO. Ниже приведен сценарий создания этой таблицы из файла schema.sql.

```
CREATE TABLE REVINFO (
    REVTSTMP BIGINT NOT NULL
    , REV INT NOT NULL AUTO_INCREMENT
    , PRIMARY KEY (REVTSTMP, REV)
);
```

Столбец REV служит для хранения номера каждой версии, который автоматически инкрементируется при создании новой записи в таблице предыстории. В столбце REVTSTMP сохраняется отметка времени, соответствующая моменту создания версии. Она хранится в числовом формате.

Конфигурирование фабрики диспетчера сущностей для контроля их версий

Модуль Hibernate Envers реализован в форме приемников EJB, которые могут быть сконфигурированы в компоненте Spring Bean типа LocalContainerEntity ManagerFactory. Для этой цели ниже приведен конфигурационный класс Java. А демонстрировать конфигурацию в формате XML не имеет смысла, потому что

единственное отличие примера из этого раздела состоит в наличии целого ряда дополнительных свойств, специально настраиваемых в модуле Hibernate Envers.

```
package com.apress.prospring5.ch8.config;

import org.hibernate.envers.boot.internal.EnversServiceImpl;
import org.hibernate.envers.event.spi.EnversPostUpdateEventListenerImpl;
import org.hibernate.event.spi.PostUpdateEventListener;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.domain.support.AuditingEntityListener;
import org.springframework.data.jpa.repository.config.EnableJpaAuditing;
import org.springframework.data.jpa.repository.config
          .EnableJpaRepositories;
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseType;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa
          .LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor
          .HibernateJpaVendorAdapter;
import org.springframework.transaction
          .PlatformTransactionManager;
import org.springframework.transaction.annotation
          .EnableTransactionManagement;

import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;
import java.util.Properties;

@Configuration
@EnableTransactionManagement
@ComponentScan(basePackages =
              {"com.apress.prospring5.ch8"})
@EnableJpaRepositories(basePackages =
                      {"com.apress.prospring5.ch8.repos"})
@EnableJpaAuditing(auditorAwareRef = "auditorAwareBean")
public class EnversConfig {

  private static Logger logger =
          LoggerFactory.getLogger(EnversConfig.class);

  @Bean
```

```
public DataSource dataSource() {
  try {
    EmbeddedDatabaseBuilder dbBuilder =
            new EmbeddedDatabaseBuilder();
    return dbBuilder.setType(EmbeddedDatabaseType.H2)
       .addScripts("classpath:db/schema.sql",
                   "classpath:db/test-data.sql").build();
  } catch (Exception e) {
    logger.error("Embedded DataSource bean cannot "
              + "be created!", e);
    return null;
  }
}

@Bean
public PlatformTransactionManager transactionManager() {
  return new JpaTransactionManager(entityManagerFactory());
}

@Bean
public JpaVendorAdapter jpaVendorAdapter() {
  return new HibernateJpaVendorAdapter();
}

@Bean
public Properties hibernateProperties() {
  Properties hibernateProp = new Properties();
  hibernateProp.put("hibernate.dialect",
                    "org.hibernate.dialect.H2Dialect");
  hibernateProp.put("hibernate.format_sql", true);
  hibernateProp.put("hibernate.show_sql", true);
  hibernateProp.put("hibernate.max_fetch_depth", 3);
  hibernateProp.put("hibernate.jdbc.batch_size", 10);
  hibernateProp.put("hibernate.jdbc.fetch_size", 50);
  // Свойства для настройки модуля Hibernate Envers
  hibernateProp.put(
        "org.hibernate.envers.audit_table_suffix", "_H");
  hibernateProp.put(
        "org.hibernate.envers.revision_field_name",
        "AUDIT_REVISION");
  hibernateProp.put(
        "org.hibernate.envers.revision_type_field_name",
        "ACTION_TYPE");
  hibernateProp.put(
        "org.hibernate.envers.audit_strategy",
        "org.hibernate.envers.strategy"
      + ".ValidityAuditStrategy");
  hibernateProp.put("org.hibernate.envers"
                  + ".audit_strategy_validity"
```

```
                    + "_end_rev_field_name",
                      "AUDIT_REVISION_END");
    hibernateProp.put("org.hibernate.envers"
                    + ".audit_strategy_validity"
                    + "_store_revend_timestamp",
                      "True");
    hibernateProp.put("org.hibernate.envers."
                    + "audit_strategy_validity_revend"
                    + "_timestamp_field_name",
                    "AUDIT_REVISION_END_TS");
    return hibernateProp;
  }

  @Bean
  public EntityManagerFactory entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean factoryBean =
         new LocalContainerEntityManagerFactoryBean();
    factoryBean.setPackagesToScan(
                "com.apress.prospring5.ch8.entities");
    factoryBean.setDataSource(dataSource());
    factoryBean.setJpaVendorAdapter(
                new HibernateJpaVendorAdapter());
    factoryBean.setJpaProperties(hibernateProperties());
    factoryBean.setJpaVendorAdapter(jpaVendorAdapter());
    factoryBean.afterPropertiesSet();
    return factoryBean.getObject();
  }
}
```

Приемник событий аудита в модуле Hibernate Envers (`org.hibernate.envers.event.AuditEventListener`) присоединяется к различным событиям сохраняемости. Этот приемник перехватывает события после вставки, обновления или удаления и сохраняет копии моментального снимка класса сущности в таблице предыстории перед обновлением. Приемник присоединяется также к событиям обновления связей (`pre-collection-update`, `pre-collection-remove` и `pre-collection-recreate`) для их обработки и выполнения операций по обновлению связей класса сущности. В модуле Hibernate Envers имеется возможность отслеживать предысторию сущностей в их отношении (например, "один ко многим" или "многие ко многим"). Для конфигурирования модуля Hibernate Envers определен также ряд свойств, вкратце описанных в табл. 8.3 (префикс свойств, `org.hibernate.envers` опущен для ясности)[3].

[3] Полный перечень свойств для конфигурирования модуля Hibernate Envers можно найти в официальной документации на Hibernate, доступной по адресу `https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html#envers-configuration`.

Таблица 8.3. Свойства конфигурации модуля Hibernate Envers

Свойство	Описание
`audit_table_suffix`	Суффикс имени таблицы для сущности с контролируемыми версиями. Например, для класса сущности `SingerAudit`, преобразуемого в таблицу `SINGER_AUDIT`, модуль Hibernate Envers будет сохранять предысторию в таблице `SINGER_AUDIT_H`, поскольку в свойстве `audit_table_suffix` установлено значение `_H`
`revision_field_name`	Столбец таблицы предыстории для хранения номера версии каждой записи в предыстории
`revision_type_field_name`	Столбец таблицы предыстории для хранения типа действия обновления
`audit_strategy`	Стратегия аудита, применяемая для контроля версий сущностей
`audit_strategy_validity_end_rev_field_name`	Столбец таблицы предыстории для хранения номера конечной версии каждой записи в предыстории. Требуется только в том случае, если применяется стратегия аудита достоверности
`audit_strategy_validity_store_revend_timestamp`	Обозначает, следует ли сохранять отметки времени при обновлении номера конечной версии каждой записи в предыстории. Требуется только в том случае, если применяется стратегия аудита достоверности
`audit_strategy_validity_revend_timestamp_field_name`	Столбец таблицы предыстории для хранения метки времени, когда обновляется номер конечной версии каждой записи в предыстории. Требуется только в том случае, если применяется стратегия аудита достоверности, а также при условии, что в предыдущем свойстве установлено логическое значение `true`

Включение режима контроля версий сущностей и извлечения предыстории

Чтобы активизировать режим контроля версий конкретной сущности, необходимо снабдить класс сущности аннотацией `@Audited`. Эту аннотацию можно применять на уровне класса, чтобы проводить полный аудит вносимых изменений. Если же требуется исключить некоторые поля из аудита, то их можно снабдить аннотацией `@NotAudited`. Ниже приведен фрагмент исходного кода класса сущности `SingerAudit`, где применяется аннотация `@Audited`.

```
package com.apress.prospring5.ch8.entities;

import org.hibernate.envers.Audited;
import org.springframework.data.jpa.domain.support
          .AuditingEntityListener;
...

@Entity
@Table(name = "singer_audit")
@Audited
```

```
@EntityListeners(AuditingEntityListener.class)
public class SingerAudit implements Serializable {
  ...
}
```

Этот класс сущности снабжен аннотацией `@Audited`, которую приемники Hibernate Envers выявляют, а затем контролируют версии обновляемых сущностей. По умолчанию в модуле Hibernate Envers предпринимается также попытка контролировать предысторию связей между сущностями. Если же такой контроль не требуется, то следует употребить аннотацию `@NotAudited`, как упоминалось выше.

Для извлечения записей из предыстории в модуле Hibernate Envers предоставляется интерфейс `org.hibernate.envers.AuditReader`, который можно получить из класса `AuditReaderFactory`. Добавим в интерфейс `SingerAuditService` новый метод `findAuditByRevision()`, предназначенный для извлечения записи из предыстории (в данном случае — аудита изменений в классе `SingerAudit`) по номеру версии. Ниже приведен обновленный вариант интерфейса `ContactAuditService` с упомянутым выше методом.

```
package com.apress.prospring5.ch8.services;

import com.apress.prospring5.ch8.entities.SingerAudit;
import java.util.List;

public interface SingerAuditService {
  List<SingerAudit> findAll();
  SingerAudit findById(Long id);
  SingerAudit save(SingerAudit singerAudit);
  SingerAudit findAuditByRevision(Long id, int revision);
}
```

Один из вариантов извлечения записи из предыстории предусматривает передачу идентификатора сущности и номера версии. Ниже приведена реализация метода `findAuditByRevision()`, предназначенного для этой цели.

```
package com.apress.prospring5.ch8.services;

import com.apress.prospring5.ch8.entities.SingerAudit;
import com.apress.prospring5.ch8.repos
          .SingerAuditRepository;
import com.google.common.collect.Lists;
import org.hibernate.envers.AuditReader;
import org.hibernate.envers.AuditReaderFactory;
import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction
          .annotation.Transactional;
```

```
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.List;

@Service("singerAuditService")
@Transactional
public class SingerAuditServiceImpl
       implements SingerAuditService {

  @Autowired
  private SingerAuditRepository singerAuditRepository;

  @PersistenceContext
  private EntityManager entityManager;

  @Transactional(readOnly = true)
  public List<SingerAudit> findAll() {
    return Lists.newArrayList(
           singerAuditRepository.findAll());
  }

  public SingerAudit findById(Long id) {
    return singerAuditRepository.findOne(id).get();
  }

  public SingerAudit save(SingerAudit singer) {
    return singerAuditRepository.save(singer);
  }

  @Transactional(readOnly = true)
  @Override
  public SingerAudit
         findAuditByRevision(Long id, int revision) {
    AuditReader auditReader =
                AuditReaderFactory.get(entityManager);
    return auditReader.find(SingerAudit.class, id, revision);
  }
}
```

Экземпляр типа `EntityManager` внедряется в данный класс и затем передается экземпляру типа `AuditReaderFactory` для извлечения экземпляра типа `AuditReader`. После этого можно вызвать метод `AuditReader.find()`, чтобы извлечь экземпляр класса сущности `SingerAudit` с конкретной версией.

Тестирование контроля версий сущностей

А теперь выясним, каким образом действует контроль версий сущностей. Ниже приведен фрагмент кода, предназначенный для тестирования рассмотренного выше кода, реализующего контроль версий сущностей. Код начальной загрузки контекста

типа ApplicationContext и метода listContacts() совпадает с соответствующим кодом из класса SpringJpaDemo.

```
package com.apress.prospring5.ch8;

import java.util.GregorianCalendar;
import java.util.List;
import java.util.Date;
import com.apress.prospring5.ch8.config.EnversConfig;
import com.apress.prospring5.ch8.entities.SingerAudit;
import com.apress.prospring5.ch8.services
          .SingerAuditService;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;
import org.springframework.context.support
          .GenericXmlApplicationContext;

public class SpringEnversJPADemo {
  public static void main(String... args) {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicati onContext(
               EnversConfig.class);

    SingerAuditService singerAuditService =
          ctx.getBean(SingerAuditService.class);

    System.out.println("Add new singer");
    SingerAudit singer = new SingerAudit();
    singer.setFirstName("BB");
    singer.setLastName("King");
    singer.setBirthDate(new Date(
           (new GregorianCalendar(1940, 8, 16))
           .getTime().getTime()));
    singerAuditService.save(singer);
    listSingers(singerAuditService.findAll());

    System.out.println("Update singer");
    singer.setFirstName("Riley B.");
    singerAuditService.save(singer);
    listSingers(singerAuditService.findAll());
    SingerAudit oldSinger =
          singerAuditService.findAuditByRevision(41, 1);
    System.out.println("");
    System.out.println("Old Singer with id 1 and rev 1:"
                    + oldSinger);
    System.out.println("");
    oldSinger = singerAuditService.findAuditByRevision(41, 2);
```

```
    System.out.println("");
    System.out.println("Old Singer with id 1 and rev 2:"
                    + oldSinger);
    System.out.println("");

    ctx.close();
  }

  private static void
          listSingers(List<SingerAudit> singers) {
    System.out.println("");
    System.out.println("Listing singers:");
    for (SingerAudit singer: singers) {
      System.out.println(singer);
      System.out.println();
    }
  }
}
```

Сначала в приведенном выше тестовом коде создается новый объект, представляющий певца, а затем он обновляется. После этого извлекаются сущности типа `SingerAudit` с номерами версий `1` и `2` соответственно. Выполнение этого тестового кода дает следующий результат:

```
Listing singers:
...//
Singer - Id: 4, First name: BB, Last name: King,
   Birthday: 1940-09-16,
Created by: prospring5, Create date:
            2017-05-11 23:50:14.778,
Modified by: prospring5, Modified date:
            2017-05-11 23:50:14.778

Old Singer with id 1 and rev 1:Singer - Id: 4,
First name: BB, Last name: King, Birthday: 1940-09-16,
Created by: prospring5, Create date:
            2017-05-11 23:50:14.778,
Modified by: prospring5, Modified date:
            2017-05-11 23:50:14.778
Old Singer with id 1 and rev 2:Singer - Id: 4,
First name: Riley B., Last name: King,
  Birthday: 1940-09-16,
Created by: prospring5, Create date:
            2017-05-11 23:50:14.778,
Modified by: prospring5, Modified date:
            2017-05-11 23:50:15.0
```

Как следует из приведенного выше результата, после операции обновления имя певца в объекте типа `SingerAudit` было изменено на `Riley B.`. Но при просмотре

предыстории версии 1 выявилось имя певца BB. А в версии 2 его имя изменилось на Riley B.. Обратите также внимание на то, что дата последней модификации версии 2 верно отражает обновленные дату и время.

Стартовая библиотека Spring Boot для JPA

До сих пор было показано, как конфигурировать самые разные компоненты, включая сущности, базу данных, информационные хранилища и службы. Как и следовало ожидать, в модуле Spring Boot должна присутствовать стартовая библиотека, помогающая ускорить разработку проектов и свести к минимуму затраты труда на конфигурирование. А поскольку стартовая библиотека в модуле Spring Boot зависит от прикладного интерфейса JPA, то ей сопутствует предварительно сконфигурированная встроенная база данных, и все это требуется для внедрения зависимости в путь к классам. А в библиотеке Hibernate аналогичный компонент служит для поддержки абстракции на уровне сохраняемости, где автоматически обнаруживаются интерфейсы из иерархии интерфейса Repository. Таким образом, разработчикам остается лишь предоставить сущности, расширения информационного хранилища и класс Application, чтобы пользоваться всеми этими средствами. В конечном счете можно разработать класс и для заполнения базы данных. И здесь поясняется, как это сделать.

Прежде всего необходимо внедрить стартовую библиотеку Spring Boot для JPA в качестве зависимости. Это делается таким же образом, как и при внедрении зависимостей от всех остальных библиотек. В частности, необходимо ввести версию, идентификаторы группы стартовой библиотеки в корневом файле конфигурации build.gradle, а также определить зависимости в файле конфигурации chapter08/boot-jpa/build.gradle. Ниже приведены соответствующие фрагменты кода из этих файлов конфигурации.

```
// Файл конфигурации pro-spring-15/build.gradle
ext {
    // Библиотеки Spring
    springVersion = '5.0.0.M5'
    bootVersion = '2.0.0.BUILD-SNAPSHOT'
    springDataVersion = '2.0.0.M2'
    ...

    // boot = [
       ...
       starterJdbc : "org.springframework.boot:
              spring-boot-starter-jdbc:$bootVersion",
       starterJpa : "org.springframework.boot:
              spring-boot-starter-data-jpa:$bootVersion"
    ]
```

```
    testing = [
       junit: "junit:junit:$junitVersion"
    ]

    db = [
        ...
        h2 : "com.h2database:h2:$h2Version"
    ]
}

// Файл конфигурации chapter08/boot-jpa/build.gradle
buildscript {
    repositories {
       mavenLocal
       mavenCentral
       maven { url "http://repo.spring.io/release" }
       maven { url "http://repo.spring.io/milestone" }
       maven { url "http://repo.spring.io/snapshot" }
       maven { url "https://repo.spring.io/libs-snapshot" }
    }
    dependencies {
        classpath boot.springBootPlugin
    }
}

    apply plugin: 'org.springframework.boot'

    dependencies {
        compile boot.starterJpa, db.h2
    }
}
```

После внедрения этих зависимостей и обновления проекта должно появиться все дерево зависимостей от стартовой библиотеки `spring-bootstarter-data-jpa`, как показано на примере представления Gradle Projects в IDE IntelliJ IDEA на рис. 8.2.

Сущности окажутся теми же, что и прежде (`Singer`, `Album` и `Instrument`), поэтому отображать их не нужно. Для их инициализации служит компонент Spring Bean типа `DBInitializer`, т.е. служебный класс, в котором применяются компоненты Spring Beans типа `SingerRepository` и `InstrumentRepository`, предоставляемые модулем Spring Boot для сохранения ряда объектов в базе данных. Ниже приведен исходный код этого класса.

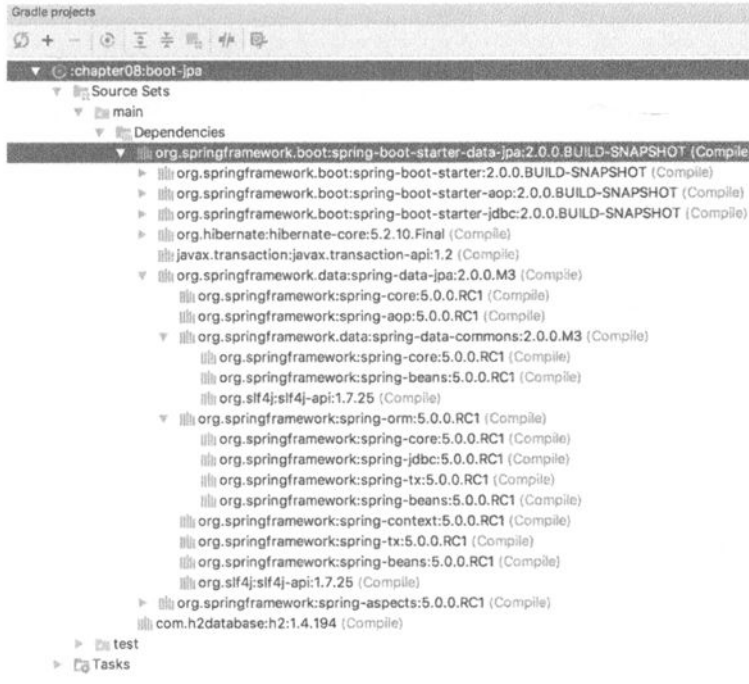


Рис. 8.2. Дерево зависимостей от стартовой библиотеки Spring Boot для JPA

```
package com.apress.prospring5.ch8.config;

import com.apress.prospring5.ch8.InstrumentRepository;
import com.apress.prospring5.ch8.SingerRepository;
import com.apress.prospring5.ch8.entities.Album;
import com.apress.prospring5.ch8.entities.Instrument;
import com.apress.prospring5.ch8.entities.Singer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.stereotype.Service;

import javax.annotation.PostConstruct;
import java.util.Date;
import java.util.GregorianCalendar;

@Service
public class DBInitializer {
  private Logger logger =
          LoggerFactory.getLogger(DBInitializer.class);

  @Autowired
```

```
SingerRepository singerRepository;

@Autowired
InstrumentRepository instrumentRepository;

@PostConstruct
public void initDB() {
  logger.info("Starting database initialization...");

  Instrument guitar = new Instrument();
  guitar.setInstrumentId("Guitar");
  instrumentRepository.save(guitar);
  Instrument piano = new Instrument();

  piano.setInstrumentId("Piano");
  instrumentRepository.save(piano);

  Instrument voice = new Instrument();
  voice.setInstrumentId("Voice");
  instrumentRepository.save(voice);

  Singer singer = new Singer();
  singer.setFirstName("John");
  singer.setLastName("Mayer");
  singer.setBirthDate(new Date(
         (new GregorianCalendar(1977, 9, 16))
         .getTime().getTime()));
  singer.addInstrument(guitar);
  singer.addInstrument(piano);

  Album album1 = new Album();
  album1.setTitle("The Search For Everything");
  album1.setReleaseDate(new java.sql.Date(
         (new GregorianCalendar(2017, 0, 20))
         .getTime().getTime()));
  singer.addAbum(album1);

  Album album2 = new Album();
  album2.setTitle("Battle Studies");
  album2.setReleaseDate(new java.sql.Date(
         (new GregorianCalendar(2009, 10, 17))
         .getTime().getTime()));
  singer.addAbum(album2);

  singerRepository.save(singer);

  singer = new Singer();
  singer.setFirstName("Eric");
  singer.setLastName("Clapton");
```

```
    singer.setBirthDate(new Date(
          (new GregorianCalendar(1945, 2, 30))
          .getTime().getTime()));
    singer.addInstrument(guitar);

    Album album = new Album();
    album.setTitle("From The Cradle");
    album.setReleaseDate(new java.sql.Date(
          (new GregorianCalendar(1994, 8, 13))
          .getTime().getTime()));
    singer.addAbum(album);

    singerRepository.save(singer);

    singer = new Singer();
    singer.setFirstName("John");
    singer.setLastName("Butler");
    singer.setBirthDate(new Date(
          (new GregorianCalendar(1975, 3, 1))
          .getTime().getTime()));
    singer.addInstrument(guitar);

    singerRepository.save(singer);
    logger.info("Database initialization finished.");
  }
}
```

В данном примере интерфейсы `SingerRepository` и `InstrumentRepository` довольно просты:

```
// Исходный файл InstrumentRepository.java
package com.apress.prospring5.ch8;

import com.apress.prospring5.ch8.entities.Instrument;
import org.springframework.data.repository.CrudRepository;

public interface InstrumentRepository
  extends CrudRepository<Instrument, Long> {
}

// Исходный файл SingerRepository.java
package com.apress.prospring5.ch8;

import com.apress.prospring5.ch8.entities.Singer;
import org.springframework.data.repository.CrudRepository;
import java.util.List;

public interface SingerRepository
extends CrudRepository<Singer, Long> {
```

```
  List<Singer> findByFirstName(String firstName);
  List<Singer> findByFirstNameAndLastName(String firstName,
                                          String lastName);
}
```

Интерфейс `SingerRepository` внедряется непосредственно в аннотированный класс `Application` из модуля Spring Boot и служит для извлечения записей обо всех певцах и порожденных ими записей из базы данных с последующим их выводом на консоль. Ниже приведен исходный код класса `Application`.

```
package com.apress.prospring5.ch8;

import com.apress.prospring5.ch8.entities.Singer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure
          .SpringBootApplication;
import org.springframework.context
          .ConfigurableApplicationContext;
import org.springframework.transaction.annotation
          .Transactional;
import java.util.List;

@SpringBootApplication(scanBasePackages =
                       "com.apress.prospring5.ch8.config")
public class Application implements CommandLineRunner {
  private static Logger logger =
          LoggerFactory.getLogger(Application.class);

  @Autowired
  SingerRepository singerRepository;

  public static void main(String... args)
         throws Exception {
    ConfigurableApplicationContext ctx =
          SpringApplication.run(Application.class, args);

    System.in.read();
    ctx.close();
  }

  @Transactional(readOnly = true)
  @Override public void run(String... args)
           throws Exception {
    List<Singer> singers =
```

```
                singerRepository.findByFirstName("John");
    listSingersWithAlbum(singers);
  }

  private static void
          listSingersWithAlbum(List<Singer> singers) {
    logger.info(" ---- Listing singers with instruments:");
    singers.forEach(singer -> {
      logger.info(singer.toString());
      if (singer.getAlbums() != null) {
        singer.getAlbums().forEach(
           album -> logger.info("\t" + album.toString()));
      }
      if (singer.getInstruments() != null) {
        singer.getInstruments().forEach(
           instrument -> logger.info("\t"
                      + instrument.getInstrumentId()));
      }
    });
  }
}
```

В классе `Application` внедрено нечто новое и, в частности, интерфейс `CommandLineRunner`. Этот интерфейс служит для того, чтобы предписать модулю Spring Boot выполнить метод `run()`, когда данный компонент Spring Bean содержится в приложении.

Атрибут `scanBasePackages = "com.apress.prospring5.ch8.config"` служит для активизации просмотра компонентов в одном или нескольких пакетах, указанных в качестве аргумента, чтобы входящие в их состав компоненты Spring Beans были созданы и введены в контекст приложения. Это требуется в том случае, когда компоненты Spring Beans определены не в том пакете, где определен класс `Application`.

Следует также заметить, что в данном случае не требуется ни другого конфигурационного класса, ни сценария SQL для инициализации базы данных и никаких других аннотаций в классе `Application`. Очевидно, что модуль Spring Boot с его стартовыми библиотеками оказывается очень удобным в том случае, если требуется уделить основное внимание логике работы приложения.

Если выполнить исходный код класса `Application`, то будет получен вполне ожидаемый результат, как показано ниже. (Однако следует иметь в виду, что, прежде чем выйти из данного приложения, необходимо нажать любую клавишу.)

```
INFO c.a.p.c.c.DBInitializer -
     Starting database initialization...
INFO c.a.p.c.c.DBInitializer -
     Database initialization finished.
INFO c.a.p.c.Application -
     ---- Listing singers with instruments:
```

```
INFO c.a.p.c.Application - Singer - Id: 1,
     First name: John, Last name: Mayer,
     Birthday: 1977-10-16
INFO c.a.p.c.Application - Album - id: 1, Singer id: 1,
     Title: Battle Studies, Release Date: 2009-11-17
INFO c.a.p.c.Application - Album - id: 2, Singer id: 1,
Title: The Search For Everything, Release Date: 2017-01-20
INFO c.a.p.c.Application - Piano
INFO c.a.p.c.Application - Guitar
INFO c.a.p.c.Application - Singer - Id: 3, First name: John,
     Last name: Butler, Birthday: 1975-04-01
INFO c.a.p.c.Application - Guitar
INFO c.a.p.c.Application -
     Started Application in 3.464 seconds
     (JVM running for 4.0)
```

Соображения по поводу применения прикладного интерфейса JPA

Несмотря на относительно большой объем материала, изложенного в этой главе, в ней была раскрыта только малая доля функциональных возможностей прикладного интерфейса JPA. Так, в главе не обсуждалось применение JPA для вызова хранимых процедур из базы данных. JPA — это завершенный и эффективный стандарт на доступ к данным преобразований ORM, и с помощью таких сторонних средств, как проект Spring Data JPA и модуль Hibernate Envers, можно относительно просто реализовать разнообразную сквозную функциональность.

Прикладной интерфейс JPA является стандартным для платформы JEE и поддерживается большинством крупных сообществ разработчиков программного обеспечения с открытым кодом, а также коммерческими поставщиками программного обеспечения (например, JBoss, GlassFish, WebSphere, WebLogic и т.д.). Таким образом, применение JPA в качестве стандарта для доступа к данным является вполне обоснованным решением. Если же требуется абсолютный контроль над запросом, в таком случае можно воспользоваться поддержкой собственных запросов в JPA вместо того, чтобы обращаться непосредственно к услугам интерфейса JDBC.

В заключение следует заметить, что если приложения на платформе JEE разрабатываются средствами Spring, то прикладной интерфейс JPA рекомендуется применять для реализации уровня доступа к данным. Если же возникают особые потребности в доступе к данным, то при желании можно по-прежнему пользоваться также услугами интерфейса JDBC. Не следует, однако, забывать, что каркас Spring позволяет легко сочетать технологии доступа к данным с прозрачным управлением транзакциями. А если требуется еще больше упростить процесс разработки, это можно сделать с помощью модуля Spring Boot и его предварительно сконфигурированных компонентов Spring Beans и специально настраиваемых конфигураций.

Резюме

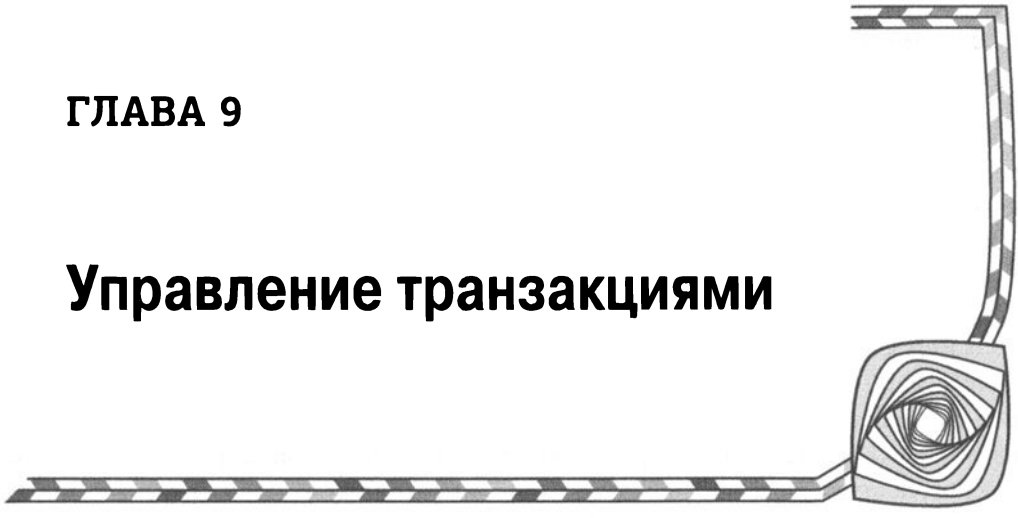
В этой главе были сначала рассмотрены основные понятия прикладного интерфейса JPA и показано, каким образом конфигурируется компонент JPA типа `EntityManagerFactory` в Spring с помощью библиотеки Hibernate, служащей в качестве поставщика услуг сохраняемости. Затем обсуждалось применение прикладного интерфейса JPA для выполнения основных операций в базе данных. В качестве дополнительных вопросов было рассмотрено применение собственных запросов и строго типизированного прикладного интерфейса JPA Criteria API.

Далее в главе пояснялось, каким образом абстракция интерфейса `Repository` в проекте Spring Data JPA помогает упростить разработку приложений в JPA и как пользоваться приемниками сущностей из этого проекта для отслеживания основной информации аудита классов сущностей. Кроме того, здесь обсуждалось применение модуля Hibernate Envers для полного контроля версий классов сущностей.

И в заключение было рассмотрено применение модуля Spring Boot вместе с прикладным интерфейсом JPA, позволяющее значительно упростить конфигурирование, чтобы уделить основное внимание разработке функциональных средств, требующихся в приложениях. В следующей главе речь пойдет об управлении транзакциями в Spring.

ГЛАВА 9

Управление транзакциями



Транзакции — одна из самых важных составляющих построения надежного корпоративного приложения. Наиболее распространенным типом транзакции является операция в базе данных. В ходе типичной операции обновления начинается транзакция в базе данных, где данные обновляются, а по результатам выполнения этой операции в базе данных транзакция фиксируется или откатывается. Но зачастую в зависимости от требований к приложению и ресурсов сервера базы данных (например, СУРБД, промежуточного программного обеспечения для обработки сообщений, системы управления предприятием (ERP) и т.д.), с которыми должно взаимодействовать приложение, управление транзакциями может оказаться намного более сложным.

На заре разработки приложений на Java (после появления интерфейса JDBC, но до того, как стал доступным стандарт JEE или каркас приложений, подобный Spring) разработчики программно контролировали и управляли транзакциями в прикладном коде. А когда появился стандарт JEE, точнее — стандарт EJB, разработчики получили возможность пользоваться управляемыми контейнером транзакциями (CMT), чтобы управлять ими декларативно. Но сложное объявление транзакций в дескрипторе развертывания EJB все же было трудно поддерживать, а кроме того, оно излишне усложняло обработку транзакций. Некоторые разработчики предпочитали иметь больший контроль над транзакциями и поэтому — выбирали управляемые компонентами транзакции (BMT), чтобы управлять ими программно. Тем не менее сложность программирования с помощью JTA (Java Transaction API — прикладной интерфейс Java для транзакций) также препятствовала повышению производительности труда разработчиков.

Как обсуждалось в главе 5, посвященной АОП, управление транзакциями является сквозной функциональностью и не должно программироваться в самой бизнес-

логике. Самый подходящий способ реализовать управление транзакциями — дать возможность разработчикам самостоятельно определить требования к транзакциям декларативно, а таким программным средствам, как Spring, JEE или АОП, — автоматически привязать логику управления транзакциями. В этой главе будет показано, каким образом каркас Spring может упростить реализацию логики обработки транзакций. В каркасе Spring предоставляется поддержка как декларативного, так и программного способа управления транзакциями.

В каркасе Spring обеспечивается отличная поддержка декларативных транзакций, а это значит, что бизнес-логика не загромождается кодом управления транзакциями. Все, что требуется сделать, — это объявить методы (в классах или на соответствующих уровнях абстракции), которые должны принимать участие в транзакции, вместе с подробностями конфигурации транзакций, а каркас Spring сам позаботится о поддержке управления транзакциями. В этой главе будут, в частности, рассмотрены следующие вопросы.

- **Уровень абстракции транзакций в Spring.** В этой части описаны основные составляющие классов абстрагирования транзакций в Spring, а также пояснено, как применять эти классы для управления свойствами транзакций.
- **Декларативное управление транзакциями.** В этой части показано, как пользоваться каркасом Spring и простыми объектами Java для реализации декларативного управления транзакциями. Все это будет продемонстрировано на конкретных примерах декларативного управления транзакциями, где применяются XML-файлы конфигурации, а также аннотации Java.
- **Программное управление транзакциями.** Несмотря на то что программное управление транзакциями применяется очень редко, в этой части поясняется, как пользоваться предоставляемым в Spring классом `TransactionTemplate`, обеспечивающим полный контроль над кодом управления транзакциями.
- **Глобальные транзакции через прикладной интерфейс JTA.** Что касается глобальных транзакций, которые должны охватывать несколько ресурсов сервера базы данных, то в этой части будет показано, как конфигурировать и реализовать глобальные транзакции в Spring, используя прикладной интерфейс JTA.

Исследование уровня абстракции транзакций в Spring

Независимо от того, используется ли каркас Spring или нет, при разработке приложения, взаимодействующего с базой данных, приходится делать главный выбор между локальными и глобальными транзакциями. Локальные транзакции характерны для отдельного транзакционного ресурса (например, подключения через интерфейс JDBC), тогда как глобальные находятся под управлением контейнера и могут охватывать несколько транзакционных ресурсов.

Типы транзакций

Локальные транзакции просты в управлении, и если все операции в приложении должны взаимодействовать только с одним транзакционным ресурсом (например, транзакция через интерфейс JDBC), то локальных транзакций окажется достаточно для этой цели. Но если программные средства Spring не применяются при разработке приложений, то для управления транзакциями приходится писать немало кода, а если в дальнейшем понадобится расширить область действия транзакций, то код управления локальными транзакциями придется переписать, чтобы пользоваться глобальными транзакциями.

В среде Java глобальные транзакции были реализованы в JTA (Java Transaction API — прикладной интерфейс Java для транзакций). В таком случае совместимый с JTA диспетчер транзакций подключается ко многим транзакционным ресурсам через соответствующие диспетчеры ресурсов, способные взаимодействовать с диспетчером транзакций по протоколу XA (открытому стандарту, определяющему распределенные транзакции). Кроме того, с помощью механизма двухфазной фиксации (2 Phase Commit — 2PC) обеспечивается правильное обновление или откат всех участвующих в транзакции источников данных, находящихся на сервере базы данных. При отказе любого из этих ресурсов будет выполнен откат всей транзакции, а следовательно, и откат обновлений других ресурсов.

На рис. 9.1 схематически представлены глобальные транзакции, реализуемые в прикладном интерфейсе JTA. Как следует из рис. 9.1, в глобальной транзакции (в общем называемой *распределенной транзакцией*) принимают участие четыре ос-

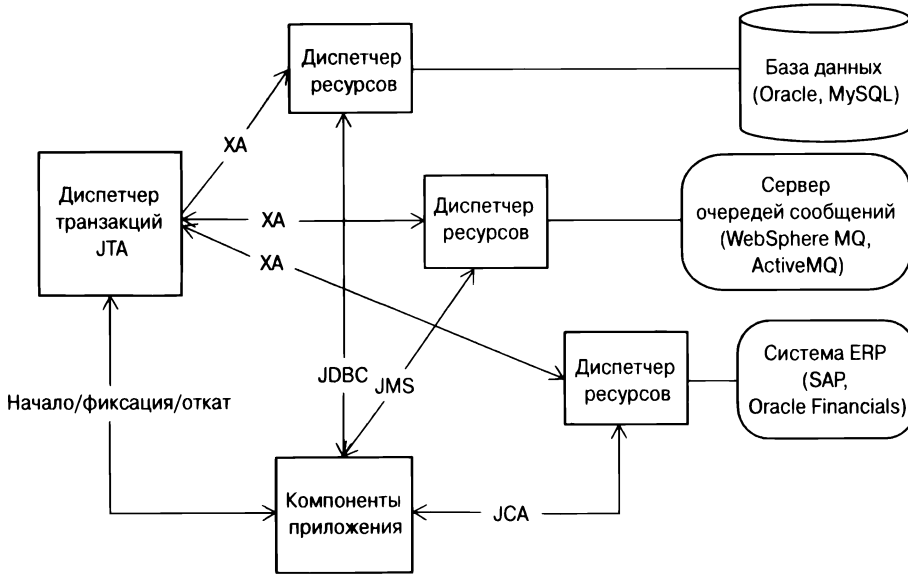


Рис. 9.1. Общее представление глобальных транзакций, реализуемых в прикладном интерфейсе JTA

новные стороны. Первая сторона — это ресурс сервера базы данных (например, СУРБД), промежуточное программное обеспечение для обмена сообщениями, система управления предприятием (ERP) и т.д.

Второй стороной является диспетчер ресурсов, который обычно предоставляется поставщиком ресурсов сервера базы данных и отвечает за взаимодействие с этим ресурсом. Например, при подключении к базе данных MySQL приходится взаимодействовать с классом `MysqlXADataSource`, предоставляемым специальным средством Java для подключения к базе данных MySQL. Другие ресурсы сервера базы данных (MQ, ERP и т.д.) также предоставляют свои диспетчеры ресурсов.

Третьей стороной является диспетчер транзакций JTA, отвечающий за управление, координацию и синхронизацию состояния транзакции со всеми диспетчерами ресурсов, принимающими участие в транзакции. Для этой цели служит протокол XA — открытый стандарт, широко применяемый для обработки распределенных транзакций. В диспетчере транзакций JTA поддерживается также механизм 2PC, поэтому все изменения будут фиксироваться вместе, и если обновление любого ресурса завершается неудачно, то производится откат всей транзакции, а следовательно, ни один из ресурсов не будет обновлен. Этот механизм полностью описан в спецификации службы транзакций Java (Java Transaction Service — JTS).

И последней, четвертой стороной является приложение. Транзакцией управляет (начинает, фиксирует, открывает и т.д.) само приложение, базовый контейнер или каркас Spring, в котором выполняется приложение. В то же время приложение взаимодействует с ресурсами сервера базы данных через разные стандарты, определенные на платформе JEE. Как показано на рис. 9.1, приложение подключается к СУРБД через интерфейс JDBC, к промежуточному программному обеспечению MQ по стандарту JMS и к системе ERP по спецификации на архитектуру средств подключения на платформе Java EE (Java EE Connector Architecture — JCA).

Прикладной интерфейс JTA поддерживается всеми полноценными серверами приложений, совместимыми со стандартом JEE (например, JBoss, WebSphere, WebLogic и GlassFish), где транзакция доступна благодаря поиску через интерфейс JNDI. Как и для автономных приложений или веб-контейнеров (вроде Tomcat и Jetty), существуют как коммерческие решения, так и решения с открытым кодом, обеспечивающие поддержку JTA/XA в подобных средах (например, Atomikos, JOTM (Java Open Transaction Manager — открытый диспетчер транзакций Java) и Bitronix).

Реализации интерфейса `PlatformTransactionManager`

Для создания транзакций и управления ими в каркасе Spring вместе с интерфейсом `PlatformTransactionManager` применяются интерфейсы `TransactionDefinition` и `TransactionStatus`. Чтобы реализовать эти интерфейсы на практике, придется досконально разобраться в принципе действия диспетчера транзакций.

На рис. 9.2 приведены реализации интерфейса `PlatformTransactionManager` в Spring.

В каркасе Spring предоставляется обширный ряд реализаций интерфейса `PlatformTransactionManager`. Так, в классе `CciLocalTransactionManager` поддерживаются стандарты JEE, JCA и CCI (Common Client Interface — общий клиентский интерфейс). А класс `DataSourceTransactionManager` предназначен для обобщенных подключений через интерфейс JDBC. Для объектно-реляционного преобразования (ORM) предусмотрен целый ряд реализаций, включая прикладной интерфейс JPA (класс `JpaTransactionManager`[1]), а также Hibernate 3 и библиотеку Hibernate 5 (класс `HibernateTransactionManager`[2]). В классе `JmsTransactionManager` поддерживаются также реализации по стандарту JMS 2.0[3], а для реализации прикладного интерфейса JTA имеется обобщенный класс `JtaTransactionManager`. Кроме того, в Spring предоставляется ряд классов диспетчеров транзакций JTA, характерных для конкретных серверов приложений. В этих классах поддерживается WebSphere (класс `WebSphereUowTransactionManager`), WebLogic (класс `WebLogicJtaTransactionManager`) и Oracle OC4J (класс `OC4JJtaTransactionManager`).

Анализ свойств транзакций

В этом разделе описываются свойства транзакций, поддерживаемые в Spring, с акцентом на взаимодействие с СУРБД как с ресурсом сервера базы данных. Транзакции обладают четырьмя хорошо известными свойствами ACID (atomicity, consistency, isolation, durability — атомарность, согласованность, изолированность, долговечность), и за их поддержку отвечают транзакционные ресурсы. Контролировать атомарность, согласованность и долговечность транзакции нельзя, но можно контролировать ее распространение и время распространения, а также настроить в конфигурации доступность транзакции только для чтения и задавать уровень изоляции.

Все эти установки инкапсулированы средствами Spring в интерфейсе `TransactionDefinition`. Этот интерфейс применяется в главном интерфейсе `PlatformTransactionManager` для поддержки транзакций, реализации которого отвечают за управление транзакциями на конкретной платформе вроде JDBC или JTA. Базовый метод `PlatformTransactionManager.getTransaction()` получает интерфейс `TransactionDefinition` в качестве аргумента и возвращает интерфейс `TransactionStatus`. Последний служит для контроля над выполнением транзакции, а точнее, для того, чтобы установить результат транзакции и проверить, завершена ли транзакция и не оказывается ли она новой.

---

[1] Начиная с версии Spring 5 спецификация JDO не поддерживается, поэтому класс `JdoTransactionManager` отсутствует в иерархии, приведенной на рис. 9.2.

[2] В версии 5 каркас Spring взаимодействует только с версией 5 библиотеки Hibernate, тогда как реализации версий Hibernate 3 и Hibernate 4 исключены.

[3] Начиная с версии Spring 5 поддержка стандарта JMS 1.1 прекращена.

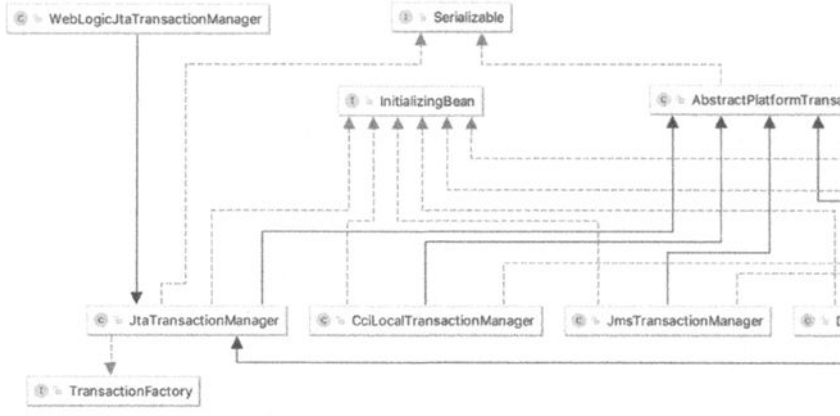


Рис. 9.2. Реализации интерфейса **`PlatformTransactionManager`** в Spring

Интерфейс `TransactionDefinition`

Как упоминалось ранее, интерфейс `TransactionDefinition` управляет свойствами транзакции. Рассмотрим этот интерфейс более подробно и опишем его методы. Ниже приведено объявление этого интерфейса.

```
package org.springframework.transaction;

import java.sql.Connection;

public interface TransactionDefinition {
  // здесь опущены операторы объявления переменных

  ...

  int getPropagationBehavior();

  int getIsolationLevel();

  int getTimeout();

  boolean isReadOnly();

  String getName();
}
```

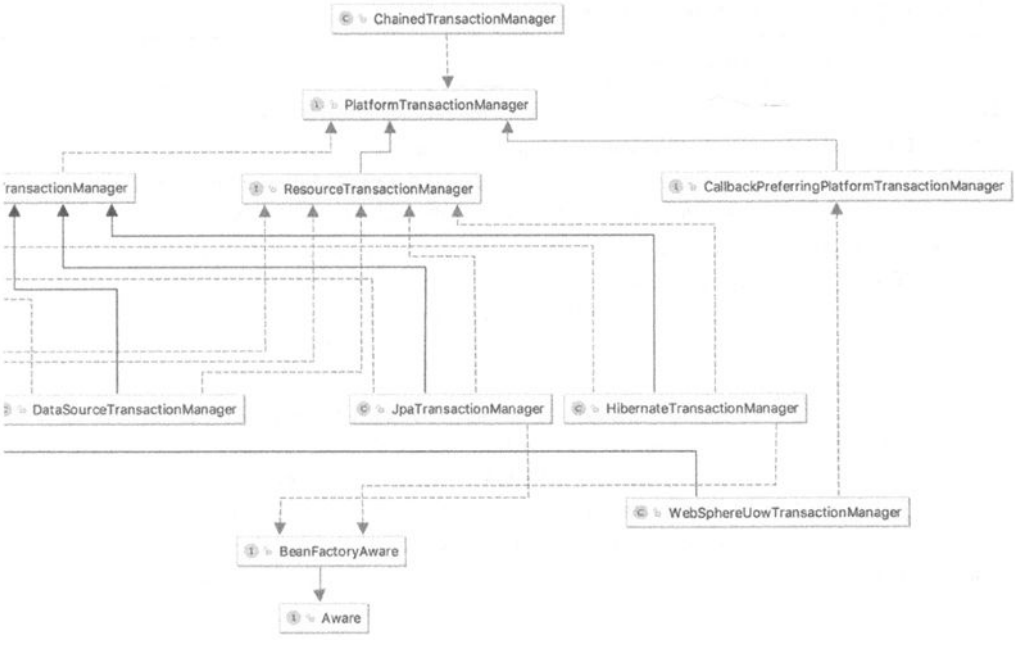


Рис. 9.2. Продолжение

Простым и самоочевидным в этом интерфейсе является метод `getTimeout()`, возвращающий промежуток времени (в секундах), в течение которого транзакция должна быть завершена, а также метод `isReadOnly()`, указывающий, доступна ли транзакция только для чтения. В реализации диспетчера транзакций это значение может служить для того, чтобы оптимизировать выполнение и проверить, выполняются ли в транзакции только операции чтения. А метод `getName()` возвращает имя транзакции.

Оставшиеся методы `getPropagationBehavior()` и `getIsolationLevel()` заслуживают более пристального внимания. Начнем с метода `getIsolationLevel()`, где определяются изменения в данных, доступных другим транзакциям. В табл. 9.1 перечислены доступные для применения уровни изоляции транзакций, а также описаны изменения, доступные другим транзакциям в текущей транзакции.

Выбор подходящего уровня изоляции очень важен для согласованности данных, но он оказывает заметное влияние на производительность. Самый высокий уровень изоляции, `TransactionDefinition.ISOLATION_SERIALIZABLE`, является особенно дорогостоящим для сопровождения.

В методе `getPropagationBehavior()` определяется, что произойдет с транзакционным вызовом, в зависимости от того, существует ли активная транзакция. Значе-

Таблица 9.1. Уровни изоляции транзакций

Уровень изоляции	Описание
`ISOLATION_DEFAULT`	Стандартный уровень изоляции базового информационного хранилища
`ISOLATION_READ_UNCOMMITTED`	Самый низкий уровень изоляции, на котором транзакцию едва ли можно назвать таковой, поскольку на нем доступны данные, модифицированные в других незафиксированных транзакциях
`ISOLATION_READ_COMMITTED`	Стандартный уровень изоляции в большинстве баз данных. На этом уровне гарантируется, что в других транзакциях не нельзя читать данные, которые еще не зафиксированы в текущей транзакции. Но данные, прочитанные в одной транзакции, могут быть обновлены в других транзакциях
`ISOLATION_REPEATABLE_READ`	Более строгий уровень, чем `ISOLATION_READ_COMMITED`. На этом уровне гарантируется, что после выборки данных можно произвести повторную выборку, по крайней мере, того же самого набора данных. Если же новые данные были введены в других транзакциях, вновь введенные данные могут быть извлечены
`ISOLATION_SERIALIZABLE`	Самый дорогостоящий и надежный уровень изоляции. На этом уровне все транзакции трактуются как выполняемые последовательно одна за другой

ния, которыми оперирует этот метод, перечислены в табл. 9.2. Режимы распространения, представленные статическими значениями, определяются в интерфейсе `TransactionDefinition`.

Таблица 9.2. Режимы распространения транзакций

Режим распространения	Описание
`PROPAGATION_REQUIRED`	Поддерживает транзакцию, если она уже существует. А если транзакция отсутствует, то начинается новая транзакция
`PROPAGATION_SUPPORTS`	Поддерживает транзакцию, если она уже существует. А если транзакция отсутствует, то выполнение осуществляется без транзакций
`PROPAGATION_MANDATORY`	Поддерживает транзакцию, если она уже существует. А если активная транзакция отсутствует, то генерируется исключение
`PROPAGATION_REQUIRES_NEW`	Всегда начинает новую транзакцию. Если активная транзакция уже существует, она приостанавливается
`PROPAGATION_NOT_SUPPORTED`	Не поддерживает выполнение с активной транзакцией. Всегда выполняется без транзакций и приостанавливает любые существующие транзакции
`PROPAGATION_NEVER`	Всегда выполняется без транзакций, даже если имеется активная транзакция. Если же активная транзакция существует, генерируется исключение
`PROPAGATION_NESTED`	Выполняется во вложенной транзакции, если существует активная транзакция. А если активная транзакция отсутствует, то выполняется так, как будто установлено значение `PROPAGATION_REQUIRED`

Интерфейс TransactionStatus

Интерфейс TransactionStatus, объявление которого приведено ниже, позволяет диспетчеру транзакций управлять выполнением транзакции. В прикладном коде, реализующем данный интерфейс, можно проверить, является ли транзакция новой или доступна только для чтения, и на основании полученного результата может быть инициирован откат транзакции.

```
package org.springframework.transaction;

public interface TransactionStatus
       extends SavepointManager {

  boolean isNewTransaction();

  boolean hasSavepoint();

  void setRollbackOnly();

  boolean isRollbackOnly();

  void flush();

  boolean isCompleted();
}
```

Методы из интерфейса TransactionStatus самоочевидны. Наиболее примечательным из них является метод setRollbackOnly(), вызывающий откат и завершающий активную транзакцию.

Метод hasSavePoint() возвращает признак, обозначающий, имеется ли в транзакции точка сохранения (т.е. эта транзакция создана как вложенная, исходя из точки сохранения). Метод flush() сбрасывает текущий сеанс работы в информационное хранилище, если это уместно (например, когда используется библиотека Hibernate). А метод isCompleted() возвращает признак завершения транзакции (т.е. фиксации или отката).

Модель выборочных данных и инфраструктура для исходного кода примеров

В этом разделе представлен обзор модели выборочных данных и инфраструктуры, применяемых в последующих примерах, демонстрирующих управление транзакциями. В этих примерах будет применяться прикладной интерфейс JPA с библиотекой Hibernate в качестве уровня сохраняемости для реализации логики доступа к данным. А в целях упрощения разработки основных операций в базе данных будет также использован проект Spring Data JPA и его абстракция информационного хранилища.

Создание простого проекта Spring JPA с зависимостями

Начнем с создания простого проекта. Поскольку в данном случае применяется прикладной интерфейс JPA, в новый проект необходимо внедрить обязательные зависимости, требующиеся для примеров из этой главы, как показано ниже.

```
// Файл конфигурации pro-spring-15/build.gradle
ext {
    // Библиотеки Spring
    springVersion = '5.0.0.RC1'
    bootVersion = '2.0.0.BUILD-SNAPSHOT'
    springDataVersion = '2.0.0.M3'

    // Библиотеки протоколирования
    slf4jVersion = '1.7.25'
    logbackVersion = '1.2.3'
    guavaVersion = '21.0'
    junitVersion = '4.12'

    aspectjVersion = '1.9.0.BETA-5'

    // Библиотека базы данных
    h2Version = '1.4.194'

    // Библиотеки сохраняемости
    hibernateVersion = '5.2.10.Final'
    hibernateJpaVersion = '1.0.0.Final'
    atomikosVersion = '4.0.0M4'

    spring = [
        context : "org.springframework:spring-context:
                  $springVersion",
        aop : "org.springframework:spring-aop:
               $springVersion",
        aspects : "org.springframework:spring-aspects:
                  $springVersion",
        tx : "org.springframework:spring-tx:
              $springVersion",
        jdbc : "org.springframework:spring-jdbc:
                $springVersion",
        contextSupport: "org.springframework:
                        spring-context-support:
                        $springVersion",
        orm : "org.springframework:spring-orm:
               $springVersion",
        data : "org.springframework.data:spring-data-jpa:
                $springDataVersion",
```

```
        test : "org.springframework:spring-test:
                $springVersion"
    ]

    hibernate = [
        ...
        em : "org.hibernate:hibernate-entitymanager:
              $hibernateVersion",
        tx : "com.atomikos:transactions-hibernate4:
              $atomikosVersion"
    ]

    boot = [
        ...
        springBootPlugin : "org.springframework.boot:
                            spring-boot-gradle-plugin:
                            $bootVersion",
        starterJpa : "org.springframework.boot:
                      spring-boot-starter-data-jpa:
                      $bootVersion"
    ]

    testing = [
        junit: "junit:junit:$junitVersion"
    ]

    misc = [
        ...
        slf4jJcl : "org.slf4j:jcl-over-slf4j:
                    $slf4jVersion",
        logback : "ch.qos.logback:logback-classic:
                   $logbackVersion",
        aspectjweaver: "org.aspectj:aspectjweaver:
                        $aspectjVersion",
        lang3 : "org.apache.commons:commons-lang3:3.5",
        guava : "com.google.guava:guava:$guavaVersion"
    ]

    db = [
        ...
        h2 : "com.h2database:h2:$h2Version"
    ]
}

// Файл конфигурации chapter09/build.gradle
dependencies {
    // Эти зависимости указываются для всех подмодулей,
    // кроме модуля начальной загрузки,
    // который определяется отдельно
```

```
    if !project.name.contains"boot" {
      // исключить транзитивные зависимости,
      // поскольку об этом позаботится модуль spring-data
      compile spring.contextSupport {
      exclude module: 'spring-context'
      exclude module: 'spring-beans'
      exclude module: 'spring-core'
    }
    // исключить транзитивную зависимость 'hibernate',
    // чтобы получить контроль над применяемой версией
    compile hibernate.tx {
      exclude group: 'org.hibernate', module: 'hibernate'
    }

    compile spring.orm, spring.context, misc.slf4jJcl,
            misc.logback, db.h2, misc.lang3,
    hibernate.em
  }
  testCompile testing.junit
}
```

Чтобы исследовать поведение исходного кода примера при изменении свойств транзакции, установим также уровень протоколирования `DEBUG` в файле конфигурации `logback.xml`. Ниже приведен соответствующий фрагмент кода из этого файла.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

    <contextListener class="ch.qos.logback.classic.jul
                            .LevelChangePropagator">
        <resetJUL>true</resetJUL>
    </contextListener>

    <appender name="console" class="ch.qos.logback.core
                                    .ConsoleAppender">
         <encoder>
             <pattern>%d{HH:mm:ss.SSS}
                      %thread %-5level %logger{5} - %msg%n
             </pattern>
        </encoder>
    </appender>

    <logger name=
            "com.apress.prospring5.ch8" level="debug"/>

    <logger name="org.springframework.transaction"
           level="info"/>

    <logger name="org.hibernate.SQL" level="debug"/>
```

```
    <root level="info">
        <appender-ref ref="console" />
    </root>
</configuration>
```

Модель выборочных данных и общие классы

Чтобы упростить дело, воспользуемся в рассматриваемых далее примерах двумя таблицами, SINGER и ALBUM, применявшимися в предыдущих главах, посвященных средствам доступа к данным. Для создания этих таблиц никакого сценария SQL не требуется, поскольку можно установить значение create-drop в свойстве hibernate.hbm2ddl.auto из библиотеки Hibernate, чтобы всегда начинать выполнение заново всякий раз, когда требуется что-нибудь проверить. Обе таблицы будут сформированы на основании сущностей типа Singer и Album. Ниже приведены фрагменты кода с аннотированными полями из каждой таблицы.

```
// Исходный файл Singer.java
package com.apress.prospring5.ch9.entities;
...

@Entity
@Table(name = "singer")
@NamedQueries({
      @NamedQuery(name=Singer.FIND_ALL,
                  query="select s from Singer s"),
      @NamedQuery(name=Singer.COUNT_ALL,
                  query="select count(s) from Singer s")
})
public class Singer implements Serializable {

  public static final String FIND_ALL = "Singer.findAll";
  public static final String COUNT_ALL = "Singer.countAll";

  @Id
  @GeneratedValue(strategy = IDENTITY)
  @Column(name = "ID")
  private Long id;

  @Version
  @Column(name = "VERSION")
  private int version;

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

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

```
  @Temporal(TemporalType.DATE)
  @Column(name = "BIRTH_DATE")
  private Date birthDate;

  @OneToMany(mappedBy = "singer", cascade=CascadeType.ALL,
             orphanRemoval=true)
  private Set<Album> albums = new HashSet<>();
  ...
}

// Исходный файл Album.java
package com.apress.prospring5.ch9.entities;
...

@Entity
@Table(name = "album")
public class Album implements Serializable {
  @Id
  @GeneratedValue(strategy = IDENTITY)
  @Column(name = "ID")
  private Long id;

  @Version
  @Column(name = "VERSION")
  private int version;

  @Column
  private String title;

  @Temporal(TemporalType.DATE)
  @Column(name = "RELEASE_DATE")
  private Date releaseDate;

  @ManyToOne
  @JoinColumn(name = "SINGER_ID")
  private Singer singer;
  ...
}
```

Классы `Singer` и `Album` будут изолированы в проекте `base-dao`, который послужит в качестве зависимости для всех проектов транзакций. Помимо сущностей, в данном проекте определяются интерфейсы информационного хранилища, которые будут описаны несколько позже. Потребуется также приведенный ниже конфигурационный класс, чтобы определить компонент Spring Bean типа `DataSource`. Исходя из практических и учебных целей, учетные данные, драйвер и URL для подключения к базе данных будут задаваться непосредственно в коде, а не вводиться из внешнего файла, как это обычно происходит в реальных условиях эксплуатации приложений.

```
package com.apress.prospring5.ch9.config;
...

@Configuration
@EnableJpaRepositories(basePackages =
                       {"com.apress.prospring5.ch9.repos"})
public class DataJpaConfig {

  private static Logger logger =
          LoggerFactory.getLogger(DataJpaConfig.class);

  @SuppressWarnings("unchecked")
  @Bean
  public DataSource dataSource() {
    try {
      SimpleDriverDataSource dataSource =
                  new SimpleDriverDataSource();
      Class<? extends Driver> driver =
           (Class<? extends Driver>)
            Class.forName("org.h2.Driver");
      dataSource.setDriverClass(driver);
      dataSource.setUrl("jdbc:h2:musicdb");
      dataSource.setUsername("prospring5");
      dataSource.setPassword("prospring5");
      return dataSource;
    } catch (Exception e) {
      logger.error("Populator DataSource bean cannot "
                + "be created!", e);
      return null;
    }
  }

  @Bean
  public Properties hibernateProperties() {
    Properties hibernateProp = new Properties();
    hibernateProp.put("hibernate.dialect",
                      "org.hibernate.dialect.H2Dialect");
    hibernateProp.put("hibernate.hbm2ddl.auto",
                      "create-drop");
    // hibernateProp.put("hibernate.format_sql", true);
    hibernateProp.put("hibernate.show_sql", true);
    hibernateProp.put("hibernate.max_fetch_depth", 3);
    hibernateProp.put("hibernate.jdbc.batch_size", 10);
    hibernateProp.put("hibernate.jdbc.fetch_size", 50);
    return hibernateProp;
  }

  @Bean
  public JpaVendorAdapter jpaVendorAdapter() {
```

```
    return new HibernateJpaVendorAdapter();
  }

  @Bean
  public EntityManagerFactory entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean factoryBean =
         new LocalContainerEntityManagerFactoryBean();
    factoryBean.setPackagesToScan(
                "com.apress.prospring5.ch9.entities");
    factoryBean.setDataSource(dataSource());
    factoryBean.setJpaVendorAdapter(
                new HibernateJpaVendorAdapter());
    factoryBean.setJpaProperties(hibernateProperties());
    factoryBean.setJpaVendorAdapter(jpaVendorAdapter());
    factoryBean.afterPropertiesSet();
    return factoryBean.getNativeEntityManagerFactory();
  }
}
```

В приведенном выше коде определяется встроенная база данных H2. Учетные данные задаются непосредственно в коде, а интерфейс `DataSource` реализуется в классе `SimpleDriverDataSource`, предназначенном для применения только в простых, тестирующих или учебных целях.

В настоящее время для определения требований к транзакциям в Spring чаще всего применяются аннотации. Их главное преимущество заключается в том, что требование к транзакции вместе с ее свойствами, определяющими время ожидания, уровень изоляции, режим распространения и прочее, задаются непосредственно в коде, благодаря чему упрощается отслеживание изменений и сопровождение приложений. Конфигурирование выполняется также с помощью аннотаций и конфигурационных классов Java. А для того чтобы активизировать поддержку аннотаций для управления транзакциями в Spring, используя конфигурацию в формате XML, необходимо ввести дескриптор разметки `<tx:annotation-driven>` в XML-файл конфигурации. Ниже приведен фрагмент кода транзакционной конфигурации с соответствующими транзакционными пространствами имен. Если же вас заинтересует полная конфигурация, ее можно найти в файле `tx-annotation-app-context.xml` рассматриваемого здесь проекта в предыдущем издании.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       ...
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation=
          "http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans
                /spring-beans.xsd
          http://www.springframework.org/schema/tx
```

```
        http://www.springframework.org/schema/tx
              /spring-tx.xsd ...">

    <bean id="transactionManager"
        class="org.springframework.orm.jpa
                  .JpaTransactionManager">
        <property name="entityManagerFactory" ref="emf"/>
    </bean>

    <tx:annotation-driven />

    <bean id="emf"
        class="org.springframework.orm.jpa
                  .LocalContainerEntityManagerFactoryBean">
      ...
    </bean>

    <context:component-scan
        base-package="com.apress.prospring5.ch9" />

    <jpa:repositories base-package=
             "com.apress.prospring5.ch9.repos"
             entity-manager-factory-ref="emf"
             transaction-manager-ref="transactionManager"/>
</beans>
```

В данном случае применяется прикладной интерфейс JPA, поэтому в приведенной выше конфигурации определяется компонент Spring Bean типа `JpaTransaction Manager`. Так, в дескрипторе разметки `<tx:annotation-driven>` указывается, что для управления транзакциями применяются аннотации. В столь простом определении каркасу Spring предписывается искать компонент `transactionManager` типа `PlatformTransactionManager`. Если же транзакционный компонент называется иначе (скажем, `customTransactionManager`), то определение элемента должно быть объявлено с помощью атрибута `transaction-manager`, которому должно быть присвоено в качестве значения имя компонента Spring Bean, предназначенного для управления транзакциями.

```
<tx:annotation-driven transaction-manager=
    "customTransactionManager"/>
```

Затем в рассматриваемой здесь конфигурации определяется компонент Spring Bean типа `EntityManagerFactory`, а вслед за ним режим просмотра классов уровня обслуживания в дескрипторе разметки `<context:component-scan>`. И, наконец, в дескрипторе `<jpa:repositories>` активизируется абстракция информационного хранилища в Spring Data JPA. Этот элемент разметки заменяется конфигурационным классом `DataJpaConfiguration` с аннотацией `@EnableJpaReposito ries`.

В профессиональных средах обычно принято отделять конфигурацию сохраняемости (объект DAO) от транзакционной конфигурации (службы). Именно поэтому содержимое представленного ранее XML-файла конфигурации разбито на два конфигурационных класса Java. Так, представленный ранее класс `DataJpaConfig` содержит только компоненты Spring Beans, предназначенные для доступа к данным, а приведенный ниже класс `ServicesConfig` — только компоненты Spring Beans, имеющие отношение к управлению транзакциями.

```
package com.apress.prospring5.ch9.config;

import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.orm.jpa
          .JpaTransactionManager;
import org.springframework.transaction
          .PlatformTransactionManager;
import org.springframework.transaction.annotation
          .EnableTransactionManagement;
import javax.persistence.EntityManagerFactory;

@Configuration
@EnableTransactionManagement
@ComponentScan(basePackages = "com.apress.prospring5.ch9")
public class ServicesConfig {
  @Autowired EntityManagerFactory entityManagerFactory;

  @Bean
  public PlatformTransactionManager transactionManager() {
    return new JpaTransactionManager(entityManagerFactory);
  }
}
```

Чтобы реализовать интерфейс `SingerService`, создадим сначала класс с пустой реализацией всех методов из интерфейса `SingerService`, а затем реализуем метод `SingerService.findAll()`. Ниже приведен исходный код класса `SingerServiceImpl` с реализованным методом `findAll()`.

```
package com.apress.prospring5.ch9.services;

import com.apress.prospring5.ch9.entities.Singer;
import com.apress.prospring5.ch9.repos.SingerRepository;
import com.google.common.collect.Lists;
import org.springframework.beans.factory.annotation
```

```
          .Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation
          .Propagation;
import org.springframework.transaction.annotation
          .Transactional;
import java.util.List;

@Service("singerService")
@Transactional
public class SingerServiceImpl implements SingerService {

  private SingerRepository singerRepository;

  @Autowired
  public void setSingerRepository(
         SingerRepository singerRepository) {
     this.singerRepository = singerRepository;
  }

  @Override
  @Transactional(readOnly = true)
  public List<Singer> findAll() {
    return Lists.newArrayList(singerRepository.findAll());
  }
}
```

Если управление транзакциями основывается на аннотациях, то приходится иметь дело только с аннотацией `@Transactional`. Так, в приведенном выше коде аннотация `@Transactional` применяется на уровне класса, а это означает, что по умолчанию каркас Spring гарантирует, что транзакция должна присутствовать прежде выполнения каждого метода из класса. В аннотации `@Transactional` поддерживается целый ряд атрибутов, которые могут быть предоставлены для переопределения стандартного поведения. Доступные атрибуты, а также их возможные значения по умолчанию приведены в табл. 9.3.

Таблица 9.3. Атрибуты, доступные в аннотации `@Transactional`

Имя атрибута	Значение по умолчанию	Возможные значения

`propagation`	`Propagation.REQUIRED`	`Propagation.REQUIRED`
		`Propagation.SUPPORTS`
		`Propagation.MANDATORY`
		`Propagation.REQUIRES_NEW`
		`Propagation.NOT_SUPPORTED`
		`Propagation.NEVER`
		`Propagation.NESTED`

Окончание табл. 9.3

Имя атрибута	Значение по умолчанию	Возможные значения

`isolation`	`Isolation.DEFAULT` (уровень изоляции базового ресурса по умолчанию)	`Isolation.DEFAULT`
		`Isolation.READ_UNCOMMITTED`
		`Isolation.READ_COMMITTED`
		`Isolation.REPEATABLE_READ`
		`Isolation.SERIALIZABLE`

`timeout`	`TransactionDefinition.TIMEOUT_DEFAULT` (время ожидания базового ресурса в секундах по умолчанию)	Целочисленное значение больше нуля, обозначающее количество секунд времени ожидания
`readOnly`	`false`	`{true, false}`
`rollbackFor`	Классы исключений, для которых будет произведен откат транзакции	Отсутствуют
`rollbackForClassName`	Классы исключений, для которых будет произведен откат транзакции	Отсутствуют
`noRollbackFor`	Классы исключений, для которых будет произведен откат транзакции	Отсутствуют
`noRollbackForClassName`	Классы исключений, для которых будет произведен откат транзакции	Отсутствуют
`value`	`""` (значение описателя указанной транзакции)	Отсутствуют

Исходя из данных, приведенных в табл. 9.3, аннотация `@Transactional` без всяких атрибутов означает, что транзакция должна быть распространена, а также выбран уровень изоляции, время ожидания по умолчанию, режим чтения и записи. Так, если обратиться к представленному ранее методу `findAll()`, то следует отметить, что он снабжен аннотацией `@Transactional(readOnly=true)`. Она переопределяет аннотацию, применяемую по умолчанию на уровне класса, не изменяя ни одного атрибута, но устанавливая транзакцию в режим доступа только для чтения. Ниже приведена тестовая программа для проверки метода `findAll()`.

```
package com.apress.prospring5.ch9;

import java.util.List;
import com.apress.prospring5.ch9.config.DataJpaConfig;
import com.apress.prospring5.ch9.config.ServicesConfig;
import com.apress.prospring5.ch9.entities.Singer;
import com.apress.prospring5.ch9.services.SingerService;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;

public class TxAnnotationDemo {
```

```
  public static void main(String... args) {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               ServicesConfig.class, DataJpaConfig.class);

    SingerService singerService =
                  ctx.getBean(SingerService.class);

    List<Singer> singers = singerService.findAll();
    singers.forEach(s -> System.out.println(s));
    ctx.close();
  }
}
```

Если активизирован надлежащий режим протоколирования, например, следующим образом:

```
<logger name="org.springframework.orm.jpa" level="debug"/>
```

то на консоль будут выведены протокольные сообщения, связанные с обработкой транзакций. Выполнение данной тестовой программы приведет к выводу на консоль приведенного ниже результата (подробнее см. журнал отладки, выводимый на консоль).

```
DEBUG o.s.o.j.JpaTransactionManager
   - Creating new transaction with name[4]
     [com.apress.prospring5.ch9.services
         .SingerServiceImpl.findAll]:
   PROPAGATION_REQUIRED,ISOLATION_DEFAULT,readOnly; ''
DEBUG o.s.o.j.JpaTransactionManager
   - Participating in existing transaction[5]
Hibernate: select singer0_.ID as ID1_1_,
           singer0_.BIRTH_DATE as BIRTH_DA2_1_,
           singer0_.FIRST_NAME as FIRST_NA3_1_,
           singer0_.LAST_NAME as LAST_NAM4_1_,
           singer0_.VERSION as VERSION5_1_
           from singer singer0_
DEBUG o.s.o.j.JpaTransactionManager
   - Closing JPA EntityManager
     [...] after transaction[6]

DEBUG o.s.o.j.JpaTransactionManager
   - Initiating transaction commit[7]
```

[4] Создание новой транзакции под именем...

[5] Участие в существующей транзакции

[6] После транзакции

[7] Инициирование фиксации транзакции

```
Singer - Id: 1, First name: John, Last name: Mayer,
   Birthday: 1977-10-16
Singer - Id: 2, First name: Eric, Last name: Clapton,
   Birthday: 1945-03-30
Singer - Id: 3, First name: John, Last name: Butler,
   Birthday: 1975-04-01
```

Ради большей ясности из приведенного выше результата исключены не относящиеся к делу операторы вывода. Прежде всего перед выполнением метода `findAll()` в компоненте Spring Bean типа `JpaTransactionManager` создается новая транзакция с атрибутами по умолчанию и под именем, полностью совпадающим с уточненным именем класса и его метода, но эта транзакция становится доступной только для чтения, как определено в аннотации `@Transactional`, объявляемой на уровне метода. Затем передается запрос, а по его завершении без ошибок транзакция фиксируется. Компонент типа `JpaTransactionManager` отвечает за выполнение операций создания и фиксации транзакций.

А теперь перейдем к реализации операции обновления. Для этого необходимо реализовать методы `findById()` и `save()` из интерфейса `SingerService`, как показано ниже.

```
package com.apress.prospring5.ch9.services;
...

@Service("singerService")
@Transactional
public class SingerServiceImpl implements SingerService {
  private SingerRepository singerRepository;

  @Autowired
  public void setSingerRepository(
              SingerRepository singerRepository) {
    this.singerRepository = singerRepository;
  }

  @Override
  @Transactional(readOnly = true)
  public List<Singer> findAll() {
    return Lists.newArrayList(singerRepository.findAll());
  }

  @Override
  @Transactional(readOnly = true)
  public Singer findById(Long id) {
    return singerRepository.findById(id).get();
  }

  @Override
  public Singer save(Singer singer) {
```

```
    return singerRepository.save(singer);
  }
}
```

Метод `findById()` также снабжен аннотацией `@Transactional(readOnly=true)`. Как правило, атрибут `readOnly=true` должен применяться ко всем методам поиска. Объясняется это, главным образом, тем, что большинство поставщиков услуг сохраняемости выполняют оптимизацию доступных только для чтения транзакций на определенном уровне. Например, в Hibernate не поддерживаются моментальные снимки управляемых экземпляров, извлекаемых из базы данных, работающей в режиме только для чтения.

Что же касается метода `save()`, то в нем просто вызывается метод `CrudRepository.save()` и не объявляется никакой аннотации. Это означает, что будет использована аннотация, определяющая доступную только для чтения транзакцию на уровне класса. А теперь внесем коррективы в класс `TxAnnotationDemo`, чтобы проверить метод `save()`, как показано ниже.

```
package com.apress.prospring5.ch9;
...
public class TxAnnotationDemo {
  public static void main(String... args) {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               ServicesConfig.class, DataJpaConfig.class);

    SingerService singerService =
          ctx.getBean(SingerService.class);

    List<Singer> singers = singerService.findAll();
    singers.forEach(s -> System.out.println(s));

    Singer singer = singerService.findById(1L);
    singer.setFirstName("John Clayton");
    singer.setLastName("Mayer");
    singerService.save(singer);
    System.out.println("Singer saved successfully: "
                    + singer);
    ctx.close();
  }
}
```

Сначала в приведенном выше тестовом коде извлекается объект типа `Singer` с идентификатором `1`, а затем имя певца обновляется и сохраняется в базе данных. После выполнения этого тестового кода на консоль будет выведен следующий результат:

```
Singer saved successfully: Singer - Id: 1,
       First name: John Clayton,
Last name: Mayer, Birthday: 1977-10-16
```

Метод `save()` получает атрибуты, наследуемые по умолчанию из аннотации `@Transactional`, объявляемой на уровне класса. По завершении операции обновления компонент Spring Bean типа `JpaTransactionManager` инициирует фиксацию транзакции, в результате чего библиотека Hibernate очищает контекст сохраняемости и фиксирует исходное соединение с базой данных через интерфейс JDBC.

И, наконец, рассмотрим метод `countAll()` и, в частности, две конфигурации транзакций. Несмотря на то что для этой цели вполне подойдет метод `CrudRepository.count()`, он здесь не применяется. Вместо этого для целей демонстрации реализуется другой метод, главным образом, потому, что методы, определяемые с помощью интерфейса `CrudRepository` в проекте Spring Data, уже помечены соответствующими атрибутами транзакций.

Ниже показано, как новый метод `countAllSingers()` определяется в интерфейсе `SingerRepository`.

```
package com.apress.prospring5.ch9.repos;

import com.apress.prospring5.ch9.entities.Singer;
import org.springframework.data.jpa.repository
          .JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository
          .CrudRepository;

public interface SingerRepository
       extends CrudRepository<Singer, Long> {
  @Query("select count(s) from Singer s")
  Long countAllSingers();
}
```

К вновь определяемому методу `countAllSingers()` применяется аннотация `@Query` со значением, обозначающим оператор языка JPQL, в котором подсчитывается количество певцов в базе данных. Ниже приведена реализация метода `countAll()` в классе `SingerServiceImpl`.

```
package com.apress.prospring5.ch9.services;
...

@Service("singerService")
@Transactional
public class SingerServiceImpl implements SingerService {

  private SingerRepository singerRepository;

  @Autowired
```

```
  public void setSingerRepository(
              SingerRepository singerRepository) {
    this.singerRepository = singerRepository;
  }

  @Override
  @Transactional(readOnly=true)
  public long countAll() {
    return singerRepository.countAllSingers();
  }
}
```

Аннотация `@Transactional`, применяемая к методу `countAll()`, ничем не отличается от аналогичных аннотаций в других методах поиска. Чтобы поверить метод `countAll()`, достаточно ввести приведенный ниже оператор в тело метода `main()` из класса `TxAnnotationDemo`, выполнить его исходный код и понаблюдать за выводом на консоль. Если в итоге появится сообщение вроде `"Singer count: 3"`, значит, тестируемый метод был выполнен правильно.

```
System. out.println("Singer count: "
                  + contactService.countAll());
```

Из результата, выводимого на консоль, ясно видно, что для метода `countAll()` была, как и предполагалось, создана транзакция, доступная только для чтения. Но ведь к этому методу вообще не требуется привлекать транзакцию, а также поручать управление выводимым результатом базовому диспетчеру сущностей типа `EntityManager` из прикладного интерфейса JPA. Вместо этого требуется лишь получить подсчитанное количество певцов в базе данных. В таком случае можно сменить режим распространения транзакций на `Propagation.NEVER`. Ниже приведен соответственно переделанный вариант метода `countAll()`.

```
package com.apress.prospring5.ch9.services;
...

@Service("singerService")
@Transactional
public class SingerServiceImpl implements SingerService {
  ...
  @Override
  @Transactional(propagation = Propagation.NEVER)
  public long countAll() {
    return singerRepository.countAllSingers();
  }
}
```

Если выполнить тестовый код снова, то можно обнаружить, что транзакция не будет создана для метода `countAll()` при отладочном выводе. В этом разделе были рассмотрены основные конфигурации, которые приходится настраивать при повсед-

невной обработке транзакций. В особых случаях, возможно, придется определить время ожидания, уровень изоляции, откат (или его отсутствие) для конкретных исключительных ситуаций и т.д.

На заметку В компоненте Spring Bean типа **`JpaTransactionManager`** не поддерживается специальный уровень изоляции. Вместо этого в нем всегда применяется уровень изоляции, устанавливаемый по умолчанию для базового информационного хранилища. Если в качестве поставщика услуг сохраняемости в прикладном интерфейсе применяется библиотека Hibernate, то в качестве обходного приема можно расширить класс **`HibernateJpaDialect`** для поддержки специального уровня изоляции.

Конфигурирование управления транзакциями в АОП

Еще один общий подход к декларативному управлению транзакциями состоит в том, чтобы воспользоваться поддержкой АОП в Spring. До версии Spring 2 требовалось пользоваться классом `TransactionProxyFactoryBean`, чтобы определить требования к транзакциям для компонентов Spring Beans. Но, начиная с версии 2, в Spring предоставляется намного более простой способ внедрения пространства имен `aop` и применения общей методики конфигурирования в АОП для определения требований к транзакциям. Но и такой способ конфигурирования транзакций, безусловно, устарел после внедрения аннотаций. Впрочем, о нем полезно все же знать на тот случай, если потребуется инкапсулировать код транзакций, не относящийся к текущему проекту, а отредактировать его нельзя, чтобы ввести в него аннотации `@Transaction`.

В следующем фрагменте кода пример из предыдущего раздела сконфигурирован в формате XML, чтобы воспользоваться пространством имен `aop`:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi=
            "http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context=
            "http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans
                 /spring-beans.xsd
           http://www.springframework.org/schema/tx
           http://www.springframework.org/schema/tx
                 /spring-tx.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context
                 /spring-context.xsd
```

```
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop
              /spring-aop.xsd">

    <bean name="dataJpaConfig"
         class="com.apress.prospring5.ch9
                   .config.DataJpaConfig" />

    <aop:config>
        <aop:pointcut id="serviceOperation"
               expression= "execution(* com.apress
                     .prospring5.ch9.*ServiceImpl.*(..))"/>
        <aop:advisor pointcut-ref="serviceOperation"
                       advice-ref="txAdvice"/>
    </aop:config>

    <tx:advice id="txAdvice">
        <tx:attributes>
             <tx:method name="find*" read-only="true"/>
             <tx:method name="count*" propagation="NEVER"/>
             <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

    <bean id="transactionManager"
       class="org.springframework.orm.jpa
                  .JpaTransactionManager
         <property name="entityManagerFactory"
                    ref="entityManagerFactory"/>
    </bean>

    <context:component-scan
       base-package="com.apress.prospring5.ch9.services" />
</beans>
```

Приведенная выше конфигурация очень похожа на конфигурацию в формате XML, представленную вначале этого раздела. По существу, дескриптор разметки `<tx:annotation-driven>` исключен из этой конфигурации, а в дескрипторе разметки `<context:component-scan>` изменено имя пакета, применяемого для декларативного управления транзакциями. Самыми важными в этой конфигурации являются дескрипторы разметки `<aop:config>` и `<tx:advice>`.

В дескрипторе разметки `<aop:config>` определяется срез для всех операций на уровне услуг (т.е. всех реализаций в классах из пакета `com.apress.prospring5.ch9.services`), а в дескрипторе разметки `<tx:advice>` — совет со ссылкой на компонент Spring Bean с идентификатором `txAdvice`. Кроме того, в дескрипторе разметки `<tx:advice>` задаются атрибуты транзакции для различных методов, которые должны участвовать в транзакции. В частности, все методы поиска (с префиксом **`find`**) указываются в этом дескрипторе как доступные только для чтения, а все

методы подсчета (с префиксом **count**) — как не участвующие в транзакции. А для остальных методов будет действовать устанавливаемый по умолчанию режим распространения транзакций. Данная конфигурация подобна той, что приведена в примере с аннотациями.

Управление транзакциями осуществляется непосредственно через пространство имен `aop`, и поэтому аннотация `@Transactional` больше не требуется в классе `SingerServiceImpl` или его методах. Чтобы проверить описанную выше конфигурацию, можно воспользоваться следующим классом:

```
package com.apress.prospring5.ch9;

import java.util.List;
import com.apress.prospring5.ch9.entities.Singer;
import com.apress.prospring5.ch9.services.SingerService;
import org.springframework.context.support
          .GenericXmlApplicationContext;

public class TxDeclarativeDemo {
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
          new GenericXmlApplicationContext();
    ctx.load("classpath:"
           + "spring/tx-declarative-app-context.xml");
    ctx.refresh();

    SingerService singerService =
          ctx.getBean(SingerService.class);

    // Проверка метода findAll()
    List<Singer> singers = singerService.findAll();
    singers.forEach(s -> System.out.println(s));

    // Проверка метода save()
    Singer singer = singerService.findById(1L);
    singer.setFirstName("John Clayton");
    singerService.save(singer);
    System.out.println("Singer saved successfully: "
                    + singer);

    // Проверка метода countAll()
    System.out.println("Singer count: "
                    + singerService.countAll());
    ctx.close();
  }
}
```

Тестирование исходного кода из данного примера и анализ выводимых на консоль результатов выполнения операций, связанных с обработкой транзакций в Spring и

Hibernate, оставляется вам в качестве упражнения. По существу, это делается так же, как и в примере с аннотациями.

Применение программных транзакций

Третий способ предусматривает программное управление поведением транзакций. В этом случае доступны две возможности. Во-первых, внедрить экземпляр типа `PlatformTransactionManager` в компонент Spring Bean и взаимодействовать непосредственно с диспетчером транзакций. И, во-вторых, воспользоваться предоставляемым в Spring классом `TransactionTemplate`, значительно упрощающим дело. В этом разделе будет продемонстрировано взаимодействие с классом `TransactionTemplate`. Ради простоты сосредоточим основное внимание только на реализации метода `SingerService.countAll()`. Ниже приведен вариант конфигурационного класса `ServiceConfig`, переделанный для применения программных транзакций.

```
package com.apress.prospring5.ch9.config;

import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context
          .annotation.ComponentScan;
import org.springframework.context
          .annotation.Configuration;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.transaction
          .PlatformTransactionManager;
import org.springframework.transaction
          .TransactionDefinition;
import org.springframework.transaction.support
          .TransactionTemplate;
import javax.persistence.EntityManagerFactory;

@Configuration
@ComponentScan(basePackages = "com.apress.prospring5.ch9")
public class ServicesConfig {

  @Autowired EntityManagerFactory entityManagerFactory;

  @Bean
  public TransactionTemplate transactionTemplate() {
    TransactionTemplate tt = new TransactionTemplate();
    tt.setPropagationBehavior(
       TransactionDefinition.PROPAGATION_NEVER);
    tt.setTimeout(30);
    tt.setTransactionManager(transactionManager());
    return tt;
  }
```

```
  @Bean
  public PlatformTransactionManager transactionManager() {
    return new JpaTransactionManager(entityManagerFactory);
  }
}
```

Из приведенного выше конфигурационного класса исключен совет АОП для транзакции. Кроме того, компонент `transactionTemplate` определен в нем вместе с несколькими атрибутами с помощью класса `org.springframework.transaction.support.TransactionTemplate`. Из данного конфигурационного класса удалена также аннотация `@EnableTransactionManagement`, поскольку управление транзакциями теперь осуществляется не вручную. Ниже показано, каким образом реализуется метод `countAll()`.

```
package com.apress.prospring5.ch9.services;

import com.apress.prospring5.ch9.entities.Singer;
import com.apress.prospring5.ch9.repos.SingerRepository;
import com.google.common.collect.Lists;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.support
          .TransactionTemplate;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.List;

@Service("singerService")
@Repository
public class SingerServiceImpl implements SingerService {
  @Autowired
  private SingerRepository singerRepository;

  @Autowired
  private TransactionTemplate transactionTemplate;

  @PersistenceContext
  private EntityManager em;

  @Override
  public long countAll() {
    return transactionTemplate.execute(
    transactionStatus -> em.createNamedQuery(
                           Singer.COUNT_ALL, Long.class)
                          .getSingleResult());
  }
}
```

В приведенном выше коде сначала внедряется класс `TransactionTemplate` из Spring. Затем в методе `countAll()` вызывается метод `TransactionTemplate.execute()`, которому передается лямбда-выражение, назначение которого не совсем ясно. Поэтому ниже приведена расширенная версия метода `countAll()`, написанная до внедрения лямбда-выражений. Этому методу передается внутренний класс, где реализуется интерфейс `TransactionCallback<T>` и переопределяется метод `doInTransaction()` с требуемой логикой. Эта логика будет действовать в атрибутах, как определено в компоненте `transactionTemplate`.

```
public long countAll() {
  return transactionTemplate.execute(
    new TransactionCallback<Long>() {
      public Long doInTransaction(
        TransactionStatus transactionStatus) {
          return em.createNamedQuery(Singer.COUNT_ALL,
                                     Long.class).getSingleResult();
      }
  });
}
```

Исходный код соответствующей тестовой программы приведен ниже.

```
package com.apress.prospring5.ch9;

import com.apress.prospring5.ch9.config.DataJpaConfig;
import com.apress.prospring5.ch9.config.ServicesConfig;
import com.apress.prospring5.ch9.services.SingerService;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;

public class TxProgrammaticDemo {
  public static void main(String... args) {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               ServicesConfig.class, DataJpaConfig.class);
    SingerService singerService =
                  ctx.getBean(SingerService.class);
    System.out.println("Singer count: "
                    + singerService.countAll());

    ctx.close();
  }
}
```

Оставляем вам выполнение этой тестовой программы и анализ полученных результатов в качестве упражнения. Попробуйте настроить атрибуты транзакции и посмотреть, что произойдет при обработке транзакции из метода `countAll()`.

Соображения по поводу управления транзакциями

После обсуждения разных способов реализации управления транзакциями осталось выяснить, какой из них следует выбрать для практического применения. Декларативный способ рекомендуется применять во всех случаях, стараясь по возможности не реализовывать управление транзакциями непосредственно в коде. Если же возникает потребность реализовать логику управления транзакциями в прикладном коде, это зачастую объясняется неудачным проектным решением. В таком случае реализуемую логику следует вынести в управляемые фрагменты кода и декларативно определить в них требования к транзакциям.

Конфигурированию в формате XML и с помощью аннотаций при декларативном подходе присущи свои достоинства и недостатки. Одни разработчики предпочитают не объявлять требования к транзакциям в прикладном коде, тогда как другие отдают предпочтение аннотациям из-за простоты их сопровождения, поскольку в исходном коде можно обнаружить все объявления требований к транзакциям. И в этом случае принимать решение следует, исходя из требований к приложению. Если же в команде разработчиков или в организации стандартизирован конкретный стиль конфигурирования, то необходимо придерживаться именно его.

Обработка глобальных транзакций в Spring

Во многих корпоративных приложениях на Java требуется доступ к различным ресурсам сервера базы данных. Например, при получении фрагмента информации о заказчике от внешнего бизнес-партнера может потребоваться обновление баз данных сразу в нескольких системах (CRM, ERP и т.д.), а в ряде случаев — формирование и отправка соответствующего сообщения серверу MQ через службу JMS для всех остальных приложений в компании, которые заинтересованы в информации о заказчиках. Транзакции, которые охватывают многие ресурсы сервера базы данных, называются *глобальными* (или *распределенными*).

Главной особенностью глобальной транзакции является гарантия атомарности. Это означает, что обновлению подлежат все задействованные ресурсы, а иначе — ни один из них. Диспетчер транзакций должен поддерживать сложную логику координации и синхронизации. В среде Java фактическим стандартом для реализации глобальных транзакций служит прикладной интерфейс JTA.

Транзакции JTA поддерживаются в Spring так же полно, как и локальные транзакции, а соответствующая логика скрывается от кода предметной области. В этом разделе поясняется, каким образом глобальные транзакции реализуются средствами JTA в Spring.

Инфраструктура для реализации примера применения JTA

Воспользуемся в рассматриваемом здесь примере той же таблицей, что и в предыдущих примерах из этой главы. Но в связи с тем, что во встроенной базе данных H2 отсутствует полная поддержка протокола XA (во всяком случае, на момент написания данной книги), в данном примере в качестве базы данных на сервере применяется MySQL.

Требуется также показать, каким образом глобальные транзакции реализуются средствами JTA в автономном приложении или в среде веб-контейнера. Поэтому в данном примере применяется Atomikos (`http://www.atomikos.com/Main/TransactionsEssentials`) —широко распространенный диспетчер транзакций JTA с открытым кодом, предназначенный для сред, отличающихся от JEE.

Для демонстрации глобальных транзакций в действии потребуются по крайней мере два ресурса сервера базы данных. Ради простоты в данном примере используется одна база данных MySQL, но два диспетчера сущностей JPA. Это даст один и тот же результат, поскольку для различения баз данных на сервере имеется несколько единиц сохраняемости JPA.

Итак, создадим в базе данных MySQL две схемы и соответствующих пользователей, как показано в приведенном ниже сценарии DDL. Завершив подготовку базы данных к работе, можно перейти непосредственно к конфигурированию и реализации глобальных транзакций в Spring.

```
CREATE USER 'prospring5_a'@'localhost'
       IDENTIFIED BY 'prospring5_a';
CREATE SCHEMA MUSICDB_A;
GRANT ALL PRIVILEGES ON MUSICDB_A . *
      TO 'prospring5_a'@'localhost';
      PRIVILEGES;

CREATE USER 'prospring5_b'@'localhost'
       IDENTIFIED BY 'prospring5_b';
CREATE SCHEMA MUSICDB_B;
GRANT ALL PRIVILEGES ON MUSICDB_B . *
      TO 'prospring5_b'@'localhost';
      PRIVILEGES;
```

Реализация глобальных транзакций средствами JTA

Рассмотрим сначала конфигурирование глобальных транзакций в Spring. Ниже приведен исходный код конфигурационного класса `XAJpaConfig`, в котором объявляются компоненты Spring Beans для доступа к двум базам данных.

```
package com.apress.prospring5.ch9.config;

import com.atomikos.jdbc.AtomikosDataSourceBean;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.data.jpa.repository.config
          .EnableJpaRepositories;
import org.springframework.jdbc.datasource
          .SimpleDriverDataSource;
import org.springframework.orm.jpa.JpaVendorAdapter;

import org.springframework.orm.jpa
          .LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor
          .HibernateJpaVendorAdapter;
import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;
import java.sql.Driver;
import java.util.Properties;

@Configuration
@EnableJpaRepositories
public class XAJpaConfig {

  private static Logger logger =
          LoggerFactory.getLogger(XAJpaConfig.class);

  @SuppressWarnings("unchecked")
  @Bean(initMethod = "init", destroyMethod = "close")
  public DataSource dataSourceA() {
    try {
      AtomikosDataSourceBean dataSource =
              new AtomikosDataSourceBean();
      dataSource.setUniqueResourceName("XADBMSA");
      dataSource.setXaDataSourceClassName(
                "com.mysql.cj.jdbc.MysqlXADataSource");
      dataSource.setXaProperties(xaAProperties());
      dataSource.setPoolSize(1);
      return dataSource;
    } catch (Exception e) {
      logger.error("Populator DataSource bean cannot "
                + "be created!", e);
      return null;
    }
  }
```

```
@Bean
public Properties xaAProperties() {
  Properties xaProp = new Properties();
  xaProp.put("databaseName", "musicdb_a");
  xaProp.put("user", "prospring5_a");
  xaProp.put("password", "prospring5_a");
  return xaProp;
}

@SuppressWarnings("unchecked")
@Bean(initMethod = "init", destroyMethod = "close")
public DataSource dataSourceB() {
  try {
    AtomikosDataSourceBean dataSource =
            new AtomikosDataSourceBean();
    dataSource.setUniqueResourceName("XADBMSB");
    dataSource.setXaDataSourceClassName(
        "com.mysql.cj.jdbc.MysqlXADataSource");
    dataSource.setXaProperties(xaBProperties());
    dataSource.setPoolSize(1);
    return dataSource;
  } catch (Exception e) {
    logger.error("Populator DataSource bean cannot "
              + "be created!", e);
    return null;
  }
}

@Bean
public Properties xaBProperties() {
  Properties xaProp = new Properties();
  xaProp.put("databaseName", "musicdb_b");
  xaProp.put("user", "prospring5_b");
  xaProp.put("password", "prospring5_b");
  return xaProp;
}

@Bean
public Properties hibernateProperties() {
  Properties hibernateProp = new Properties();
  hibernateProp.put(
      "hibernate.transaction.factory_class",
      "org.hibernate.transaction.JTATransactionFactory");
  hibernateProp.put("hibernate.transaction.jta.platform",
                    "com.atomikos.icatch.jta.hibernate4"
                  + ".AtomikosPlatform");
  // требуется в версии Hibernate 5
  hibernateProp.put(
       "hibernate.transaction.coordinator_class", "jta");
```

```
    hibernateProp.put("hibernate.dialect",
                  "org.hibernate.dialect.MySQL5Dialect");
    // будет действовать только в том случае, если создать
    // сначала каталог users/schemas, а затем
    // использовать сценарий из файла ddl.sql
    hibernateProp.put("hibernate.hbm2ddl.auto",
                      "create-drop");
    hibernateProp.put("hibernate.show_sql", true);
    hibernateProp.put("hibernate.max_fetch_depth", 3);
    hibernateProp.put("hibernate.jdbc.batch_size", 10);
    hibernateProp.put("hibernate.jdbc.fetch_size", 50);
    return hibernateProp;
  }

  @Bean
  public EntityManagerFactory emfA() {
    LocalContainerEntityManagerFactoryBean factoryBean =
         new LocalContainerEntityManagerFactoryBean();
    factoryBean.setPackagesToScan(
                "com.apress.prospring5.ch9.entities");
    factoryBean.setDataSource(dataSourceA());
    factoryBean.setJpaVendorAdapter(
                new HibernateJpaVendorAdapter());
    factoryBean.setJpaProperties(hibernateProperties());
    factoryBean.setPersistenceUnitName("emfA");
    factoryBean.afterPropertiesSet();
    return factoryBean.getObject();
  }

  @Bean
  public EntityManagerFactory emfB() {
    LocalContainerEntityManagerFactoryBean factoryBean =
         new LocalContainerEntityManagerFactoryBean();
    factoryBean.setPackagesToScan(
                "com.apress.prospring5.ch9.entities");
    factoryBean.setDataSource(dataSourceB());
    factoryBean.setJpaVendorAdapter(
                new HibernateJpaVendorAdapter());
    factoryBean.setJpaProperties(hibernateProperties());
    factoryBean.setPersistenceUnitName("emfB");
    factoryBean.afterPropertiesSet();
    return factoryBean.getObject();
  }
}
```

Приведенная выше конфигурация длинная, хотя и не особенно сложная. Прежде всего в ней определяются два компонента Spring Beans для обозначения источников данных, представляющих два разных ресурса баз данных. Эти компоненты называются `dataSourceA` и `dataSourceB` и подключаются к схемам `musicdb_a` и

musicdb_b. В обоих компонентах, реализующих источники данных, применяется класс com.atomikos.jdbc.AtomikosDataSourceBean, в котором поддерживается совместимый с протоколом XA источник данных, а в определениях этих компонентов Spring Beans указывается класс com.mysql.jdbc.jdbc2.optional.MysqlXADataSource, реализующий интерфейс DataSource по протоколу XA в базе данных MySQL. Этот класс служит диспетчером ресурсов для MySQL. Затем в данной конфигурации предоставляются сведения о подключении к базе данных. Обратите внимание на то, что в атрибуте poolSize задается количество соединений, доступных в пуле соединений, который должен поддерживаться в Atomikos, хотя это и не обязательно. Но если атрибут poolSize не задан, то в Atomikos будет использоваться стандартное значение 1.

Далее в рассматриваемой здесь конфигурации определяются компоненты Spring Beans типа EntityManagerFactory под названием emfA и emfB. Общие для JPA свойства размещаются вместе в компоненте hibernateProperties. Единственное отличие обоих компонентов Spring Beans заключается в том, что они внедрены с соответствующим источником данных (т.е. компонент emfA с источником данных dataSourceA и компоненте emfB с источником данных dataSourceB). Следовательно, компонент emfA будет соединен со схемой musicdb_a базы данных MySQL через источник данных dataSourceA, тогда как компонент emfB — со схемой musicdb_b через источник данных dataSourceB. Рассмотрим свойства hibernate.transaction.factory_class и hibernate.transaction.jta.platform из компонента emfB. Оба эти свойства очень важны, потому что они используются в библиотеке Hibernate для поиска базовых компонентов Spring Beans типа UserTransaction и TransactionManager и участия в контексте сохраняемости, управляющем глобальной транзакцией. Не менее важным является и свойство hibernate.transaction.coordinator_class, требующееся в классах Atomikos, совместимых с версией Hibernate 4, для взаимодействия с версией Hibernate 5[8].

Ниже приведен исходный код конфигурационного класса ServicesConfig, где объявляются компоненты Spring Beans, применяемые для реализации управления глобальными транзакциями.

```
package com.apress.prospring5.ch9.config;

import com.atomikos.icatch.config.UserTransactionService;
import com.atomikos.icatch.config
          .UserTransactionServiceImp;
import com.atomikos.icatch.jta.UserTransactionImp;
import com.atomikos.icatch.jta.UserTransactionManager;
```

[8] Данная конфигурация была составлена по описанию интеграции с Spring в официальной документации на Atomikos, доступной по адресу https://www.atomikos.com/Documentation/SpringIntegration, а также по материалам сайта Stack Overflow, доступным по адресу https://stackoverflow.com/questions/33127854/hibernate-5-with-spring-jta.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.context.annotation.DependsOn;
import org.springframework.transaction
          .PlatformTransactionManager;
import org.springframework.transaction.annotation
          .EnableTransactionManagement;
import org.springframework.transaction.jta
          .JtaTransactionManager;
import javax.transaction.SystemException;
import javax.transaction.UserTransaction;
import java.util.Properties;

@Configuration
@EnableTransactionManagement
@ComponentScan(basePackages =
               "com.apress.prospring5.ch9.services")
public class ServicesConfig {
  private Logger logger =
          LoggerFactory.getLogger(ServicesConfig.class);

  @Bean(initMethod = "init",
        destroyMethod = "shutdownForce")
  public UserTransactionService userTransactionService() {
    Properties atProps = new Properties();
    atProps.put("com.atomikos.icatch.service",
                "com.atomikos.icatch.standalone"
              + ".UserTransactionServiceFactory");
    return new UserTransactionServiceImp(atProps);
  }

  @Bean (initMethod = "init", destroyMethod = "close")
  @DependsOn("userTransactionService")
  public UserTransactionManager
         atomikosTransactionManager() {
    UserTransactionManager utm =
        new UserTransactionManager();
    utm.setStartupTransactionService(false);
    utm.setForceShutdown(true);
    return utm;
  }

  @Bean
  @DependsOn("userTransactionService")
```

```
  public UserTransaction userTransaction() {
    UserTransactionImp ut = new UserTransactionImp();
    try {
      ut.setTransactionTimeout(300);
    } catch (SystemException se) {
      logger.error("Configuration exception.", se);
      return null;
    }
    return ut;
  }

  @Bean
  public PlatformTransactionManager transactionManager() {
    JtaTransactionManager ptm =
       new JtaTransactionManager();
    ptm.setTransactionManager(atomikosTransactionManager());
    ptm.setUserTransaction(userTransaction());
    return ptm;
  }
}
```

Для части, относящейся к Atomikos, в приведенной выше конфигурации определяются два компонента Spring Beans: `atomikosTransactionManager` и `atomikosUserTransaction`. В классах, предоставляемых в Atomikos, реализуются стандартные для Spring интерфейсы `org.springframework.transaction.PlatformTransactionManager` и `org.springframework.transaction.PlatformTransactionManager` соответственно. Эти компоненты Spring Beans обеспечивают координацию транзакций и синхронизации услуг, требующихся в прикладном интерфейсе JTA, а также связываются с диспетчерами ресурсов по протоколу XA при поддержке механизма 2PC. Далее определяется компонент Spring Bean под названием `transactionManager` (с классом реализации `org.springframework.transaction.jta.JtaTransactionManager`), включая внедрение двух компонентов транзакций, предоставляемых в Atomikos. Тем самым каркасу Spring предписывается употребить прикладной интерфейс Atomikos JTA для управления транзакциями. Обратите также внимание на компонент Spring Bean типа `UserTransactionService`, применяемый для конфигурирования службы транзакций Atomikos, предоставляющей услуги администрирования незавершенных транзакций[9].

Ниже приведен исходный код класса `SingerServiceImpl` для применения прикладного интерфейса JTA. Однако ради простоты в этом классе полностью реализован только метод `save()`.

---

[9] Данная конфигурация адаптирована с помощью аннотаций конфигурацией в формате XML, приведенной в документации на Atomikos по адресу `https://www.atomikos.com/Documentation/SpringIntegration#The_Advanced_Case_40As_of_3.3_41`.

```
package com.apress.prospring5.ch9.services;

import com.apress.prospring5.ch9.entities.Singer;
import org.apache.commons.lang3.NotImplementedException;
import org.springframework.orm.jpa.JpaSystemException;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction
          .annotation.Transactional;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.PersistenceException;
import java.util.List;

@Service("singerService")
@Repository
@Transactional
public class SingerServiceImpl implements SingerService {

  @PersistenceContext(unitName = "emfA")
  private EntityManager emA;
  @PersistenceContext(unitName = "emfB")
  private EntityManager emB;

  @Override
  @Transactional(readOnly = true)
  public List<Singer> findAll() {
    throw new NotImplementedException("findAll");
  }

  @Override
  @Transactional(readOnly = true)
  public Singer findById(Long id) {
    throw new NotImplementedException("findById");
  }

  @Override
  public Singer save(Singer singer) {
    Singer singerB = new Singer();
    singerB.setFirstName(singer.getFirstName());
    singerB.setLastName(singer.getLastName());
    if (singer.getId() == null) {
      emA.persist(singer);
      emB.persist(singerB);
      // throw new JpaSystemException(
      //           new PersistenceException());
    } else {
      emA.merge(singer);
```

```
      emB.merge(singer);
    }
    return singer;
  }

  @Override
  public long countAll() {
    return 0;
  }
}
```

В приведенном выше коде определены два диспетчера сущностей, внедренные в классе `SingerServiceImpl`. В методе `save()` объект, представляющий певца, сохраняется в двух схемах. Не обращайте пока что внимания на оператор `throw` для генерирования исключения; мы воспользуемся им в дальнейшем, чтобы проверить, произошел ли откат транзакции при неудачном завершении операции сохранения в схеме `musicdb_b`. Ниже приведен исходный код соответствующей тестовой программы.

```
package com.apress.prospring5.ch9;

import com.apress.prospring5.ch9.config.ServicesConfig;
import com.apress.prospring5.ch9.config.XAJpaConfig;
import com.apress.prospring5.ch9.entities.Singer;
import com.apress.prospring5.ch9.services.SingerService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;

import java.util.Date;
import java.util.GregorianCalendar;

public class TxJtaDemo {
  private static Logger logger =
          LoggerFactory.getLogger(TxJtaDemo.class);

  public static void main(String... args) {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               ServicesConfig.class, XAJpaConfig.class);
    SingerService singerService =
          ctx.getBean(SingerService.class);
    Singer singer = new Singer();
    singer.setFirstName("John");
    singer.setLastName("Mayer");
    singer.setBirthDate(new Date(
```

```
            (new GregorianCalendar(1977, 9, 16))
            .getTime().getTime()));
    singerService.save(singer);
    if (singer.getId() != null) {
      logger.info("--> Singer saved successfully");
    } else {
      logger.info("--> Singer was not saved, check "
                + "the configuration!!");
    }
    ctx.close();
  }
}
```

В данной тестовой программе создается новый объект, представляющий певца, а также вызывается метод `SingerService.save()`. В проверяемой реализации этого метода предпринимается попытка сохранить один и тот же объект в двух базах данных. Если не произойдет ничего непредвиденного, то выполнение данной программы приведет к выводу на консоль следующего результата (здесь он показан не полностью):

```
--> Singer saved successfully[10]
```

В диспетчере транзакций Atomikos создается составная транзакция, устанавливается связь с источником данных по протоколу XA (в данном случае — с базой данных MySQL), выполняется синхронизация, фиксируется транзакция и т.д. Новая запись о певце сохраняется в обеих схемах базы данных. Но если требуется проверить результат сохранения непосредственно в коде, то для этой цели можно предоставить реализацию метода `findAll()`, где это будет сделано автоматически:

```
package com.apress.prospring5.ch9.services;
...
@Service("singerService")
@Repository
@Transactional
public class SingerServiceImpl implements SingerService {

  private static final String FIND_ALL=
          "select s from Singer s";

  @PersistenceContext(unitName = "emfA")
  private EntityManager emA;
  @PersistenceContext(unitName = "emfB")
  private EntityManager emB;

  @Override
  @Transactional(readOnly = true)
  public List<Singer> findAll()
```

[10] Певец успешно сохранен

```
  {
    List<Singer> singersFromA = findAllInA();
    List<Singer> singersFromB = findAllInB();
    if (singersFromA.size()!= singersFromB.size()) {
      throw new AsyncXAResourcesException("XA resources "
              + "do not contain the same expected data.");
    }
    Singer sA = singersFromA.get(0);
    Singer sB = singersFromB.get(0);
    if (!sA.getFirstName().equals(sB.getFirstName())) {
      throw new AsyncXAResourcesException("XA resources "
             + "do not contain the same expected data.");
    }

    List<Singer> singersFromBoth = new ArrayList<>();
    singersFromBoth.add(sA);
    singersFromBoth.add(sB);
    return singersFromBoth;
  }

  private List<Singer> findAllInA() {
    return emA.createQuery(FIND_ALL).getResultList();
  }

  private List<Singer> findAllInB() {
    return emB.createQuery(FIND_ALL).getResultList();
  }
  ...
}
```

Таким образом, код проверки сохраняемого объекта, представляющего певца, в обеих базах данных можно изменить следующим образом:

```
package com.apress.prospring5.ch9;
...
public class TxJtaDemo {
  private static Logger logger =
          LoggerFactory.getLogger(TxJtaDemo.class);

  public static void main(String... args) {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               ServicesConfig.class, XAJpaConfig.class);
    SingerService singerService =
          ctx.getBean(SingerService.class);
    Singer singer = new Singer();
    singer.setFirstName("John");
    singer.setLastName("Mayer");
    singer.setBirthDate(new Date(
```

```
            (new GregorianCalendar(1977, 9, 16))
            .getTime().getTime()));
    singerService.save(singer);
    if (singer.getId() != null) {
      logger.info("--> Singer saved successfully");
    } else {
      logger.error("--> Singer was not saved, check "
                  + "the configuration!!");
    }
    // проверить результат сохранения в обеих базах данных
    List<Singer> singers = singerService.findAll();
    if (singers.size()!= 2) {
      logger.error("--> Something went wrong.");
    } else {
      logger.info("--> Singers form both DBs: " + singers);
    }

    ctx.close();
  }
}
```

А теперь выясним, как действует откат транзакции. Как показано ниже, вместо вызова `emB.persist()` можно сгенерировать исключение, чтобы сымитировать неудачный исход операции сохранения информации во второй базе данных.

```
package com.apress.prospring5.ch9.services;
...
@Service("singerService")
@Repository
@Transactional
public class SingerServiceImpl implements SingerService {

  private static final String FIND_ALL=
          "select s from Singer s";

  @PersistenceContext(unitName = "emfA")
  private EntityManager emA;
  @PersistenceContext(unitName = "emfB")
  private EntityManager emB;
  ...
  @Override
  public Singer save(Singer singer) {
    Singer singerB = new Singer();
    singerB.setFirstName(singer.getFirstName());
    singerB.setLastName(singer.getLastName());
    if (singer.getId() == null) {
      emA.persist(singer);
      if(true) {
        throw new JpaSystemException(
```

```
                new PersistenceException(
                    "Simulation of something going wrong."));
            }
            emB.persist(singerB);
        } else {
            emA.merge(singer);
            emB.merge(singer);
        }
        return singer;
    }

    @Override
    public long countAll() {
        return 0;
    }
}
```

Если снова выполнить программу из рассматриваемого здесь примера, то на консоль будет выведен следующий результат:

```
...
INFO o.h.h.i.QueryTranslatorFactoryInitiator - HHH000397:
   Using ASTQueryTranslatorFactory
INFO o.s.o.j.LocalContainerEntityManagerFactoryBean
     - Initialized JPA
   EntityManagerFactory for persistence unit 'emfA'
INFO o.s.o.j.LocalContainerEntityManagerFactoryBean
     - Initialized JPA
   EntityManagerFactory for persistence unit 'emfB'
INFO o.s.t.j.JtaTransactionManager
     - Using JTA UserTransaction:
   com.atomikos.icatch.jta.UserTransactionImp@6da9dc6
INFO o.s.t.j.JtaTransactionManager
     - Using JTA TransactionManager:
   com.atomikos.icatch.jta.UserTransactionManager@2216effc
DEBUG o.s.t.j.JtaTransactionManager
     - Creating new transaction with name
[com.apress.prospring5.ch9.services.SingerServiceImpl.save]:
   PROPAGATION_REQUIRED,ISOLATION_DEFAULT; ''
DEBUG o.s.o.j.EntityManagerFactoryUtils
     - Opening JPA EntityManager
DEBUG o.s.o.j.EntityManagerFactoryUtils
     - Registering transaction synchronization
   for JPA EntityManager
Hibernate: insert into singer (BIRTH_DATE, FIRST_NAME,
     LAST_NAME, VERSION)
values (?, ?, ?, ?)
DEBUG o.s.o.j.EntityManagerFactoryUtils
     - Closing JPA EntityManager
```

```
DEBUG o.s.t.j.JtaTransactionManager
     - Initiating transaction rollback
WARN c.a.j.AbstractConnectionProxy
     - Forcing close of pending statement:
   com.mysql.cj.jdbc.PreparedStatementWrapper@3f685162
Exception in thread "main"
     org.springframework.orm.jpa.JpaSystemException:
   Simulation of something going wrong.;
   ...
Caused by: javax.persistence.PersistenceException:
   Simulation of something going wrong.
```

Как следует из приведенного выше результата, сведения о первом певце были сохранены (обратите внимание на оператор `insert`). Но при сохранении во втором источнике данных генерируется исключение, из-за чего Atomikos производит откат всей транзакции. Можете просмотреть схему `musicdb_a` и удостовериться, что запись о новом певце не сохранена.

Стартовая библиотека Spring Boot для JTA

В модуле Spring Boot имеется готовая стартовая библиотека с рядом предварительно сконфигурированных компонентов Spring Beans для прикладного интерфейса JTA, помогающих разработчикам сосредоточить основное внимание на функциональных средствах предметной области, а не на установке рабочей среды. И поскольку это основное назначение всех стартовых библиотек из модуля Spring Boot независимо от конкретного компонента, то предыдущее пояснение может показаться несколько излишним. В состав стартовой библиотеки Spring Boot для JTA входит отдельный компонент, предназначенный для применения Atomikos. Этот компонент извлекает соответствующие библиотеки Atomikos и автоматически конфигурирует их.

Чтобы приспособить предыдущий пример к модулю Spring Boot, достаточно импортировать конфигурации источника данных и диспетчера транзакций в приложение Spring Boot. Но в этом разделе потребуется другой пример, поскольку в нем преследуется цель показать, каким образом предварительно сконфигурированные компоненты из модуля Spring Boot могут оказать помощь в ускорении процесса разработки приложений, в которых предусмотрено управление глобальными транзакциями. Допустим, что в очередь обмена сообщениями требуется передать сообщение, чтобы известить о получении нового экземпляра типа `Singer`. Очевидно, что если операция сохранения записи о певце в базе данных завершится неудачно, то придется совершить откат транзакции и запретить отправку подобного сообщения. Таким образом, в данном примере необходимо сделать следующее.

- Сконфигурировать в Gradle проект Spring Boot для JTA, чтобы применить службу JMS, как показано ниже.

```
// Файл конфигурации build.gradle
ext {
    ...
    bootVersion = '2.0.0.M1'
    atomikosVersion = '4.0.4'

    boot = [
            ...
      starterJpa : "org.springframework.boot:
            spring-boot-starter-data-jpa:$bootVersion",
      starterJta : "org.springframework.boot:
            spring-boot-starter-jta-atomikos:$bootVersion",
      starterJms : "org.springframework.boot:
            spring-boot-starter-artemis:$bootVersion"
    ]

    misc = [
          ...
      artemis : "org.apache.activemq:
                artemis-jms-server:2.1.0"
    ]

    db = [
          ...
      h2 : "com.h2database:h2:$h2Version"
    ]
}

// Файл конфигурации chapter09/boot-jta/build.gradle
buildscript {
    repositories {
      ...
    }

    dependencies {
      classpath boot.springBootPlugin
    }
}

apply plugin: 'org.springframework.boot'

dependencies {
    compile boot.starterJpa, boot.starterJta,
            boot.starterJms, db.h2
    compilemisc.artemis {
      exclude group: 'org.apache.geronimo.specs',
             module: 'geronimo-jms_2.0_spec'
    }
}
```

- На рис. 9.3 показаны стартовые библиотеки из модуля Spring Boot, объявленные ранее как зависимости для рассматриваемого здесь проекта, а также те зависимости, которые они вносят в данный проект.

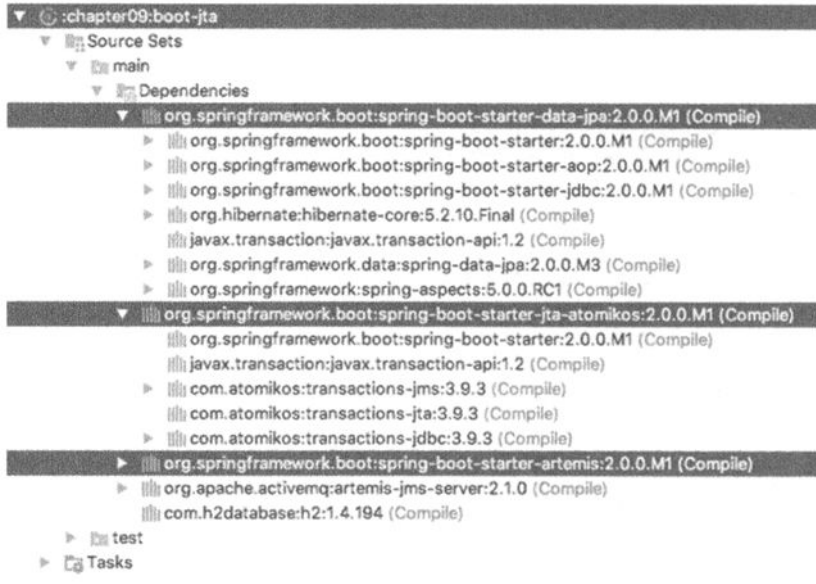


Рис. 9.3. Стартовые библиотеки из модуля Spring Boot и их зависимости

- Определить класс сущности `Singer` и сопутствующее ему информационное хранилище. Структура этого класса должна быть такой же, как и прежде, но без всяких связанных с ним сущностей. А интерфейс `SingerRepository` останется пустым, поскольку в данном примере требуются лишь методы `save()` и `count()`, предоставляемые из интерфейса `CrudRepository`.
- Определить служебный класс, предназначенный для сохранения записи о певце и отправки подтверждающего сообщения, как показано ниже.

```
package com.apress.prospring5.ch9.services;

import com.apress.prospring5.ch9.entities.Singer;
import com.apress.prospring5.ch9.ex
          .AsyncXAResourcesException;
import com.apress.prospring5.ch9.repos.SingerRepository;
import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction
          .annotation.Transactional;
```

```
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Service("singerService")
@Transactional
public class SingerServiceImpl implements SingerService {

  private SingerRepository singerRepository;
  private JmsTemplate jmsTemplate;

  public SingerServiceImpl(
         SingerRepository singerRepository,
         JmsTemplate jmsTemplate) {
    this.singerRepository = singerRepository;
    this.jmsTemplate = jmsTemplate;
  }

  @Override
  public Singer save(Singer singer) {
    jmsTemplate.convertAndSend("singers",
                               "Just saved:" + singer);
    if(singer == null) {
      throw new AsyncXAResourcesException(
            "Simulation of something going wrong.");
    }
    singerRepository.save(singer);
    return singer;
  }

  @Override public long count() {
    return singerRepository.count();
  }
}
```

- Для внедрения компонента Spring Bean информационного хранилища не требуется ни аннотация `@Autowired`, ни класс `JmsTemplate`, поскольку модуль Spring Boot автоматически внедряет требующиеся компоненты Spring Beans, если они объявлены.
- Определить компонент Spring Bean для приема сообщений, отправляемых в очередь обмена сообщениями, и последующего их вывода на консоль, как показано ниже.

```
package com.apress.prospring5.ch9;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

@Component
```

```
public class Messages {
  private static Logger logger =
          LoggerFactory.getLogger(Messages.class);

  @JmsListener(destination="singers")
  public void onMessage(String content) {
     logger.info("--> Received content: " + content);
  }
}
```

- Сконфигурировать сервер Artemis JMS, чтобы создать встроенную очередь `singers`. Для этого в свойстве `spring.artemis.embedded.queues` устанавливается значение `singers` непосредственно в файле свойств `application.properties`, который может быть использован для конфигурирования конкретного приложения Spring Boot.

```
spring.artemis.embedded.queues=singers
spring.jta.log-dir=out
```

- В приведенном выше фрагменте кода конфигурации представлено содержимое файла свойств `application.properties`. Помимо свойства `spring.artemis.embedded.queues`, в этом файле устанавливается свойство `spring.jta.log-dir`, определяющее место для вывода журнала регистрации JTA из Atomikos (в данном случае — в каталог `out`).
- Определить класс приложения `Application`, чтобы заключить в нем все остальное и проверить его, как показано ниже.

```
package com.apress.prospring5.ch9;

import com.apress.prospring5.ch9.entities.Singer;
import com.apress.prospring5.ch9.services.SingerService;
import com.atomikos.jdbc.AtomikosDataSourceBean;
import org.h2.jdbcx.JdbcDataSource;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure
          .SpringBootApplication;
import org.springframework.context
          .ConfigurableApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.orm.jpa
          .LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor
          .HibernateJpaVendorAdapter;
```

```
import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.Properties;
import static org.hibernate.cfg.AvailableSettings.*;
import static org.hibernate.cfg.AvailableSettings
                .STATEMENT_FETCH_SIZE;

@SpringBootApplication(scanBasePackages =
          "com.apress.prospring5.ch9.services")
public class Application implements CommandLineRunner {

  private static Logger logger =
          LoggerFactory.getLogger(Application.class);

  public static void main(String... args)
         throws Exception {
    ConfigurableApplicationContext ctx =
           SpringApplication.run(Application.class, args);

    System.in.read();
    ctx.close();
  }

  @Autowired SingerService singerService;
  @Override public void run(String... args)
           throws Exception {
    Singer singer = new Singer();
    singer.setFirstName("John");
    singer.setLastName("Mayer");
    singer.setBirthDate(new Date(
          (new GregorianCalendar(1977, 9, 16))
          .getTime().getTime()));
    singerService.save(singer);

    long count = singerService.count();
    if (count == 1) {
      logger.info("--> Singer saved successfully");
    } else {
      logger.error("--> Singer was not saved, "
                + "check the configuration!!");
    }

    try {
      singerService.save(null);
    } catch (Exception ex) {
      logger.error(ex.getMessage() + "Final count:"
                + singerService.count());
```

```
        }
      }
    }
```

Если выполнить исходный код класса `Application`, то на консоль будет выведен результат, аналогичный приведенному ниже.

```
...
INFO c.a.j.AtomikosConnectionFactoryBean
   - AtomikosConnectionFactoryBean
     'jmsConnectionFactory': init...
INFO o.s.t.j.JtaTransactionManager
   - Using JTA UserTransaction:
    com.atomikos.icatch.jta.UserTransactionManager@408a247c
INFO c.a.j.AtomikosJmsXaSessionProxy
   - atomikos xa session proxy for resource
     jmsConnectionFactory:
        calling createQueue on JMS driver session...
INFO c.a.j.AtomikosJmsXaSessionProxy
   - atomikos xa session proxy for resource
     jmsConnectionFactory:
        calling getTransacted on JMS driver session...
DEBUG o.s.t.j.JtaTransactionManager
   - Participating in existing transaction
DEBUG o.s.t.j.JtaTransactionManager
    - Initiating transaction commit
INFO c.a.d.x.XAResourceTransaction - XAResource.start ...
INFO c.a.d.x.XAResourceTransaction - XAResource.end ...
DEBUG o.s.t.j.JtaTransactionManager
   - Initiating transaction commit
DEBUG o.s.t.j.JtaTransactionManager
   - Creating new transaction with name
     [com.apress.prospring5.ch9.services
         .SingerServiceImpl.save]:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT; ''
INFO c.a.i.i.BaseTransactionManager
   - createCompositeTransaction ( 10000 ):
     created new ROOT transaction with
     id 127.0.0.1.tm0000200001
DEBUG o.s.t.j.JtaTransactionManager
   - Participating in existing transaction
INFO c.a.p.c.Application - --> Singer saved successfully
...// и т.д.
```

Из приведенного выше результата ясно видно, что для каждой операции создается и повторно используется глобальная транзакция. Чтобы выйти из данного приложения, как обычно, нажав сначала любую клавишу, а затем клавишу <Enter>, следует проявить терпение, поскольку для корректного завершения данного приложения потребуется некоторое время.

Разрабатывая приложения JTA с помощью модуля Spring Boot, можно прийти к некоторым выводам. Несмотря на кажущуюся простоту процесса разработки, он все равно не избавляет от необходимости работать с несколькими источниками данных и конфигурировать рабочую среду. Дело еще больше усложняется, если поставщик услуг JTA предоставляется сервером приложений JEE, хотя для примеров приложений, создаваемых в учебных целях и для тестирования такой подход оказывается вполне практичным.

Соображения по поводу применения JTA

Среди разработчиков ведется горячая полемика о том, следует ли пользоваться прикладным интерфейсом JTA для управления глобальными транзакциями. Например, команда разработчиков Spring не рекомендует пользоваться JTA для глобальных транзакций. Общий принцип состоит в том, что когда приложение разворачивается на полномасштабном сервере приложений JEE, то нет никаких причин не применять прикладной интерфейс JTA, поскольку все производители распространенных серверов приложений JEE оптимизируют собственные реализации JTA для своих платформ. И это одно из главных функциональных средств, применение которого не дается даром.

Если приложение развертывается автономно или в веб-контейнере, то принимать решение следует, руководствуясь требованиями к приложению. Нагрузочное тестирование необходимо проводить как можно раньше, чтобы удостовериться, что применение прикладного интерфейса JTA не приводит к снижению производительности.

Примечательно, что каркас Spring бесперебойно работает с локальными и глобальными транзакциями в большинстве основных веб- и JEE-контейнеров, поэтому при переходе от одной стратегии управления транзакциями к другой обычно модификация кода не потребуется. Если вы решите употребить прикладной интерфейс JTA в своем приложении, воспользуйтесь для этой цели классом `JtaTransactionManager` из Spring.

Резюме

Управление транзакциями является главной составляющей обеспечения целостности данных в приложении практически любого типа. В этой главе пояснялось, как пользоваться каркасом Spring для управления транзакциями, не оказывая почти никакого влияния на исходный код приложений. В ней было также показано, как обращаться с локальными и глобальными транзакциями. На целом ряде примеров было наглядно показано, каким образом реализуются транзакции как декларативно с помощью конфигурации в формате XML и аннотаций, так и программно.

Локальные транзакции поддерживаются как на самом сервере приложений JEE, так и за его пределами, а для активизации поддержки локальных транзакций в Spring требуется лишь простое конфигурирование. Тем не менее настройка рабочей среды на глобальные транзакции предусматривает дополнительные затраты труда и сильно зависит от того, с каким поставщиком услуг JTA и соответствующими ресурсами сервера базы данных должно взаимодействовать приложение.

ГЛАВА 10

Проверка достоверности с преобразованием типов и форматированием данных

Проверка достоверности данных имеет критическое значение в корпоративных приложениях. Назначение проверки достоверности данных — убедиться, что обрабатываемые данные удовлетворяют всем заранее определенным требованиям из предметной области, а также обеспечить целостность и пригодность данных на других уровнях приложения.

При разработке приложений проверка достоверности данных всегда упоминается вместе с их преобразованием и форматированием. Дело в том, что формат источника данных, скорее всего, отличается от формата, употребляемого в приложении. Например, в веб-приложении пользователь вводит данные во внешнем интерфейсе веб-браузера. Когда пользователь сохраняет эти данные, они отправляются серверу (по завершении локальной проверки достоверности данных). На стороне сервера выполняется процесс привязки данных, в ходе которого данные извлекаются из HTTP-запроса, преобразуются и привязываются к соответствующим объектам предметной области (к примеру, пользователь вводит сведения о певце в HTML-форме, которая затем привязывается к объекту типа `Singer` на сервере) по правилам форматирования, определенным для каждого атрибута (например, по шаблону формата даты `гггг-мм-дд`). По завершении привязки данных к объекту предметной области применяются правила проверки их достоверности для поиска любого нарушения накладываемых ограничений. Если все прошло нормально, данные сохраняются, а пользователю выводится сообщение об удачном исходе данной операции. В противном случае пользователю выводится сообщения об ошибках проверки достоверности данных.

В этой главе поясняется, каким образом в Spring предоставляется развитая поддержка преобразования типов, форматирования полей и проверки достоверности данных. В частности, здесь будут рассмотрены следующие вопросы.

- **Система преобразования типов данных в Spring и интерфейс поставщика услуг форматирования.** В этой части описана обобщенная система преобразования типов и интерфейс поставщика услуг форматирования данных (Formatter SPI). Здесь будут представлены новые службы, которыми можно пользоваться вместо прежней поддержки редакторов свойств, а также показано, как они выполняют взаимное преобразование любых типов данных Java.
- **Проверка достоверности данных в Spring.** В этой части пояснено, каким образом в Spring поддерживается проверка достоверности объектов предметной области. Сначала будет сделано краткое введение в интерфейс `Validator` из Spring, а затем основное внимание будет уделено поддержке спецификации JSR-349 (Bean Validation — проверка достоверности компонентов Spring Beans).

Зависимости

Как и в предыдущих главах, для примеров кода, представленных в этой главе, требуется ряд приведенных ниже зависимостей. Обратите особое внимание на зависимость `joda-time`. Если вы пользуетесь версией Java 8, то имейте в виду, что и в версии Spring 5 поддерживается также спецификация JSR-310, в которой описывается прикладной интерфейс API под названием `javax.time`.

```
// Файл конфигурации pro-spring-15/build.gradle
ext {
    // Библиотеки Spring
    springVersion = '5.0.0.RC1'

    jodaVersion = '2.9.9'
    javaxValidationVersion = '2.0.0.Beta2' // 1.1.0.Final
    javaElVersion = '3.0.1-b04' // 3.0.0
    glasshfishELVersion = '2.2.1-b05' // 2.2
    hibernateValidatorVersion = '6.0.0.Beta2' //5.4.1.Final
    spring = [...]

    hibernate = [
      validator : "org.hibernate:hibernate-validator:
                  $hibernateValidatorVersion",
      ...
    ]

     misc = [
      validation : "javax.validation:validation-api:
                   $javaxValidationVersion",
      joda : "joda-time:joda-time:$jodaVersion",
      ...
    ]
    ...
```

```
}

// Файл конфигурации chapter10/build.gradle
dependencies {
    compile spring.contextSupport, misc.slf4jJcl,
     misc.logback, db.h2, misc.lang3, hibernate.em,
    hibernate.validator, misc.joda, misc.validation

    testCompile testing.junit
}
```

Система преобразования типов данных в Spring

В версии Spring 3 появилась новая система преобразования типов данных, предоставляющая эффективный способ для выполнения взаимных преобразований любых типов данных Java в приложениях Spring. В этом разделе поясняется, каким образом в этой системе реализуются такие же функциональные возможности, как и у ранее применявшихся редакторов свойств, а также поддержка взаимных преобразований любых типов данных Java. Кроме того, будет продемонстрирована реализация специального преобразователя типов с помощью интерфейса Converter SPI.

Преобразование строковых данных с помощью редакторов свойств

Как пояснялось в главе 4, строковые данные типа `String` из файлов свойств преобразуются в свойства объектов POJO благодаря поддержке в Spring редакторов свойств, реализующих интерфейс `PropertyEditor`. Сделаем сначала краткий обзор, а затем рассмотрим интерфейс Converter SPI, доступный с версии Spring 3.0, в качестве более эффективной альтернативы редакторам свойств.

Итак, рассмотрим в качестве примера следующий упрощенный вариант класса `Singer`:

```
package com.apress.prospring5.ch10;

import java.net.URL;
import java.text.SimpleDateFormat;
import org.joda.time.DateTime;

public class Singer {
  private String firstName;
  private String lastName;
  private DateTime birthDate;
  private URL personalSite;

  // Методы получения и установки
  ...
```

```
  public String toString() {
    SimpleDateFormat sdf =
          new SimpleDateFormat("yyyy-MM-dd");
    return String.format("{First name: %s, Last name: %s,
                          Birthday: %s, Site: %s}",
    firstName, lastName, sdf.format(birthDate.toDate()),
                                    personalSite);
  }
}
```

Для установки атрибута `birthDate`, задающего дату дня рождения, в приведенном выше коде используется класс `DateTime` из библиотеки Joda-Time. Имеется также поле типа `URL`, где указывается персональный веб-сайт певца, если это уместно. А теперь допустим, что требуется построить объекты типа `Singer` в контексте типа `ApplicationContext` с помощью значений, хранящихся в файле конфигурации Spring или в файле свойств. В качестве примера ниже приведено содержимое XML-файла `prop-editor-app-context.xml` конфигурации Spring.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context=
            "http://www.springframework.org/schema/context"
       xmlns:util=
            "http://www.springframework.org/schema/util"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation=
          "http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans
                /spring-beans.xsd
          http://www.springframework.org/schema/context
          http://www.springframework.org/schema/context
                /spring-context.xsd
          http://www.springframework.org/schema/util
          http://www.springframework.org/schema/util
                /spring-util.xsd">

    <context:annotation-config/>

    <context:property-placeholder
            location="classpath:application.properties"/>

    <bean id="customEditorConfigurer"
       class="org.springframework.beans.factory.config
                .CustomEditorConfigurer"
       p:propertyEditorRegistrars-ref=
                "propertyEditorRegistrarsList"/>

    <util:list id="propertyEditorRegistrarsList">
```

```
        <bean class="com.apress.prospring5.ch10
                        .DateTimeEditorRegistrar">
            <constructor-arg value="${date.format.pattern}"/>
        </bean>
    </util:list>

    <bean id="eric"
        class="com.apress.prospring5.ch10.Singer"
        p:firstName="Eric"
        p:lastName="Clapton"
        p:birthDate="1945-03-30"
        p:personalSite="http://www.ericclapton.com"/>

    <bean id="countrySinger"
        class="com.apress.prospring5.ch10.Singer"
        p:firstName="${countrySinger.firstName}"
        p:lastName="${countrySinger.lastName}"
        p:birthDate="${countrySinger.birthDate}"
        p:personalSite="${countrySinger.personalSite}"/>
</beans>
```

В приведенной выше конфигурации строятся два разных компонента Spring Beans типа `Singer`. В частности, компонент `chris` создан со значениями, предоставляемыми в файле конфигурации, тогда как атрибуты для компонента `countrysinger` вынесены во внешний файл свойств. Кроме того, в этой конфигурации определяется специальный редактор свойств, предназначенный для преобразования строковых данных типа `String` в тип `DateTime` из библиотеки Joda-Time, а шаблон формата даты и времени также вынесен во внешний файл свойств. Ниже приведено содержимое файла свойств `application.properties`.

```
date.format.pattern=yyyy-MM-dd

countrySinger.firstName=John
countrySinger.lastName=Mayer
countrySinger.birthDate=1977-10-16
countrySinger.personalSite=http://johnmayer.com/
```

Ниже представлен специальный редактор свойств для преобразования строковых данных типа `String` в тип `DateTime` из библиотеки Joda-Time.

```
package com.apress.prospring5.ch10;

import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;
import org.springframework.beans.PropertyEditorRegistrar;
import org.springframework.beans.PropertyEditorRegistry;
import java.beans.PropertyEditorSupport;
```

```
public class DateTimeEditorRegistrar
       implements PropertyEditorRegistrar {
  private DateTimeFormatter dateTimeFormatter;

  public DateTimeEditorRegistrar(String dateFormatPattern) {
    dateTimeFormatter =
        DateTimeFormat.forPattern(dateFormatPattern);
  }

  @Override
  public void registerCustomEditors(
             PropertyEditorRegistry registry) {
    registry.registerCustomEditor(DateTime.class,
             new DateTimeEditor(dateTimeFormatter));
  }

  private static class DateTimeEditor
          extends PropertyEditorSupport {
    private DateTimeFormatter dateTimeFormatter;
    public DateTimeEditor(
           DateTimeFormatter dateTimeFormatter) {
      this.dateTimeFormatter = dateTimeFormatter;
    }

    @Override
    public void setAsText(String text)
           throws IllegalArgumentException {
      setValue(DateTime.parse(text, dateTimeFormatter));
    }
  }
}
```

Сначала в классе `DateTimeEditorRegistrar` реализуется интерфейс `PropertyEditorRegister` для регистрации специального класса `PropertyEditor`. Затем определяется внутренний класс `DateTimeEditor`, выполняющий преобразование строковых данных типа `String` в тип `DateTime`. В данном примере применяется внутренний класс, поскольку он доступен только в реализации интерфейса `PropertyEditorRegister`. А теперь проверим исходный код из данного примера. Для этой цели послужит приведенная ниже тестовая программа.

```
package com.apress.prospring5.ch10;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.support
          .GenericXmlApplicationContext;

public class PropEditorDemo {
```

```
  private static Logger logger =
          LoggerFactory.getLogger(PropEditorDemo.class);

  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();

    ctx.load(
        "classpath:spring/prop-editor-app-context.xml");
    ctx.refresh();

    Singer eric = ctx.getBean("eric", Singer.class);
    logger.info("Eric info: " + eric);
    Singer countrySinger = ctx.getBean("countrySinger",
                                       Singer.class);
    logger.info("John info: " + countrySinger);

    ctx.close();
  }
}
```

Как видите, из контекста типа `ApplicationContext` извлекаются и выводятся на консоль два компонента типа Spring Beans типа `Singer`. Выполнение этой тестовой программы дает следующий результат:

```
[main] INFO c.a.p.c.PropEditorDemo
   - Eric info: {First name: Eric,
     Last name: Clapton, Birthday: 1945-03-30,
          Site: http://www.ericclapton.com}
[main] INFO c.a.p.c.PropEditorDemo
   - John info: {First name: John,
     Last name: Mayer, Birthday: 1977-10-16,
          Site: http://johnmayer.com/}
```

Как следует из приведенного выше результата, свойства были преобразованы и применены к компонентам Spring Beans типа `Singer`. Конфигурация в формате XML используется в данном примере вместо конфигурационного класса потому, что внедряемые данные объявляются как текстовые строки, а их преобразование выполняется в Spring прозрачно в фоновом режиме.

Введение в систему преобразования типов данных в Spring

В версии Spring 3.0 стала доступной обобщенная система преобразования типов, которая находится в пакете `org.springframework.core.convert`. Эту систему преобразования типов данных можно рассматривать не только как альтернативу редакторам свойств, но и сконфигурировать для выполнения взаимных преобразований

типов данных Java и объектов POJO, тогда как редакторы свойств служат только для преобразования строковых данных из файла свойств в типы данных Java.

Реализация специального преобразователя

Чтобы продемонстрировать систему преобразования типов данных в действии, вернемся к предыдущему примеру и воспользуемся тем же самым классом `Singer`. На этот раз допустим, что систему преобразования типов данных требуется применять для преобразования даты из строкового формата `String` в свойство `birthDate` из класса `Singer`, которое относится к типу `DateTime` из библиотеки Joda-Time. Для поддержки такого преобразования вместо специального редактора свойства создадим в данном примере специальный преобразователь, реализовав интерфейс `org.springframework.core.convert.converter.Converter<S,T>`. Ниже приведен исходный код этого специального преобразователя.

```
package com.apress.prospring5.ch10;

import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;
import org.springframework.core.convert.converter.Converter;
import javax.annotation.PostConstruct;

public class StringToDateTimeConverter
       implements Converter<String, DateTime> {
  private static final String DEFAULT_DATE_PATTERN =
          "yyyy-MM-dd";
  private DateTimeFormatter dateFormat;
  private String datePattern = DEFAULT_DATE_PATTERN;

  public String getDatePattern() {
    return datePattern;
  }

  public void setDatePattern(String datePattern) {
    this.datePattern = datePattern;
  }

  @PostConstruct
  public void init() {
    dateFormat = DateTimeFormat.forPattern(datePattern);
  }

  @Override
  public DateTime convert(String dateString) {
    return dateFormat.parseDateTime(dateString);
  }
}
```

В исходном коде приведенного выше преобразователя реализуется интерфейс `Converter<String, DateTime>`, а это означает, что такой преобразователь отвечает за преобразование строковых данных типа `String` (исходного типа `S`) в тип `DateTime` (целевой тип `T`). Внедрять шаблон даты и времени совсем не обязательно, и чтобы сделать это, достаточно вызвать метод установки `setDatePattern()`. Если же такой шаблон не внедрен, то используется стандартный шаблон `гггг-мм-дд`. После этого в методе инициализации `init()`, снабженном аннотацией `@PostConstruct`, получается экземпляр класса `DateTimeFormat` из библиотеки Joda-Time, который выполнит преобразование по заданному шаблону. И, наконец, в методе `convert()` реализована вся логика подобного преобразования.

Конфигурирование интерфейса ConversionService

Чтобы применить службу преобразования вместо редактора свойств, придется сконфигурировать экземпляр интерфейса `org.springframework.core.convert.ConversionService` в контексте типа `ApplicationContext`. Ниже приведен исходный код предназначенного для этой цели конфигурационного класса Java.

```
package com.apress.prospring5.ch10.config;

import com.apress.prospring5.ch10.Singer;
import com.apress.prospring5.ch10
          .StringToDateTimeConverter;
import org.joda.time.DateTime;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context
          .annotation.ComponentScan;
import org.springframework.context
          .annotation.Configuration;
import org.springframework.context
          .annotation.PropertySource;
import org.springframework.context.support
          .ConversionServiceFactoryBean;
import org.springframework.context.support
          .PropertySourcesPlaceholderConfigurer;
import org.springframework.core.convert
          .converter.Converter;

import java.net.URL;
import java.util.HashSet;
import java.util.Set;

@PropertySource("classpath:application.properties")
@Configuration
@ComponentScan(basePackages = "com.apress.prospring5.ch10")
```

```
public class AppConfig {

  @Value("${date.format.pattern}")
  private String dateFormatPattern;

  @Bean
  public static PropertySourcesPlaceholderConfigurer
               propertySourcesPlaceholderConfigurer() {
    return new PropertySourcesPlaceholderConfigurer();
  }

  @Bean
  public Singer john(@Value("${countrySinger.firstName}")
       String firstName,
       @Value("${countrySinger.lastName}") String lastName,
       @Value("${countrySinger.personalSite}")
              URL personalSite,
       @Value("${countrySinger.birthDate}")
              DateTime birthDate)
       throws Exception {
    Singer singer = new Singer();
    singer.setFirstName(firstName);
    singer.setLastName(lastName);
    singer.setPersonalSite(personalSite);
    singer.setBirthDate(birthDate);
    return singer;
  }

  @Bean
  public ConversionServiceFactoryBean
         conversionService() {
    ConversionServiceFactoryBean
           conversionServiceFactoryBean =
              new ConversionServiceFactoryBean();
    Set<Converter> convs = new HashSet<>();
    convs.add(converter());
    conversionServiceFactoryBean.setConverters(convs);
    conversionServiceFactoryBean.afterPropertiesSet();
    return conversionServiceFactoryBean;
  }

  @Bean
  StringToDateTimeConverter converter() {
    StringToDateTimeConverter conv =
          new StringToDateTimeConverter();
    conv.setDatePattern(dateFormatPattern);
    conv.init();
    return conv;
  }
}
```

В приведенном выше коде значения читаются из указанного файла свойств с таким же содержимым, как и в примере из предыдущего раздела. Эти значения внедряются в создаваемый компонент Spring Bean с помощью аннотации `@Value`.

В данном случае каркасу Spring предписывается использовать систему преобразования типов данных, объявив компонент `conversionService` с помощью класса `ConversionServiceFactoryBean`. Если же ни один из компонентов службы преобразования не определен, каркас Spring воспользуется системой, основанной на редакторах свойств.

По умолчанию в службе преобразования типов поддерживается взаимное преобразование таких общих типов данных, как строки, числа, перечисления, коллекции, отображения и т.д. Поддерживается также преобразование строковых данных типа `String` в типы данных Java в системе, основанной на редакторах свойств. В компоненте `conversionService` сконфигурирован специальный преобразователь строковых данных типа `String` в тип `DateTime`. Ниже приведена тестовая программа для проверки такого преобразователя.

```
package com.apress.prospring5.ch10;

import com.apress.prospring5.ch10.config.AppConfig;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;

public class ConvServDemo {
  private static Logger logger =
          LoggerFactory.getLogger(ConvServDemo.class);

  public static void main(String... args) {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               AppConfig.class);

    Singer john = ctx.getBean("john", Singer.class);
    logger.info("Singer info: " + john);
    ctx.close();
  }
}
```

Выполнение этой тестовой программы дает приведенный ниже результат. Как видите, преобразование свойств компонента `john` приводит к такому же результату, как и в том случае, когда применяются редакторы свойств.

```
15:41:09.960 main INFO c.a.p.c.ConvServDemo
   - Singer info: {First name: John,
```

```
Last name: Mayer, Birthday: 1977-10-16,
   Site: http://johnmayer.com/}
```

Взаимное преобразование произвольных типов данных

Истинный потенциал системы преобразования типов заключается в ее способности выполнять взаимные преобразования произвольных типов данных. Чтобы продемонстрировать эту способность в действии, создадим еще один класс `AnotherSinger`, подобный классу `Singer`. Его исходный код приведен ниже.

```
package com.apress.prospring5.ch10;

import java.net.URL;
import java.text.SimpleDateFormat;
import org.joda.time.DateTime;

public class AnotherSinger {
  private String firstName;
  private String lastName;
  private DateTime birthDate;
  private URL personalSite;

  // Методы установки и получения
  ...

  public String toString() {
    SimpleDateFormat sdf =
          new SimpleDateFormat("yyyy-MM-dd");
    return String.format("{First name: %s, Last name: %s,
           Birthday: %s, Site: %s}", firstName, lastName,
           sdf.format(birthDate.toDate()), personalSite);
  }
}
```

В данном случае требуется преобразовать любой экземпляр класса `Singer` в экземпляр класса `AnotherSinger`. В результате преобразования атрибуты `firstName` и `lastName` из класса `Singer` станут соответственно атрибутами `lastName` и `firstName` из класса `AnotherSinger`. Реализуем специальный преобразователь для выполнения такого преобразования, как показано ниже.

```
package com.apress.prospring5.ch10;

import org.springframework.core.convert
          .converter.Converter;

public class SingerToAnotherSingerConverter
       implements Converter<Singer, AnotherSinger> {
```

```
  @Override
  public AnotherSinger convert(Singer singer) {
    AnotherSinger anotherSinger = new AnotherSinger();
    anotherSinger.setFirstName(singer.getLastName());
    anotherSinger.setLastName(singer.getFirstName());
    anotherSinger.setBirthDate(singer.getBirthDate());
    anotherSinger.setPersonalSite(singer.getPersonalSite());

    return anotherSinger;
  }
}
```

Этот класс очень прост. В нем лишь меняются местами значения атрибутов `firstName` и `lastName` в классах `Singer` и `AnotherSinger`. Чтобы зарегистрировать специальный преобразователь в контексте типа `ApplicationContext`, достаточно заменить определение компонента `conversionService` в конфигурационном классе `AppConfig` приведенным ниже фрагментом кода.

```
package com.apress.prospring5.ch10.config;

import com.apress.prospring5.ch10.Singer;
import com.apress.prospring5.ch10
          .SingerToAnotherSingerConverter;
import com.apress.prospring5.ch10
          .StringToDateTimeConverter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.context.support
          .ConversionServiceFactoryBean;
import org.springframework.core.convert.converter
         .Converter;

import java.net.URL;
import java.util.HashSet;
import java.util.Set;

@Configuration
@ComponentScan(basePackages = "com.apress.prospring5.ch10")
public class AppConfig {

  @Bean
  public Singer john() throws Exception {
    Singer singer = new Singer();
    singer.setFirstName("John");
    singer.setLastName("Mayer");
    singer.setPersonalSite(
```

```
            new URL("http://johnmayer.com/"));
    singer.setBirthDate(converter().convert("1977-10-16"));
    return singer;
  }

  @Bean
  public ConversionServiceFactoryBean conversionService() {
    ConversionServiceFactoryBean
              conversionServiceFactoryBean =
                        new ConversionServiceFactoryBean();
    Set<Converter> convs = new HashSet<>();
    convs.add(converter());
    convs.add(singerConverter());
    conversionServiceFactoryBean.setConverters(convs);
    conversionServiceFactoryBean.afterPropertiesSet();
    return conversionServiceFactoryBean;
  }

  @Bean
  StringToDateTimeConverter converter() {
    return new StringToDateTimeConverter();
  }

  @Bean
  SingerToAnotherSingerConverter singerConverter() {
    return new SingerToAnotherSingerConverter();
  }
}
```

Чтобы проверить упомянутый выше специальный преобразователь, воспользуемся тестовой программой из предыдущего примера, а по существу, приведенным ниже классом `MultipleConvServDemo`.

```
package com.apress.prospring5.ch10;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;
import org.springframework.core.convert.ConversionService;
import com.apress.prospring5.ch10.config.AppConfig;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
```

```
public class MultipleConvServDemo {
  private static Logger logger =
       LoggerFactory.getLogger(MultipleConvServDemo.class);

  public static void main(String... args) {
    GenericApplicationContext ctx =
          new AnnotationConfigApplicationContext(
              AppConfig.class);

    Singer john = ctx.getBean("john", Singer.class);

    logger.info("Singer info: " + john);

    ConversionService conversionService =
              ctx.getBean(ConversionService.class);

    AnotherSinger anotherSinger =
      conversionService.convert(john, AnotherSinger.class);
    logger.info("Another singer info: " + anotherSinger);
    String[] stringArray =
        conversionService.convert("a,b,c", String[].class);
    logger.info("String array: " + stringArray[0]
              + stringArray[1] + stringArray[2]);
    List<String> listString = new ArrayList<>();
    listString.add("a");
    listString.add("b");
    listString.add("c");

    Set<String> setString =
      conversionService.convert(listString, HashSet.class);

    for (String string: setString)
      System.out.println("Set: " + string);
  }
}
```

В приведенном выше коде ссылка на интерфейс `ConversionService` получается из контекста типа `ApplicationContext`. А поскольку интерфейс `ConversionService` уже зарегистрирован в контексте типа `ApplicationContext` со специальными преобразователями, его можно применять для преобразования объекта типа `Singer`, а также для взаимных преобразований других типов данных, которые поддерживаются в соответствующей службе. Для целей демонстрации в приведенный выше код были введены примеры преобразования разделяемых запятыми символьных строк типа `String` в массив типа `Array`, а также списка типа `List` в множество типа `Set`. Выполнение данной тестовой программы дает следующий результат:

```
[main] INFO c.a.p.c.MultipleConvServDemo - Singer info:
   {First name: John, Last name: Mayer,
    Birthday: 1977-10-16, Site: http://johnmayer.com/}
```

```
[main] INFO c.a.p.c.MultipleConvServDemo
   - Another singer info:
     {First name: Mayer, Last name: John,
      Birthday: 1977-10-16, Site: http://johnmayer.com/}
[main] INFO c.a.p.c.MultipleConvServDemo
   - String array: abc
Set: a
Set: b
Set: c
```

Как следует из приведенного выше результата, преобразование типов `Singer` и `AnotherSinger` выполнено верно, и то же самое можно сказать о преобразованиях типа `String` в тип `Array` и типа `List` в тип `Set`. Для службы преобразования типов данных в Spring можно легко создавать специальные преобразователи и выполнять преобразования на любом уровне приложения. Один из возможных вариантов использования предусматривает наличие двух систем с одной и той же информацией о певцах, которую требуется обновлять. Но структуры баз данных в этих системах отличаются (например, фамилия в одной системе означает имя в другой и т.п.). Службу преобразования типов можно применять для преобразования объектов перед их сохранением в каждой системе в отдельности.

Начиная с версии Spring 3.0, служба преобразования, а также интерфейс Formatter SPI, рассматриваемый в следующем разделе, широко применяется в модуле Spring MVC. С помощью дескриптора разметки `<mvc:annotation-driven/>` в XML-файле конфигурации или аннотации `@EnableWebMvc` (начиная с версии Spring 3.1) в конфигурационном классе Java можно автоматически зарегистрировать в конфигурации контекста веб-приложения все используемые по умолчанию преобразователи (например, `StringToArrayConverter`, `StringToBooleanConverter` и `StringToLocaleConverter` из пакета `org.springframework.core.convert.support`) и средства форматирования (например, `CurrencyFormatter`, `DateFormatter` и `NumberFormatter` из различных подпакетов, входящих в пакет `org.springframework.format`). Подробнее о разработке веб-приложений в Spring речь пойдет в главе 16.

Форматирование полей в Spring

Помимо системы преобразования типов данных, разработчикам приложений Spring доступно также другое превосходное средство — интерфейс Formatter SPI. Как и следовало ожидать, этот интерфейс помогает настраивать все свойства, связанные с форматированием полей.

Для реализации средств форматирования главным в Formatter SPI служит интерфейс `org.springframework.format.Formatter<T>`. В каркасе Spring предоставляется несколько его реализаций для часто применяемых типов данных, в том числе классы `CurrencyFormatter`, `DateFormatter`, `NumberFormatter` и `PercentFormatter`.

Реализация специального средства форматирования

Реализация специального средства форматирования довольно проста. Воспользуемся снова классом `Singer` и реализуем специальное средство форматирования для взаимного преобразования значения типа `DateTime` из атрибута `birthDate` в символьную строку типа `String`. Но на этот раз применим другой подход, расширив класс `org.springframework.format.support.FormattingConversionService FactoryBean` из Spring, чтобы предоставить специальное средство форматирования. `FormattingConversionServiceFactoryBean` — это фабричный класс, обеспечивающий удобный доступ к базовому классу `FormattingConversionService`, поддерживающему систему преобразования типов, а также форматирование полей по правилам, определенным для каждого типа поля.

Ниже приведен исходный код специального класса, расширяющего класс `Format tingConversionServiceFactoryBean` и определяющего специальное средство форматирования, предназначенное для форматирования данных типа `DateTime` из библиотеки Joda-Time.

```
package com.apress.prospring5.ch10;

import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.format.Formatter;
import org.springframework.format.support
          .FormattingConversionServiceFactoryBean;
import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;
import java.text.ParseException;
import java.util.HashSet;
import java.util.Locale;
import java.util.Set;

@Component("conversionService")
public class ApplicationConversionServiceFactoryBean
       extends FormattingConversionServiceFactoryBean {
  private static Logger logger =
      LoggerFactory.getLogger(
            ApplicationConversionServiceFactoryBean.class);
  private static final String DEFAULT_DATE_PATTERN =
          "yyyy-MM-dd";
  private DateTimeFormatter dateFormat;
```

```
  private String datePattern = DEFAULT_DATE_PATTERN;
  private Set<Formatter<?>> formatters = new HashSet<>();
  public String getDatePattern() {
    return datePattern;
  }

  @Autowired(required = false)
  public void setDatePattern(String datePattern) {
    this.datePattern = datePattern;
  }

  @PostConstruct
  public void init() {
    dateFormat = DateTimeFormat.forPattern(datePattern);
    formatters.add(getDateTimeFormatter());
    setFormatters(formatters);
  }

  public Formatter<DateTime> getDateTimeFormatter() {
    return new Formatter<DateTime>() {

      @Override
      public DateTime parse(String dateTimeString,
                    Locale locale) throws ParseException {
        logger.info("Parsing date string: "
                    + dateTimeString);
        return dateFormat.parseDateTime(dateTimeString);
      }

      @Override
      public String print(DateTime dateTime,
                          Locale locale) {
        logger.info("Formatting datetime: " + dateTime);
        return dateFormat.print(dateTime);
      }
    };
  }
}
```

В приведенном выше классе исходный код специального средства форматирования подчеркнут. В нем реализуется интерфейс `Formatter<DateTime>` и оба определенных в нем метода. В частности, метод `parse()` служит для синтаксического анализа и преобразования из формата символьных строк типа `String` в формат даты и времени типа `DateTime` (для целей локализации этому методу передаются также соответствующие региональные настройки), а метод `print()` — для преобразования из формата экземпляров типа `DateTime` в формат символьных строк типа `String`. Шаблон даты может быть внедрен в компонент Spring Bean, а иначе будет употреблен выбираемый по умолчанию шаблон `гггг-мм-дд`. Кроме того, в методе `init()` реги-

стрируется специальное средство форматирования путем вызова метода `setFormatters()`. Добавлять можно столько средств форматирования, сколько потребуется в приложении.

Конфигурирование компонента типа ConversionServiceFactoryBean

Объявив компонент Spring Bean типа `ConversionServiceFactoryBean`, можно существенно сократить объем исходного кода конфигурационного класса `AppConfig`, как показано ниже.

```
package com.apress.prospring5.ch10.config;

import com.apress.prospring5.ch10
          .ApplicationConversionServiceFactoryBean;
import com.apress.prospring5.ch10.Singer;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
          .Configuration;
import java.net.URL;
import java.util.Locale;

@Configuration
@ComponentScan(basePackages = "com.apress.prospring5.ch10")
public class AppConfig {

  @Autowired
  ApplicationConversionServiceFactoryBean conversionService;

  @Bean
  public Singer john() throws Exception {
    Singer singer = new Singer();
    singer.setFirstName("John");
    singer.setLastName("Mayer");
    singer.setPersonalSite(new
           URL("http://johnmayer.com/"));
    singer.setBirthDate(conversionService.
    getDateTimeFormatter()
           .parse("1977-10-16", Locale.ENGLISH));
    return singer;
  }
}
```

А ниже приведена соответствующая тестовая программа.

```
package com.apress.prospring5.ch10;

import com.apress.prospring5.ch10.config.AppConfig;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;
import org.springframework.core.convert
          .ConversionService;

public class ConvFormatServDemo {

  private static Logger logger =
    LoggerFactory.getLogger(ConvFormatServDemo.class);

  public static void main(String... args) {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               AppConfig.class);

    Singer john = ctx.getBean("john", Singer.class);
    logger.info("Singer info: " + john);
    ConversionService conversionService = ctx.getBean(
             "conversionService", ConversionService.class);
    logger.info("Birthdate of singer is : " +
    conversionService.convert(john.getBirthDate(),
                              String.class));

    ctx.close();
  }
}
```

Выполнение этой тестовой программы дает следующий результат:

```
Parsing date string: 1977-10-16
[main] INFO c.a.p.c.ConvFormatServDemo - Singer info: {
     First name: John, Last name: Mayer,
       Birthday: 1977-10-16,
       Site: http://johnmayer.com/}
Formatting datetime: 1977-10-16T00:00:00.000+03:00
       [main] INFO c.a.p.c.ConvFormatServDemo -
Birthdate of singer is : 1977-10-16
```

Как следует из приведенного выше результата, в Spring вызывается метод `parse()` из специального средства форматирования с целью преобразовать значение атрибута `birthDate` из строкового типа String в тип `DateTime` атрибута. А если вызвать метод `ConversionService.convert()`, передав ему атрибут `birthDate`, то в Spring будет вызван метод `print()` для форматирования вывода.

Проверка достоверности данных в Spring

Проверка достоверности данных является крайне важной составляющей любого приложения. Правила проверки достоверности данных, применяемые к объектам предметной области, гарантируют, что все данные вполне структурированы и удовлетворяют всем требованиям к предметной области. В идеальном случае все правила проверки достоверности поддерживаются централизованно, причем один и тот же набор правил применяется к одному и тому же типу данных, независимо от того, из какого источника они поступили (например, от пользователя веб-приложения, из удаленного приложения через веб-службы, из сообщения JMS или файла).

Обсуждая проверку достоверности данных, следует также иметь в виду преобразование и форматирование, потому что, прежде чем проверять фрагмент данных, его необходимо преобразовать в требуемый простой объект POJO по правилам форматирования, определенным для каждого типа данных. Допустим, пользователь сначала вводит сведения о каком-нибудь певце через интерфейс веб-приложения в браузере, а затем передает введенные им данные серверу на обработку. Если веб-приложение на стороне сервера было разработано с помощью модуля Spring MVC, то каркас Spring извлечет данные из HTTP-запроса и выполнит преобразование строковых данных типа `String` в требуемый тип по правилам форматирования (например, символьная строка типа `String`, представляющая дату, будет преобразована в поле типа `Date` по правилу форматирования `гггг-ммм-дд`). Этот процесс называется *привязкой данных*. Как только привязка данных будет завершена, а объект предметной области сконструирован, этот объект подвергается проверке достоверности, а возникшие ошибки возвращаются и отображаются пользователю. Если же проверка достоверности завершится удачно, проверенный таким образом объект сохраняется в базе данных

В Spring поддерживаются два основных типа проверки достоверности данных. Один из них предоставляется самим каркасом Spring и предусматривает создание специальных средств проверки достоверности путем реализации интерфейса `org.springframework.validation.Validator`. А другой тип проверки достоверности опирается на поддержку спецификации JSR-349 (Bean Validation — Проверка достоверности компонентов Spring Beans). Оба типа проверки достоверности данных рассматриваются в последующих разделах.

Применение интерфейса `Validator` в Spring

Создав класс для реализации интерфейса `Validator` в Spring, можно разработать логику проверки достоверности данных. Выясним, как это делается. Допустим, в классе `Singer`, с которым мы имели дело до сих пор, имя певца не должно быть пустым. Чтобы проверить объекты типа `Singer` на предмет соблюдения этого правила, можно создать специальное средство проверки достоверности. Класс, реализующий такое средство проверки достоверности, приведен ниже.

```
package com.apress.prospring5.ch10;

import org.springframework.stereotype.Component;
import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

@Component("singerValidator")
public class SingerValidator implements Validator {
  @Override
  public boolean supports(Class<?> clazz) {
    return Singer.class.equals(clazz);
  }

  @Override
  public void validate(Object obj, Errors e) {
    ValidationUtils.rejectIfEmpty(e, "firstName",
                                  "firstName.empty");
  }
}
```

В приведенном выше классе `SingerValidator` реализуется интерфейс `Validator` и два его метода. Так, в методе `supports()` указывается, поддерживается ли в средстве проверки достоверности данных подобная проверка передаваемого ему класса, тогда как в методе `validate()` проверяется достоверность передаваемого ему объекта, а результат проверки сохраняется в экземпляре реализации интерфейса `org.springframework.validation.Errors`. В методе `validate()` проверяется только атрибут `firstName` и с помощью удобного метода `ValidationUtils.rejectIfEmpty()` гарантируется, что имя певца не окажется пустым. В качестве второго аргумента этому методу передается код ошибки, который может быть использован для поиска в пакетах ресурсов сообщений, связанных с проверкой достоверности, и последующего вывода локализованных сообщений об ошибках.

Ниже приведен исходный код соответствующего конфигурационного класса.

```
package com.apress.prospring5.ch10.config;

import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
          .Configuration;

@Configuration
@ComponentScan(basePackages = "com.apress.prospring5.ch10")
  public class AppConfig {
}
```

В следующем фрагменте кода приведена соответствующая тестовая программа:

```
package com.apress.prospring5.ch10;

import com.apress.prospring5.ch10.config.AppConfig;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;
import org.springframework.validation
          .BeanPropertyBindingResult;
import org.springframework.validation.ObjectError;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;
import java.util.List;

public class SpringValidatorDemo {

  private static Logger logger =
          LoggerFactory.getLogger(
                        SpringValidatorDemo.class);

  public static void main(String... args) {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               AppConfig.class);

    Singer singer = new Singer();
    singer.setFirstName(null);
    singer.setLastName("Mayer");
    Validator singerValidator =
           ctx.getBean("singerValidator", Validator.class);
    BeanPropertyBindingResult result =
        new BeanPropertyBindingResult(singer, "John");

    ValidationUtils.invokeValidator(
                    singerValidator, singer, result);

    List<ObjectError> errors = result.getAllErrors();
    logger.info("No of validation errors: "
              + errors.size());
    errors.forEach(e -> logger.info(e.getCode()));

    ctx.close();
  }
}
```

В приведенном выше коде тестовой программы конструируется объект типа `Singer`, где имя певца устанавливается равным `null`. Затем из контекста типа

ApplicationContext извлекается средство проверки достоверности. Для сохранения результата проверки достоверности создается экземпляр класса BeanPropertyBindingResult. А для выполнения проверки достоверности вызывается метод ValidationUtils.invokeValidator(). После этого производится проверка на предмет наличия ошибок. Выполнение этой тестовой программы дает приведенный ниже результат. Как видите, в ходе проверки достоверности произошла одна ошибка, код которой был правильно отображен.

```
[main] INFO c.a.p.c.SpringValidatorDemo
    - No of validation errors: 1
[main] INFO c.a.p.c.SpringValidatorDemo - firstName.empty
```

Применение спецификации JSR-349 (Bean Validation)

В версии 4 в каркасе Spring реализована полноценная поддержка спецификации JSR-349 на прикладной интерфейс Bean Validation API для проверки достоверности данных. В этом прикладном интерфейсе определяется ряд ограничений из пакета javax.validation.constraints в форме аннотаций Java (например, @NotNull), которые могут быть наложены на объекты предметной области. Кроме того, с помощью аннотаций могут быть разработаны и применены специальные средства проверки достоверности (например, на уровне класса).

Применение прикладного интерфейса Bean Validation API освобождает от привязки к конкретному поставщику услуг проверки достоверности. Прикладной интерфейс Bean Validation API позволяет использовать стандартные аннотации для реализации логики проверки достоверности в объектах предметной области, даже не зная базового поставщика услуг проверки достоверности. Эталонным поставщиком таких услуг по спецификации JSR-349 служит, например, Hibernate Validator версии 5 (http://hibernate.org/validator/).

В каркасе Spring предоставляется цельная поддержка прикладного интерфейса Bean Validation API. К основным поддерживаемым функциональным возможностям относится определение ограничений проверки достоверности с помощью стандартных аннотаций по спецификации JSR-349, специальные средства проверки достоверности и ее конфигурирования по спецификации JSR-349 в контексте типа ApplicationContext. Все эти возможности рассматриваются по очереди в последующих разделах. В каркасе Spring все еще полностью поддерживается спецификация JSR-303, если в пути к классам указывается средство проверки достоверности Hibernate Validator версии 4, а также прикладной интерфейс API версии 1.0 для проверки достоверности.

Наложение ограничений проверки достоверности на свойства объектов

Начнем с наложения ограничений проверки достоверности на свойства объектов предметной области. Ниже приведен усовершенствованный вариант класса Singer,

дополненный ограничениями проверки достоверности, наложенными на свойства `firstName` и `genre`.

```
package com.apress.prospring5.ch10.obj;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class Singer {

  @NotNull
  @Size(min=2, max=60)
  private String firstName;

  private String lastName;

  @NotNull
  private Genre genre;

  private Gender gender;

  // Методы установки и получения
  ...
}
```

В приведенном выше коде накладываемые ограничения специально подчеркнуты. В частности, на свойство `firstName` накладываются два ограничения. Первое из них задается в аннотации `@NotNull`, где указывается, что значение данного свойства не должно быть пустым. А в аннотации `@Size` задается длина атрибута `firstName`. Ограничение из аннотации `@NotNull` накладывается также на свойство `customer Type`. Ниже представлены классы перечислений `Genre` и `Gender` соответственно. В первом из них определяется музыкальный жанр, к которому относится певец, а во втором — пол, хотя это и не имеет никакого отношения к музыкальной карьере певца, а следовательно, поле, определяющее пол, может быть пустым.

```
// Исходный файл Genre.java
package com.apress.prospring5.ch10.obj;
public enum Genre {
  POP("P"),
  JAZZ("J"),
  BLUES("B"),
  COUNTRY("C");
  private String code;

  private Genre(String code) {
   this.code = code;
  }
```

```
  public String toString() {
    return this.code;
  }
}

// Исходный файл Gender.java
package com.apress.prospring5.ch10.obj;

public enum Gender {
  MALE("M"), FEMALE("F");
  private String code;

  Gender(String code) {
    this.code = code;
  }

  @Override
  public String toString() {
    return this.code;
  }
}
```

Конфигурирование поддержки проверки достоверности для компонентов Spring Beans

Чтобы сконфигурировать поддержку прикладного интерфейса Bean Validation API в контексте типа `ApplicationContext`, следует определить экземпляр класса `org.springframework.validation.beanvalidation.LocalValidatorFactoryBean` в конфигурации, составляемой в Spring. Ниже приведен соответствующий конфигурационный класс.

```
package com.apress.prospring5.ch10.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.validation.beanvalidation
          .LocalValidatorFactoryBean;

@Configuration
@ComponentScan(basePackages = "com.apress.prospring5.ch10")
public class AppConfig {
  @Bean LocalValidatorFactoryBean validator() {
    return new LocalValidatorFactoryBean();
  }
}
```

Как видите, достаточно указать объявление компонента Spring Bean типа `LocalValidatorFactoryBean`. По умолчанию каркас Spring будет искать библиотеку Hibernate Validator по пути к классам. А теперь создадим служебный класс, предоставляющий услуги проверки достоверности для класса `Singer`. Этот служебный класс приведен ниже.

```
package com.apress.prospring5.ch10;

import com.apress.prospring5.ch10.obj.Singer;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.stereotype.Service;
import javax.validation.ConstraintViolation;
import javax.validation.Validator;
import java.util.Set;

@Service("singerValidationService")
public class SingerValidationService {

 @Autowired
  private Validator validator;

  public Set<ConstraintViolation<Singer>>
         validateSinger(Singer singer) {
    return validator.validate(singer);
  }
}
```

Здесь внедряется экземпляр класса `javax.validation.Validator` (обратите внимание на его отличие от интерфейса `Validator`, предоставляемого в Spring под полностью уточненным именем `org.springframework.validation.Validator`). Как только компонент Spring Bean типа `LocalValidatorFactoryBean` будет определен, в любом месте приложения можно создать ссылку на интерфейс `Validator`. Для проверки достоверности простого объекта POJO вызывается метод `Validator.validate()`, а результаты проверки достоверности возвращаются в виде списка типа `List` объектов, реализующих интерфейс `ConstraintViolation<T>`.

Ниже приведена соответствующая тестовая программа.

```
package com.apress.prospring5.ch10;

import com.apress.prospring5.ch10.config.AppConfig;
import com.apress.prospring5.ch10.obj.Singer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;
```

```
import javax.validation.ConstraintViolation;
import java.util.Set;

public class Jsr349Demo {
  private static Logger logger =
          LoggerFactory.getLogger(Jsr349Demo.class);

  public static void main(String... args) {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               AppConfig.class);

    SingerValidationService singerBeanValidationService =
          ctx.getBean(SingerValidationService.class);

    Singer singer = new Singer();
    singer.setFirstName("J");
    singer.setLastName("Mayer");
    singer.setGenre(null);
    singer.setGender(null);

    validateSinger(singer, singerBeanValidationService);

    ctx.close();
  }

  private static void validateSinger(Singer singer,
         SingerValidationService singerValidationService) {
    Set<ConstraintViolation<Singer>> violations =
        singerValidationService.validateSinger(singer);
    listViolations(violations);
  }

  private static void listViolations(
          Set<ConstraintViolation<Singer>> violations) {
    logger.info("No. of violations: " + violations.size());
    for (ConstraintViolation<Singer> violation :
         violations) {
      logger.info("Validation error for property: "
                + violation.getPropertyPath()
                + " with value: "
                + violation.getInvalidValue()
                + " with error message: "
                + violation.getMessage());
    }
  }
}
```

Как следует из приведенного выше тестового кода, объект типа `Singer` конструируется со значениями свойств `firstName` и `genre`, нарушающими ограничения. В методе `validateCustomer()` вызывается метод `MyBeanValidationService.validateCustomer()`, который, в свою очередь, обращается к прикладному интерфейсу Bean Valiadation API. Выполнение данной тестовой программы дает следующий результат:

```
[main] INFO o.h.v.i.u.Version - HV000001:
   Hibernate Validator 6.0.0.Beta2
[main] INFO c.a.p.c.Jsr349Demo - No. of violations: 2
[main] INFO c.a.p.c.Jsr349Demo
     - Validation error for property:
   firstName with value: J with error message: size
       must be between 2 and 60
[main] INFO c.a.p.c.Jsr349Demo
     - Validation error for property:
   genre with value: null with error message:
       may not be null
```

Как следует из приведенного выше результата, при проверке достоверности были обнаружены два нарушения, о которых выведены соответствующие сообщения. Кроме того, библиотека Hibernate Validator уже составила на основании аннотации стандартные сообщения об ошибках проверки достоверности. Можно также предоставить собственные сообщения об ошибках, как демонстрируется в следующем разделе.

Создание специального средства проверки достоверности

Проверку достоверности данных можно выполнять не только на уровне свойств, но и на уровне классов. Например, в классе `Singer` требуется убедиться, что свойства `lastName` и `gender` не содержат пустое значение `null` для отдельных певцов в стиле кантри. И для такой проверки достоверности данных можно разработать специальное средство. Разработка специального средства проверки достоверности с помощью прикладного интерфейса Bean Validation API осуществляется в две стадии. На первой стадии создается тип аннотации для средства проверки достоверности, как показано ниже, а на второй — класс, реализующий логику проверки достоверности данных.

```
package com.apress.prospring5.ch10;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import javax.validation.Constraint;
import javax.validation.Payload;
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Constraint(validatedBy=CountrySingerValidator.class)
@Documented
public @interface CheckCountrySinger {
  String message() default "Country Singer should
         have gender and last name defined";
  Class<?> groups() default {};
  Class<? extends Payload> payload() default {};
}
```

Аннотация `@Target(ElementType.TYPE)` означает, что создаваемая аннотация должна применяться только на уровне классов. В аннотации `@Constraint` указывается, что определяется средство проверки достоверности, а в атрибуте `validatedBy` задается класс, предоставляющий логику проверки достоверности. В теле создаваемой аннотации определены три атрибута в форме методов, как поясняется ниже.

- Атрибут `mess+age` определяет возвращаемое сообщение (или код ошибки) для возврата в том случае, если нарушено ограничение. В аннотации может быть также предоставлено стандартное сообщение.
- Атрибут `groups` обозначает группу проверки достоверности, если это уместно. Средства проверки достоверности допускается назначать для разных групп и выполнять проверку в отдельной группе.
- Атрибут `payload` обозначает дополнительные объекты полезной информации, относящиеся к классу, реализующему интерфейс `javax.validation.Payload`. Этот атрибут позволяет снабдить ограничение дополнительной информацией (например, объект с полезной информацией может указывать на серьезность нарушения наложенного ограничения).

Ниже приведен класс `CountrySingerValidator`, в котором предоставляется логика специальной проверки достоверности.

```
package com.apress.prospring5.ch10;

import com.apress.prospring5.ch10.obj.Singer;
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class CountrySingerValidator
       implements ConstraintValidator<CheckCountrySinger,
                                       Singer> {
  @Override
  public void initialize(CheckCountrySinger
                         constraintAnnotation) {
  }

  @Override
```

```
  public boolean isValid(Singer singer,
                ConstraintValidatorContext context) {
    boolean result = true;
    if (singer.getGenre() != null
        && (singer.isCountrySinger()
        && (singer.getLastName() == null
        || singer.getGender() == null))) {
      result = false;
    }
    return result;
  }
}
```

Класс `CountrySingerValidator` реализует интерфейс `ConstraintValidator <CheckCountrySinger, Singer>`, а это означает, что создаваемое средство проверки достоверности проверяет наличие аннотации `@CheckCountrySinger` в классе `Singer`. В данном классе реализуется метод `isValid()`, которому базовый поставщик услуг проверки достоверности (например, Hibernate Validator) передает проверяемый экземпляр. В этом методе проверяется, относится ли певец к категории исполнителей музыки в стиле кантри, а свойства `lastName` и `gender` не содержат пустое значение `null`. В итоге получается логическое значение, обозначающее результат проверки достоверности.

Чтобы активизировать проверку достоверности с помощью специально созданного средства, достаточно применить аннотацию `@CheckCountrySinger` в классе `Singer`, как показано ниже.

```
package com.apress.prospring5.ch10.obj;

import com.apress.prospring5.ch10.CheckCountrySinger;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

@CheckCountrySinger
public class Singer {

  @NotNull
  @Size(min = 2, max = 60)
  private String firstName;

  private String lastName;

  @NotNull
  private Genre genre;

  private Gender gender;

  // Методы получения и установки
  ...
```

```
  public boolean isCountrySinger() {
    return genre == Genre.COUNTRY;
  }
}
```

Чтобы проверить вновь созданное специальное средство проверки достоверности, достаточно получить экземпляр типа `Singer` и заполнить его контрольными данными, как показано в приведенном ниже тестовом классе `Jsr349CustomDemo`.

```
public class Jsr349CustomDemo {
  ...

  Singer singer = new Singer();
  singer.setFirstName("John");
  singer.setLastName("Mayer");
  singer.setGenre(Genre.COUNTRY);
  singer.setGender(null);

  validateSinger(singer, singerValidationService);

  ...
}
```

Выполнение приведенного выше тестового кода дает следующий (приведенный не полностью) результат:

```
[main] INFO o.h.v.i.u.Version - HV000001:
   Hibernate Validator 6.0.0.Beta2
[main] INFO c.a.p.c.Jsr349CustomDemo - No. of violations: 1
[main] INFO c.a.p.c.Jsr349CustomDemo
   - Validation error for property:
     with value:
       com.apress.prospring5.ch10.obj.Singer@3116c353
     with error message:
       Country Singer should have gender and last name defined
```

Как следует из приведенного выше результата, проверяемое значение (в данном случае — экземпляр типа `Singer`) нарушает правило проверки достоверности для певцов в стиле кантри, поскольку свойство `gender` содержит пустое значение `null`. Обратите также внимание на то, что путь к данному свойству не указан, поскольку ошибка проверки достоверности выявлена на уровне класса.

Специальная проверка достоверности с помощью аннотации `@AssertTrue`

Помимо специального средства проверки достоверности, для реализации специальной проверки достоверности в прикладном интерфейсе Bean Validation API можно также воспользоваться аннотацией `@AssertTrue`. Покажем, как это делается, на

примере класса `Singer`, удалив аннотацию `@CheckCountrySinger` и внеся соответствующие коррективы в метод `isCountrySinger()`.

```
public class Singer {

  @NotNull
  @Size(min = 2, max = 60)
  private String firstName;

  private String lastName;

  @NotNull
  private Genre genre;

  private Gender gender;
  ...

  @AssertTrue(message="ERROR! Individual customer should
              have gender and last name defined")
  public boolean isCountrySinger() {
    boolean result = true;
    if (genre!= null && (genre.equals(Genre.COUNTRY)
        && (gender == null || lastName == null))) {
      result = false;
    }
    return result;
  }
}
```

Из приведенного выше исходного кода можно сделать вывод, что аннотация `@CheckCountrySinger` и класс `CountrySingerValidator` больше не требуются. В класс `Singer` введен метод `isCountrySinger()`, снабженный аннотацией `@AssertTrue` (из пакета `javax.validation.constraints`). При обращении к проверке достоверности соответствующий поставщик услуг инициирует эту проверку и удостоверится, что она дает истинный результат. По спецификации JSR-349 предоставляется также аннотация `@AssertFalse` для проверки условий, которые должны быть ложными. Если выполнить тестовый код (из класса `Jsr349AssertTrue Demo`) снова, то будет получен такой же результат, как и в предыдущем примере, где применялось специальное средство проверки достоверности.

Соображения по поводу специальной проверки достоверности

Так какой же подход выбрать к специальной проверке достоверности по спецификации JSR-349: реализовать специальное средство проверки достоверности или воспользоваться аннотацией `@AssertTrue`? Как правило, пользоваться аннотацией

`@AssertTrue` проще, а правила проверки доступны непосредственно в коде, реализующем объекты предметной области. Но при более сложной логике проверки достоверности, когда, например, требуется внедрить служебный класс, получить доступ к базе данных и проверить достоверность целого ряда значений, следует выбрать реализацию специального средства проверки достоверности, поскольку внедрять объекты уровня обслуживания в объекты предметной области совсем не желательно. Кроме того, специальные средства проверки достоверности можно применять неоднократно к сходным объектам предметной области.

Выбор прикладного интерфейса API для проверки достоверности

Итак, рассмотрев функциональные возможности интерфейса `Validator` из Spring и прикладного интерфейса Bean Validation API, остается выяснить, каким из них лучше пользоваться в приложениях? Разумеется, лучше выбрать прикладной интерфейс Bean Validation API по спецификации JSR-349, поскольку для этого имеются следующие веские основания.

- JSR-349 — это стандарт JEE, имеющий обширную поддержку во многих каркасах для клиентской и серверной частей прикладных систем (например, в Spring, JPA 2, Spring MVC и GWT).
- Спецификация JSR-349 предоставляет стандартный прикладной интерфейс API, скрывающий исходного поставщика услуг проверки достоверности, не привязывая разработчиков приложений к конкретному поставщику.
- Начиная с версии 4, каркас Spring тесно связан со спецификацией JSR-349. Например, в веб-контроллере Spring MVC аргумент метода можно снабдить аннотацией `@Valid` (из пакета `javax.validation`), и благодаря этому каркас Spring будет автоматически выполнять проверку достоверности по спецификации JSR-349 в процессе привязки данных. Кроме того, в конфигурации контекста веб-приложения Spring MVC с помощью простого дескриптора `<mvc:annotation-driven/>` можно настроить Spring на автоматическое включение системы преобразования типов и форматирования полей, а также на поддержку спецификации JSR-349 (Bean Validation).
- Если применяется прикладной интерфейс JPA 2, то поставщик услуг автоматически выполнит проверку достоверности сущности по спецификации JSR-349, прежде чем сохранить ее, предоставляя тем самым еще один уровень защиты.

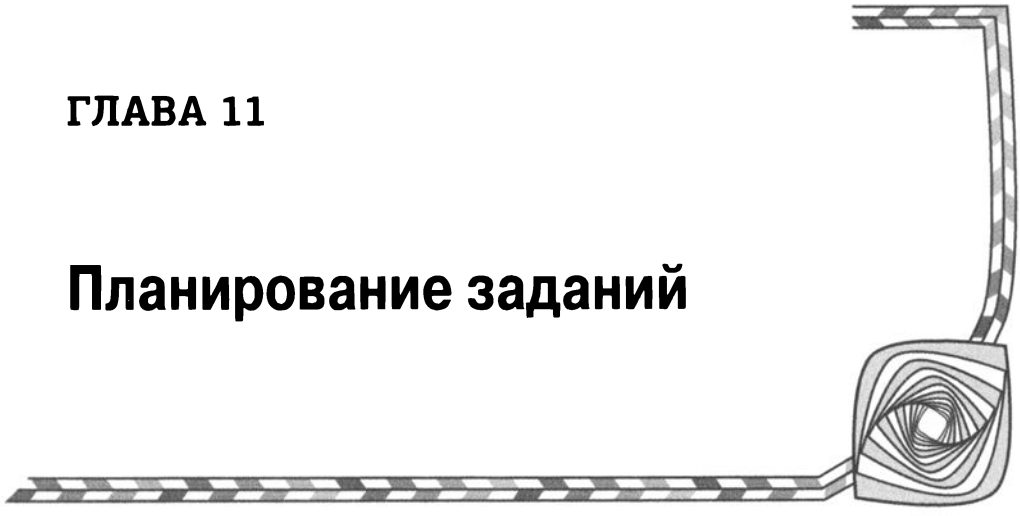
Подробнее о применении прикладного интерфейса Bean Validation API по спецификации JSR-349 вместе с Hibernate Validator в качестве поставщика услуг проверки достоверности можно узнать в документации на Hibernate Validator, оперативно доступной по адресу `http://docs.jboss.org/hibernate/validator/5.1/reference/en-US/html`.

Резюме

В этой главе были рассмотрены система преобразования типов данных в Spring, а также интерфейс SPI для средств форматирования полей. В ней пояснялось, как пользоваться новой системой для взаимного преобразования произвольных типов данных, помимо поддержки редакторов свойств. Кроме того, была описана поддержка в Spring проверки достоверности данных с помощью интерфейса `Validator` и по рекомендуемой спецификации JSR-349 (Bean Validation).

ГЛАВА 11

Планирование заданий



Планирование заданий относится к типичным функциональным возможностям корпоративных приложений. Оно состоит из трех частей: задания, т.е. порции бизнес-логики, которая должна выполняться в определенное время или же регулярно, средства запуска, задающего условие, при котором задание должно быть выполнено, а также планировщика, выполняющего задание на основании информации, получаемой из средства запуска. В этой главе будут, в частности, рассмотрены следующие вопросы.

- **Планирование заданий в Spring.** В этой части описана поддержка планирования заданий в Spring с акцентом на абстракции интерфейса `TaskScheduler`, появившегося в версии Spring 3. Здесь представлены также сценарии планирования, в том числе через фиксированные промежутки времени и с помощью выражений формата `cron`.
- **Асинхронное выполнение заданий.** В этой части показано, как пользоваться аннотацией `@Async` в Spring для асинхронного выполнения заданий.
- **Выполнение заданий в Spring.** В этой части вкратце описан интерфейс `TaskExecutor`, а также особенности выполнения заданий в Spring.

Зависимости для примеров планирования заданий

В приведенном ниже фрагменте кода конфигурации Gradle представлены зависимости, требующиеся для примеров из этой главы.

```
// Файл конфигурации pro-spring-15/build.gradle
ext {
    springDataVersion = '2.0.0.M3'

    // Библиотеки протоколирования
```

```
    slf4jVersion = '1.7.25'
    logbackVersion = '1.2.3'

    guavaVersion = '21.0'
    jodaVersion = '2.9.9'
    utVersion = '6.0.1.GA'

    junitVersion = '4.12'

    spring = [
        data : "org.springframework.data:
                spring-data-jpa:$springDataVersion",
        ...
    ]

    testing = [
       junit: "junit:junit:$junitVersion"
    ]

    misc = [
       slf4jJcl : "org.slf4j:jcl-over-slf4j:$slf4jVersion",
        logback : "ch.qos.logback:logback-classic:
                   $logbackVersion",
        guava : "com.google.guava:guava:$guavaVersion",
        joda : "joda-time:joda-time:$jodaVersion",
        usertype : "org.jadira.usertype:usertype.core:$utVersion",
        ...
    ]
    ...
}
...
// Файл конфигурации chapter11/build.gradle
dependencies {

    compile (spring.contextSupport) {
        exclude module: 'spring-context'
        exclude module: 'spring-beans'
        exclude module: 'spring-core'
    }
    compile misc.slf4jJcl, misc.logback, misc.lang3,
           spring.data, misc.guava, misc.joda,
            misc.usertype, db.h2

    testCompile testing.junit
}
```

Планирование заданий в Spring

Корпоративные приложения часто нуждаются в планировании заданий. Во многих приложениях разнообразные задания (вроде отправки заказчикам уведомлений

по электронной почте, запуска заданий в конце дня, обслуживания данных, обновления данных в пакетах и т.п.) должны планироваться для регулярного выполнения, через фиксированные промежутки времени (например, через каждый час) или по заданному расписанию (например, ежедневно в 20:00 с понедельника по пятницу). Как упоминалось ранее, планирование заданий состоит из трех частей: составления календарного плана (средство запуска), выполнения задания (планировщик) и самого задания.

Запустить задание на выполнение в Spring можно самыми разными способами. Один из них предусматривает внешний запуск заданий на выполнение из системы календарного планирования, которая уже существует в среде развертывания приложений. Так, на многих предприятиях для планирования заданий используются такие коммерческие системы, как Control-M или CA AutoSys. Если приложение выполняется на платформе Linux/Unix, то можно воспользоваться планировщиком `crontab`. Чтобы запустить задание на выполнение, достаточно отправить соответствующий запрос веб-службы RESTful-WS приложению Spring, и в результате контроллер Spring MVC инициирует выполнение задания.

Еще один способ предполагает использование поддержки планирования заданий в Spring. Для планирования заданий в Spring доступны следующие возможности.

- **Поддержка объекта типа `Timer` из JDK.** Для планирования заданий в Spring поддерживается объект класса `Timer` из комплекта JDK.
- **Интеграция с Quartz.** Каркас Spring интегрирован с Quartz Scheduler[1] — широко распространенной библиотекой планирования с открытым кодом.
- **Абстракция интерфейса `TaskScheduler` в Spring.** В версии Spring 3 появилась абстракция интерфейса `TaskScheduler`, предоставляющая простой способ планирования заданий и поддерживающая наиболее типичные требования.

Далее в этом разделе основное внимание уделяется применению абстракции интерфейса `TaskScheduler` для планирования заданий.

Введение в абстракцию интерфейса `TaskScheduler`

Абстракция интерфейса `TaskScheduler` в Spring состоит из следующих основных частей.

- **Интерфейс `Trigger`.** Интерфейс `org.springframework.scheduling.Trigger` обеспечивает поддержку, необходимую для определения механизма запуска заданий на выполнение. В каркасе Spring доступны две реализации интерфейса `Trigger`. Так, в классе `CronTrigger` поддерживается запуск заданий на выполнение, исходя из выражений формата `cron`, а в классе `Periodic`

[1] Официальный веб-сайт Quartz Scheduler находится по адресу `www.quartz-scheduler.org`.

`Trigger` — запуск сначала с некоторой задержкой, а затем через фиксированные промежутки времени.

- **Задание.** Это часть бизнес-логики, выполнение которой требуется запланировать. В каркасе Spring задание может быть определено как метод в любом компоненте Spring Bean.
- **Интерфейс `TaskScheduler`.** Интерфейс `org.springframework.scheduling.TaskScheduler` обеспечивает поддержку, необходимую для планирования заданий. В каркасе Spring доступны три реализации интерфейса `TaskScheduler`. Так, класс `TimerManagerTaskScheduler` (из пакета `org.springframework.scheduling.commonj`) служит оболочкой для интерфейса `commonj.timers.TimerManager`, который входит в состав прикладного интерфейса CommonJ и обычно применяется на коммерческих серверах приложений JEE вроде WebSphere и WebLogic. А классы `ConcurrentTaskScheduler` и `ThreadPoolTaskScheduler` (из пакета `org.springframework.scheduling.concurrent`) служат оболочкой для класса `java.util.concurrent.ScheduledThreadPoolExecutor` и поддерживают запуск заданий из общего пула потоков исполнения.

На рис. 11.1 схематически показаны отношения между интерфейсом `Trigger`, интерфейсом `TaskScheduler` и заданием, реализующим интерфейс `java.lang.Runnable`. Планировать задания, применяя абстракцию интерфейса `TaskScheduler` в Spring, можно двумя способами. В одном из них предусматривается применение пространства имен `task` при конфигурировании приложений Spring в формате XML, а в другом — аннотаций. Рассмотрим оба эти способа по очереди.

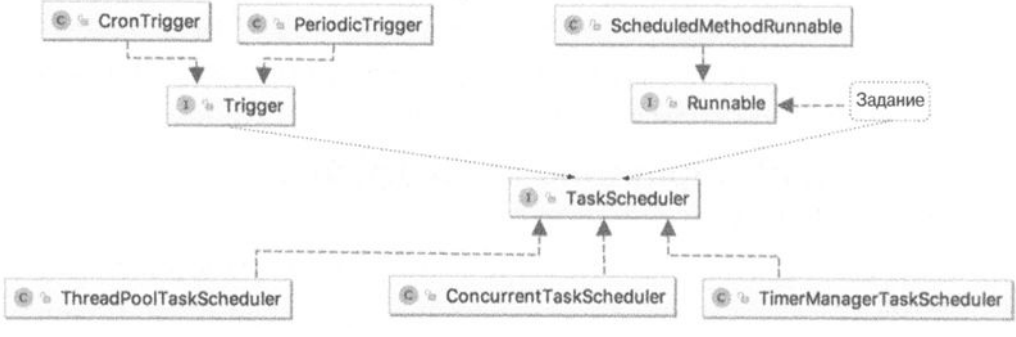


Рис. 11.1. Отношения между средством запуска, заданием и планировщиком

Анализ примера задания

Чтобы продемонстрировать особенности планирования заданий в Spring, реализуем сначала простое задание, а именно: приложение, в котором ведется база данных со сведениями об автомобилях. Ниже приведен класс `Car`, реализованный как класс сущности из прикладного интерфейса JPA.

```
package com.apress.prospring5.ch11;

import static javax.persistence.GenerationType.IDENTITY;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Version;

import org.hibernate.annotations.Type;
import org.joda.time.DateTime;

@Entity
@Table(name="car")
public class Car {
  @Id
  @GeneratedValue(strategy = IDENTITY)

  @Column(name = "ID")
  private Long id;

  @Column(name="LICENSE_PLATE")
  private String licensePlate;

  @Column(name="MANUFACTURER")
  private String manufacturer;

  @Column(name="MANUFACTURE_DATE")
  @Type(type="org.jadira.usertype.dateandtime
                .joda.PersistentDateTime")
  private DateTime manufactureDate;
  @Column(name="AGE")
  private int age;

  @Version
  private int version;

  // Методы получения и установки
  ...

  @Override
  public String toString() {
    SimpleDateFormat sdf =
          new SimpleDateFormat("yyyy-MM-dd");
    return String.format("{License: %s, Manufacturer: %s,
                         Manufacture Date: %s, Age: %d}",
                         licensePlate, manufacturer,
                      sdf.format(manufactureDate.toDate()), age);
  }
}
```

Этот класс сущности служит в качестве модели данных для таблицы CAR, формируемой средствами Hibernate. Конфигурирование уровня доступа к данным и уровня обслуживания для всех примеров из этой главы сведено в один конфигурационный класс, исходный код которого приведен ниже.

```
package com.apress.prospring5.ch11.config;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.data.jpa.repository.config
          .EnableJpaRepositories;
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseType;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa
          .LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor
          .HibernateJpaVendorAdapter;
import org.springframework.transaction
          .PlatformTransactionManager;

import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;
import java.util.Properties;

@Configuration
@EnableJpaRepositories(basePackages =
          {"com.apress.prospring5.ch11.repos"})
@ComponentScan(basePackages =
          {"com.apress.prospring5.ch11"} )
public class DataServiceConfig {

  private static Logger logger =
          LoggerFactory.getLogger(DataServiceConfig.class);

  @Bean
  public DataSource dataSource() {
    try {
      EmbeddedDatabaseBuilder dbBuilder =
              new EmbeddedDatabaseBuilder();
```

```
        return dbBuilder.setType(EmbeddedDatabaseType.H2)
                        .build();
      } catch (Exception e) {
        logger.error("Embedded DataSource bean cannot "
                   + "be created!", e);
        return null;
      }
    }

    @Bean
    public Properties hibernateProperties() {
      Properties hibernateProp = new Properties();
      hibernateProp.put("hibernate.dialect",
                        "org.hibernate.dialect.H2Dialect");
      hibernateProp.put("hibernate.hbm2ddl.auto",
                        "create-drop");
      // hibernateProp.put("hibernate.format_sql", true);
      hibernateProp.put("hibernate.show_sql", true);
      hibernateProp.put("hibernate.max_fetch_depth", 3);
      hibernateProp.put("hibernate.jdbc.batch_size", 10);
      hibernateProp.put("hibernate.jdbc.fetch_size", 50);
      return hibernateProp;
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
      return new JpaTransactionManager(entityManagerFactory());
    }

    @Bean
    public JpaVendorAdapter jpaVendorAdapter() {
      return new HibernateJpaVendorAdapter();
    }

    @Bean
    public EntityManagerFactory entityManagerFactory() {
      LocalContainerEntityManagerFactoryBean factoryBean =
           new LocalContainerEntityManagerFactoryBean();
      factoryBean.setPackagesToScan(
                  "com.apress.prospring5.ch11.entities");
      factoryBean.setDataSource(dataSource());
      factoryBean.setJpaVendorAdapter(
                  new HibernateJpaVendorAdapter());
      factoryBean.setJpaProperties(hibernateProperties());
      factoryBean.setJpaVendorAdapter(jpaVendorAdapter());
      factoryBean.afterPropertiesSet();
      return factoryBean.getNativeEntityManagerFactory();
    }
}
```

Ниже приведен класс `DBInitializer`, отвечающий за заполнение данными таблицы `CAR`.

```
package com.apress.prospring5.ch11.config;

import com.apress.prospring5.ch11.entities.Car;
import com.apress.prospring5.ch11.repos.CarRepository;
import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import javax.annotation.PostConstruct;

@Service
public class DBInitializer {

  private Logger logger =
          LoggerFactory.getLogger(DBInitializer.class);
  @Autowired CarRepository carRepository;

  @PostConstruct
  public void initDB() {
    logger.info("Starting database initialization...");
    DateTimeFormatter formatter =
        DateTimeFormat.forPattern("yyyy-MM-dd");

    Car car = new Car();
    car.setLicensePlate("GRAVITY-0405");
    car.setManufacturer("Ford");
    car.setManufactureDate(DateTime.parse("2006-09-12",
                           formatter));
    carRepository.save(car);

    car = new Car();
    car.setLicensePlate("CLARITY-0432");
    car.setManufacturer("Toyota");
    car.setManufactureDate(DateTime.parse("2003-09-09",
                           formatter));
    carRepository.save(car);

    car = new Car();
    car.setLicensePlate("ROSIE-0402");
    car.setManufacturer("Toyota");
    car.setManufactureDate(DateTime.parse("2017-04-16",
                           formatter));
    carRepository.save(car);
```

```
    logger.info("Database initialization finished.");
  }
}
```

А теперь определим уровень объектов DAO для сущности типа `Car`. Воспользуемся проектом Spring Data JPA и, в частности, абстрактной поддержкой в нем информационного хранилища. Ниже приведен интерфейс `CarRepository`, который служит в качестве простого расширения интерфейса `CrusRepository`, поскольку в данном случае нас вообще не интересуют специальные операции над объектами DAO.

```
package com.apress.prospring5.ch11.repos;

import com.apress.prospring5.ch11.entities.Car;
import org.springframework.data.repository.CrudRepository;

public interface CarRepository
       extends CrudRepository<Car, Long> {
}
```

Уровень обслуживания представлен интерфейсом `CarService` и его реализацией в классе `CarServiceImpl`, как показано ниже.

```
package com.apress.prospring5.ch11.services;
// Архивный JAR-файл CarService.jar

import com.apress.prospring5.ch11.entities.Car;
import java.util.List;

public interface CarService {
  List<Car> findAll();
  Car save(Car car);
  void updateCarAgeJob();
  boolean isDone();
}

// Архивный JAR-файл CarServiceImpl.jar
...
@Service("carService")
@Repository
@Transactional
public class CarServiceImpl implements CarService {
  public boolean done;

  final Logger logger = LoggerFactory.getLogger(
                        CarServiceImpl.class);

  @Autowired
  CarRepository carRepository;
  @Override
  @Transactional(readOnly=true)
  public List<Car> findAll() {
```

```
    return Lists.newArrayList(carRepository.findAll());
  }

 @Override
  public Car save(Car car) {
    return carRepository.save(car);
  }

  @Override
  public void updateCarAgeJob() {
    List<Car> cars = findAll();

    DateTime currentDate = DateTime.now();
    logger.info("Car age update job started");

    cars.forEach(car -> {
      int age = Years.yearsBetween(
                    car.getManufactureDate(),
                    currentDate).getYears();
      car.setAge(age);
      save(car);
      logger.info("Car age update --> " + car);
    });

    logger.info("Car age update job completed "
              + "successfully");
    done = true;
  }

  @Override
  public boolean isDone() {
    return done;
  }
}
```

В приведенном выше коде предоставляются четыре метода, выполняющих следующие действия.

- Первый метод извлекает сведения обо всех автомашинах: `List<Car> findAll()`.
- Второй метод сохраняет обновленный объект типа `Car`: `Car save(Car car)`.
- Третий метод, `void updateCarAgeJob()`, представляет задание, которое требуется регулярно выполнять для обновления срока службы автомашины, исходя из даты ее изготовления и текущей даты эксплуатации.
- И четвертый метод, `boolean isDone()`, служит для того, чтобы выяснить, когда завершится задание, чтобы корректно закрыть приложение.

Как и в других поддерживаемых в Spring пространствах имен, в пространстве имен `task-namespace` предоставляется упрощенная конфигурация для календарного планирования заданий с помощью абстракции интерфейса `TaskScheduler` в

Spring. В приведенном ниже фрагменте кода из XML-файла `task-namespace-app-context.xml` демонстрируется часть конфигурации приложения Spring, содержащая запланированные задания. Как видите, пользоваться пространством имен `task-namespace` совсем не трудно.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:task="http://www.springframework.org
                        /schema/task"
       xmlns:xsi="http://www.w3.org/2001
                       /XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org
                                /schema/beans
          http://www.springframework.org/schema/beans
                /spring-beans.xsd
          http://www.springframework.org/schema/task
          http://www.springframework.org/schema/task
                /spring-task.xsd">

       <task:scheduler id="carScheduler" pool-size="10"/>

       <task:scheduled-tasks scheduler="carScheduler">
         <task:scheduled ref="carService"
            method="updateCarAgeJob" fixed-delay="10000"/>
         </task:scheduled-tasks>
</beans>
```

Когда в Spring обнаруживается дескриптор разметки `<task:scheduler>`, получается экземпляр класса `ThreadPoolTaskScheduler`, тогда как в атрибуте `pool-size` задается размер пула потоков исполнения, которым может пользоваться планировщик. В дескрипторе разметки `<task:scheduled-tasks>` планируется одно или больше заданий, причем в отдельном задании можно обращаться к компоненту Spring Bean (в данном случае — `carService`) и конкретному методу этого компонента (в данном случае — `updateCarAgeJob()`). А в атрибуте `fixed-delay` каркасу Spring предписывается получить экземпляр типа `PeriodicTrigger` в виде реализации интерфейса `TaskScheduler` в классе `Trigger`.

Конфигурацию планирования заданий можно сочетать с конфигурацией доступа к данным, объявив новый конфигурационный класс и импортировав обе конфигурации с помощью аннотации `@Import` для конфигурационных классов и аннотации `@Import Resource` для XML-файлов конфигурации, как показано ниже.

```
package com.apress.prospring5.ch11.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.context.annotation
          .ImportResource;
```

```
@Configuration
@Import({ DataServiceConfig.class })
@ImportResource(
       "classpath:spring/task-namespace-app-context.xml")
public class AppConfig {}
```

В приведенном ниже тестовом коде конфигурационный класс `AppConfig` служит для создания контекста типа `ApplicationContext`, чтобы проверить возможности Spring для планирования заданий.

```
package com.apress.prospring5.ch11;

import com.apress.prospring5.ch11.config.AppConfig;
import com.apress.prospring5.ch11.services.CarService;
import com.apress.prospring5.ch11.services.CarServiceImpl;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;

public class ScheduleTaskDemo {

  final static Logger logger = LoggerFactory.getLogger(
                               CarServiceImpl.class);

  public static void main(String... args)
         throws Exception {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               AppConfig.class);
    CarService carService = ctx.getBean("carService",
                                        CarService.class);

    while (!carService.isDone()) {
      logger.info("Waiting for scheduled job to end ...");
      Thread.sleep(250);
    }
    ctx.close();
  }
}
```

Выполнение приведенного выше тестового кода приводит к выводу на консоль результата из пакетного задания.

```
[main] INFO c.a.p.c.s.CarServiceImpl
     - Waiting for scheduled job to end ...
[carScheduler-1] INFO c.a.p.c.s.CarServiceImpl
     - Car age update job started
```

```
[carScheduler-1] INFO c.a.p.c.s.CarServiceImpl
     - Car age update --> {License: GRAVITY-0405,
       Manufacturer: Ford, Manufacture Date: 2006-09-12,
       Age: 10}
[carScheduler-1] INFO c.a.p.c.s.CarServiceImpl
    - Car age update --> {License: CLARITY-0432,
      Manufacturer: Toyota, Manufacture Date: 2003-09-09,
      Age: 13}
[carScheduler-1] INFO c.a.p.c.s.CarServiceImpl
     - Car age update --> {License: ROSIE-0402,
       Manufacturer: Toyota, Manufacture Date: 2017-04-16,
       Age: 0}
[carScheduler-1] INFO c.a.p.c.s.CarServiceImpl
     - Car age update job completed successfully
```

В данном примере выполнение приложения было остановлено лишь после однократного выполнения запланированного задания. Как было объявлено в атрибуте `fixed-delay="10000"`, задание должно выполняться через каждые 10 секунд. Но повторное выполнение приложения можно разрешить до тех пор, пока пользователь не нажмет любую клавишу. С этой целью в класс `ScheduleTaskDemo` следует ввести коррективы, как показано ниже.

```
package com.apress.prospring5.ch11;

import com.apress.prospring5.ch11.config.AppConfig;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;

public class ScheduleTaskDemo {
  public static void main(String... args)
         throws Exception {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               AppConfig.class);
    System.in.read();
    ctx.close();
  }
}
```

Как следует из выводимого на консоль результата, свойства `age` объектов, представляющих автомашины, обновляются. Помимо фиксированного промежутка времени, имеется и более гибкий механизм планирования заданий, который состоит в применении выражений в формате `cron`. Чтобы воспользоваться этим механизмом, достаточно заменить в XML-файле конфигурации следующую строку кода:

```
<task:scheduled ref="carService" method="updateCarAgeJob"
       fixed-delay="10000"/>
```

на такую строку кода:

```
<task:scheduled ref="carService" method="updateCarAgeJob"
      cron="0 * * * * *"/>
```

Если после этой замены снова выполнить исходный код тестового класса `ScheduleTaskDemo` и дать приложению возможность выполняться дольше одной минуты, то можно обнаружить, что запланированное задание будет выполняться каждую минуту.

Планирование заданий с помощью аннотаций

Еще один способ планировать задания с помощью абстракции интерфейса `Task Scheduler` в Spring состоит в применении аннотаций. Для этой цели в Spring предусмотрена аннотация `@Scheduled`. Чтобы активизировать поддержку аннотаций для планирования заданий, необходимо ввести дескриптор разметки `<task:annotation-driven>` в XML-файл конфигурации приложения Spring. А с другой стороны, можно воспользоваться конфигурационным классом, снабдив его аннотацией `@Enable Scheduling`. Воспользуемся именно этим способом, чтобы полностью избавиться от необходимости конфигурировать в формате XML.

```
package com.apress.prospring5.ch11.config;

import org.springframework.context.annotation
          .Configuration;
import org.springframework.context.annotation
          .Import;
import org.springframework.scheduling.annotation
          .EnableScheduling;

@Configuration
@Import({DataServiceConfig.class})
@EnableScheduling
public class AppConfig {
}
```

Вот, собственно, и все, что требуется сделать. Не нужно даже объявлять сам планировщик, поскольку Spring сделает это автоматически. Аннотация `@Enable Scheduling`, применяемая в конфигурационном классе, активизирует режим обнаружения аннотаций `@Scheduled` в любых компонентах Spring Beans, находящихся в контейнере инверсии управления, или же в их методах. Но самое замечательное, что методы, снабженные аннотацией `@Scheduled`, могут быть даже объявлены непосредственно в конфигурационных классах. Эта аннотация предписывает каркасу Spring искать определение связанного с ней планировщика в особом компоненте Spring Bean типа `TaskScheduler`, находящемся в данном контексте, в компоненте Spring Bean типа `TaskScheduler` под названием `taskScheduler` или же в компоненте Spring Bean типа `ScheduledExecutorService`. Если ни один из этих компо-

нентов Spring Beans не удастся обнаружить, то будет создан стандартный однопоточный планировщик для применения в регистраторе.

Чтобы запланировать выполнение конкретного метода из компонента Spring Bean, этот метод необходимо снабдить аннотацией `@Scheduled` и передать ему требования к планированию. В качестве примера ниже приведен вариант класса `CarService Impl`, расширенный для объявления нового компонента Spring Bean с запланированным методом, переопределяющим метод `updateCarAgeJob()` из родительского класса для того, чтобы употребить в нем аннотацию `@Scheduled`.

```
package com.apress.prospring5.ch11.services;

import com.apress.prospring5.ch11.entities.Car;
import org.joda.time.DateTime;
import org.joda.time.Years;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.scheduling.annotation
          .Scheduled;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation
      .Transactional;
import java.util.List;

@Service("scheduledCarService")
@Repository
@Transactional
public class ScheduledCarServiceImpl
       extends CarServiceImpl {

  @Override
  @Scheduled(fixedDelay=10000)
  public void updateCarAgeJob() {
    List<Car> cars = findAll();

    DateTime currentDate = DateTime.now();

    logger.info("Car age update job started");

    cars.forEach(car -> {
      int age = Years.yearsBetween(
                car.getManufactureDate(),
                currentDate).getYears();
      car.setAge(age);
      save(car);
      logger.info("Car age update --> " + car);
    });
```

```
    logger.info("Car age update job completed "
              + "successfully");
  }
}
```

Ниже приведен исходный код соответствующей тестовой программы.

```
package com.apress.prospring5.ch11;

import com.apress.prospring5.ch11.config.AppConfig;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;

public class ScheduleTaskAnnotationDemo {
  public static void main(String... args)
         throws Exception {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               AppConfig.class);
    System.in.read();
    ctx.close();
  }
}
```

Как показано ниже, выполнение этой тестовой программы дает почти такой же результат, как и в предыдущем примере, где применялось пространство имен `task-namespace`. Можете опробовать другой механизм запуска заданий на выполнение, изменив атрибут в аннотации `@Scheduled` (т.е. `fixedDelay`, `fixedRate`, `cron`) и самостоятельно проверив его.

```
[main] DEBUG o.s.s.a.ScheduledAnnotationBeanPostProcessor
     - Could not find default TaskScheduler bean
org.springframework.beans.factory
   .NoSuchBeanDefinitionException:
No qualifying bean of type
   'org.springframework.scheduling.TaskScheduler' available
... // здесь следует дополнительная трассировка стека
[main] DEBUG o.s.s.a.ScheduledAnnotationBeanPostProcessor
     - Could not find default ScheduledExecutorService bean
org.springframework.beans.factory.NoSuchBeanDefinitionException:
No qualifying bean of type
   'java.util.concurrent.ScheduledExecutorService' available
... // здесь следует дополнительная трассировка стека
[pool-1-thread-1] INFO c.a.p.c.s.CarServiceImpl
     - Car age update job started
[pool-1-thread-1] INFO c.a.p.c.s.CarServiceImpl
     - Car age update --> {License:
       GRAVITY-0405, Manufacturer: Ford,
```

```
      Manufacture Date: 2006-09-12, Age: 10}
[pool-1-thread-1] INFO c.a.p.c.s.CarServiceImpl
     - Car age update --> {License:
       CLARITY-0432, Manufacturer: Toyota,
       Manufacture Date: 2003-09-09, Age: 13}
[pool-1-thread-1] INFO c.a.p.c.s.CarServiceImpl
     - Car age update --> {License: ROSIE-0402,
       Manufacturer: Toyota, Manufacture Date: 2017-04-16,
       Age: 0}
[pool-1-thread-1] INFO c.a.p.c.s.CarServiceImpl
     - Car age update job completed successfully
```

При желании можно и самостоятельно определить компонент Spring Bean типа `TaskScheduler`. В следующем примере кода объявляется компонент Spring Bean типа `ThreadPoolTaskScheduler`, равнозначный компоненту, объявленному в конфигурации формата XML, составленной в примере из предыдущего раздела:

```
package com.apress.prospring5.ch11.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.context.annotation
          .Import;
import org.springframework.scheduling.TaskScheduler;
import org.springframework.scheduling.annotation
          .EnableScheduling;
import org.springframework.scheduling.concurrent
          .DefaultManagedTaskScheduler;
import org.springframework.scheduling.concurrent
          .ThreadPoolTaskScheduler;

@Configuration
@Import({DataServiceConfig.class})
@EnableScheduling
public class AppConfig {

  @Bean TaskScheduler carScheduler() {
    ThreadPoolTaskScheduler carScheduler =
          new ThreadPoolTaskScheduler();
    carScheduler.setPoolSize(10);
    return carScheduler;
  }
}
```

Если протестировать исходный код из данного примера, то можно обнаружить, что исключения больше не выводятся в журнал регистрации, а имя планировщика, выполняющего метод, изменилось, поскольку теперь компонент Spring Bean типа `TaskScheduler` называется `carScheduler`.

```
[carScheduler-1] INFO c.a.p.c.s.CarServiceImpl
     - Car age update job started
[carScheduler-1] INFO c.a.p.c.s.CarServiceImpl
     - Car age update --> {License: GRAVITY-0405,
        Manufacturer: Ford, Manufacture Date: 2006-09-12,
        Age: 10}
[carScheduler-1] INFO c.a.p.c.s.CarServiceImpl
     - Car age update --> {License: CLARITY-0432,
       Manufacturer: Toyota, Manufacture Date: 2003-09-09,
       Age: 13}
[carScheduler-1] INFO c.a.p.c.s.CarServiceImpl
     - Car age update --> {License:ROSIE-0402,
       Manufacturer: Toyota, Manufacture Date: 2017-04-16,
       Age: 0}
[carScheduler-1] INFO c.a.p.c.s.CarServiceImpl
     - Car age update job completed successfully
```

Асинхронное выполнение заданий в Spring

Начиная с версии 3.0, в Spring поддерживается также применение аннотаций для асинхронного выполнения заданий. Для этого достаточно снабдить конкретный метод аннотацией `@Async`, как показано ниже.

```
package com.apress.prospring5.ch11;

import java.util.concurrent.Future;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.scheduling.annotation.Async;
import org.springframework.scheduling.annotation
          .AsyncResult;
import org.springframework.stereotype.Service;

@Service("asyncService")
public class AsyncServiceImpl implements AsyncService {
  final Logger logger =
        LoggerFactory.getLogger(AsyncServiceImpl.class);

  @Async
  @Override
  public void asyncTask() {
    logger.info("Start execution of async. task");
    try {
      Thread.sleep(10000);
    } catch (Exception ex) {
      logger.error("Task Interruption", ex);
    }
    logger.info("Complete execution of async. task");
  }
```

```
  @Async
  @Override
  public Future<String> asyncWithReturn(String name) {
    logger.info("Start execution of async. task with "
              + "return for "+ name);

    try {
      Thread.sleep(5000);
    } catch (Exception ex) {
      logger.error("Task Interruption", ex);
    }

    logger.info("Complete execution of async. task with "
              + "return for " + name);
    return new AsyncResult<>("Hello: " + name);
  }
}
```

В интерфейсе `AsyncService` определяются два метода. В частности, метод `asyncTask()` асинхронно выполняет простое задание для записи информации в журнал регистрации, а метод `asyncWithReturn()` принимает аргумент типа `String` и возвращает экземпляр типа `java.util.concurrent.Future<V>`. По завершении метода `asyncWithReturn()` результат сохраняется в экземпляре класса `org.springframework.scheduling.annotation.AsyncResult<V>`, реализующего интерфейс `Future<V>`. Таким образом, результат выполнения может быть извлечен в вызывающем коде не сразу, а впоследствии.

Аннотация `@Async` выбирается в результате активизации асинхронного режима выполнения методов в Spring. С этой целью конфигурационный класс Java снабжается аннотацией `@EnableAsync`, как показано ниже[2].

```
package com.apress.prospring5.ch11.config;

import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.context.annotation
          .ImportResource;
import org.springframework.scheduling.annotation
          .EnableAsync;
import org.springframework.scheduling.annotation
          .EnableScheduling;

@Configuration
```

[2] При конфигурировании в формате XML для этого достаточно сделать соответствующее объявление в дескрипторе разметки `<task:annotation-driven/>`.

```
@EnableAsync
@ComponentScan(basePackages =
              {"com.apress.prospring5.ch11"} )
public class AppConfig {
}
```

Ниже приведена соответствующая тестовая программа.

```
package com.apress.prospring5.ch11;

import java.util.concurrent.Future;
import com.apress.prospring5.ch11.config.AppConfig;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context
          .support.GenericApplicationContext;

public class AsyncTaskDemo {
  private static Logger logger =
          LoggerFactory.getLogger(AsyncTaskDemo.class);

  public static void main(String... args)
         throws Exception {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               AppConfig.class);

    AsyncService asyncService = ctx.getBean("asyncService",
                                AsyncService.class);

    for (int i = 0; i < 5; i++) {
      asyncService.asyncTask();
    }

    Future<String> result1 =
           asyncService.asyncWithReturn("John Mayer");
    Future<String> result2 =
           asyncService.asyncWithReturn("Eric Clapton");
    Future<String> result3 =
           asyncService.asyncWithReturn("BB King");
    Thread.sleep(6000);

    logger.info("Result1: " + result1.get());
    logger.info("Result2: " + result2.get());
    logger.info("Result3: " + result3.get());
    System.in.read();
    ctx.close();
  }
}
```

В приведенной выше тестовой программе метод `asyncTask()` вызывается пять раз, а метод `asyncWithReturn()` — три раза с разными аргументами. Полученный результат извлекается по истечении шести секунд. Выполнение этой тестовой программы дает следующий результат:

```
...
17:55:31.851 [SimpleAsyncTaskExecutor-1] INFO
   c.a.p.c.AsyncServiceImpl - Start execution of async. task[3]
17:55:31.851 [SimpleAsyncTaskExecutor-2] INFO
   c.a.p.c.AsyncServiceImpl - Start execution of async. task
17:55:31.851 [SimpleAsyncTaskExecutor-3] INFO
   c.a.p.c.AsyncServiceImpl - Start execution of async. task
17:55:31.851 [SimpleAsyncTaskExecutor-4] INFO
   c.a.p.c.AsyncServiceImpl - Start execution of async. task
17:55:31.852 [SimpleAsyncTaskExecutor-5] INFO
   c.a.p.c.AsyncServiceImpl - Start execution of async. task
17:55:31.852 [SimpleAsyncTaskExecutor-6] INFO
   c.a.p.c.AsyncServiceImpl - Start execution of async. task
   with return for John Mayer[4]
17:55:31.852 [SimpleAsyncTaskExecutor-7] INFO
   c.a.p.c.AsyncServiceImpl - Start execution of async. task
   with return for Eric Clapton[5]
17:55:31.852 [SimpleAsyncTaskExecutor-8] INFO
   c.a.p.c.AsyncServiceImpl - Start execution of async. task
   with return for BB King
17:55:36.856 [SimpleAsyncTaskExecutor-8] INFO
   c.a.p.c.AsyncServiceImpl - Complete execution of async. task
   with return for BB King
17:55:36.856 [SimpleAsyncTaskExecutor-6] INFO
   c.a.p.c.AsyncServiceImpl - Complete execution of async. task
   with return for John Mayer
17:55:36.856 [SimpleAsyncTaskExecutor-7] INFO
   c.a.p.c.AsyncServiceImpl - Complete execution of async. task
   with return for Eric Clapton
17:55:37.852 [main] INFO c.a.p.c.AsyncTaskDemo
   - Result1: Hello: John Mayer
17:55:37.853 [main] INFO c.a.p.c.AsyncTaskDemo
   - Result2: Hello: Eric Clapton
17:55:37.853 [main] INFO c.a.p.c.AsyncTaskDemo
   - Result3: Hello: BB King
17:55:41.852 [SimpleAsyncTaskExecutor-1] INFO
   c.a.p.c.AsyncServiceImpl - Complete execution of async. task[6]
17:55:41.852 [SimpleAsyncTaskExecutor-4] INFO
```

---

[3] Начало выполнения асинхронной задачи

[4] Начало выполнения асинхронной задачи с возвратом Ф.И.О. певца

[5] Завершение выполнения асинхронной задачи с возвратом Ф.И.О. певца

[6] Завершение выполнения асинхронной задачи

```
   c.a.p.c.AsyncServiceImpl - Complete execution of async. task
17:55:41.852 [SimpleAsyncTaskExecutor-3] INFO
   c.a.p.c.AsyncServiceImpl - Complete execution of async. task
17:55:41.852 [SimpleAsyncTaskExecutor-5] INFO
   c.a.p.c.AsyncServiceImpl - Complete execution of async. task
17:55:41.852 [SimpleAsyncTaskExecutor-2] INFO
   c.a.p.c.AsyncServiceImpl - Complete execution of async. task
```

Как следует из приведенного выше результата, все вызовы начались в одно и то же время. Сначала завершились три вызова метода `asyncWithReturn()`, а возвращаемые значения были выведены на консоль. А после этого завершились пять вызовов метода `asyncTask()`.

Выполнение заданий в Spring

Начиная с версии 2.0, для выполнения заданий в Spring предоставляется абстракция интерфейса `TaskExecutor`, который делает именно то, что и подразумевает его название, выполняя задание, представленное реализацией интерфейса `Runnable` на языке Java. Для разных потребностей в каркасе Spring предлагается целый ряд готовых реализаций интерфейса `TaskExecutor`. С полным перечнем реализаций интерфейса `TaskExecutor` можно ознакомиться по адресу `http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/core/task/TaskExecutor.html`.

Ниже перечислены наиболее употребительные реализации интерфейса `TaskExecutor`.

- Класс `SimpleAsyncTaskExecutor`. Создает новые потоки исполнения при каждом вызове, а существующие потоки исполнения не используются повторно.
- Класс `SyncTaskExecutor`. Не выполняется асинхронным образом, а вызов происходит в вызывающем потоке исполнения.
- Класс `SimpleThreadPoolTaskExecutor`. Это подкласс, производный от класса `SimpleThreadPool` из библиотеки Quartz и применяемый в том случае, когда пул потоков исполнения требуется совместно использовать в компонентах, как входящих, так и не входящих в состав библиотеки Quartz.
- Класс `ThreadPoolTaskExecutor`. В этой реализации интерфейса `TaskExecutor` предоставляется возможность сконфигурировать экземпляр класса `ThreadPoolExecutor` через свойства компонента Spring Bean и сделать его доступным как реализацию интерфейса `TaskExecutor` в Spring.

Каждая реализация интерфейса `TaskExecutor` служит собственным целям, хотя соглашения об их вызовах одинаковы. Единственное отличие состоит в конфигурации, где определяется, какую именно реализацию интерфейса `TaskExecutor` требуется использовать, а также ее свойств, если таковые имеются. Рассмотрим простой пример, в котором на консоль выводится ряд сообщений. В качестве реализации ин-

терфейса TaskExecutor воспользуемся классом SimpleAsyncTaskExecutor. С этой целью создадим сначала класс компонента Spring Bean, содержащий логику выполнения задания:

```
package com.apress.prospring5.ch11;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.core.task.TaskExecutor;
import org.springframework.stereotype.Component;

@Component
public class TaskToExecute {
  private final Logger logger =
          LoggerFactory.getLogger(TaskToExecute.class);

  @Autowired
  private TaskExecutor taskExecutor;

  public void executeTask() {
    for(int i=0; i < 10; ++ i) {
      taskExecutor.execute(() ->
        logger.info("Hello from thread: "
                    + Thread.currentThread().getName()));
    }
  }
}
```

В приведенном выше классе реализуется обычный компонент Spring Bean, в который требуется внедрить зависимость от интерфейса TaskExecutor и определить метод executeTask(). В методе executeTask() вызывается метод execute(), предоставляемый в интерфейсе TaskExecutor, для чего создается новый экземпляр типа Runnable, содержащий логику, которую требуется выполнить для данного задания. Хотя это и не вполне очевидно, поскольку для получения экземпляра типа Runnable здесь применяется лямбда-выражение. Конфигурация в данном примере довольно проста и подобна конфигурации, представленной в примере из предыдущего раздела, как показано ниже. Следует лишь принять во внимание необходимость предоставить объявление компонента Spring Bean типа TaxExecutor, который требуется внедрить в компонент Spring Bean типа TaskToExecute.

```
package com.apress.prospring5.ch11.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
```

```
          .Configuration;
import org.springframework.core.task
          .SimpleAsyncTaskExecutor;
import org.springframework.core.task.TaskExecutor;
import org.springframework.scheduling.TaskScheduler;
import org.springframework.scheduling.annotation
          .EnableAsync;
import org.springframework.scheduling.concurrent
          .ThreadPoolTaskScheduler;

@Configuration
@EnableAsync
@ComponentScan(basePackages =
               {"com.apress.prospring5.ch11"})
public class AppConfig {
  @Bean TaskExecutor taskExecutor() {
    return new SimpleAsyncTaskExecutor();
  }
}
```

В приведенном выше конфигурационном классе объявляется простой компонент `taskExecutor` типа `SimpleAsyncTaskExecutor`. Этот компонент Spring Bean внедряется с помощью контейнера инверсии управления в Spring в компонент Spring Bean типа `TaskToExecute`. Для проверки исходного кода из данного примера можно воспользоваться следующей тестовой программой:

```
package com.apress.prospring5.ch11;

import com.apress.prospring5.ch11.config.AppConfig;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;

public class TaskExecutorDemo {

  public static void main(String... args)
         throws Exception {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               AppConfig.class);

    TaskToExecute taskToExecute =
                  ctx.getBean(TaskToExecute.class);
    taskToExecute.executeTask();

    System.in.read();
    ctx.close();
  }
}
```

Если выполнить исходный код из данного примера, то на консоль должен быть выведен результат, аналогичный следующему:

```
Hello from thread: SimpleAsyncTaskExecutor-1
Hello from thread: SimpleAsyncTaskExecutor-5
Hello from thread: SimpleAsyncTaskExecutor-3
Hello from thread: SimpleAsyncTaskExecutor-10
Hello from thread: SimpleAsyncTaskExecutor-8
Hello from thread: SimpleAsyncTaskExecutor-6
Hello from thread: SimpleAsyncTaskExecutor-2
Hello from thread: SimpleAsyncTaskExecutor-4
Hello from thread: SimpleAsyncTaskExecutor-9
Hello from thread: SimpleAsyncTaskExecutor-7
```

Как следует из приведенного выше результата, каждое задание (т.е. сообщение, выводимое на консоль) отображается по мере ее выполнения. Каждое сообщение выводится вместе с именем потока исполнения, которым по умолчанию является имя класса (`SimpleAsyncTaskExecutor`), а также номер потока исполнения.

Резюме

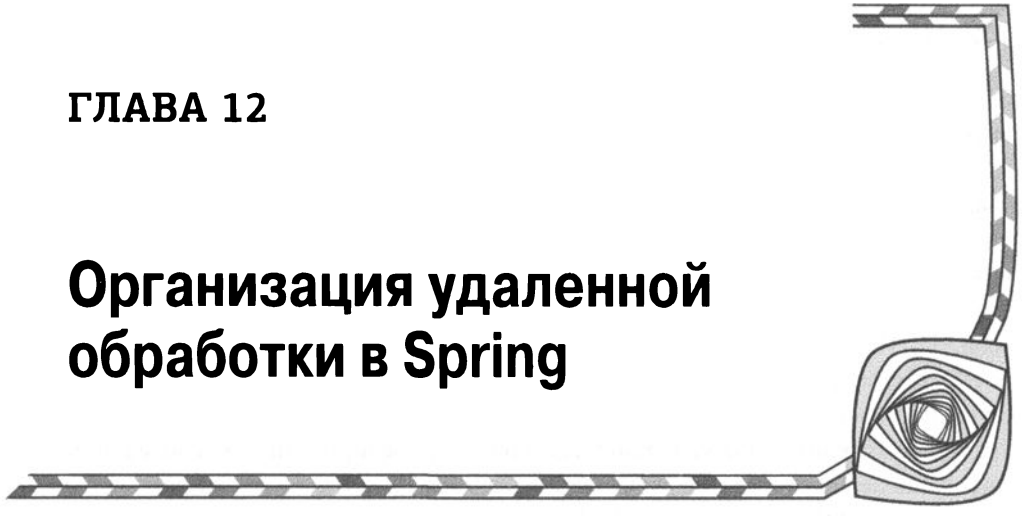
В этой главе была рассмотрена поддержка планирования заданий в Spring. Основное внимание в ней было уделено встроенной в Spring абстракции интерфейса `TaskScheduler` и продемонстрировано ее применение для удовлетворения потребностей планирования заданий на примере пакетного задания по обновлению данных. Было также показано, каким образом в Spring поддерживаются аннотации для асинхронного выполнения заданий. Кроме того, здесь был вкратце описан доступный в Spring интерфейс `TaskExecutor` и его распространенные реализации.

В этой главе не потребовался отдельный раздел, посвященный модулю Spring Boot, поскольку планирование и асинхронное выполнение заданий реализовано в библиотеке `spring-context`, а их аннотации следует употреблять в конфигурации Spring Boot. Кроме того, конфигурирование запланированных и асинхронных заданий нетрудно осуществить и в Spring. И модуль Spring Boot мало чем может помочь в усовершенствовании этой процедуры[7].

[7] Но если вам любопытно и требуется перенести свой проект в Spring Boot, небольшой учебный материал на эту тему вы можете найти по адресу `https://spring.io/guides/gs/scheduling-tasks/`.

ГЛАВА 12

Организация удаленной обработки в Spring



Корпоративное приложение обычно нуждается во взаимодействии с другими приложениями. Возьмем, к примеру, компанию, продающую товары. Когда пользователь размещает заказ, система обработки заказов обрабатывает его и формирует транзакцию. Во время обработки заказа делается запрос складской системы с целью проверить, доступен ли нужный товар на складе. После подтверждения заказа системе исполнения заказов отправляется уведомление о необходимости доставить товар заказчику. И, наконец, соответствующая информация посылается системе бухгалтерского учета, формируется счет-фактура и обрабатывается платеж.

Зачастую этот бизнес-процесс не удается реализовать в одном приложении, и поэтому для удовлетворения всех его требований совместно действует несколько приложений. Одни из них можно разработать собственными силами, а другие — приобрести у независимых поставщиков. Кроме того, приложения могут выполняться на отдельных, расположенных в разных местах машинах и реализовываться с применением различных технологий и языков программирования (например, Java, .NET и C++). Подтверждение установления связи между приложениями для организации эффективного бизнес-процесса всегда является критически важной задачей при построении архитектуры и реализации приложения. Такому приложению требуется поддержка удаленной обработки с помощью различных протоколов и технологий, чтобы оно могло принимать активное участие в корпоративной среде.

В среде Java поддержка удаленной обработки существует с момента зарождения этого языка. На ранней стадии его развития, начиная с версии Java 1.*x*, большинство требований к удаленной обработке были реализованы с помощью традиционных сокетов сетевого протокола TCP или RMI (Remote Method Invocation — удаленный вызов методов). С появлением платформы J2EE для организации взаимодействия

приложений на сервере обычно стали выбираться средства EJB и JMS. Быстрое развитие языка XML и Интернета привело к появлению удаленной поддержки средствами XML через сетевой протокол HTTP, включая прикладной интерфейс Java API для удаленных вызовов процедур в формате XML (спецификация JAX-RPC), прикладной интерфейс Java API для веб-служб XML (спецификация JAX-RPC) и технологий, основанных на сетевом протоколе HTTP (например, Hessian и Burlap). В каркасе Spring также предлагается своя поддержка удаленной обработки на основе сетевого протокола HTTP в виде механизма вызова Spring HTTP. Чтобы как-то отвечать бурному росту Интернета и требованиям к более оперативному реагированию веб-приложений на внешние воздействия (например, с помощью технологии Ajax), в последние годы крайне необходимой для успешной деятельности предприятий стала более простая и эффективная удаленная поддержка приложений. В итоге был создан прикладной интерфейс Java API для веб-служб REST (спецификация JAX-RS), который быстро нашел широкое применение. Многих разработчиков привлекают и другие протоколы вроде Comet и HTML5 WebSocket. Нечего и говорить, что технологии удаленной обработки продолжают развиваться быстрыми темпами.

Как пояснялось выше, в Spring предоставляется своя поддержка удаленной обработки (через механизм вызова Spring HTTP), а также многих упомянутых выше технологий (например, RMI, EJB, JMS, Hessian, Burlap[1], JAX-RPC, JAX-WS и JAX-RS). Описать все эти технологии удаленной обработки в одной главе просто невозможно, поэтому мы уделим внимание только наиболее употребительным технологиям. В этой главе будут, в частности, рассмотрены следующие вопросы.

- **Механизм вызова Spring HTTP.** Если приложения, которые должны взаимодействовать, основаны на Spring, то механизм вызова Spring HTTP предоставляет простой и эффективный способ обращения к службам, предоставляемым другими приложениями. В этой части показано, как пользоваться механизмом вызова Spring HTTP для доступа к подобной службе на его уровне обслуживания и как вызывать службы, предоставляемые удаленным приложением.
- **Применение службы JMS в Spring.** Служба обмена сообщениями в Java (Java Messaging Service — JMS) предоставляет еще один асинхронный и слабо связный способ обмена сообщениями между приложениями. В этой части поясняется, как Spring упрощает разработку приложений с помощью службы JMS.
- **Применение веб-служб REST в Spring.** Веб-службы REST, специально предназначенные для сетевого протокола HTTP, относятся к технологии, наиболее часто употребляемой для удаленной поддержки приложений, а также интерактивных пользовательских интерфейсов веб-приложений, в которых применяется технология Ajax. В этой части продемонстрирована всесторонняя поддержка доступа к веб-службам с помощью прикладного интерфейса JAX-RS в модуле Spring MVC, а также пояснено, как обращаться к веб-службам с помощью

[1] В выпущенной версии Spring 4 упоминалось, что дальнейшая активная разработка и поддержка технологии Burlap будет прекращена.

класса `RestTemplate`. Здесь будут также обсуждаться вопросы защиты веб-служб от несанкционированного доступа.

- **Применение протокола AMQP в Spring.** В родственном проекте Spring AMQP предоставляется типичная для Spring абстракция протокола AMQP (Advanced Message Queuing Protocol — усовершенствованный протокол организации очередей сообщений) наряду с реализацией системы обмена сообщениями RabbitMQ. В данном проекте предоставляется обширный ряд функциональных возможностей, но в этой главе основное внимание уделяется тем его функциональным возможностям, которые имеют непосредственное отношение к удаленной обработке через удаленный вызов процедур (RPC).

Модель выборочных данных для исходного кода примеров

В примерах из этой главы применяется простая модель выборочных данных, содержащая только таблицу `SINGER` для хранения информации. Эта таблица формируется библиотекой Hibernate на основании приведенного ниже класса `Singer`. Этот класс и его свойства снабжаются соответствующими стандартными аннотациями из прикладного интерфейса JPA.

```
package com.apress.prospring5.ch12.entities;

import javax.persistence.*;
import java.io.Serializable;
import java.util.Date;
import static javax.persistence.GenerationType.IDENTITY;

@Entity
@Table(name = "singer")
public class Singer implements Serializable {
  @Id
  @GeneratedValue(strategy = IDENTITY)
  @Column(name = "ID")
  private Long id;

  @Version
  @Column(name = "VERSION")
  private int version;

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

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

```
  @Temporal(TemporalType.DATE)
  @Column(name = "BIRTH_DATE")
  private Date birthDate;

  // Методы установки и получения
  ...
}
```

Чтобы заполнить таблицу, можно воспользоваться инициализирующим компонентом Spring Bean. Ниже приведен класс, реализующий этот компонент.

```
package com.apress.prospring5.ch12.services;

import com.apress.prospring5.ch12.entities.Singer;
import com.apress.prospring5.ch12.repos.SingerRepository;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.stereotype.Service;

import javax.annotation.PostConstruct;
import java.util.Date;
import java.util.GregorianCalendar;

@Service
public class DBInitializer {

  private Logger logger =
          LoggerFactory.getLogger(DBInitializer.class);
  @Autowired
  SingerRepository singerRepository;

  @PostConstruct
  public void initDB() {
    logger.info("Starting database initialization...");
    Singer singer = new Singer();
    singer.setFirstName("John");
    singer.setLastName("Mayer");
    singer.setBirthDate(new Date(
           (new GregorianCalendar(1977, 9, 16))
           .getTime().getTime()));
    singerRepository.save(singer);
    singer = new Singer();

    singer.setFirstName("Eric");
    singer.setLastName("Clapton");
    singer.setBirthDate(new Date(
           (new GregorianCalendar(1945, 2, 30)).getTime().getTime()));
```

```
    singerRepository.save(singer);
    singer = new Singer();
    singer.setFirstName("John");
    singer.setLastName("Butler");
    singer.setBirthDate(new Date(
           (new GregorianCalendar(1975, 3, 1))
           .getTime().getTime()));

    singerRepository.save(singer);
    logger.info("Database initialization finished.");
  }
}
```

Внедрение обязательных зависимостей для серверной части JPA

В рассматриваемый здесь проект должны быть внедрены обязательные зависимости. В приведенном ниже фрагменте кода конфигурации описаны зависимости, требующиеся для реализации уровня обслуживания средствами JPA 2 и Hibernate в качестве поставщика услуг сохраняемости. Кроме того, здесь будет использоваться проект Spring Data JPA.

```
// Файл конфигурации pro-spring-15/build.gradle
ext {
    // Библиотеки Spring
    springVersion = '5.0.0.RC1'
    springDataVersion = '2.0.0.M3'

    // Библиотеки протоколирования
    slf4jVersion = '1.7.25'
    logbackVersion = '1.2.3'

    junitVersion = '4.12'

    // Библиотека базы данных
    h2Version = '1.4.194'

    // Библиотеки сохраняемости
    hibernateVersion = '5.2.10.Final'
    hibernateJpaVersion = '1.0.0.Final'
    atomikosVersion = '4.0.0M4'

    spring = [
        context : "org.springframework:spring-context:
                   $springVersion",
        aop : "org.springframework:spring-aop:
               $springVersion",
```

```
        aspects : "org.springframework:spring-aspects:
                   $springVersion",
        tx : "org.springframework:spring-tx:
              $springVersion",
        jdbc : "org.springframework:spring-jdbc:
                $springVersion",
        contextSupport: "org.springframework:
                         spring-context-support:
                         $springVersion",
        orm : "org.springframework:spring-orm:
               $springVersion",
        data : "org.springframework.data:spring-data-jpa:
                $springDataVersion",
        test : "org.springframework:spring-test:
                $springVersion"
    ]

    hibernate = [
        ...
        em : "org.hibernate:hibernate-entitymanager:
              $hibernateVersion",
        tx : "com.atomikos:transactions-hibernate4:
              $atomikosVersion"
    ]

    testing = [
        junit: "junit:junit:$junitVersion"
    ]

    misc = [
        ...
        slf4jJcl : "org.slf4j:jcl-over-slf4j:$slf4jVersion",
        logback : "ch.qos.logback:logback-classic:
                   $logbackVersion",
        lang3 : "org.apache.commons:commons-lang3:3.5",
        guava : "com.google.guava:guava:$guavaVersion"
    ]

    db = [
        ...
        h2 : "com.h2database:h2:$h2Version"
    ]
}

// Файл конфигурации chapter12.gradle
dependencies {
    // Эти зависимости указываются для всех подмодулей,
    // кроме модуля начальной загрузки, который
    // определяется отдельно
```

```
    if (!project.name.contains("boot")) {
      // исключить транзитивные зависимости,
      // поскольку об этом позаботится модуль spring-data
      compile (spring.contextSupport) {
         exclude module: 'spring-context'
         exclude module: 'spring-beans'
         exclude module: 'spring-core'
      }
      // исключить транзитивную зависимость 'hibernate',
      // чтобы получить контроль над применяемой версией
      compile (hibernate.tx) {
        exclude group: 'org.hibernate', module: 'hibernate'
      }
      compile misc.slf4jJcl, misc.logback, misc.lang3,
              hibernate.em, misc.guava
    }
    testCompile testing.junit
}
```

Реализация и конфигурирование интерфейса `SingerService`

Внедрив упомянутые выше зависимости, можно приступить к реализации и конфигурированию уровня обслуживания для примеров из этой главы. В последующих разделах сначала обсуждается реализация интерфейса `SingerService` средствами JPA 2, Spring Data JPA и Hibernate в качестве поставщика услуг сохраняемости. А затем будет показано, как сконфигурировать уровень обслуживания в проекте Spring.

Реализация интерфейса `SingerService`

В примерах из этой главы будет показано, как получить доступ к службам для выполнения различных операций над сведениями о певцах на стороне удаленных клиентов. И для начала ниже приведено определение интерфейса `SingerService`.

```
package com.apress.prospring5.ch12.services;

import com.apress.prospring5.ch12.entities.Singer;
import java.util.List;

public interface SingerService {
  List<Singer> findAll();
  List<Singer> findByFirstName(String firstName);
  Singer findById(Long id);
  Singer save(Singer singer);
  void delete(Singer singer);
}
```

Методы из этого интерфейса самоочевидны. Но поскольку в данном случае будет использоваться поддержка информационного хранилища в проекте Spring Data JPA, то реализуем приведенный ниже интерфейс `SingerRepository`.

```
package com.apress.prospring5.ch12.repos;

import com.apress.prospring5.ch12.entities.Singer;
import org.springframework.data.repository.CrudRepository;
import java.util.List;

public interface SingerRepository
       extends CrudRepository<Singer, Long> {
  List<Singer> findByFirstName(String firstName);
}
```

Расширив интерфейс `CrudRepository<T,ID extends Serializable>` методами из интерфейса `SingerService`, остается лишь объявить вручную метод `findByFirstName()`. В приведенном ниже фрагменте кода демонстрируется класс, реализующий интерфейс `SingerService`. На этом реализация, по существу, завершается, а на следующей стадии предстоит сконфигурировать службу в контексте типа `ApplicationContext` конкретного веб-проекта, как поясняется в следующем разделе.

```
package com.apress.prospring5.ch12.services;

import com.apress.prospring5.ch12.entities.Singer;
import com.apress.prospring5.ch12.repos.SingerRepository;
import com.google.common.collect.Lists;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation
          .Transactional;
import java.util.List;

@Service("singerService")
@Transactional
public class SingerServiceImpl
       implements SingerService {

  @Autowired
  private SingerRepository singerRepository;

  @Override
  @Transactional(readOnly = true)
  public List<Singer> findAll() {
    return Lists.newArrayList(singerRepository.findAll());
  }
```

```
  @Override
  @Transactional(readOnly = true)
  public List<Singer> findByFirstName(String firstName) {
    return singerRepository.findByFirstName(firstName);
  }

  @Override
  @Transactional(readOnly = true)
  public Singer findById(Long id) {
    return singerRepository.findById(id).get();
  }

  @Override
  public Singer save(Singer singer) {
    return singerRepository.save(singer);
  }

  @Override
  public void delete(Singer singer) {
    singerRepository.delete(singer);
  }
}
```

Конфигурирование службы типа `SingerService`

Для доступа к данным и транзакциям можно воспользоваться простым конфигурационным классом Java, представленным ранее и еще раз приведенным ниже.

```
package com.apress.prospring5.ch12.config;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.data.jpa.repository.config
          .EnableJpaRepositories;
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseType;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa
          .LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor
          .HibernateJpaVendorAdapter;
```

```
import org.springframework.transaction
          .PlatformTransactionManager;

import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;
import java.util.Properties;

@Configuration
@EnableJpaRepositories(basePackages =
                    {"com.apress.prospring5.ch12.repos"})
@ComponentScan(basePackages = {"com.apress.prospring5.ch12"} )
public class DataServiceConfig {

  private static Logger logger =
          LoggerFactory.getLogger(DataServiceConfig.class);

  @Bean
  public DataSource dataSource() {
    try {
      EmbeddedDatabaseBuilder dbBuilder =
              new EmbeddedDatabaseBuilder();
      return dbBuilder.setType(EmbeddedDatabaseType.H2)
                    .build();
    } catch (Exception e) {
      logger.error("Embedded DataSource bean cannot "
                + "be created!", e);
      return null;
    }
  }

  @Bean
  public Properties hibernateProperties() {
    Properties hibernateProp = new Properties();
    hibernateProp.put("hibernate.dialect",
                    "org.hibernate.dialect.H2Dialect");
    hibernateProp.put("hibernate.hbm2ddl.auto",
                    "create-drop");
    hibernateProp.put("hibernate.show_sql", true);
    hibernateProp.put("hibernate.max_fetch_depth", 3);
    hibernateProp.put("hibernate.jdbc.batch_size", 10);
    hibernateProp.put("hibernate.jdbc.fetch_size", 50);
    return hibernateProp;
  }

  @Bean
  public PlatformTransactionManager transactionManager() {
    return new JpaTransactionManager(entityManagerFactory());
  }
```

```
  @Bean
  public JpaVendorAdapter jpaVendorAdapter() {
    return new HibernateJpaVendorAdapter();
  }

  @Bean
  public EntityManagerFactory entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean factoryBean =
         new LocalContainerEntityManagerFactoryBean();
    factoryBean.setPackagesToScan("com.apress.prospring5.ch12.entities");
    factoryBean.setDataSource(dataSource());
    factoryBean.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
    factoryBean.setJpaProperties(hibernateProperties());
    factoryBean.setJpaVendorAdapter(jpaVendorAdapter());
    factoryBean.afterPropertiesSet();
    return factoryBean.getNativeEntityManagerFactory();
  }
}
```

В данном случае организуется доступ к средству вызова HTTP Spring через модуль Spring MVC, и в связи с этим возникает потребность сконфигурировать веб-приложение. Это можно сделать, не прибегая к формату XML с помощью двух конфигурационных классов, как поясняется ниже.

- В одном из конфигурационных классов реализуется интерфейс `WebMvcConfigurer`. Он был внедрен в версии Spring 3.1, и в нем определяются методы обратного вызова, предназначенные для специальной настройки основанной на Java конфигурации модуля Spring MVC, активизируемого с помощью аннотации `@EnableWebMvc`. В данном случае нужно лишь организовать доступ к HTTP-службе, для чего веб-интерфейс не требуется, и поэтому можно вполне обойтись и пустой реализацией. Реализация данного интерфейса, снабжаемая аннотацией `@EnableWebMvc`, заменяет конфигурацию приложения Spring в формате XML с помощью пространства имен `mvc`. Более сложный пример будет подробно рассмотрен в главе 16.

```
package com.apress.prospring5.ch12.config;

import org.springframework.context.annotation
          .Configuration;
import org.springframework.web.servlet.config.annotation
          .EnableWebMvc;
import org.springframework.web.servlet.config.annotation
          .WebMvcConfigurer;

@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {
}
```

- В еще одном конфигурационном классе реализуется интерфейс `org.springframework.web.WebApplicationInitializer` или расширяется одна из готовых его реализаций в Spring. Этот интерфейс должен быть реализован в средах, начиная с версии Spring 3.0, чтобы программно сконфигурировать контекст типа `ServletContext`. Тем самым исключается необходимость предоставлять XML-файл `web.xml` для конфигурирования веб-приложения. В данном классе импортируется конфигурация доступа к данным и обработки транзакций, и на этой основе создается контекст корневого приложения. А контекст веб-приложения создается с помощью класса `WebConfig` и того конфигурационного класса, в котором определяется конфигурация механизма вызова Spring HTTP. И хотя эти классы могут быть объединены в один, применяя Spring, следует все же придерживаться более подходящей нормы практики, храня специальные службы и компоненты Spring Beans в разных классах.

```
package com.apress.prospring5.ch12.config;

import org.springframework.web.servlet.support.
AbstractAnnotationConfigDispatcherServletInitializer;

public class WebInitializer extends
     AbstractAnnotationConfigDispatcherServletInitializer {

  @Override
  protected Class<?>[] getRootConfigClasses() {
    return new Class<?>[] {
      DataServiceConfig.class
    };
  }

  @Override
  protected Class<?>[] getServletConfigClasses() {
    return new Class<?>[] {
      HttpInvokerConfig.class, WebConfig.class
    };
  }

  @Override
  protected String[] getServletMappings() {
    return new String[]{"/invoker/*"};
  }
}
```

Здесь речь идет о веб-приложении Spring MVC, поэтому необходимо создать файл формата WAR и развернуть контейнер сервлетов. Это можно сделать самыми разными способами, например, развернув автономный контейнер сервлетов Tomcat, запустив экземпляр контейнера сервлетов Tomcat в IDE или встроенный его экземпляр с помощью инструментального средства сборки проектов вроде Maven. Выбор кон-

кретного способа зависит от насущных потребностей, но в локальной среде разработки встроенный экземпляр контейнера сервлетов рекомендуется запускать из инструментального средства сборки проектов или непосредственно из IDE. В примерах исходного кода к этой книге используется контейнер сервлетов Tomcat Server версии 9.*x*, а в IDE Intellij IDEA устанавливается модуль запуска веб-приложений. Подробнее об этом см. в приложении к данной книге. Следуя этим рекомендациям, вы должны построить веб-приложение и развернуть его избранным вами способом. Если попытаться загрузить его по следующему URL: `http://localhost:8080/`, то в окне браузера появится такое сообщение:

```
Spring Remoting: Simplifying Development of
       Distributed Applications
RMI services over HTTP should be correctly exposed
    when this page is visible.
```

Это означает, веб-приложение развернуто правильно и теперь компонент `singerService` должен быть доступен по следующему URL:

```
http://localhost:8080/invoker/httpInvoker/singerService URL
```

Организация доступа к удаленной службе

Если приложение, с которым планируется взаимодействовать, также приводится в действие каркасом Spring, то удобно выбрать механизм вызова Spring HTTP. Этот механизм предоставляет удаленным клиентам очень простой способ доступа к удаленным службам в контексте типа `WebApplicationContext`. Процедуры для организации такого доступа к службам описаны в последующих разделах. А ниже приведен конфигурационный класс `HttpInvokerConfig`, содержащий компонент Spring Bean, предназначенный для организации доступа к механизму вызова Spring HTTP.

```
package com.apress.prospring5.ch12.config;

import com.apress.prospring5.ch12.services.SingerService;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.remoting.httpinvoker
          .HttpInvokerServiceExporter;

@Configuration
public class HttpInvokerConfig {

  @Autowired
  SingerService singerService;

  @Bean(name = "/httpInvoker/singerService")
```

```
  public HttpInvokerServiceExporter
        httpInvokerServiceExporter() {
    HttpInvokerServiceExporter invokerService =
              new HttpInvokerServiceExporter();
    invokerService.setService(singerService);
    invokerService.setServiceInterface(
                  SingerService.class);
    return invokerService;
  }
}
```

Компонент `httpInvokerServiceExporter` определяется с помощью класса `HttpInvokerServiceExporter`, предназначенного для экспорта любого компонента Spring Bean в качестве службы через механизм Spring HTTP. В самом компоненте Spring Bean определены два свойства. Первым из них является свойство `service`, обозначающее компонент Spring Bean, предоставляющий нужную службу. Для этого свойства внедрен компонент `singerService`. А вторым является свойство `service Interface`, обозначающее тип интерфейса, для которого организуется доступ, и в данном случае это интерфейс `apress.prospring5.ch12.serviced.Singer Service`. Итак, уровень обслуживания завершен и подготовлен к доступу для удаленных клиентов.

Вызов удаленной службы

Обращение к службе через механизм вызова Spring HTTP осуществляется очень просто. Сначала необходимо сконфигурировать контекст типа `ApplicationContext`, как демонстрируется в следующем конфигурационном классе:

```
package com.apress.prospring5.ch12.config;

import com.apress.prospring5.ch12.services.SingerService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.remoting.httpinvoker
          .HttpInvokerProxyFactoryBean;

@Configuration
public class RmiClientConfig {

  @Bean
  public SingerService singerService() {
    HttpInvokerProxyFactoryBean factoryBean =
              new HttpInvokerProxyFactoryBean();
    factoryBean.setServiceInterface(SingerService.class);
    factoryBean.setServiceUrl("http://localhost:8080/invoker"
                            + "/httpInvoker/singerService");
    factoryBean.afterPropertiesSet();
```

```
    return (SingerService) factoryBean.getObject();
  }
}
```

В приведенном выше коде для клиентской стороны объявляется компонент Spring Bean типа `HttpInvokerProxyFactoryBean` и устанавливаются два его свойства. В частности, свойство `serviceUrl` обозначает местоположение удаленной службы (в данном случае — по адресу `http://localhost:8080/invoker/httpInvoker/singerService`). А в свойстве `serviceInterface` задается интерфейс службы (в данном случае — `SingerService`). Если для клиента разрабатывается отдельный проект, то интерфейс `SingerService` и класс сущности `Singer`должны присутствовать в пути к классам этого клиентского приложения.

В приведенном ниже фрагменте кода демонстрируется тестовый класс для вызова удаленной службы. В этом классе применяется класс `RmiClientConfig` для создания тестового контекста. А класс `SpringRunner` служит псевдонимом для класса `SpringJUnit4ClassRunner`, требующегося для выполнения модульных тестов в контексте Spring. Подробнее об этом речь пойдет в главе 13.

```
package com.apress.prospring5.ch12;

import com.apress.prospring5.ch12.config.RmiClientConfig;
import com.apress.prospring5.ch12.entities.Singer;
import com.apress.prospring5.ch12.services.DBInitializer;
import com.apress.prospring5.ch12.services.SingerService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.test.context
          .ContextConfiguration;
import org.springframework.test.context.junit4
          .SpringRunner;
import java.util.List;
import static org.junit.Assert.assertEquals;

@ContextConfiguration(classes = RmiClientConfig.class)
@RunWith(SpringRunner.class)
public class RmiTests {
  private Logger logger =
          LoggerFactory.getLogger(RmiTests.class);

  @Autowired
  private SingerService singerService;

  @Test
  public void testRmiAll() {
```

```
    List<Singer> singers = singerService.findAll();
    assertEquals(3, singers.size());
    listSingers(singers);
  }

  @Test
  public void testRmiJohn() {
    List<Singer> singers =
         singerService.findByFirstName("John");
    assertEquals(2, singers.size());
    listSingers(singers);
  }

  private void listSingers(List<Singer> singers) {
    singers.forEach(s -> logger.info(s.toString()));
  }
}
```

Приведенный выше тестовый класс должен быть выполнен после развертывания веб-приложения. Определенные в нем тесты должны пройти, а компонент `singer Service` должен возвратить список экземпляров типа `Singer`. Ниже приведен предполагаемый результат, выводимый на консоль.

```
// testRmiAll
INFO c.a.p.c.RmiTests - Singer - Id: 1, First name: John,
     Last name: Mayer, Birthday: 1977-10-16
INFO c.a.p.c.RmiTests - Singer - Id: 2, First name: Eric,
     Last name: Clapton, Birthday: 1945-03-30
INFO c.a.p.c.RmiTests - Singer - Id: 3, First name: John,
     Last name: Butler, Birthday: 1975-04-01
// testRmiJohn - все певцы с именем firstName='John'
INFO c.a.p.c.RmiTests - Singer - Id: 1, First name: John,
     Last name: Mayer, Birthday: 1977-10-16
INFO c.a.p.c.RmiTests - Singer - Id: 3, First name: John,
     Last name: Butler, Birthday: 1975-04-01
```

Применение службы JMS в Spring

Применение промежуточного программного обеспечения, ориентированного на организацию очередей сообщений и обычно называемого сервером очереди сообщений (MQ), означает еще один распространенный способ поддержки взаимодействия приложений. Основное преимущество сервера очереди сообщений заключается в том, что он предоставляет асинхронный и слабо связный способ интеграции приложений. В среде Java стандартом для подключения к серверу MQ с целью отправить или получить сообщение является служба JMS. На сервере MQ поддерживается список очередей и тем, для которых приложения могут подключаться, отправлять и получать сообщения. Ниже кратко описаны отличия между очередью и темой.

- **Очередь.** Служит для поддержки модели двухточечного обмена сообщениями. Когда поставщик отправляет сообщение в очередь, сервер MQ сохраняет его в очереди и доставляет только одному потребителю при следующем его подключении.
- **Тема.** Служит для поддержки модели обмена сообщениями типа "издатель–подписчик". На сообщение в самой теме может подписываться любое количество клиентов. Когда сообщение поступает в заданную тему, сервер MQ доставляет его всем клиентам, подписавшимся на него. Эта модель особенно удобна в том случае, когда имеется несколько приложений, заинтересованных в получении одной и той же порции информации (например, ленты новостей).

В службе JMS поставщик подключается к серверу MQ и отправляет сообщение в очередь или в тему. Потребитель также подключается к серверу MQ и принимает из очереди или темы интересующие его сообщения. В версии JMS 1.1 был унифицирован прикладной интерфейс API, чтобы избавить поставщика и потребителя от необходимости обращаться к разным прикладным интерфейсам API при взаимодействии с очередями и темами. В этом разделе мы уделим основное внимание модели двухточечного обмена сообщениями через очереди, которая чаще всего применяется в корпоративной среде.

В версии 4.0 каркаса Spring Framework внедрена поддержка версии 2.0 службы JMS. Чтобы воспользоваться функциональными возможностями JMS 2.0, достаточно ввести имя архивного JAR-файла в путь к файлам, сохранив в то же время обратную совместимость с версиями JMS 1.*x*. А, начиная с версии 5.0 каркаса Spring Framework, версия 1.1 службы JMS больше не поддерживается, поэтому приведенные далее примеры уместны лишь для версий 2.*x* службы JMS.

На момент написания данной книги версия 2.0 службы JMS не поддерживалась в ActiveMQ, и поэтому в приведенных далее примерах будет использоваться HornetQ, где служба JMS поддерживается, начиная с версии 2.4.0.Final, как брокер сообщений и автономный сервер. Описание загрузки и установки HornetQ выходит за рамки данной книги, поэтому обращайтесь за справкой к документации, доступной по адресу `http://docs.jboss.org/hornetq/2.4.0.Final/docs/quickstart-guide/html/index.html`.[2]

Для целей рассматриваемого здесь примера потребуется ряд новых зависимостей. Ниже приведены необходимые для этой цели фрагменты конфигурации.

```
// Файл конфигурации pro-spring-15/build.gradle
ext {
    jmsVersion = '2.0'
```

[2] Если вы предпочитаете программные продукты Apache, то имейте в виду, что в Apache ActiveMQ Artemis реализована версия JMS 2.0 с неблокирующей архитектурой, обеспечивающая превосходную производительность. Подробнее об этом см. по адресу `https://activemq.apache.org/artemis/`. Там же приведены примеры кода с проектом, в котором применяется Artemis ActiveMQ.

```
    hornetqVersion = '2.4.0.Final'

    spring = [
        ...
        jms : "org.springframework:spring-jms:
               $springVersion"
    ]

    misc = [
        ...
        Hornetq : "org.hornetq:hornetq-jms-client:
                  $hornetqVersion"
    ]
    ...
}

// Файл конфигурации chapter12/jms-hornetq/build.gradle
dependencies {
    compile spring.jms, misc.jms
}
```

После установки сервера необходимо создать очередь в файле конфигурации HornetQ JMS. Этот файл находится в каталоге установки HornetQ по следующему пути: `config/stand-alone/non-clustered hornetq-jms.xml`. В него необходимо ввести определение очереди, как показано ниже.

```
<configuration xmlns="urn:hornetq"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:hornetq /schema/hornetq-jms.xsd">
  ...
  <queue name="prospring5">
    <entry name="/queue/prospring5"/>
  </queue>
</configuration>
```

А теперь следует запустить сервер HornetQ, выполнив сценарий `run.sh` (в зависимости от конкретной операционной системы), и убедиться, что этот сервер запускается без ошибок. Для этого достаточно проверить, что в журнале регистрации отсутствуют сообщения о возникших исключениях и присутствуют подчеркнутые ниже строки.

```
...
00:36:21,171 INFO [org.hornetq.core.server] HQ221035:
   Live Server Obtained live lock
00:36:21,841 INFO [org.hornetq.core.server] HQ221003:
   trying to deploy queue jms.queue.DLQ
00:36:21,852 INFO [org.hornetq.core.server] HQ221003:
   // очередь, сконфигурированная в предыдущем примере
```

```
   trying to deploy queue jms.queue.prospring5[3]
00:36:21,853 INFO [org.hornetq.core.server] HQ221003:
   trying to deploy queue jms.queue.ExpiryQueue
00:36:21,993 INFO [org.hornetq.core.server] HQ221020:
   Started Netty Acceptor version 4.0.13.Final
   localhost:5455
00:36:21,996 INFO [org.hornetq.core.server] HQ221020:
Started Netty Acceptor version 4.0.13.Final localhost:5445
00:36:21,997 INFO [org.hornetq.core.server] HQ221007:
   Server is now live[4]
```

Далее необходимо предоставить конфигурацию Spring для подключения к данному серверу и доступа к настроенной выше очереди `prospring5`. Обычно для этой цели служат два конфигурационных класса: один — для отправки сообщений, другой — для их приема. Но поскольку для конфигурирования службы JMS в Spring практически требуется немного компонентов Spring Beans, это можно сделать в одном конфигурационном классе, как показано ниже.

```
package com.apress.prospring5.ch12.config;

import org.hornetq.api.core.TransportConfiguration;
import org.hornetq.core.remoting.impl.netty
          .NettyConnectorFactory;
import org.hornetq.core.remoting.impl.netty
          .TransportConstants;
import org.hornetq.jms.client.HornetQJMSConnectionFactory;
import org.hornetq.jms.client.HornetQQueue;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.jms.annotation.EnableJms;
import org.springframework.jms.config
          .DefaultJmsListenerContainerFactory;
import org.springframework.jms.config
          .JmsListenerContainerFactory;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.listener
          .DefaultMessageListenerContainer;

import javax.jms.ConnectionFactory;
import java.util.HashMap;
import java.util.Map;

@Configuration
```

[3] Попытка развернуть очередь jms.queue.prospring5

[4] Теперь сервер активно действует.

```
@EnableJms
@ComponentScan("com.apress.prospring5.ch12")
public class AppConfig {

  @Bean HornetQQueue prospring5() {
    return new HornetQQueue("prospring5");
  }

  @Bean ConnectionFactory connectionFactory() {
    Map<String, Object> connDetails = new HashMap<>();
    connDetails.put(TransportConstants.HOST_PROP_NAME,
                    "127.0.0.1");
    connDetails.put(TransportConstants.PORT_PROP_NAME,
                    "5445");
    TransportConfiguration transportConfiguration =
      new TransportConfiguration(
       NettyConnectorFactory.class.getName(), connDetails);
    return new HornetQJMSConnectionFactory(false,
               transportConfiguration);
    }

  @Bean
  public JmsListenerContainerFactory
         <DefaultMessageListenerContainer>
         jmsListenerContainerFactory() {
    DefaultJmsListenerContainerFactory factory =
           new DefaultJmsListenerContainerFactory();
    factory.setConnectionFactory(connectionFactory());
    factory.setConcurrency("3-5");
    return factory;
  }

  @Bean JmsTemplate jmsTemplate() {
    JmsTemplate jmsTemplate =
                new JmsTemplate(connectionFactory());
    jmsTemplate.setDefaultDestination(prospring5());
    return jmsTemplate;
  }
}
```

Сначала в приведенном выше коде предоставляется реализация интерфейса `javax.jms.ConnectionFactory` в классе `HornetQJMSConnectionFactory` из библиотеки HornetQ Java, чтобы установить соединение с поставщиком услуг JMS. Затем в данном коде объявляется компонент Spring Bean типа `JmsListenerContainerFactory`, предназначенный для создания контейнеров приемников сообщений, которые обращаются к простым прикладным интерфейсами API клиентов службы JMS для получения через нее сообщений. А компонент Spring Bean типа `JmsTemplate` предназначен для отправки сообщений через службу JMS в очередь `prospring5`.

Для приема сообщений через службу JMS должен быть объявлен компонент приемника сообщений с указанием места их назначения (т.е. очереди prospring5) и фабрики контейнеров jmsListenerContainerFactory().

Реализация приемника сообщений через службу JMS в Spring

До версии Spring 4.1 для разработки приемника сообщений через службу JMS приходилось создавать класс, реализующий интерфейс javax.jms.MessageListener и его метод onMessage(). А в версии Spring 4.1 была внедрена аннотация @JmsListener, с помощью которой методы помечаются в компонентах Spring Beans как целевые для приемника сообщений JMS по указанному месту назначения (очереди или темы). В следующем фрагменте кода приведен класс SimpleMessageListener простого приемника сообщений через службу JMS вместе с объявлением компонента Spring Bean:

```
package com.apress.prospring5.ch12;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.TextMessage;

public class SimpleMessageListener {
  private static final Logger logger =
          LoggerFactory.getLogger(
                        SimpleMessageListener.class);

  @JmsListener(destination = "prospring5",
          containerFactory = "jmsListenerContainerFactory")
  public void onMessage(Message message) {
    TextMessage textMessage = (TextMessage) message;

    try {
      logger.info(">>> Received: "
                + textMessage.getText());
    } catch (JMSException ex) {
      logger.error("JMS error", ex);
    }
  }
}
```

При поступлении сообщения методу onMessage(), снабженному аннотацией @JmsListener, передается экземпляр интерфейса javax.jms.Message. В этом ме-

тоде полученное сообщение приводится к экземпляру интерфейса `javax.jms.TextMessage`, а затем с помощью метода `TextMessage.getText()` извлекается тело сообщения. Список возможных форматов сообщений приведен в оперативно доступной документации на платформу JEE.

Обработка аннотации `@JmsListener` выполняется с помощью аннотации `@EnableJms` в конфигурационном классе или равнозначного ей элемента разметки `<jms:annotation-driven/>` в формате XML. А теперь выясним, каким образом сообщение посылается в очередь `propring5`.

Отправка сообщений через службу JMS в Spring

В этом разделе поясняется, каким образом сообщения отправляются через службу JMS в Spring. Для этой цели воспользуемся удобным классом `org.springframework.jms.core.JmsTemplate`. Прежде всего разработаем интерфейс `MessageSender` и реализующий его класс `SimpleMessageSender`:

```
// Исходный файл MessageSender.java
package com.apress.prospring5.ch12;

public interface MessageSender {
  void sendMessage(String message);
}

// Исходный файл SimpleMessageSender.java
package com.apress.prospring5.ch12;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;
import javax.jms.TextMessage;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;
import org.springframework.stereotype.Component;

@Component("messageSender")
public class SimpleMessageSender
       implements MessageSender {
  private static final Logger logger =
      LoggerFactory.getLogger(SimpleMessageSender.class);
  @Autowired
  private JmsTemplate jmsTemplate;

  @Override
  public void sendMessage(final String message) {
```

```
    jmsTemplate.setDeliveryDelay(5000L);
    this.jmsTemplate.send(new MessageCreator() {
      @Override
      public Message createMessage(Session session)
            throws JMSException {
        TextMessage jmsMessage =
                    session.createTextMessage(message);
        logger.info(">>> Sending: " + jmsMessage.getText());
        return jmsMessage;
      }
    });
  }
}
```

Как видите, в приведенном выше коде внедрен экземпляр типа `JmsTemplate`. В методе `sendMessage()` вызывается метод `JmsTemplate.send()` и сразу же конструируется экземпляр интерфейса `org.springframework.jms.core.MessageCreator`. А в экземпляре типа `MessageCreator` реализуется метод `createMessage()`, создающий новый экземпляр типа `TextMessage` с текстовым сообщением для последующей отправки серверу HornetQ. Объявления приемника и отправителя сообщений выбираются при просмотре компонентов.

Теперь свяжем вместе отправку и получение сообщений, чтобы посмотреть службу JMS в действии. Ниже приведена тестовая программа, в которой проверяется как отправка, так и получение сообщений.

```
package com.apress.prospring5.ch12;

import com.apress.prospring5.ch12.config.AppConfig;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;
import org.springframework.context.support
          .GenericXmlApplicationContext;
import java.util.Arrays;

public class JmsHornetQSample {
  public static void main(String... args)
         throws Exception {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               AppConfig.class);

    MessageSender messageSender = ctx.getBean(
                  "messageSender", MessageSender.class);

    for(int i=0; i < 10; ++i) {
      messageSender.sendMessage("Test message: " + i);
    }
```

```
    System.in.read();
    ctx.close();
  }
}
```

Эта тестовая программа очень проста. После ее запуска на выполнение сообщения отправляются в очередь. Приемник, реализуемый в классе `SimpleMessageListener`, получает эти сообщения, а на консоль выводится следующий результат:

```
INFO c.a.p.c.SimpleMessageSender -
     >>> Sending: Test message: 0
INFO c.a.p.c.SimpleMessageSender -
     >>> Sending: Test message: 1
INFO c.a.p.c.SimpleMessageSender -
     >>> Sending: Test message: 2
INFO c.a.p.c.SimpleMessageSender -
     >>> Sending: Test message: 3
INFO c.a.p.c.SimpleMessageSender -
     >>> Sending: Test message: 4
INFO c.a.p.c.SimpleMessageSender -
     >>> Sending: Test message: 5
INFO c.a.p.c.SimpleMessageSender -
     >>> Sending: Test message: 6
INFO c.a.p.c.SimpleMessageSender -
     >>> Sending: Test message: 7
INFO c.a.p.c.SimpleMessageSender -
     >>> Sending: Test message: 8
INFO c.a.p.c.SimpleMessageSender -
     >>> Sending: Test message: 9
INFO c.a.p.c.SimpleMessageListener -
     >>> Received: Test message: 0
INFO c.a.p.c.SimpleMessageListener -
     >>> Received: Test message: 1
INFO c.a.p.c.SimpleMessageListener -
     >>> Received: Test message: 2
INFO c.a.p.c.SimpleMessageListener -
     >>> Received: Test message: 3
INFO c.a.p.c.SimpleMessageListener -
     >>> Received: Test message: 4
INFO c.a.p.c.SimpleMessageListener -
     >>> Received: Test message: 5
INFO c.a.p.c.SimpleMessageListener -
     >>> Received: Test message: 6
INFO c.a.p.c.SimpleMessageListener -
     >>> Received: Test message: 7
INFO c.a.p.c.SimpleMessageListener -
     >>> Received: Test message: 8
INFO c.a.p.c.SimpleMessageListener -
     >>> Received: Test message: 9
```

Запуск Artemis средствами Spring Boot

О возможности использовать модуль Spring Boot, чтобы упростить разработку приложений службы JMS, вкратце упоминалось в главе 9, где был приведен пример обработки распределенных транзакций с привлечением очереди и базы данных. С помощью модуля Spring Boot можно автоматически сконфигурировать компонент Spring Bean типа `javax.jms.ConnectionFactory` при обнаружении брокера сообщений Artemis в пути к классам приложения. Встроенный в JMS брокер сообщений запускается и конфигурируется автоматически. Брокер сообщений Artemis можно использовать в самых разных режимах работы, а конфигурировать — с помощью специальных его свойств, устанавливаемых в файле свойств `application.properties`.

Брокер сообщений Artemis можно использовать в собственном режиме `native`, причем подключение к нему обеспечивается протоколом `netty`. В таком случае файл свойств `application.properties` может выглядеть следующим образом:

```
spring.artemis.mode=native
spring.artemis.host=0.0.0.0
spring.artemis.port=61617
spring.artemis.user=prospring5
spring.artemis.password=prospring5
```

Чтобы разрабатывать приложения для службы JMS с помощью модуля Spring Boot и брокера сообщений Artemis, проще всего воспользоваться встроенным сервером. Для этого достаточно указать имя очереди, в которой предстоит хранить сообщения. Следовательно, файл свойств `application.properties` будет выглядеть следующим образом:

```
spring.artemis.mode=embedded
spring.artemis.embedded.queues=prospring5
```

Именно такой способ и применяется в примерах кода из этого раздела, поскольку он требует минимальной настройки конфигурации. И для этого достаточно создать упомянутый выше файл свойств `application.properties` и приведенный ниже класс `Application`.

```
package com.apress.prospring5.ch12;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure
          .SpringBootApplication;
import org.springframework.boot.autoconfigure.
jms.DefaultJmsListenerContainerFactoryConfigurer;
import org.springframework.context
          .ConfigurableApplicationContext;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.jms.annotation.JmsListener;
import org.springframework.jms.config
        .DefaultJmsListenerContainerFactory;
import org.springframework.jms.config
        .JmsListenerContainerFactory;
import org.springframework.jms.core.JmsTemplate;

import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.TextMessage;

@SpringBootApplication
public class Application {

  private static Logger logger =
          LoggerFactory.getLogger(Application.class);

  @Bean
  public JmsListenerContainerFactory<?> connectionFactory(
         ConnectionFactory connectionFactory,
         DefaultJmsListenerContainerFactoryConfigurer
         configurer) {
    DefaultJmsListenerContainerFactory factory =
           new DefaultJmsListenerContainerFactory();
    configurer.configure(factory, connectionFactory);
    return factory;
  }

  public static void main(String... args)
         throws Exception {
    ConfigurableApplicationContext ctx =
                SpringApplication.run(
                      Application.class, args);
    JmsTemplate jmsTemplate =
                ctx.getBean(JmsTemplate.class);
    jmsTemplate.setDeliveryDelay(5000L);
    for (int i = 0; i < 10; ++i) {
      logger.info(">>> Sending: Test message: " + i);
      jmsTemplate.convertAndSend("prospring5",
                                 "Test message: " + i);
}
```

Безусловно, чтобы привести приведенный выше прикладной код в действие, придется воспользоваться стартовой библиотекой Spring Boot JMS в качестве зависимости, а также указать сервер Artemis в пути к классам. Ниже приведена соответствующая конфигурация Gradle.

```
// Файл конфигурации pro-spring-15/build.gradle
ext {
    bootVersion = '2.0.0.M1'
    artemisVersion = '2.1.0'

    boot = [
        ...
        starterJms : "org.springframework.boot:
               spring-boot-starter-artemis:$bootVersion"
    ]

    testing = [
        junit: "junit:junit:$junitVersion"
    ]

    misc = [
        ...
        artemisServer : "org.apache.activemq:
               artemis-jms-server:$artemisVersion"
    ]
    ...
}

// Файл конфигурации chapre12/boot-jms/build.gradle
buildscript {
    repositories {
        ...
    }
    dependencies {
        classpath boot.springBootPlugin
    }
  }
  apply plugin: 'org.springframework.boot'

  dependencies {
    compile boot.starterJms, misc.artemisServer
  }
}
```

Если объявить библиотеку `spring-boot-starter-artemis` в качестве зависимости, то для обработки методов, снабженных аннотацией `@JmsListener`, можно уже не пользоваться аннотацией `@EnableJms`. Компонент `jmsTemplate` создается средствами Spring Boot со стандартной конфигурацией, предоставляемой свойствами, устанавливаемыми в файле свойств `application.properties`. Этот компонент может быть использован не только для отправки, но и для приема сообщений с помощью метода `receive()`. И хотя это делается синхронно, тем не означает блокировку компонента `jmsTemplate`. Именно поэтому конфигурируемый вручную компонент Spring Bean типа `JmsListenerContainerFactory` применяется для

создания стандартного контейнера приемников сообщений типа `DefaultMessageListenerContainer`, который будет асинхронно принимать сообщения, максимально эффективно используя соединение. Если выполнить исходный код класса `Application`, то на консоль будет выведен результат, аналогичный выводимому из сервера HornetQ.

Применение веб-служб REST в Spring

В настоящее время веб-службы REST (RESTful-WS) являются, вероятно, наиболее широко применяемой технологией для удаленного доступа. Веб-службы REST широко применяются в самых разных целях: от вызова удаленных служб через сетевой протокол HTTP до поддержки интерактивного пользовательского интерфейса веб-приложения в стиле Ajax. Распространенность веб-служб REST объясняется целым рядом причин, в том числе следующими.

- **Простота понимания.** Веб-службы REST спроектированы на основе сетевого протокола HTTP. Адрес URL вместе с методом доступа по сетевому протоколу HTTP определяет назначение запроса. Например, URL типа `http://somedomain.com/restful/customer/1` вместе с методом доступа `GET` по сетевому протоколу HTTP означает, что клиенту требуется извлечь сведения о заказчике с идентификатором `1`.
- **Облегченность.** Веб-службы REST являются более легковесными по сравнению с веб-службами, построенными на основе сетевого протокола SOAP, которые требуют большого объема метаданных, чтобы обозначить, какую именно службу должен вызывать клиент. Запрос и ответ REST представляют собой не более чем запрос и ответ по сетевому протоколу HTTP, как в любом другом веб-приложении.
- **Дружественность по отношению к брандмауэру.** Веб-службы REST предназначены для доступа по сетевому протоколу HTTP (или HTTPS), поэтому приложение становится намного более дружественным к брандмауэру и легко доступным для удаленных клиентов.

В этом разделе будут рассмотрены основные понятия веб-служб REST и их поддержка в Spring с помощью модуля Spring MVC.

Введение в веб-службы REST

Аббревиатура *REST* означает *REpresentational State Transfer* (Передача состояния представлений). Технология REST определяет ряд архитектурных ограничений, которые совместно описывают *унифицированный интерфейс* для доступа к ресурсам. К основным понятиям этого унифицированного интерфейса относятся идентификация ресурсов и манипулирование ресурсами через представления.

Для идентификации ресурсов порция информации должна быть доступна через унифицированный идентификатор ресурса (Uniform Resource Identifier — URI). На-

пример, `www.somedomain.com/api/singer/1` — это URI, представляющий ресурс со сведениями о певце с идентификатором `1`. Если же певец с идентификатором `1` не существует, клиент получит код ошибки `404` по сетевому протоколу HTTP, как и в том случае, когда на веб-сайте отсутствует запрошенная страница. Еще одним примером может служить URI типа `www.somedomain.com/api/singer`, представляющий ресурс со списком сведений о певцах. Такими идентифицируемыми ресурсами можно управлять посредством различных представлений, описываемых в виде методов доступа по сетевому протоколу HTTP в табл. 12.1.

Таблица 12.1. Представления для манипулирования ресурсами

Метод доступа по протоколу HTTP	Описание
`GET`	Извлекает представление ресурса
`HEAD`	Аналогично методу `GET`, но без тела ответа; обычно служит для получения заголовка
`POST`	Создает новый ресурс
`PUT`	Обновляет ресурс
`DELETE`	Удаляет ресурс
`OPTIONS`	Извлекает методы доступа, разрешенные в сетевом протоколе HTTP

Подробное описание веб-служб REST можно найти в книге *Ajax and REST Recipes: A Problem-Solution Approach* (издательство Apress, 2006 г.).

Ввод обязательных зависимостей для примеров из этой главы

Чтобы построить приложение Spring REST, следует внедрить ряд новых зависимостей. А поскольку в представленных далее примерах придется организовать обмен объектами между клиентом и сервером, то для их сериализации и десериализации потребуется отдельная библиотека. Попутно было бы полезно продемонстрировать различные способы сериализации на примере одного и того же приложения (в данном случае — в форматах XML и JSON), а это означает, что для каждого из них потребуется отдельная библиотека. Такие библиотеки, внедряемые как зависимости для веб-служб REST, вместе с их назначением перечислены в табл. 12.2.

Таблица 12.2. Библиотеки, внедряемые как зависимости для веб-служб REST

Идентификатор группы или модуля	Версия	Назначение
`org.springframework: spring-oxm`	5.0.0.RC1	Модуль Spring для взаимного преобразования объектов и данных формата XML
`org.codehaus.jackson: jacksonDatabind`	2.9.0.pr3	Процессор Jackson JSON для поддержки данных в формате JSON

Окончание табл. 12.2

Идентификатор группы или модуля	Версия	Назначение
`org.codehaus.castor: castor-xml`	1.4.1	Библиотека Castor XML, предназначенная для сериализации и десериализации данных в формате XML
`org.apache. httpcomponents: httpclient`	4.5.3	Проект Apache HTTP Components. Библиотека клиента HTTP, предназначенная для вызова веб-служб REST

Проектирование веб-службы REST для певцов

Первая стадия процесса разработки приложения RESTful-WS состоит в проектировании структуры службы, включая поддерживаемые методы доступа по сетевому протоколу HTTP и целевые URL для различных операций. В веб-службах REST для певцов требуется поддерживать операции запрашивания, создания, обновления и удаления. Так, для запрашивания необходимо поддерживать извлечение сведений обо всех певцах или отдельном певце по идентификатору.

Такие службы будут реализованы в виде контроллера Spring MVC в классе `Singer Controller` из пакета `com.apress.prospring5.ch12`. Шаблон URL, метод доступа по протоколу HTTP, описание и соответствующие методы контроллера приведены в табл. 12.3. Все URL указываются далее относительно адреса `http://localhost:8080/ch12/restful`, а для представления данных поддерживаются форматы XML и JSON. Подходящий формат будет предоставлен в соответствии с типом среды передачи данных в заголовке клиентского HTTP-запроса.

Таблица 12.3. Методы доступа по протоколу HTTP и контроллера Spring MVC

URL	Метод доступа по протоколу HTTP	Описание	Метод контроллера
`/singer/ listdata`	`GET`	Извлекает сведения обо всех певцах	`listData()`
`/singer/id`	`GET`	Извлекает сведения об одном певце	`findBySingerId()`
`/singer`	`POST`	Создает сведения о новом певце	`create()`
`/singer/id`	`PUT`	Обновляет сведения о певце по его идентификатору	`update()`
`/singer/id`	`DELETE`	Удаляет сведения о певце по его идентификатору	`delete()`

Доступ к веб-службам REST средствами Spring MVC

В этом разделе поясняется, как пользоваться модулем Spring MVC для организации доступа к услугам для певцов через веб-службы REST, спроектированные в пре-

дыдущем разделе. Рассматриваемый здесь пример построен на основе ряда классов, реализующих интерфейс `SingerService`. Эти классы применялись ранее в примере, демонстрировавшем действие механизма вызова Spring HTTP.

В частности, класс `Singer` должен быть вам уже знаком, поэтому мы не будем приводить его исходный код снова. Но для того чтобы произвести сериализацию и десериализацию списка певцов, его придется инкапсулировать в контейнере[5]. Ниже приведен исходный код класса `Singers` с единственным свойством, содержащим список объектов типа `Singer`. Этот класс служит для поддержки преобразования данных из списка певцов, возвращаемого методом `listData()` из класса `SingerController`, в формат XML или JSON.

```
package com.apress.prospring5.ch12;

import com.apress.prospring5.ch12.entities.Singer;
import java.io.Serializable;
import java.util.List;

public class Singers implements Serializable {
  private List<Singer> singers;

  public Singers() {
  }

  public Singers(List<Singer> singers) {
    this.singers = singers;
  }

  public List<Singer> getSingers() {
    return singers;
  }

  public void setSingers(List<Singer> singers) {
    this.singers = singers;
  }
}
```

Конфигурирование библиотеки Castor XML

Для преобразования возвращаемых сведений о певцах в формат XML будет использована библиотека Castor XML (`https://castor-data-binding.github.io/castor/reference-guide/reference/xml/xml-framework.html`). В этой библиотеке поддерживается целый ряд режимов взаимного преобразования объектов POJO и данных формата XML, поэтому в рассматриваемом здесь примере для опре-

[5] Это требуется для сериализации в формате XML, тогда как для формата JSON класс контейнера не требуется.

деления подобного преобразования будет использоваться XML-файл. Содержимое этого файла преобразования (по имени `oxm-mapping.xml`) приведено ниже.

```
<mapping>
  <class name="com.apress.prospring5.ch12.Singers">
    <field name="singers"
      type="com.apress.prospring5.ch12.entities.Singer"
        collection="arraylist">
      <bind-xml name="singer"/>
    </field>
  </class>

  <class name="com.apress.prospring5.ch12.entities.Singer"
     identity="id">
     <map-to xml="singer" />

     <field name="id" type="long">
        <bind-xml name="id" node="element"/>
     </field>
     <field name="firstName" type="string">
        <bind-xml name="firstName" node="element" />
     </field>
     <field name="lastName" type="string">
        <bind-xml name="lastName" node="element" />
     </field>
     <field name="birthDate" type="string"
            handler="dateHandler">
        <bind-xml name="birthDate" node="element" />
     </field>
     <field name="version" type="integer">
        <bind-xml name="version" node="element" />
     </field>
  </class>

  <field-handler name="dateHandler"
     class=
         "com.apress.prospring5.ch12.DateTimeFieldHandler">
     <param name="date-format" value="yyyy-MM-dd"/>
  </field-handler>
</mapping>
```

Здесь определяются два преобразования. В первом дескрипторе разметки `<class>` определяется класс `Singers` со свойством `singers`, содержащим список объектов типа `Singer`, преобразуемых с помощью дескриптора разметки `<bind-xml name="singer"/>`. А во втором дескрипторе разметки `<class>` объект типа `Singer` преобразуется с помощью дескриптора разметки `<map-to xml="singer"/>`. Кроме того, для поддержки преобразования данных типа `java.util.Date` (в свойстве `birthDate` объекта типа `Singer`) реализуется специальный обработчик полей из библиотеки Castor XML. Этот обработчик полей приведен ниже.

```
package com.apress.prospring5.ch12;

import org.exolab.castor.mapping.GeneralizedFieldHandler;
import org.exolab.castor.mapping.ValidityException;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import java.text.ParseException;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Properties;

public class DateTimeFieldHandler
       extends GeneralizedFieldHandler {

  private static Logger logger =
          LoggerFactory.getLogger(
                        DateTimeFieldHandler.class);

  private static String dateFormatPattern;

  @Override
  public void setConfiguration(Properties config)
         throws ValidityException {
    dateFormatPattern = config.getProperty("date-format");
  }

  @Override
  public Object convertUponGet(Object value) {
    Date dateTime = (Date) value;
    return format(dateTime);
  }

  @Override
  public Object convertUponSet(Object value) {
    String dateTimeString = (String) value;
    return parse(dateTimeString);
  }

  @Override
  public Class<Date> getFieldType() {
    return Date.class;
  }

  protected static String format(final Date dateTime) {
    String dateTimeString = "";
    if (dateTime != null) {
      SimpleDateFormat sdf =
            new SimpleDateFormat(dateFormatPattern);
```

```
      dateTimeString = sdf.format(dateTime);
    }
    return dateTimeString;
  }

  protected static Date parse(final String dateTimeString) {
    Date dateTime = new Date();
    if (dateTimeString != null) {
      SimpleDateFormat sdf =
            new SimpleDateFormat(dateFormatPattern);
      try {
        dateTime = sdf.parse(dateTimeString);
      } catch (ParseException e) {
        logger.error("Not a valida date:"
                    + dateTimeString, e);
      }
    }
    return dateTime;
  }
}
```

В приведенном выше коде расширяется класс `org.exolab.castor.mapping.GeneralizedFieldHandler` из библиотеки Castor XML и реализуются методы `convertUponGet()`, `convertUponSet()` и `getFieldType()`. В этих методах реализуется логика взаимного преобразования типов данных `DateTime` и `String` для их последующего применения в библиотеке Castor XML. Кроме того, в данном коде определяется файл свойств для применения вместе с библиотекой Castor XML. Содержимое этого файла (под именем `castor.properties`) приведено ниже, где в свойстве `indent` библиотеке Castor XML предписывается формировать XML-разметку с отступами, что намного упрощает ее чтение во время тестирования.

```
rg.exolab.castor.indent=true
```

Реализация контроллера в классе SingerController

На следующей стадии рассматриваемого здесь процесса реализуется контроллер в классе `SingerController`. Ниже приведен исходный код класса `SingerController`, где реализуются все методы, перечисленные в табл. 12.3.

```
package com.apress.prospring5.ch12;

import com.apress.prospring5.ch12.entities.Singer;
import com.apress.prospring5.ch12.services
          .SingerService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

@Controller
@RequestMapping(value="/singer")
public class SingerController {
  final Logger logger =
        LoggerFactory.getLogger(
                      SingerController.class);

  @Autowired
  private SingerService singerService;

  @ResponseStatus(HttpStatus.OK)
  @RequestMapping(value = "/listdata",
                  method = RequestMethod.GET)
  @ResponseBody
  public Singers listData() {
    return new Singers(singerService.findAll());
  }

  @RequestMapping(value="/{id}", method=RequestMethod.GET)
  @ResponseBody
  public Singer findSingerById(@PathVariable Long id) {
    return singerService.findById(id);
  }

  @RequestMapping(value="/", method=RequestMethod.POST)
  @ResponseBody
  public Singer create(@RequestBody Singer singer) {
    logger.info("Creating singer: " + singer);
    singerService.save(singer);
    logger.info("Singer created successfully with info: "
              + singer);
    return singer;
  }

  @RequestMapping(value="/{id}", method=RequestMethod.PUT)
  @ResponseBody
  public void update(@RequestBody Singer singer,
                     @PathVariable Long id) {
    logger.info("Updating singer: " + singer);
    singerService.save(singer);
    logger.info("Singer updated successfully with info: "
              + singer);
  }
```

```
  @RequestMapping(value="/{id}",
                  method=RequestMethod.DELETE)
  @ResponseBody
  public void delete(@PathVariable Long id) {
    logger.info("Deleting singer with id: " + id);
    Singer singer = singerService.findById(id);
    singerService.delete(singer);
    logger.info("Singer deleted successfully");
  }
}
```

В приведенном выше классе необходимо отметить следующее.

- Этот класс снабжен аннотацией `@Controller`, указывающей на то, что он является контроллером Spring MVC.
- В аннотации `@RequestMapping(value="/contact")` на уровне класса определяется, что данный контроллер будет сопоставлен со всеми URL в главном веб-контексте. В рассматриваемом здесь примере данный контроллер будет обрабатывать все URL, начинающиеся с адреса `http://localhost:8080/ch12/singer`.
- Интерфейс `SingerService`, реализованный ранее в этой главе на уровне обслуживания, автоматически связан с контроллером.
- С помощью аннотаций `@RequestMapping` в каждом методе из класса `SingerController` указывается шаблон URL и сопоставляемый с ним метод доступа по протоколу HTTP. Например, метод `listData()` будет сопоставлен с URL типа `http://localhost:8080/ch12/singer/listdata` и методом доступа `GET` по протоколу HTTP, а метод `update()` — с URL типа `http://localhost:8080/ch12/singer /{id}` и методом доступа `PUT` по протоколу HTTP.
- Аннотация `@ResponseBody` распространяется на все методы. Она предписывает направлять все значений, возвращаемые из методов, непосредственно в поток вывода HTTP-ответа.
- В методах, принимающих переменные путей (например, в методе `findContactById()`), такие переменные снабжаются аннотацией `@PathVariable`. Это вынуждает модуль Spring MVC связывать переменную пути в URL (например, `http://localhost:8080/ch12/contact/1`) с аргументом `id` метода `findContactById()`. Следует иметь в виду, что аргумент `id` относится к типу `Long`, поэтому система преобразования типов Spring автоматически проеобразует тип String в тип `Long`.
- В методах `create()` и `update()` аргумент типа `Singer` снабжен аннотацией `@RequestBody`. Это вынуждает Spring автоматически привязывать содержимое тела HTTP-запроса к объекту предметной области типа `Singer`. Преобразование будет выполняться объявленными экземплярами интерфейса `Http`

MessageConverter<Object> (из пакета org.springframework.http.converter) для поддержки форматов, обсуждаемых далее в этой главе.

На заметку В версии Spring 4.0 специально для контроллеров веб-служб REST внедрена аннотация @RestController. Эта аннотация удобна тем, что она сама снабжена аннотациями @Controller и @ResponseBody. Если она применяется в классе контроллера, то все методы, снабженные аннотацией @RequestMapping, автоматически снабжаются и аннотацией @ResponseBody. Версия класса SingerController с аннотацией @RestController представлена далее в этой главе.

Конфигурирование веб-приложения Spring

Для обработки REST-запросов, посылаемых клиентом, требуется веб-приложение Spring, а следовательно, его необходимо каким-то образом сконфигурировать. Простая конфигурация веб-приложения была представлена ранее в этой главе. Эту конфигурацию можно теперь дополнить компонентами Spring Beans для преобразования сообщений сетевого протокола HTTP в форматы XML и JSON.

Веб-приложения разрабатываются в Spring по проектному шаблону "Единая точка входа" (Front Controller)[6], где все запросы получаются одним контроллером, который затем направляет их соответствующим обработчикам (классам контроллеров). В качестве такого центрального диспетчера служит экземпляр класса org.springframework.web.servlet.DispatcherServlet, регистрируемый в классе AbstractAnnotationConfigDispatcherServletInitializer, который придется расширить, чтобы заменить конфигурацию из файла web.xml. И это делается в приведенном ниже классе WebInitializer для рассматриваемых здесь примеров.

```
package com.apress.prospring5.ch12.init;

import com.apress.prospring5.ch12.config.DataServiceConfig;
import org.springframework.web.servlet.support
     .AbstractAnnotationConfigDispatcherServletInitializer;

public class WebInitializer extends
     AbstractAnnotationConfigDispatcherServletInitializer {

  @Override
  protected Class<?>[] getRootConfigClasses() {
    return new Class<?>[] {
      DataServiceConfig.class
    };
  }

  @Override
```

---

[6] Подробное описание этого проектного шаблона можно найти по адресу www.oracle.com/technetwork/java/frontcontroller-135648.html.

```
  protected Class<?>[] getServletConfigClasses() {
    return new Class<?>[] {
      WebConfig.class
    };
  }

  @Override
  protected String[] getServletMappings() {
    return new String[]{"/"};
  }
}
```

У каждого сервлета диспетчера типа `DispatchServlet` в модуле Spring MVC имеется свой контекст типа `WebApplicationContext` (но все компоненты Spring Beans уровня обслуживания, определяемые в литерале класса `DataServiceConfig.class`, т.е. в корневом контексте типа `WebApplicationContext`, будут доступны и в собственном контексте типа `WebApplicationContext` каждого сервлета). В методе `getServletMappings()` веб-контейнеру (например, Tomcat) предписывается обрабатывать все URL по шаблону / (например, `http://localhost:8080/singer`) с помощью сервлета REST. Здесь можно было бы, конечно, ввести такой контекст, как `/ch12`, но для целей примеров из этой главы требуются как можно более короткие и очевидные URL.

Ниже приведен класс `WebConfig` для конфигурирования модуля Spring MVC с преобразователями сообщений сетевого протокола HTTP.

```
package com.apress.prospring5.ch12.init;

import com.fasterxml.jackson.annotation.JsonInclude;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.SerializationFeature;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.core.io.ClassPathResource;
import org.springframework.http.MediaType;
import org.springframework.http.converter
          .HttpMessageConverter;
import org.springframework.http.converter.json
          .MappingJackson2HttpMessageConverter;
import org.springframework.http.converter.xml
          .MarshallingHttpMessageConverter;
import org.springframework.oxm.castor.CastorMarshaller;
import org.springframework.web.servlet.config.annotation
```

```
          .DefaultServletHandlerConfigurer;
import org.springframework.web.servlet.config.annotation
          .EnableWebMvc;
import org.springframework.web.servlet.config.annotation
          .WebMvcConfigurer;
import org.springframework.web.servlet.config.annotation
          .WebMvcConfigurerAdapter;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.List;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages =
               {"com.apress.prospring5.ch12"})
public class WebConfig extends WebMvcConfigurer {

  @Autowired ApplicationContext ctx;

  @Bean
  public MappingJackson2HttpMessageConverter
         mappingJackson2HttpMessageConverter() {
    MappingJackson2HttpMessageConverter
           mappingJackson2HttpMessageConverter =
                 new MappingJackson2HttpMessageConverter();
    mappingJackson2HttpMessageConverter
                  .setObjectMapper(objectMapper());
    return mappingJackson2HttpMessageConverter;
  }

  @Override
  public void configureDefaultServletHandling(
         DefaultServletHandlerConfigurer configurer) {
    configurer.enable();
  }

  @Bean
  public ObjectMapper objectMapper() {
    ObjectMapper objMapper = new ObjectMapper();
    objMapper.enable(SerializationFeature.INDENT_OUTPUT);
    objMapper.setSerializationInclusion(
              JsonInclude.Include.NON_NULL);
    DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
    objMapper.setDateFormat(df);
    return objMapper;
  }
```

```
  @Override
  public void configureMessageConverters(
         List<HttpMessageConverter<?>> converters) {
    converters.add(mappingJackson2HttpMessageConverter());
    converters.add(singerMessageConverter());
  }

  @Bean MarshallingHttpMessageConverter
        singerMessageConverter() {
    MarshallingHttpMessageConverter mc =
              new MarshallingHttpMessageConverter();
    mc.setMarshaller(castorMarshaller());
    mc.setUnmarshaller(castorMarshaller());
    List<MediaType> mediaTypes = new ArrayList<>();
    MediaType mt = new MediaType("application", "xml");
    mediaTypes.add(mt);
    mc.setSupportedMediaTypes(mediaTypes);
    return mc;
  }

  @Bean CastorMarshaller castorMarshaller() {
    CastorMarshaller castorMarshaller =
                    new CastorMarshaller();
    castorMarshaller.setMappingLocation(ctx.getResource(
                    "classpath:spring/oxm-mapping.xml"));
    return castorMarshaller;
  }
}
```

В приведенном выше классе необходимо отметить следующее.

- Аннотация `@EnableWebMvc` активизирует поддержку аннотаций в модуле Spring MVC и, в частности, аннотации `@Controller`[7], а также регистрирует систему преобразования типов и форматирования данных в Spring. Кроме того, в определении данной аннотации активизируется поддержка проверки достоверности данных по спецификации JSR-303.
- В методе `configureMessageConverters()` объявляются экземпляры типа `HttpMessageConverter`, предназначенные для преобразования сообщений в поддерживаемых форматах среды передачи[8]. В данном случае объявляются два преобразователя сообщений, поскольку предполагается поддерживать оба формата данных: JSON и XML. Первый из этих преобразователей сообщений реализован в классе `MappingJackson2HttpMessageConverter`, который слу-

[7] Это равнозначно дескриптору разметки `<mvc:annotation-driven>`.

[8] Это равнозначно дескриптору `<mvc:message-converters>`, внедренному в версии Spring 3.1.

жит для поддержки в Spring библиотеки Jackson JSON[9], а другой — в классе `MarshallingHttpMessageConverter`, предоставляемом в модуле `spring-oxm` для сериализации и десериализации данных формата XML. В классе `MarshallingHttpMessageConverter` необходимо определить применяемое средство сериализации и десериализации (в данном случае оно предоставляется библиотекой Castor XML).

- Для реализации компонента `castorMarshaller` применяется класс `org.springframework.oxm.castor.CastorMarshaller`, предоставляемый в Spring для интеграции с библиотекой Castor XML. Кроме того, предоставляется место преобразования, которое требуется для нормальной работы библиотеки Castor XML.
- Аннотация `@ComponentScan` предписывает каркасу Spring просмотреть классы контроллеров в указанном пакете[10].

Итак, создание службы на стороне сервера завершено. На данном этапе необходимо создать файл формата WAR, содержащий веб-приложение. А если вы работаете в IDE вроде IntelliJ IDEA или STS, то запустите на выполнение экземпляр контейнера сервлетов Tomcat.

Тестирование веб-служб REST средствами `curl`

А теперь проведем быстрое тестирование реализованных нами веб-служб REST. Это проще всего сделать с помощью `curl` — инструментального средства[11], работающего в режиме командной строки для передачи данных в синтаксисе URL. Чтобы воспользоваться этим инструментальным средством, загрузите его с указанного веб-сайта и установите на своем компьютере.

Например, чтобы проверить извлечение сведений обо всех певцах, откройте окно командной строки в Windows или окно терминала в Unix/Linux, разверните файл формата WAR на сервере и введите приведенную ниже команду.

```
$ curl -v -H "Accept: application/json"
  http://localhost:8080/singer/listdata
* Trying ::1...
* Connected to localhost (::1) port 8080 (#0)
> GET /singer/listdata HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.43.0
> Accept: application/json
>
< HTTP/1.1 200
< Content-Type: application/json;charset=UTF-8
```

[9] Официальный веб-сайт библиотеки Jackson JSON находится по адресу `http://jackson.codehaus.org`.

[10] Это равнозначно дескриптору `<context:component-scan>`.

[11] См. по адресу `http://curl.haxx.se`.

```
< Transfer-Encoding: chunked
< Date: Sat, 17 Jun 2017 17:16:43 GMT
<
{
  "singers" : [ {
  "id" : 1,
  "version" : 0,
  "firstName" : "John",
  "lastName" : "Mayer",
  "birthDate" : "1977-10-16"
}, {
  "id" : 2,
  "version" : 0,
  "firstName" : "Eric",
  "lastName" : "Clapton",
  "birthDate" : "1945-03-30"
}, {
  "id" : 3,
  "version" : 0,
  "firstName" : "John",
  "lastName" : "Butler",
  "birthDate" : "1975-04-01"
} ]
* Connection #0 to host localhost left intact
```

Эта команда отправляет HTTP-запрос серверной веб-службе REST. В данном случае будет вызван метод `listData()` из класса `SingerController` для извлечения и возврата сведений обо всех певцах. Кроме того, с помощью параметра `-H` в HTTP-заголовке объявляется атрибут `Accept`, обозначающий, что клиент предполагает получить данные в формате JSON. В результате выполнения данной команды первоначально заполненные сведения о певцах будут выведены на консоль в формате JSON. А теперь проверим преобразование данных в формат XML. Ниже приведена соответствующая команда и результаты ее выполнения.

```
$ curl -v -H "Accept: application/xml"
  http://localhost:8080/singer/listdata
* Trying ::1...
* Connected to localhost (::1) port 8080 (#0)
> GET /singer/listdata HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.43.0
> Accept: application/xml
>
< HTTP/1.1 200
< Content-Type: application/xml
< Transfer-Encoding: chunked
< Date: Sat, 17 Jun 2017 17:18:22 GMT
<
```

```
<?xml version="1.0" encoding="UTF-8"?>
<singers>
    <singer>
       <id>1</id>
       <firstName>John</firstName>
       <lastName>Mayer</lastName>
       <birthDate>1977-10-16</birthDate>
       <version>0</version>
    </singer>
    <singer>
       <id>2</id>
       <firstName>Eric</firstName>
       <lastName>Clapton</lastName>
       <birthDate>1945-03-30</birthDate>
       <version>0</version>
    </singer>
    <singer>
        <id>3</id>
        <firstName>John</firstName>
        <lastName>Butler</lastName>
        <birthDate>1975-04-01</birthDate>
        <version>0</version>
    </singer>
</singers>
* Connection #0 to host localhost left intact
```

Как видите, по сравнению с приведенным выше примером здесь имеется всего лишь одно отличие: вместо формата JSON в атрибуте `Accept` указан формат XML. Выполнение этой команды приводит к выводу на консоль сведений о певцах в формате XML. И это происходит благодаря тому, что в контексте типа `WebApplication Context` сервлета REST объявлены компоненты Spring Beans типа `HttpMessage Converter`, а модуль Spring MVC будет вызывать соответствующий преобразователь сообщений, исходя из значения атрибута `Accept`, указанного в HTTP-заголовке клиента, и соответственно направлять полученный результат в HTTP-ответ.

Применение класса `RestTemplate` для доступа к веб-службам REST

Для построения приложений в Spring предоставляется класс `RestTemplate`, предназначенный для доступа к веб-службам REST. В этом разделе будет показано, как пользоваться этим классом для доступа к службе для певцов на сервере. Рассмотрим сначала приведенную ниже простую конфигурацию контекста типа `ApplicationContext` для предоставляемого в Spring класса `RestTemplate`.

```
package com.apress.prosring5.ch12;

import com.apress.prospring5.ch12.CustomCredentialsProvider;
import org.apache.http.auth.Credentials;
```

```
import org.apache.http.auth.UsernamePasswordCredentials;
import org.apache.http.client.CredentialsProvider;
import org.apache.http.client.HttpClient;
import org.apache.http.impl.client.HttpClientBuilder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.ClassPathResource;
import org.springframework.http.MediaType;
import org.springframework.http.client
          .HttpComponentsClientHttpRequestFactory;
import org.springframework.http.converter
          .HttpMessageConverter;
import org.springframework.http.converter.xml
          .MarshallingHttpMessageConverter;
import org.springframework.oxm.castor.CastorMarshaller;
import org.springframework.web.client.RestTemplate;

import java.util.ArrayList;
import java.util.List;

@Configuration
public class RestClientConfig {

  @Autowired ApplicationContext ctx;

  @Bean
  public HttpComponentsClientHttpRequestFactory
         httpRequestFactory() {
    HttpComponentsClientHttpRequestFactory
        httpRequestFactory =
            new HttpComponentsClientHttpRequestFactory();
    HttpClient httpClient = HttpClientBuilder.create()
                            .build();
    httpRequestFactory.setHttpClient(httpClient);
    return httpRequestFactory;
  }

  @Bean
  public RestTemplate restTemplate() {
    RestTemplate restTemplate =
        new RestTemplate(httpRequestFactory());
    List<HttpMessageConverter<?>> mcvs = new ArrayList<>();
    mcvs.add(singerMessageConverter());
    restTemplate.setMessageConverters(mcvs);
    return restTemplate;
  }
```

```
  @Bean MarshallingHttpMessageConverter
        singerMessageConverter() {
    MarshallingHttpMessageConverter mc =
                new MarshallingHttpMessageConverter();
    mc.setMarshaller(castorMarshaller());
    mc.setUnmarshaller(castorMarshaller());
    List<MediaType> mediaTypes = new ArrayList<>();
    MediaType mt = new MediaType("application", "xml");
    mediaTypes.add(mt);
    mc.setSupportedMediaTypes(mediaTypes);
    return mc;
  }

  @Bean CastorMarshaller castorMarshaller() {
    CastorMarshaller castorMarshaller =
                    new CastorMarshaller();
    castorMarshaller.setMappingLocation(
    ctx.getResource( "classpath:spring/oxm-mapping.xml"));
    return castorMarshaller;
  }
}
```

В приведенном выше коде компонент `restTemplate` объявляется с помощью класса `RestTemplate`. В этом классе используется библиотека Castor XML для внедрения преобразователей сообщений с помощью экземпляра типа `MarshallingHttpMessageConverter` таким же образом, как и на стороне сервера. Файл сопоставления будет совместно использоваться как на стороне сервера, так и на стороне клиента. Кроме того, для компонента `restTemplate` в анонимном классе `MarshallingHttpMessageConverter` с помощью экземпляра типа `MediaType` внедряются поддерживаемые типы среды передачи данных и, в частности, указывается единственный поддерживаемый формат XML. В итоге клиент всегда предполагает получить возвращаемые данные в формате XML, а библиотека Castor XML поможет выполнить взаимное преобразование простых объектов POJO и данных формата XML.

Чтобы проверить URL всех веб-служб REST, поддерживаемых в рассматриваемом здесь веб-приложении Spring, удобно воспользоваться классом для модульного тестирования, выполняемого средствами JUnit в контексте данного приложения, определяемом в конфигурационном классе `RestClientConfig`. Исходный код такого класса для модульного тестирования приведен ниже, причем каждый его метод может быть выполнен по отдельности в IDE вроде IntelliJ IDEA или STS.

```
package com.apress.prosring5.ch12.test;

import com.apress.prospring5.ch12.Singers;
import com.apress.prospring5.ch12.entities.Singer;
import com.apress.prosring5.ch12.RestClientConfig;
import org.junit.Before;
import org.junit.Test;
```

```
import org.junit.runner.RunWith;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation
        .Autowired;
import org.springframework.test.context
        .ContextConfiguration;
import org.springframework.test.context.junit4
        .SpringJUnit4ClassRunner;
import org.springframework.test.context.junit4
        .SpringRunner;
import org.springframework.web.client.RestTemplate;

import java.util.Date;
import java.util.GregorianCalendar;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {RestClientConfig.class})
public class RestClientTest {

  final Logger logger =
        LoggerFactory.getLogger(RestClientTest.class);
  private static final String URL_GET_ALL_SINGERS =
          "http://localhost:8080/singer/listdata";
  private static final String URL_GET_SINGER_BY_ID =
          "http://localhost:8080/singer/{id}";
  private static final String URL_CREATE_SINGER =
          "http://localhost:8080/singer/";
  private static final String URL_UPDATE_SINGER =
          "http://localhost:8080/singer/{id}";
  private static final String URL_DELETE_SINGER =
          "http://localhost:8080/singer/{id}";

  @Autowired RestTemplate restTemplate;

  @Before
  public void setUp() {
    assertNotNull(restTemplate);
  }

  @Test
  public void testFindAll() {
    logger.info("--> Testing retrieve all singers");
    Singers singers = restTemplate.getForObject(
                      URL_GET_ALL_SINGERS, Singers.class);
    assertTrue(singers.getSingers().size() == 3);
```

```
    listSingers(singers);
}

@Test
public void testFindbyId() {
    logger.info("--> Testing retrieve a singer by id : 1");
    Singer singer = restTemplate.getForObject(
                    URL_GET_SINGER_BY_ID, Singer.class, 1);
    assertNotNull(singer);
    logger.info(singer.toString());
}

@Test
public void testUpdate() {
    logger.info("--> Testing update singer by id : 1");
    Singer singer = restTemplate.getForObject(
                    URL_UPDATE_SINGER, Singer.class, 1);
    singer.setFirstName("John Clayton");
    restTemplate.put(URL_UPDATE_SINGER, singer, 1);
    logger.info("Singer update successfully: " + singer);
}

@Test
public void testDelete() {
    logger.info("--> Testing delete singer by id : 3");
    restTemplate.delete(URL_DELETE_SINGER, 3);
    Singers singers = restTemplate.getForObject(
                      URL_GET_ALL_SINGERS, Singers.class);
    Boolean found = false;
    for(Singer s: singers.getSingers()) {
      if(s.getId() == 3) {
        found = true;
      }
    };
    assertFalse(found);
    listSingers(singers);
}

@Test
public void testCreate() {
    logger.info("--> Testing create singer");
    Singer singerNew = new Singer();
    singerNew.setFirstName("BB");
    singerNew.setLastName("King");
    singerNew.setBirthDate(new Date(
              (new GregorianCalendar(1940, 8, 16))
              .getTime().getTime()));
    singerNew = restTemplate.postForObject(
              URL_CREATE_SINGER, singerNew, Singer.class);
```

```
    logger.info("Singer created successfully: "
              + singerNew);
    logger.info("Singer created successfully: "
              + singerNew);
    Singers singers = restTemplate.getForObject(
                      URL_GET_ALL_SINGERS, Singers.class);
    listSingers(singers);
  }

  private void listSingers(Singers singers) {
    singers.getSingers()
           .forEach(s -> logger.info(s.toString()));
  }
}
```

Здесь объявлены URL для доступа к различным операциям, которые будут применяться в последующих примерах. В методе `main()` сначала извлекается экземпляр типа `RestTemplate`, а затем в тестовом методе `TestFindAll()` вызывается метод `RestTemplate.getForObject()`, соответствующий методу `GET` доступа по протоколу HTTP. Этому методу передается URL и предполагаемый возвращаемый тип (в данном случае — класс `Singers`, содержащий полный список певцов).

Удостоверьтесь в работоспособности сервера приложений и выполните тестовый метод `TestFindAll()`, чтобы получить приведенный ниже результат. Как видите, компонент Spring Bean типа `MarshallingHttpMessageConverter`, зарегистрированный с помощью класса `RestTemplate`, автоматически преобразует сообщение в простой объект POJO.

```
INFO c.a.p.c.t.RestClientTest - -->
     Testing retrieve all singers
INFO c.a.p.c.t.RestClientTest - Singer - Id: 1,
     First name: John, Last name: Mayer,
     Birthday: Sun Oct 16 00:00:00 EET 1977
INFO c.a.p.c.t.RestClientTest - Singer - Id: 2,
     First name: Eric, Last name: Clapton,
     Birthday: Fri Mar 30 00:00:00 EET 1945
INFO c.a.p.c.t.RestClientTest - Singer - Id: 3,
     First name: John, Last name: Butler,
     Birthday: Tue Apr 01 00:00:00 EET 1975
```

Теперь попробуем извлечь сведения о певце по его идентификатору. В тестовом методе `TestFindAll()` используется версия метода `RestTemplate.getForObject()`, которой также передается идентификатор певца для извлечения в качестве переменной пути из URL (переменная пути `{id}` в `URL_GET_SINGER_BY_ID`). Если в URL содержится несколько переменных пути, то для передачи этих переменных можно воспользоваться экземпляром типа `Map<String,Object>` или поддержкой переменного количества аргументов в методах. В последнем случае переменные пути должны быть указаны в том порядке, в каком они объявлены в URL. Если вы-

полнить тестовый метод testFindbyId(), тест должен пройти и на консоль будет выведен следующий результат:

```
INFO c.a.p.c.t.RestClientTest - -->
     Testing retrieve a singer by id : 1
INFO c.a.p.c.t.RestClientTest - Singer - Id: 1,
     First name: John, Last name: Mayer,
     Birthday: Sun Oct 16 00:00:00 EET 1977
```

Как видите, сведения о певце извлечены правильно. Теперь настала очередь для операции обновления. Сначала извлекается обновляемый объект, представляющий певца. После обновления этого объекта вызывается метод RestTemplate.put(), соответствующий методу PUT доступа по протоколу HTTP. Этому методу передается URL для обновления, видоизмененный объект, представляющий певца, а также идентификатор певца, сведения о котором обновляются. Если выполнить тестовый метод testUpdate(), на консоль будет выведен следующий результат (другие не относящиеся к делу результаты здесь опущены):

```
INFO c.a.p.c.t.RestClientTest - -->
     Testing update singer by id : 1
INFO c.a.p.c.t.RestClientTest -
     Singer update successfully: Singer - Id: 1,
     First name: John Clayton, Last name: Mayer,
     Birthday: Sun Oct 16 00:00:00 EET 1977
```

Проверим далее операцию удаления. С этой целью вызывается метод RestTemplate.delete(), соответствующий методу DELETE доступа по протоколу HTTP. Этому методу передается URL и идентификатор удаляемого певца. После этого извлекаются сведения обо всех певцах, которые еще раз отображаются с целью удостовериться в правильности операции удаления. Если выполнить тестовый метод testDelete(), на консоль будет выведен приведенный ниже результат (другие не относящиеся к делу результаты здесь опущены). Как видите, в данном случае удален певец с идентификатором 3.

```
INFO c.a.p.c.t.RestClientTest - -->
     Testing delete singer by id : 3
INFO c.a.p.c.t.RestClientTest - Singer - Id: 1,
     First name: John Clayton, Last name: Mayer,
     Birthday: Sun Oct 16 00:00:00 EET 1977
INFO c.a.p.c.t.RestClientTest -
     Singer - Id: 2, First name: Eric, Last name: Clapton,
     Birthday: Fri Mar 30 00:00:00 EET 1945
```

И, наконец, проверим операцию вставки. С этой целью сначала создается объект типа Singer, а затем вызывается метод RestTemplate.postForObject(), соответствующий методу POST доступа по протоколу HTTP. Этому методу передается URL, созданный объект типа Singer и тип класса. Если выполнить тестовый метод testCreate(), то на консоль будет выведен приведенный ниже результат. Как види-

те, на сервере был сначала создан, а затем возвращен клиенту новый объект, представляющий певца.

```
INFO c.a.p.c.t.RestClientTest - -->
     Testing create singer
INFO c.a.p.c.t.RestClientTest -
     Singer created successfully: Singer - Id: 4,
     First name: BB, Last name: King,
     Birthday: Mon Sep 16 00:00:00 EET 1940
     // перечислить всех певцов
INFO c.a.p.c.t.RestClientTest - Singer - Id: 1,
     First name: John Clayton, Last name: Mayer,
     Birthday: Sun Oct 16 00:00:00 EET 1977
INFO c.a.p.c.t.RestClientTest - Singer - Id: 2,
     First name: Eric, Last name: Clapton,
     Birthday: Fri Mar 30 00:00:00 EET 1945
INFO c.a.p.c.t.RestClientTest - Singer - Id: 4,
     First name: BB, Last name: King,
     Birthday: Mon Sep 16 00:00:00 EET 1940
```

Защита веб-служб REST средствами Spring Security

Любая удаленная служба требует защиты от несанкционированного доступа, а также извлечения данных предметной области или взаимодействия с ней, и в этом отношении веб-службы REST не являются исключением. В этом разделе поясняется, как пользоваться проектом Spring Security для защиты веб-служб REST на стороне сервера. В рассматриваемом здесь примере применяется версия Spring Security 5.0.0.M2. Это последняя на момент написания данной книги версия, в которой предоставляется удобная поддержка веб-служб REST.

Процесс применения Spring Security для защиты веб-служб REST осуществляется в три стадии. На первой стадии в дескрипторе развертывания веб-приложения (файл `web.xml`) требуется ввести фильтр защиты `springSecurityFilterChain`, но поскольку рассматриваемое здесь приложение не конфигурируется в формате XML, то данный фильтр заменяется классом, расширяющим класс `AbstractSecurityWebApplicationInitializer`. С помощью этого класса регистрируется класс `DelegatingFilterProxy`, чтобы применить компонент `springSecurityFilterChain` прежде, чем зарегистрировать любой другой фильтр. Ниже приведена реализация данного класса, которая выглядит пустой, поскольку в данном случае никакой специальной ее настройки не предполагается.

```
package com.apress.prospring5.ch12.init;

import org.springframework.security.web.context
          .AbstractSecurityWebApplicationInitializer;
```

```
public class SecurityWebApplicationInitializer
       extends AbstractSecurityWebApplicationInitializer {
}
```

Теперь необходимо ввести класс для конфигурирования защиты веб-приложения Spring, где требуется объявить тех, кому разрешается доступ к данному приложению, а также те действия, которые допускается выполнять. В рассматриваемом здесь приложении дело обстоит просто, поскольку в учебных целях аутентификация выполняется в оперативной памяти. Поэтому в приведенном ниже конфигурационном классе жестко задается имя пользователя `prospring5`, пароль `prospring5` и роль `REMOTE`.

```
package com.apress.prospring5.ch12.init;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.security.config.annotation
          .authentication.builders
          .AuthenticationManagerBuilder;
import org.springframework.security.config.annotation
          .web.builders.HttpSecurity;
import org.springframework.security.config.annotation
          .web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation
          .web.configuration.WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity
public class SecurityConfig
       extends WebSecurityConfigurerAdapter {

  private static Logger logger =
          LoggerFactory.getLogger(SecurityConfig.class);

  @Autowired
  protected void configureGlobal(
            AuthenticationManagerBuilder auth)
            throws Exception {
    try {
      auth.inMemoryAuthentication()
          .withUser("prospring5")
          .password("prospring5")
          .roles("REMOTE");
    } catch (Exception e) {
      logger.error("Could not configure "
                + "authentication!", e);
    }
  }
```

```
  @Override
  protected void configure(HttpSecurity http)
            throws Exception {
    http.sessionManagement()
        .sessionCreationPolicy(
                SessionCreationPolicy.STATELESS)
        .and()
        .authorizeRequests()
        .antMatchers("/**").permitAll()
        .antMatchers("/rest/**")
        .hasRole("REMOTE")
        .anyRequest()
        .authenticated()
        .and()
        .formLogin()
        .and()
        .httpBasic()
        .and()
        .csrf().disable();
  }
}
```

Приведенный выше конфигурационный класс снабжен аннотацией `@EnableWebSecurity` с целью активизировать безопасный режим работы в веб-приложении Spring. Как объявляется в методе `configure()`, ресурсы, доступные по URL типа `/rest/**`, должны быть защищены. А метод `sessionCreationPolicy()` позволяет указать в конфигурации, следует ли создавать сеанс связи по протоколу HTTP после аутентификации. Применяемая здесь веб-служба REST действует без сохранения состояния, поэтому устанавливается значение `SessionCreationPolicy.STATELESS`, предписывающее Spring Security не создавать сеанс связи по протоколу HTTP для всех REST-запросов. Это помогает повысить производительность веб-службы REST.

Далее делается вызов метода `antMatchers("/rest/**")`, где доступ к веб-службе REST разрешается только тем пользователям, для которых назначена роль `REMOTE`. А в методе `httpBasic()` указывается, что для веб-служб REST поддерживается только простая аутентификация по сетевому протоколу HTTP.

В методе `configureGlobal(AuthenticationManagerBuilder auth)` определяются сведения об аутентификации и, в частности, простой поставщик услуг аутентификации с жестко закодированным именем пользователя `remote` и таким же паролем, а также назначенной ролью `REMOTE`. Но в корпоративной среде аутентификация, вероятнее всего, будет выполняться с помощью базы данных или поиска по сетевому протоколу LDAP.

Метод `formLogin()` вызывается для того, чтобы дать Spring команду сформировать простую форму регистрации, с помощью которой можно проверить, правильно ли защищено приложение. Такая форма доступна по адресу `http://localhost:8080/login`.

Фильтр защиты `springSecurityFilterChain` служит для того, чтобы в Spring Security можно было перехватить HTTP-запрос на аутентификацию и проверку полномочий. Но в данном случае требуется защитить только веб-службу REST, и поэтому такой фильтр защиты применяется только к шаблону URL типа `/rest/*` (см. выше вызовы метода `antMatchers()`). И хотя защитить требуется URL всех веб-служб REST, пользователю все же должен быть разрешен доступ к главной странице веб-приложения (в данном случае — к простому HTML-файлу, отображаемому при обращении из браузера по адресу `http://localhost:8080/`). Именно в этот момент удобно ввести прикладной контекст `rest`, помимо конфигурационного класса `SecurityConfig`, в корневой контекст приложения.

```
package com.apress.prospring5.ch12.init;

import com.apress.prospring5.ch12.config.DataServiceConfig;
import org.springframework.web.servlet.support
     .AbstractAnnotationConfigDispatcherServletInitializer;

public class WebInitializer extends
     AbstractAnnotationConfigDispatcherServletInitializer {

  @Override
  protected Class<?>[] getRootConfigClasses() {
    return new Class<?>[] {
      DataServiceConfig.class, SecurityConfig.class
    };
  }

  @Override
  protected Class<?>[] getServletConfigClasses() {
    return new Class<?>[] {
      WebConfig.class
    };
  }

  @Override
  protected String[] getServletMappings() {
    return new String[]{"/rest/**"};
  }
}
```

На этом установка защиты завершается. Если развернуть рассматриваемый здесь проект снова и выполнить любой тестовый метод из класса `RestClientTest`, то на консоль будет выведен приведенный ниже результат (другие не относящиеся к делу результаты здесь опущены). Получаемый в итоге код состояния `401` по протоколу HTTP означает, что доступ к данной службе запрещен.

```
Exception in thread "main"
   org.springframework.web.cient.HttpClientErrorException:
   401 Unauthorized
```

А теперь перейдем к конфигурированию класса `RestTemplate` на стороне клиента, чтобы предоставить учетные данные серверу, как показано ниже.

```
package com.apress.prosring5.ch12;

import org.apache.http.HttpHost;
import org.apache.http.auth.AuthScope;
import org.apache.http.auth.Credentials;
import org.apache.http.auth.UsernamePasswordCredentials;
import org.apache.http.client.CredentialsProvider;
import org.apache.http.impl.client
          .BasicCredentialsProvider;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClientBuilder;
import org.apache.http.impl.client.HttpClients;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.http.MediaType;
import org.springframework.http.client
          .HttpComponentsClientHttpRequestFactory;
import org.springframework.http.converter
          .HttpMessageConverter;
import org.springframework.http.converter.xml
          .MarshallingHttpMessageConverter;
import org.springframework.oxm.castor.CastorMarshaller;
import org.springframework.web.client.RestTemplate;

import java.util.ArrayList;
import java.util.List;

@Configuration
public class RestClientConfig {

  @Autowired ApplicationContext ctx;

  @Bean Credentials credentials() {
    return new UsernamePasswordCredentials(
               "prospring5", "prospring5");
  }

  @Bean
```

```
  CredentialsProvider provider() {
    BasicCredentialsProvider provider =
                    new BasicCredentialsProvider();
    provider.setCredentials(AuthScope.ANY, credentials());
    return provider;
  }

  @Bean
  public HttpComponentsClientHttpRequestFactory
         httpRequestFactory() {
    CloseableHttpClient client = HttpClients
              .custom()
              .setDefaultCredentialsProvider(provider())
              .build();
    return new HttpComponentsClientHttpRequestFactory(
               client);
  }

  @Bean
  public RestTemplate restTemplate() {
    RestTemplate restTemplate = new RestTemplate();
    restTemplate.setRequestFactory(httpRequestFactory());
    List<HttpMessageConverter<?>> mcvs = new ArrayList<>();
    mcvs.add(singerMessageConverter());
    restTemplate.setMessageConverters(mcvs);
    return restTemplate;
  }

  @Bean MarshallingHttpMessageConverter
        singerMessageConverter() {
    MarshallingHttpMessageConverter mc =
               new MarshallingHttpMessageConverter();
    mc.setMarshaller(castorMarshaller());
    mc.setUnmarshaller(castorMarshaller());
    List<MediaType> mediaTypes = new ArrayList<>();
    MediaType mt = new MediaType("application", "xml");
    mediaTypes.add(mt);
    mc.setSupportedMediaTypes(mediaTypes);
    return mc;
  }

  @Bean CastorMarshaller castorMarshaller() {
    CastorMarshaller castorMarshaller =
                    new CastorMarshaller();
    castorMarshaller.setMappingLocation(ctx.getResource(
                    "classpath:spring/oxm-mapping.xml"));
    return castorMarshaller;
  }
}
```

Компонент `httpRequestFactory` внедряется в компонент `restTemplate` по ссылке, указываемой в аргументе конструктора. Для компонента `httpRequestFactory` используется класс `HttpComponentsClientHttpRequestFactory` из Spring, обеспечивающий поддержку библиотеки Apache HttpComponents HttpClient. Эта библиотека требуется для получения экземпляра типа `CloseableHttpClient`, предназначенного для хранения учетных данных клиента. Для поддержки внедрения учетных данных создается простой компонент Spring Bean, реализуемый в классе `UsernamePasswordCredentials`. Объект этого класса создается вместе с именем пользователя `prospring5` и таким же паролем. Как только компонент `httpRequestFactory` будет построен и внедрен в компонент `restTemplate`, все REST-запросы, инициируемые по данному шаблону, будут содержать предоставленные учетные данные. Теперь достаточно выполнить тестовые методы из класса `RestClientTest` снова, чтобы проверить, вызываются ли веб-службы, как обычно.

Реализация веб-служб REST средствами Spring Boot

Модуль Spring Boot заметно упрощает разработку, поэтому в этом разделе будет показано, как упростить разработку веб-служб REST с помощью этого модуля. Здесь используются, как и прежде, те же самые служебные классы, класс сущности `Singer`, информационное хранилище, не требующие никаких дополнительных изменений. Чтобы упростить дело и в как можно большей степени воспользоваться стандартной для модуля Spring Boot конфигурацией, в этом разделе будет также исключена сериализация в формате XML. Ведь по умолчанию в модуле Spring Boot поддерживается сериализация в формате JSON. А поскольку здесь речь пойдет о веб-приложении, то его конфигурация остается такой же, как представленная ранее, а следовательно, мы не будем обращаться к ней снова. Класс `Application` и точка входа в веб-приложение Spring Boot остаются такими же простыми, как показано ниже.

```
package com.apress.prospring5.ch12;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure
          .SpringBootApplication;
import org.springframework.context
          .ConfigurableApplicationContext;
import java.io.IOException;

@SpringBootApplication(scanBasePackages =
                       "com.apress.prospring5.ch12")
public class Application {
  private static Logger logger =
```

```
        LoggerFactory.getLogger(Application.class);
  public static void main(String args) throws IOException {
    ConfigurableApplicationContext ctx =
            SpringApplication.run(Application.class, args);
    assert (ctx != null);
    logger.info("Application Started ...");
    System.in.read();
    ctx.close();
  }
}
```

Как и было обещано ранее, ниже приведен новый и усовершенствованный класс `SingerController`, переписанный с помощью аннотации `@RestController`, а также специальных аннотаций для сопоставления с методами доступа по сетевому протоколу HTTP, внедренных в версии Spring 4.3.

```
package com.apress.prospring5.ch12.controller;

import com.apress.prospring5.ch12.entities.Singer;
import com.apress.prospring5.ch12.services.SingerService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.*;
import java.util.List;

@RestController
@RequestMapping(value = "/singer")
public class SingerController {

  final Logger logger =
        LoggerFactory.getLogger(SingerController.class);

  @Autowired
  private SingerService singerService;

  @ResponseStatus(HttpStatus.OK)
  @GetMapping(value = "/listdata")
  public List<Singer> listData() {
    return singerService.findAll();
  }

  @ResponseStatus(HttpStatus.OK)
  @GetMapping(value = "/{id}")
  public Singer findSingerById(@PathVariable Long id) {
    return singerService.findById(id);
  }
```

```
  @ResponseStatus(HttpStatus.CREATED)
  @PostMapping(value="/")
  public Singer create(@RequestBody Singer singer) {
    logger.info("Creating singer: " + singer);
    singerService.save(singer);
    logger.info("Singer created successfully with info: "
              + singer);
    return singer;
  }

  @ResponseStatus(HttpStatus.OK)
  @PutMapping(value="/{id}")
  public void update(@RequestBody Singer singer,
                     @PathVariable Long id) {
    logger.info("Updating singer: " + singer);
    singerService.save(singer);
    logger.info("Singer updated successfully with info: "
              + singer);
  }

  @ResponseStatus(HttpStatus.NO_CONTENT)
  @DeleteMapping(value="/{id}")
  public void delete(@PathVariable Long id) {
    logger.info("Deleting singer with id: " + id);
    Singer singer = singerService.findById(id);
    singerService.delete(singer);
    logger.info("Singer deleted successfully");
  }
}
```

В версии Spring 4.3 был внедрен ряд специальных разновидностей аннотаций `@RequestMapping`, предназначенных для сопоставления с основными методами доступа по сетевому протоколу HTTP. Соответствие новых и прежних аннотаций `@RequestMapping` приведено в табл. 12.4.

Таблица 12.4. Соответствие аннотаций, предназначенных для сопоставления с методами доступа и обработки запросов по сетевому протоколу HTTP

Новая аннотация	Прежняя аннотация
`@GetMapping`	`@RequestMapping(method = RequestMethod.GET)`
`@PostMapping`	`@RequestMapping(method = RequestMethod.POST`
`@PutMapping`	`@RequestMapping(method = RequestMethod.PUT)`
`@DeleteMapping`	`@RequestMapping(method = RequestMethod.DELETE)`

Кроме того, здесь применяется формат JSON, в котором поддерживаются списки и массивы, и поэтому класс `Singers` в данном случае больше не нужен. Проверить веб-приложение, разработанное с помощью модуля Spring Boot, нетрудно, поскольку для этого вообще не требуется конфигурировать класс `RestTemplate`, а достаточно

получить его экземпляр, вызвав конструктор по умолчанию. Тестовые методы остаются такими же, как и прежде.

```
package com.apress.prosring5.ch12.test;

import com.apress.prospring5.ch12.entities.Singer;
import org.junit.Before;
import org.junit.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.client.RestTemplate;

import java.util.Arrays;
import java.util.Date;
import java.util.GregorianCalendar;
import static org.junit.Assert.*;

public class RestClientTest {

  final Logger logger =
        LoggerFactory.getLogger(RestClientTest.class);

  private static final String URL_GET_ALL_SINGERS =
          "http://localhost:8080/singer/listdata";
  ...
  RestTemplate restTemplate;

  @Before
  public void setUp() {
    restTemplate = new RestTemplate();
  }

  @Test
  public void testFindAll() {
    logger.info("--> Testing retrieve all singers");
    Singer singers = restTemplate.getForObject(
                     URL_GET_ALL_SINGERS, Singer.class);
    assertTrue(singers.length == 3);
    listSingers(singers);
  }
  ...
}
```

Чтобы проверить данное веб-приложение Spring Boot, достаточно выполнить исходный код класса `Application`, а затем каждый из тестовых методов по очереди. Если же требуется убедиться, что данное веб-приложение работает нормально, а сериализация экземпляров, представляющих отдельных певцов, правильно выполняется в формате JSON, для этой цели можно воспользоваться средствами `curl`:

```
curl -v -H "Accept: application/json"
    http://localhost:8080/singer/listdata
* Trying ::1...
* Connected to localhost (::1) port 8080 (#0)
> GET /singer/listdata HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.43.0
> Accept: application/json
>
< HTTP/1.1 200
< Content-Type: application/json;charset=UTF-8
< Transfer-Encoding: chunked
< Date: Sun, 18 Jun 2017 11:14:17 GMT
<
* Connection #0 to host localhost left intact
[{"id":1, "version":1, "firstName":"John Clayton",
  "lastName":"Mayer", "birthDate":245797200000},
 {"id":2, "version":0, "firstName":"Eric",
  "lastName":"Clapton","birthDate":-781326000000},
 {"id":4, "version":0, "firstName":"BB", "last Name":"King",
  "birthDate":-924404400000}]
```

Если интерпретация полученного результата вызовет у вас определенные трудности, имейте в виду, что содержимое полей типа `Date` будет отображаться простыми числами без соответствующего форматирования, если не объявить явным образом преобразователь сообщений в формате JSON.

С помощью модуля Spring Boot можно также очень просто защитить ресурсы. Но этот вопрос будет более подробно рассмотрен в главе 16.

Применение протокола AMQP в Spring

Удаленную обработку можно также организовать, используя передачу удаленного вызова процедур (RPC) по транспортному протоколу AMQP (Advanced Message Queuing Protocol — расширенный протокол организации очередей сообщений). Этот открытый стандартный протокол служит для реализации промежуточного программного обеспечения, ориентированного на обмен сообщениями (MOM).

Приложение JMS пригодно для работы в любой операционной системе, но оно поддерживает только платформу Java, поэтому все приложения для обмена данными должны быть разработаны на Java. В частности, стандартный протокол AMQP может быть использован для разработки на разных языках приложений, которые могут легко обмениваться данными.

Как и в службе JMS, для обмена сообщениями в протоколе AMQP применяется брокер сообщений. В рассматриваемом здесь примере в качестве сервера AMQP применяется брокер сообщений RabbitMQ[12], поскольку в самом каркасе Spring отсут-

[12] См. `www.rabbitmq.org`.

ствуют функциональные возможности для организации удаленной обработки. Но вместо этого поддерживается родственный проект Spring AMQP[13], которым можно воспользоваться в качестве базового прикладного интерфейса API для обмена сообщениями. Проект Spring AMQP предоставляет базовую абстракцию для протокола AMQP, а также реализацию для установления связи с брокером сообщений RabbitMQ. В этой главе недостаточно места, чтобы описать все возможности протокола AMQP или проекта Spring AMQP, поэтому здесь будут рассмотрены только те функциональные возможности, которые имеются для организации удаленной обработки через передачу удаленных вызовов процедур (RPC).

Проект Spring AMQP делится на две части: `spring-amqp` — базовую абстракцию, а также `springrabbit` — реализацию брокера оообщений RabbitMQ. Самой устойчивой на момент написания данной книги считалась версия 2.0.0.M4 проекта Spring AMQP.

Прежде всего необходимо загрузить брокер сообщений RabbitMQ по адресу `www.rabbitmq.com/download.html` и запустить сервер. Никаких изменений в конфигурацию RabbitMQ вносить не придется, поскольку этот брокер сообщений изначально работает должным образом. Запустив RabbitMQ на выполнение, необходимо далее создать интерфейс удаленной службы. В рассматриваемом здесь примере будет создана простая метеослужба (Weather Service), возвращающая прогноз погоды по указанному коду штата. Итак, начнем с объявления интерфейса `WeatherService`, как показано ниже.

```
package com.apress.prospring5.ch12;

public interface WeatherService {
  String getForecast(String stateCode);
}
```

Далее реализуем интерфейс `WeatherService` в приведенном ниже классе, где в ответ на предоставленный код штата просто выдается прогноз погоды или сообщение в том случае, если прогноз отсутствует.

```
package com.apress.prospring5.ch12;
import org.springframework.stereotype.Component;

@Component
public class WeatherServiceImpl implements WeatherService {
  @Override
  public String getForecast(String stateCode) {
    if ("FL".equals(stateCode)) {
      return "Hot";
    } else if ("MA".equals(stateCode)) {
      return "Cold";
    }
```

[13] См. `http://projects.spring.io/spring-amqp`.

```
    return "Not available at this time";
  }
}
```

Реализовав метеослужбу типа `WeatherService`, перейдем к созданию файла конфигурации (`amqp-rpcapp-context.xml`), в котором настраивается связь по протоколу AMQP и открывается доступ к данной метеослужбе, как показано ниже.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi=
            "http://www.w3.org/2001/XMLSchema-instance"
       xmlns:rabbit=
            "http://www.springframework.org/schema/rabbit"
       xsi:schemaLocation=
          "http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans
                /spring-beans.xsd
          http://www.springframework.org/schema/rabbit
          http://www.springframework.org/schema/rabbit
                /spring-rabbit.xsd">

    <rabbit:connection-factory id="connectionFactory"
            host="localhost" />

    <rabbit:template id="amqpTemplate"
            connection-factory="connectionFactory"
            reply-timeout="2000" routing-key="forecasts"
            exchange="weather" />

    <rabbit:admin connection-factory="connectionFactory" />

    <rabbit:queue name="forecasts" />

    <rabbit:direct-exchange name="weather">
        <rabbit:bindings>
            <rabbit:binding queue="forecasts"
                    key="forecasts" />
            </rabbit:bindings>
        </rabbit:direct-exchange>

    <bean id="weatherServiceProxy"
      class="org.springframework.amqp.remoting.client
                .AmqpProxyFactoryBean">
      <property name="amqpTemplate" ref="amqpTemplate" />
      <property name="serviceInterface"
        value="com.apress.prospring5.ch12.WeatherService" />
    </bean>

    <rabbit:listener-container
           connection-factory="connectionFactory">
```

```
    <rabbit:listener ref="weatherServiceExporter"
           queue-names="forecasts" />
    </rabbit:listener-container>
    <bean id="weatherServiceExporter"
       class="org.springframework.amqp.remoting.service
                .AmqpInvokerServiceExporter">
       <property name="amqpTemplate" ref="amqpTemplate" />
       <property name="serviceInterface"
           value="com.apress.prospring5.ch12
                      .WeatherService" />
       <property name="service">
           <bean class="com.apress.prospring5.ch12
                           .WeatherServiceImpl"/>
       </property>
    </bean>
</beans>
```

Сначала в приведенной выше конфигурации настраивается соединение с брокером сообщений RabbitMQ, а также информация для обмена и организации очереди. Затем с помощью класса `AmqpProxyFactoryBean` создается компонент Spring Bean, которым клиент может воспользоваться как заместителем, делая запрос RPC. А для ответа на этот запрос используется класс `AmqpInvokerServiceExporter`, который связывается с контейнером приемника сообщений. Этот контейнер отвечает за перехват сообщений AMQP и их передачу удаленной метеослужбе. Как видите, эта конфигурация очень похожа на конфигурацию службы JMS в отношений подключений, очередей, контейнеров приемников сообщений и т.д. Но, несмотря на сходство в конфигурации, служба JMS и протокол AMQP являются совершенно разными средствами обмена сообщениями на транспортном уровне, поэтому рекомендуется посетить веб-сайт AMQP[14], чтобы подробнее ознакомиться с этим протоколом.

Итак, подготовив описанную выше конфигураци, остается лишь создать пример класса для выполнения вызовов RPC, как показано ниже.

```
package com.apress.prospring5.ch12;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.support
          .GenericXmlApplicationContext;

public class AmqpRpcDemo {
  private static Logger logger =
          LoggerFactory.getLogger(AmqpRpcDemo.class);
  public static void main(String... args) {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/amqp-rpc-app-context.xml");
```

[14] См. по адресу `www.amqp.org`.

```
    ctx.refresh();

    WeatherService weatherService =
                  ctx.getBean(WeatherService.class);
    logger.info("Forecast for FL: "
              + weatherService.getForecast("FL"));
    logger.info("Forecast for MA: "
              + weatherService.getForecast("MA"));
    logger.info("Forecast for CA: "
              + weatherService.getForecast("CA"));
    ctx.close();
  }
}
```

Если выполнить приведенный выше исходный код, то на консоль будет выведен следующий результат:

```
INFO c.a.p.c.AmqpRpcDemo - Forecast for FL: Hot
INFO c.a.p.c.AmqpRpcDemo - Forecast for MA: Cold
INFO c.a.p.c.AmqpRpcDemo - Forecast for CA:
     Not available at this time
```

Разумеется, упомянутую выше конфигурацию можно без особого труда преобразовать в конфигурационные классы Java. Но для этого придется внести ряд изменений и в другие классы. В частности, реализовывать интерфейс `WeatherService` в классе `WeatherServiceImpl` больше не придется, поскольку в нем лишь объявляется метод для приема сообщений, размещаемых в очереди `forecasts`.

```
package com.apress.prospring5.ch12;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.amqp.rabbit.annotation
          .RabbitListener;
import org.springframework.stereotype.Service;

@Service
public class WeatherServiceImpl {

  private static Logger logger =
    LoggerFactory.getLogger(WeatherServiceImpl.class);
  @RabbitListener(containerFactory=
                  "rabbitListenerContainerFactory",
                  queues="forecasts")
  public void getForecast(String stateCode) {
    if ("FL".equals(stateCode)) {
      logger.info("Hot");
    } else if ("MA".equals(stateCode)) {
      logger.info("Cold");
    } else {
```

```
      logger.info("Not available at this time");
    }
  }
}
```

Компонент `rabbitListenerContainerFactory` относится к типу `RabbitListenerContainerFactory` и служит для создания обычного контейнера приемников сообщений типа `SimpleMessageListenerContainer`. Впрочем, вся конфигурация на Java для рассматриваемого здесь примера приведена ниже.

```
package com.apress.prospring5.ch12.config;

import com.apress.prospring5.ch12.WeatherService;
import com.apress.prospring5.ch12.WeatherServiceImpl;
import org.springframework.amqp.core.*;
import org.springframework.amqp.rabbit.annotation
          .EnableRabbit;
import org.springframework.amqp.rabbit.config
          .SimpleRabbitListenerContainerFactory;
import org.springframework.amqp.rabbit.connection
          .CachingConnectionFactory;
import org.springframework.amqp.rabbit.core.RabbitAdmin;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.amqp.rabbit.listener
          .SimpleMessageListenerContainer;
import org.springframework.amqp.remoting.client
          .AmqpProxyFactoryBean;
import org.springframework.amqp.remoting.service
          .AmqpInvokerServiceExporter;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.apress.prospring5.ch12")
@EnableRabbit
public class RabbitMQConfig {

  final static String queueName = "forecasts";
  final static String exchangeName = "weather";

  @Bean CachingConnectionFactory connectionFactory() {
    return new CachingConnectionFactory("127.0.0.1");
  }

  @Bean RabbitTemplate amqpTemplate() {
    RabbitTemplate rabbitTemplate = new RabbitTemplate();
    rabbitTemplate.setConnectionFactory(
```

```
                    connectionFactory());
    rabbitTemplate.setReplyTimeout(2000);
    rabbitTemplate.setRoutingKey(queueName);
    rabbitTemplate.setExchange(exchangeName);
    return rabbitTemplate;
  }

  @Bean Queue forecasts() {
    return new Queue(queueName, true);
  }

  @Bean Binding dataBinding(DirectExchange directExchange,
                            Queue queue) {
    return BindingBuilder.bind(queue).to(directExchange)
                         .with(queueName);
  }

  @Bean RabbitAdmin admin() {
    RabbitAdmin admin =
                new RabbitAdmin(connectionFactory());
    admin.declareQueue(forecasts());
    admin.declareBinding(
          dataBinding(weather(), forecasts()));
    return admin;
  }

  @Bean DirectExchange weather() {
    return new DirectExchange(exchangeName, true, false);
  }

  @Bean
  public SimpleRabbitListenerContainerFactory
                      rabbitListenerContainerFactory() {
    SimpleRabbitListenerContainerFactory factory =
                new SimpleRabbitListenerContainerFactory();
    factory.setConnectionFactory(connectionFactory());
    factory.setMaxConcurrentConsumers(5);
    return factory;
  }
}
```

Все компоненты Spring Beans в приведенной выше конфигурации нетрудно сопоставить с их аналогами из конфигурации в формате XML. Новым элементом в данной конфигурации является лишь аннотация `@EnableRabbit`. Если она употребляется в классе, снабженном аннотацией `@Configuration`, то активизирует аннотированные конечные точки приемника сообщений от RabbitMQ, которые подспудно создаются компонентом Spring Bean типа `RabbitListenerContainerFactory`.

Чтобы проверить вновь созданную метеослужбу, необходимо также внести соответствующие коррективы в тестовую программу, а также в метод `amqpTemplate()`,

предназначенный для отправки сообщений в очередь `forecasts`, откуда они извлекаются методом `WeatherServiceImpl.getForecast()` и выводятся как прогноз погоды на консоль.

```
package com.apress.prospring5.ch12;

import com.apress.prospring5.ch12.config.RabbitMQConfig;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support
          .GenericApplicationContext;

public class AmqpRpcDemo {
  public static void main(String... args)
         throws Exception {
    GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               RabbitMQConfig.class);
    RabbitTemplate rabbitTemplate =
                   ctx.getBean(RabbitTemplate.class);
    rabbitTemplate.convertAndSend("FL");
    rabbitTemplate.convertAndSend("MA");
    rabbitTemplate.convertAndSend("CA");

    System.in.read();
    ctx.close();
  }
}
```

Если запустить сервер RabbitMQ и выполнить приведенную выше тестовую программу, то на консоль будет выведен следующий результат:

```
[SimpleAsyncTaskExecutor-1]
       INFO c.a.p.c.WeatherServiceImpl - Hot
[SimpleAsyncTaskExecutor-1]
       INFO c.a.p.c.WeatherServiceImpl - Cold
[SimpleAsyncTaskExecutor-1]
       INFO c.a.p.c.WeatherServiceImpl -
       Not available at this time
```

Применение протокола AMQP вместе с модулем Spring Boot

Модуль Spring Boot помогает также разрабатывать приложения AMQP, и для этой цели служит его стартовая библиотека `spring-boot-starter-amqp`. Значительно упрощается и конфигурирование подобных приложений. В частности, определять компоненты Spring Beans типа `RabbitTemplate`, `RabbitAdmin` и `SimpleRabbit`

ListenerContainerFactory больше не придется, поскольку они создаются и конфигурируются в модуле Spring Boot автоматически. В данном случае реализация класса WeatherServiceImpl из упомянутого выше примера удаленной метеослужбы также не претерпит заметных изменений, как показано ниже. Но поскольку за создание и конфигурирование компонента Spring Bean типа SimpleRabbitListener ContainerFactory теперь отвечает модуль Spring Boot, то его больше не нужно указывать как значение в аннотации @RabbitListener.

```
package com.apress.prospring5.ch12;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.amqp.rabbit.annotation
          .RabbitListener;
import org.springframework.stereotype.Service;

@Service
public class WeatherServiceImpl {

  private static Logger logger =
       LoggerFactory.getLogger(WeatherServiceImpl.class);

  @RabbitListener(queues="forecasts")
  public void getForecast(String stateCode) {
    if ("FL".equals(stateCode)) {
      logger.info("Hot");
    } else if ("MA".equals(stateCode)) {
      logger.info("Cold");
    } else {
      logger.info("Not available at this time");
    }
  }
}
```

Класс Application, снабженный аннотацией @SpringBootApplication, служит одновременно в качестве конфигурационного и исполняющего тесты класса, как показано ниже.

```
package com.apress.prospring5.ch12;

import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.DirectExchange;
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.rabbit.connection
          .CachingConnectionFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.amqp.rabbit.listener
          .SimpleMessageListenerContainer;
```

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure
          .SpringBootApplication;
import org.springframework.context
         .ConfigurableApplicationContext;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class Application {
  final static String queueName = "forecasts";
  final static String exchangeName = "weather";

  @Bean Queue forecasts() {
    return new Queue(queueName, true);
  }

  @Bean DirectExchange weather() {
    return new DirectExchange(exchangeName, true, false);
  }

  @Bean Binding dataBinding(DirectExchange directExchange,
                            Queue queue) {
    return BindingBuilder.bind(queue).to(directExchange)
                         .with(queueName);
  }

  @Bean CachingConnectionFactory connectionFactory() {
    return new CachingConnectionFactory("127.0.0.1");
  }

  @Bean
  SimpleMessageListenerContainer
              messageListenerContainer() {
    SimpleMessageListenerContainer container =
                new SimpleMessageListenerContainer();

    container.setConnectionFactory(connectionFactory());
    container.setQueueNames(queueName);
    return container;
  }

  public static void main(String... args)
         throws java.lang.Exception {
    ConfigurableApplicationContext ctx =
         SpringApplication.run(Application.class, args);

    RabbitTemplate rabbitTemplate =
                  ctx.getBean(RabbitTemplate.class);
    rabbitTemplate.convertAndSend(Application.queueName, "FL");
    rabbitTemplate.convertAndSend(Application.queueName, "MA");
```

```
    rabbitTemplate.convertAndSend(Application.queueName, "CA");

    System.in.read();
    ctx.close();
  }
}
```

Как видите, аннотация `@EnableRabbit` больше не нужна, и хотя конфигурация от этого не особенно сократилась, она все же улучшилась. Если выполнить исходный код приведенного выше класса, то полученный результат окажется практически таким же, как и в предыдущих примерах.

```
DEBUG c.a.p.c.Application -
   Running with Spring Boot v2.0.0.M1, Spring v5.0.0.RC1
INFO c.a.p.c.Application - No active profile set,
   falling back to default profiles: default
INFO c.a.p.c.Application - Started Application
   in 2.211 seconds JVM running for 2.801
[SimpleAsyncTaskExecutor-1]
   INFO c.a.p.c.WeatherServiceImpl - Cold
[SimpleAsyncTaskExecutor-1]
   INFO c.a.p.c.WeatherServiceImpl - Hot
[SimpleAsyncTaskExecutor-1]
   INFO c.a.p.c.WeatherServiceImpl -
   Not available at this time
```

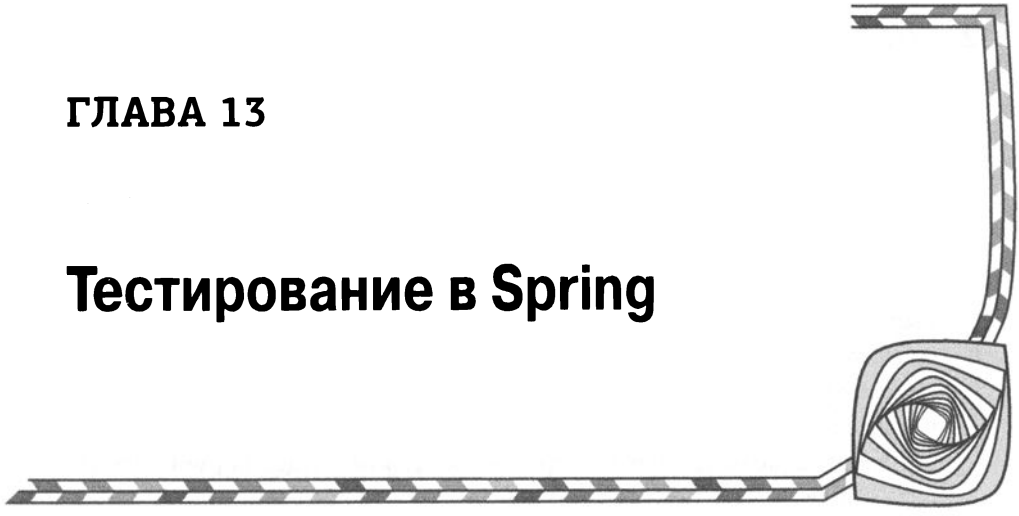
Резюме

В этой главе были представлены наиболее употребительные технологии удаленной обработки в приложениях Spring. Если взаимодействующие приложения построены на основе Spring, в таком случае удобно воспользоваться средством вызова HTTP Spring. Если же требуется асинхронный режим работы или слабо связанная интеграция, то для этих целей обычно выбирается служба JMS. В этой главе было также показано, как пользоваться веб-службами REST в Spring для удаленного доступа к службам, в том числе и с помощью класса `RestTemplate`. И, наконец, в этой главе пояснялось применение проекта Spring AMQP для организации удаленной обработки в стиле RPC с помощью брокера сообщений RabbitMQ. Чтобы упростить применение каждой из рассмотренных в этой главе технологий, включая RPC, REST и JMS, при разработке приложений в Spring можно всегда прибегнуть к услугам модуля Spring Boot.

В следующей главе речь пойдет о применении Spring для тестирования приложений. В ней будут рассмотрены методики тестирования, упрощающие задачу разработчиков.

ГЛАВА 13

Тестирование в Spring



При разработке корпоративных приложений тестирование позволяет воочию убедиться, что завершенное приложение действует именно так, как и предполагалось, и удовлетворяет всем требованиям, включая архитектурные, пользовательские и пр. Всякий раз, когда в прикладной код вносятся изменения, необходимо удостовериться, что они не повлияли на уже реализованную логику. Поддерживать среду построения и тестирования в рабочем состоянии крайне важно для обеспечения высокого качества приложений. Воспроизводимые тесты с большой степенью покрытия проверяемого кода позволяют развертывать новые приложения и изменения в них с высоким уровнем доверительности.

В среде разработки корпоративных приложений имеется немало видов тестирования, нацеленных на каждый уровень корпоративного приложения. У каждого вида тестирования свои особенности и требования. В этой главе представлены основные понятия, связанные с тестированием приложений на разных уровнях и, в частности, приложений Spring. В ней также поясняется, каким образом каркас Spring упрощает задачу разработчиков по реализации контрольных примеров для тестирования приложений на разных уровнях. В этой главе будут, в частности, рассмотрены следующие вопросы.

- **Среда тестирования корпоративных приложений.** В этой части вкратце описывается среда тестирования корпоративных приложений. Здесь будут представлены разные виды тестирования и их назначение. Основное внимание будет уделено модульному тестированию, нацеленному на разные уровни приложения.
- **Модульное тестирование логики.** Самый мелкий модульный тест предполагает проверку только логики выполнения методов в отдельном классе, а для всех остальных зависимостей имитируется корректное поведение. В этой части

описано, каким образом модульное тестирование логики работы классов, реализующих контроллеры из модуля Spring MVC, осуществляется с помощью библиотеки Java, имитирующей корректное поведение зависимостей этих классов.

- **Комплексно-модульное тестирование.** В среде тестирования корпоративных приложений проверка взаимодействия группы классов осуществляется на разных уровнях приложения для отдельной части бизнес-логики. Как правило, в среде комплексного тестирования уровень обслуживания должен проверяться вместе с уровнем сохраняемости, т.е. с доступной на сервере базой данных. Но по мере развития архитектуры приложений и обретения зрелости облегченных баз данных, размещаемых в оперативной памяти, общей нормой практики теперь становится модульное тестирование уровня обслуживания вместе с уровнем сохраняемости и базой данных на сервере как единого целого. Например, в этой главе прикладной интерфейс JPA 2 будет использован вместе с библиотекой Hibernate и прикладным интерфейсом Spring Data JPA в качестве поставщика услуг сохраняемости, а H2 — в качестве базы данных. В такой архитектуре менее важно имитировать Hibernate и Spring Data JPA при тестировании уровня обслуживания. Поэтому в этой части главы тестирование уровня обслуживания будет рассмотрено вместе с уровнем сохраняемости и базой данных H2, размещаемой в оперативной памяти. Такой вид тестирования обычно называют *комплексно-модульным тестированием*, которое находится посредине между модульным и полноценным комплексным тестированием.
- **Модульное тестирование клиентской части веб-приложений.** Даже если тестируется каждый уровень приложения, после его развертывания все равно необходимо удостовериться, что все приложение функционирует именно так, как и предполагалось. В частности, после развертывания веб-приложения в среде непрерывной сборки следует провести тестирование его клиентской части, чтобы проверить, правильно ли работает пользовательский интерфейс. Например, в приложении для певцов необходимо удостовериться, что функциональные средства нормально работают на каждой стадии выполнения, а также протестировать исключительные ситуации, когда, например, введенные пользователем данные не проходят стадию проверки достоверности. В этой части вкратце описана среда тестирования клиентской части веб-приложений.

Описание разных видов тестирования

Под *средой тестирования корпоративных приложений* понимаются тестирующие действия во всем жизненном цикле приложения в целом. На разных стадиях этого процесса будут выполняться разные тестирующие действия, цель которых — проверить, работают ли функциональные средства приложения предполагаемым образом в соответствии с заданными требованиями предметной области и технического состояния.

На каждой стадии тестирования выполняются разные контрольные примеры. Одни из них автоматизированы, тогда как другие выполняются вручную. Но в любом случае полученные результаты проверяются соответствующим персоналом (например, бизнес-аналитиками, пользователями приложения и т.д.). В табл. 13.1 описаны характеристики и цели каждого вида тестирования, а также общие инструментальные средства и библиотеки, применяемые для реализации контрольных примеров.

Таблица 13.1. Разные виды тестирования, применяемые на практике

Вид тестирования	Описание	Типичные инструментальные средства
Модульное тестирование логики	Проверяет отдельный объект, не принимая во внимание его роль в окружающей системе	Модульное тестирование: JUnit, TestNG Имитирующие объекты: Mockito, EasyMock
Комплексно-модульное тестирование	Проверяет взаимодействие компонентов в среде, близкой к реальной. В ходе такого тестирования осуществляется взаимодействие с контейнером (встроенной базой данных, веб-контейнером и т.д.)	Встроенная база данных: H2 Тестирование баз данных: DbUnit Веб-контейнер в оперативной памяти: Jetty
Модульное тестирование клиентской части веб-приложений	Проверяет пользовательский интерфейс и преследует цель убедиться, что каждый пользовательский интерфейс правильно реагирует на действия пользователей и формирует выводимый для них результат	Selenium
Непрерывная сборка и контроль качества	Кодовая база приложения должна собираться регулярно, чтобы обеспечивать соответствие качества кода существующему стандарту (например, наличие комментариев, отсутствие пустых блоков перехвата исключений и т.п.). Кроме того, степень покрытия тестами должна быть как можно более высокой, чтобы гарантировать проверку всех строк проверяемого кода	Качество кода: PMD, Checkstyle, FindBugs, Sonar Покрытие тестами: Cobertura, EclEmma Инструментальные средства сборки: Gradle, Maven Непрерывная сборка: Hudson, Jenkins
Комплексное тестирование системы	Проверяет правильность взаимодействия всех программ в новой системе, а также новой системы со всеми внешними интерфейсами. Такое тестирование должно также подтвердить, что новая система работает в соответствии со спецификацией функциональных требований и эффективно действует в операционной среде, не оказывая отрицательного влияния на другие системы	IBM Rational Functional Tester, HP Unified Functional Testing

Окончание табл. 13.1

Вид тестирования	Описание	Типичные инструментальные средства
Контроль качества системы	Проверяет, удовлетворяет ли разработанное приложение ряду нефункциональных требований. При этом зачастую выполняется тестирование производительности приложения, чтобы удостовериться, удовлетворяются ли требования к параллельной работе пользователей в системе и к рабочей нагрузке. К числу других проверяемых нефункциональных требований относятся безопасность, высокая доступность и т.п.	Apache JMeter, HP LoadRunner
Приемочное пользовательское тестирование	Эмулирует действительные условия эксплуатации новой системы, в том числе руководства пользователя и процедуры работы в системе. Широкое привлечение пользователей на данной стадии тестирования дает им неоценимый опыт работы в ней, а программистам и проектировщикам — возможность оценить восприятие новых программ конечным пользователем. Такое совместное участие в тестировании способствует более быстрому переходу пользователей и технического персонала на новую систему	IBM Rational TestManager, HP QualityCenter

В этой главе основное внимание уделяется реализации трех видов модульного тестирования (логики, клиентской части и в комплексе), а также показано, каким образом каркас Spring TestContext и другие инструментальные средства и библиотеки, поддерживающие модульное тестирование, могут оказать помощь в разработке контрольных примеров. Вместо того чтобы подробно описывать все классы, предоставляемые в Spring TestContext для тестирования, по ходу описания контрольных примеров в этой главе будут рассмотрены наиболее употребительные шаблоны, поддерживающие интерфейсы и классы из Spring TestContext.

Применение тестовых аннотаций в Spring

Прежде чем переходить к тестам логики и комплексным тестам, следует отметить, что, помимо стандартных аннотаций (вроде `@Autowired` и `@Resource`), в Spring предоставляются аннотации, специально предназначенные для тестирования. Эти аннотации можно применять в тестах логики, модульных и комплексных тестах, предоставляя самые разные функциональные возможности, в том числе упрощенную загрузку файла контекста, профили, синхронизацию выполнения тестов и многое другое. В табл. 13.2 сведены тестовые аннотации и примеры их применения.

Таблица 13.2. Тестовые аннотации в Spring

Аннотация	Описание
`@ContextConfiguration`	Аннотация, действующая на уровне классов и позволяющая определить способ загрузки и конфигурирования контекста типа **`ApplicationContext`** для комплексного тестирования
`@WebAppConfiguration`	Аннотация, действующая на уровне классов и позволяющая указать, что загружаемым контекстом приложения должен быть контекст типа **`WebApplicationContext`**
`@ContextHierarchy`	Аннотация, действующая на уровне классов и позволяющая указать активный профиль компонента Spring Bean
`@DirtiesContext`	Аннотация, действующая на уровне классов и методов и позволяющая указать, что во время выполнения теста контекст был каким-то образом модифицирован или поврежден, и поэтому он должен быть закрыт и построен заново для последующих тестов
`@TestExecutionListeners`	Аннотация, действующая на уровне классов и предназначенная для конфигурирования экземпляров типа **`TestExecutionListener`**, которые должны быть зарегистрированы диспетчером типа **`TestContextManager`**
`@TransactionConfiguration`	Аннотация, действующая на уровне классов и позволяющая сконфигурировать транзакцию, например, установить параметры отката и диспетчер транзакций (если только требуемый диспетчер транзакций не носит имя компонента **`transactionManager`**)
`@Rollback`	Аннотация, действующая на уровне классов и методов и позволяющая указать, должен ли быть произведен откат транзакции для снабженного ею тестового метода. Аннотации, действующие на уровне классов, служат для тестирования стандартных параметров классов
`@BeforeTransaction`	Аннотация, действующая на уровне методов и позволяющая указать, что снабженный ею метод должен быть вызван перед началом транзакции для тестовых методов, помеченных аннотацией **`@Transactional`**
`@AfterTransaction`	Аннотация, действующая на уровне методов и позволяющая указать, что снабженный ею метод должен быть вызван по завершении транзакции для тестовых методов, помеченных аннотацией **`@Transactional`**
`@IfProfileValue`	Аннотация, действующая на уровне классов и методов и позволяющая указать, что выполнение тестового метода должно быть разрешено для конкретного ряда внешних условий
`@ProfileValueSourceConfiguration`	Аннотация, действующая на уровне классов и позволяющая указать источник значений профиля типа **`ProfileValueSource`**, используемый в аннотации **`@IfProfileValue`**. Если эта аннотация не объявлена в тесте, по умолчанию применяется источник значений системного профиля типа **`SystemProfileValueSource`**
`@Timed`	Аннотация, действующая на уровне методов и позволяющая указать, что тест должен быть завершен в течение заданного периода времени

Окончание табл. 13.2

Аннотация	Описание
`@Repeat`	Аннотация, действующая на уровне методов и позволяющая указать, что снабженный ею тестовый метод должен быть повторен заданное количество раз

Реализация модульных тестов логики

Как пояснялось ранее, модульный тест логики действует на самом мелком уровне тестирования. Его назначение — проверить поведение отдельного класса, а предполагаемое поведение всех зависимостей этого класса имитируется. В этом разделе применение модульного теста логики будет продемонстрировано на конкретных контрольных примерах тестирования класса `ContactController` с имитацией предполагаемого поведения на уровне обслуживания. Для целей имитации предполагаемого поведения на уровне обслуживания будет использован широко распространенный каркас имитации Mockito (`http://site.mockito.org/`).

В модуле `spring-tes`, входящем в состав каркаса Spring Framework, обеспечивается первоклассная поддержка комплексного тестирования. Чтобы предоставить контекст для комплексных тестов, демонстрируемых в этом разделе, мы воспользуемся библиотекой `spring-test.jar`, в которой имеются классы, особенно ценные для комплексного тестирования с помощью контейнера инверсии управления в Spring.

Внедрение требующихся зависимостей

Прежде всего в рассматриваемый здесь проект необходимо внедрить требующиеся зависимости, как показано в приведенном ниже фрагменте кода конфигурации. Кроме того, данный проект будет построен на основании классов и интерфейсов, упоминавшихся в примерах из предыдущих глав книги, в том числе `Singer`, `Singer Service` и пр.

```
// Файл конфигурации pro-spring-15\build.gradle
ext {
    // Библиотеки Spring
    springVersion = '5.0.0.RC1'
    bootVersion = '2.0.0.M1'

    // Библиотеки для тестирования
    mockitoVersion = '2.0.2-beta'
    junitVersion = '4.12'
    hamcrestVersion = '1.3'
    dbunitVersion = '2.5.3'
    poiVersion = '3.16'
    junit5Version = '5.0.0-M4'
```

```
    spring = [
      test : "org.springframework:spring-test:
               $springVersion",
      ...
    ]

    boot = [
        starterTest : "org.springframework.boot:
                       spring-boot-starter-test:
                       $bootVersion",
        ...
    ]

    testing = [
        junit : "junit:junit:$junitVersion",
        junit5 : "org.junit.jupiter:junit-jupiter-engine:
                  $junit5Version",
        junitJupiter: "org.junit.jupiter:junit-jupiter-api:
                       $junit5Version",
        mockito : "org.mockito:mockito-all:$mockitoVersion",
        easymock : "org.easymock:easymock:3.4",
        jmock : "org.jmock:jmock:2.8.2",
        hamcrestCore: "org.hamcrest:hamcrest-core:
                       $hamcrestVersion",
        hamcrestLib : "org.hamcrest:hamcrest-library:
                       $hamcrestVersion",
        dbunit : "org.dbunit:dbunit:$dbunitVersion"
    ]

    misc = [
        ...
        poi : "org.apache.poi:poi:$poiVersion"
    ]
    ...
}
```

Модульное тестирование контроллеров Spring MVC

На уровне представления классы контроллеров обеспечивают интеграцию пользовательского интерфейса с уровнем обслуживания. Методы из классов контроллеров будут сопоставляться с HTTP-запросами. В теле такого метода запрос обрабатывается, привязывается к объектам модели и взаимодействует с уровнем обслуживания, внедренным в классы контроллеров с помощью механизма внедрения зависимостей в Spring, для последующей обработки данных. В зависимости от результата обработки класс контроллера обновляет модель и состояние представления (например, пользовательские сообщения, объекты для веб-служб REST и т.п.) и возвращает логичес-

кое представление (или модель вместе с представлением) модулю Spring MVC, чтобы выявить конкретное представление для его отображения пользователю.

Основная цель модульного тестирования классов контроллеров — проверить, правильно ли методы контроллеров обновляют модель и состояния представлений и возвращают корректные представления. В данном случае требуется проверить только поведение классов контроллеров, и поэтому необходимо сымитировать уровень обслуживания с корректным поведением. Так, для тестирования методов `listData()` и `create()` из класса `SingerController` требуется разработать подходящие контрольные примеры. Необходимые для этого действия будут описаны в последующих разделах.

Тестирование метода `listData()`

Создадим первый контрольный пример для тестирования метода `ContactController.listData()`. Нам надо удостовериться, что, когда вызван тестируемый метод, после извлечения списка певцов из уровня обслуживания сведения о них правильно сохраняются в модели, а также возвращаются нужные объекты. Такой контрольный пример приведен в следующем фрагменте кода:

```
package com.apress.prospring5.ch13;

import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

import java.util.ArrayList;
import java.util.List;

import com.apress.prospring5.ch13.entities.Singer;
import com.apress.prospring5.ch13.entities.Singers;
import org.junit.Before;
import org.junit.Test;

import org.mockito.invocation.InvocationOnMock;
import org.mockito.stubbing.Answer;
import org.springframework.test.util.ReflectionTestUtils;
import org.springframework.ui.ExtendedModelMap;

public class SingerControllerTest {
  private final List<Singer> singers = new ArrayList<>();

  @Before
  public void initSingers() {
    Singer singer = new Singer();
    singer.setId(11);
    singer.setFirstName("John");
    singer.setLastName("Mayer");
```

```
    singers.add(singer);
  }

  @Test
  public void testList() throws Exception {
    SingerService singerService =
                  mock(SingerService.class);
    when(singerService.findAll()).thenReturn(singers);

    SingerController singerController =
                    new SingerController();

    ReflectionTestUtils.setField(singerController,
                        "singerService", singerService);

    ExtendedModelMap uiModel = new ExtendedModelMap();
    uiModel.addAttribute("singers",
                         singerController.listData());

    Singers modelSingers = (Singers) uiModel.get("singers");

    assertEquals(1, modelSingers.getSingers().size());
  }
}
```

Во-первых, в данном контрольном примере вызывается метод `initSingers()`, к которому применяется аннотация `@Before`, указывающая платформе JUnit, что этот метод должен выполняться перед запуском каждого контрольного примера (в том случае, если определенную логику требуется выполнить непосредственно перед самим тестовым классом, следует воспользоваться аннотацией `@BeforeClass`). В методе `initSingers()` список певцов инициализируется жестко закодированными значениями.

Во-вторых, в методе `testList()` применяется аннотация `@Test`, указывающая на то, что это контрольный пример, который должен быть запущен в JUnit. В этом контрольном примере закрытая переменная `singerService` типа `SingerService` имитируется с помощью метода `Mockito.mock()` из каркаса Mockito (обратите внимание на оператор `import static`). Для имитации метода `SingerService.findAll()`, который будет использоваться в классе `SingerController`, в Mockito предоставляется также метод `when()`.

В-третьих, в данном контрольном примере получается экземпляр типа `SingerController`, а в его переменной `singerService`, внедряемой Spring в нормальных ситуациях, устанавливается экземпляр, имитируемый с помощью метода `setField()` из класса `ReflectionTestUtils`, предоставляемого в Spring. В этом классе на основании рефлексии предоставляется ряд служебных методов, предназначенных для применения в контрольных примерах модульного и комплексного тестиро-

вания. Кроме того, конструируется экземпляр класса `ExtendedModelMap`, реализующего интерфейс `org.springframework.ui.Model`.

Далее вызывается метод `SingerController.listData()`. Результат его вызова проверяется с помощью различных методов утверждений, предоставляемых в JUnit, чтобы удостовериться в правильности сохранения списка певцов в модели, применяемой в представлении.

Теперь контрольный пример можно запустить, и он должен выполниться успешно. В этом можно убедиться с помощью системы сборки или IDE. Перейдем далее к тестированию метода `create()`.

Тестирование метода `create()`

Ниже приведен фрагмент кода, предназначенный для тестирования метода `create()`.

```
package com.apress.prospring5.ch13;

import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;
import java.util.ArrayList;
import java.util.List;

import com.apress.prospring5.ch13.entities.Singer;
import com.apress.prospring5.ch13.entities.Singers;
import org.junit.Before;
import org.junit.Test;

import org.mockito.invocation.InvocationOnMock;
import org.mockito.stubbing.Answer;
import org.springframework.test.util.ReflectionTestUtils;
import org.springframework.ui.ExtendedModelMap;

public class SingerControllerTest {
  private final List<Singer> singers = new ArrayList<>();

  @Test
  public void testCreate() {
    final Singer newSinger = new Singer();
    newSinger.setId(999l);
    newSinger.setFirstName("BB");
    newSinger.setLastName("King");
    SingerService singerService =
                  mock(SingerService.class);
     when(singerService.save(newSinger))
                      .thenAnswer(new Answer<Singer>() {
       public Singer answer(InvocationOnMock invocation)
                    throws Throwable {
```

```
            singers.add(newSinger);
            return newSinger;
        }
      });

    SingerController singerController =
                    new SingerController();
    ReflectionTestUtils.setField(singerController,
                        "singerService", singerService);
    Singer singer = singerController.create(newSinger);
    assertEquals(Long.valueOf(9991), singer.getId());
    assertEquals("BB", singer.getFirstName());
    assertEquals("King", singer.getLastName());
    assertEquals(2, singers.size());
  }
}
```

Метод `SingerService.save()` имитируется с целью смоделировать ввод нового объекта типа `Singer` в список певцов. Обратите внимание на применение интерфейса `org.mockito.stubbing.Answer<T>` для имитации данного метода с предполагаемой логикой работы и возврата из него значения.

Затем вызывается метод `SingerController.create()`, после чего с помощью операций утверждения проверяется результат. Запустите тестовый сценарий снова и обратите внимание на результаты его выполнения.

Для тестирования метода `create()` необходимо создать дополнительные контрольные примеры, чтобы проверить самые разные ситуации. Например, необходимо проверить тот случай, когда во время операции сохранения возникают ошибки доступа к данным.

Все, что было рассмотрено до сих пор, можно было бы сделать и с помощью библиотеки JMock (`www.jmock.org/`), и версия класса `SingerControllerTest`, где применяется библиотека JMock, включена в пример кода из этого раздела. Но мы не будем останавливаться на этом подробно. Ведь наша цель — пояснить принцип имитации зависимостей, а не продемонстрировать библиотеку, с помощью которой это можно было бы сделать.[1]

Реализация комплексного тестирования

В этом разделе нам предстоит реализовать комплексное тестирование на уровне обслуживания. Основные услуги в приложении для певцов предоставляются в классе `SingerServiceImpl`, реализующем интерфейс `SingerService` средствами JPA.

Для размещения модели данных и хранения тестовой информации в ходе модульного тестирования на уровне обслуживания будет использоваться база данных H2,

[1] В качестве еще одной альтернативы можно было бы воспользоваться библиотекой EasyMock (`http://easymock.org/`).

функционирующая в оперативной памяти вместе с поставщиками услуг JPA (библиотекой Hibernate и абстракцией информационного хранилища в Spring Data JPA). Цель состоит в том, чтобы удостовериться, что класс `SingerServiceImpl` правильно выполняет свои рабочие функции. В последующих разделах будет показано, как протестировать методы поиска и операцию сохранения в классе `SingerServiceImpl`.

Внедрение требующихся зависимостей

Для реализации контрольных примеров вместе с базой данных потребуется библиотека, которая поможет заполнить базу данных требующейся тестовой информацией перед запуском контрольного примера, а затем без особого труда выполнить необходимые операции в базе данных. А для того чтобы упростить подготовку тестовых данных, воспользуемся поддержкой формата электронных таблиц Microsoft Excel.

Для достижения намеченных целей понадобятся дополнительные библиотеки. Так, на стороне базы данных потребуется распространенная библиотека DbUnit (`http://dbunit.sourceforge.net`), помогающая реализовать тестирование, связанное с базой данных. А библиотека из проекта Apache POI (`http://poi.apache.org`) поможет произвести синтаксический анализ тестовых данных, подготовленных в формате Microsoft Excel.

Конфигурирование профиля для тестирования на уровне обслуживания

Средство конфигурирования профилей, предназначенное для определения компонентов Spring Beans и появившееся в версии Spring 3.1, очень удобно для реализации контрольного примера с соответствующей конфигурацией тестируемых компонентов. Чтобы упростить тестирование на уровне обслуживания, для конфигурирования контекста типа `ApplicationContext` будут также использоваться возможности профилей. В целях тестирования рассматриваемого здесь примера приложения для певцов понадобятся следующие профили.

- **Профиль разработки (`dev`).** Это профиль, сконфигурированный для среды разработки. Например, в среде разработки должны быть выполнены сценарии создания и заполнения первоначальной информацией базы данных H2 на стороне сервера.
- **Профиль тестирования (`test`).** Это профиль, сконфигурированный для среды тестирования. Например, в среде тестирования разработки должен быть выполнен сценарий создания базы H2 на стороне сервера, а ее заполнение информацией будет сделано в контрольном примере.

Итак, сконфигурируем среду профилей приложения для певцов. В этом приложении конфигурация сервера базы данных, включая источник данных, прикладной ин-

терфейс JPA, обработку транзакций и т.д., была определена в XML-файле datasourcetx-jpa.xml. Но нам требуется сконфигурировать источник данных в этом файле только для профиля dev. Для этого необходимо ввести определение компонента источника данных в конфигурацию профиля. Ниже показаны изменения, которые требуется внести в данную конфигурацию.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

    <bean id="transactionManager"
       class="org.springframework.orm.jpa
                 .JpaTransactionManager">
      <property name="entityManagerFactory" ref="emf"/>
    </bean>

    <tx:annotation-driven
        transaction-manager="transactionManager" />

    <bean id="emf"
       class="org.springframework.orm.jpa
                 .LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="jpaVendorAdapter">
        <bean class= "org.springframework.orm.jpa.vendor
                        .HibernateJpaVendorAdapter" />
      </property>
      <property name="packagesToScan"
                value="com.apress.prospring5.ch13"/>
      <property name="jpaProperties">
        <props>
          <prop key="hibernate.dialect">org.hibernate
                                .dialect.H2Dialect</prop>
          <prop key="hibernate.max_fetch_depth">3</prop>
          <prop key="hibernate.jdbc.fetch_size">50</prop>
          <prop key="hibernate.jdbc.batch_size">10</prop>
          <prop key="hibernate.show_sql">true</prop>
      </props>
      </property>
    </bean>

    <context:annotation-config/>

    <jpa:repositories
         base-package="com.apress.prospring5.ch13"
         entity-manager-factory-ref="emf"
         transaction-manager-ref="transactionManager"/>
    <beans profile="dev">
      <jdbc:embedded-database id="dataSource" type="H2">
```

```
        <jdbc:script
              location="classpath:config/schema.sql"/>
        <jdbc:script
              location="classpath:config/test-data.sql"/>
      </jdbc:embedded-database>
    </beans>
</beans>
```

Как следует из приведенного выше фрагмента кода конфигурации, компонент `dataSource` определен в дескрипторе разметки `<beans>`, где атрибуту `profile` присвоено значение `dev`, обозначающее, что источник данных можно применять только в системе разработки. Напомним, что профили можно активизировать, передав, например, системный параметр **`-Dspring.profiles.active=dev`** виртуальной машине JVM.

Вариант конфигурирования на языке Java

Как только началось внедрение конфигурационных классов Java, конфигурирование в формате XML стало постепенно терять свое прежнее положение. Именно поэтому в примерах из данной книги делается акцент на применении конфигурационных классов Java, тогда как конфигурирование в формате XML рассматривается лишь для того, чтобы продемонстрировать постепенное развитие конфигурирования в Spring. Приведенную выше конфигурацию в формате XML можно разделить на две части: одну — для конфигурирования источника данных в зависимости от конкретного профиля, а другую — для конфигурирования транзакций, которое является общим для настройки обеих сред разработки и тестирования. Обе эти части реализованы в приведенных ниже конфигурационных классах. Такое конфигурирование на языке Java оказывается более совершенным в том отношении, что схема базы данных формируется автоматически, как по волшебству, благодаря установке значения `create-drop` в свойстве `hibernate.hbm2ddl.auto` из библиотеки Hibernate.

```
// Исходный файл DataConfig.java
package com.apress.prospring5.ch13.config;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseType;
import javax.sql.DataSource;
```

```
@Profile("dev")
@Configuration
@ComponentScan(basePackages =
               {"com.apress.prospring5.ch13.init"} )
public class DataConfig {

  private static Logger logger =
          LoggerFactory.getLogger(DataConfig.class);

  @Bean
  public DataSource dataSource() {
    try {
      EmbeddedDatabaseBuilder dbBuilder =
              new EmbeddedDatabaseBuilder();
      return dbBuilder.setType(EmbeddedDatabaseType.H2)
                      .build();
    } catch (Exception e) {
      logger.error("Embedded DataSource bean cannot "
                + "be created!", e);
      return null;
    }
  }
}

// Исходный файл ServiceConfig.class
package com.apress.prospring5.ch13.config;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.data.jpa.repository.config
          .EnableJpaRepositories;
import org.springframework.orm.jpa.JpaTransactionManager;

import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa
          .LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor
          .HibernateJpaVendorAdapter;
import org.springframework.transaction
          .PlatformTransactionManager;

import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;
import java.util.Properties;
```

```
@Configuration
@EnableJpaRepositories(basePackages = {"com.apress.prospring5.ch13.repos"})
@ComponentScan(basePackages =
               {"com.apress.prospring5.ch13.entities",
                "com.apress.prospring5.ch13.services"})
public class ServiceConfig {

  @Autowired
  DataSource dataSource;

  @Bean
  public Properties hibernateProperties() {
    Properties hibernateProp = new Properties();
    hibernateProp.put("hibernate.dialect",
                      "org.hibernate.dialect.H2Dialect");
    hibernateProp.put("hibernate.hbm2ddl.auto",
                      "create-drop");
    hibernateProp.put("hibernate.show_sql", true);
    hibernateProp.put("hibernate.max_fetch_depth", 3);
    hibernateProp.put("hibernate.jdbc.batch_size", 10);
    hibernateProp.put("hibernate.jdbc.fetch_size", 50);
    return hibernateProp;
  }

  @Bean
  public PlatformTransactionManager transactionManager() {
    return new JpaTransactionManager(
               entityManagerFactory());
  }

  @Bean
  public JpaVendorAdapter jpaVendorAdapter() {
    return new HibernateJpaVendorAdapter();
  }

  @Bean
  public EntityManagerFactory entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean factoryBean =
         new LocalContainerEntityManagerFactoryBean();
    factoryBean.setPackagesToScan(
                "com.apress.prospring5.ch13.entities");
    factoryBean.setDataSource(dataSource);
    factoryBean.setJpaVendorAdapter(
                new HibernateJpaVendorAdapter());
    factoryBean.setJpaProperties(hibernateProperties());
    factoryBean.setJpaVendorAdapter(jpaVendorAdapter());
    factoryBean.afterPropertiesSet();
    return factoryBean.getNativeEntityManagerFactory();
  }
}
```

Реализация классов для среды тестирования

Прежде чем приступить к реализации отдельного контрольного примера, необходимо реализовать ряд классов, поддерживающих заполнение базы данных тестовой информацией из файла Excel. А для того чтобы упростить разработку контрольного примера, необходимо ввести специальную аннотацию `@DataSets`, которая принимает имя файла Excel в качестве аргумента. Разработаем с этой целью специальный приемник событий, наступающих при выполнении тестов. Это средство поддерживается средой тестирования в Spring и предназначено для проверки наличия аннотации `@DataSets` и соответствующей загрузки данных.

Реализация специального приемника событий при выполнении тестов

В интерфейсе `org.springframework.test.context.TestExecutionListener` из модуля `spring-test` определяется прикладной интерфейс API приемника, способного перехватывать события на различных стадиях выполнения контрольного примера (например, до и после тестируемого класса, до и после тестируемого метода и т.д.). Для тестирования на уровне обслуживания требуется реализовать специальный приемник событий, чтобы заполнить базу данных тестовой информацией с помощью простой аннотации `@DataSets` в контрольном примере. В частности, чтобы проверить метод `SingerService.findAll()`, потребуется исходный код, подобный приведенному ниже.

```
@DataSets(setUpDataSet= "/com/apress/prospring5/ch13
                         /SingerServiceImplTest.xls")
@Test
public void testFindAll() throws Exception {
  List<Singer> result = singerService.findAll();
  ...
}
```

Применение аннотации `@DataSets` в контрольном примере означает, что перед запуском теста необходимо загрузить тестовую информацию в базу данных из указанного файла Excel. С этой целью следует прежде всего объявить специальную аннотацию:

```
package com.apress.prospring5.ch13;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)

public @interface DataSets {
```

```
  String setUpDataSet() default "";
}
```

Специальная аннотация `@DataSets` относится к категории аннотаций, действующих на уровне методов. А реализовав интерфейс `TestExecutionListener`, можно разработать специальный класс для приемника событий, наступающих при выполнении тестов, как показано ниже.

```
package com.apress.prospring5.ch13;

import org.dbunit.IDatabaseTester;
import org.dbunit.dataset.IDataSet;
import org.dbunit.util.fileloader.XlsDataFileLoader;
import org.springframework.test.context.TestContext;
import org.springframework.test.context
          .TestExecutionListener;

public class ServiceTestExecutionListener
       implements TestExecutionListener {
  private IDatabaseTester databaseTester;

  @Override
  public void afterTestClass(TestContext arg0)
         throws Exception {
  }

  @Override
  public void afterTestMethod(TestContext arg0)
         throws Exception {
    if (databaseTester != null) {
      databaseTester.onTearDown();
    }
  }

  @Override
  public void beforeTestClass(TestContext arg0)
         throws Exception {
  }

  @Override
  public void beforeTestMethod(TestContext testCtx)
         throws Exception {
    DataSets dataSetAnnotation = testCtx.getTestMethod()
             .getAnnotation(DataSets.class);
    if (dataSetAnnotation == null ) {
      return;
    }

    String dataSetName = dataSetAnnotation.setUpDataSet();
```

```
    if (!dataSetName.equals("") ) {
      databaseTester = (IDatabaseTester)
      testCtx.getApplicationContext()
             .getBean("databaseTester");
      XlsDataFileLoader xlsDataFileLoader =
                       (XlsDataFileLoader)
      testCtx.getApplicationContext()
             .getBean("xlsDataFileLoader");
      IDataSet dataSet = xlsDataFileLoader
                         .load(dataSetName);
      databaseTester.setDataSet(dataSet);
      databaseTester.onSetup();
    }
  }

  @Override
  public void prepareTestInstance(TestContext arg0)
         throws Exception {
  }
}
```

После реализации интерфейса `TestExecutionListener` осталось реализовать ряд методов. Но в данном случае нас интересуют только методы `beforeTestMethod()` и `afterTestMethod()`, в которых осуществляется заполнение и очистка тестовой информации до и после выполнения каждого тестового метода. Обратите внимание на то, что каждому методу из Spring будет передаваться экземпляр класса `TestContext`, и поэтому метод может получить доступ к базовому контексту типа `ApplicationContext`, первоначально загруженному каркасом Spring Framework для тестирования.

Метод `beforeTestMethod()` представляет особый интерес. Во-первых, в нем проверяется наличие аннотации `@DataSets` для тестового метода. Если аннотация присутствует, тестовая информация будет загружена из указанного файла Excel. В этом случае интерфейс `IDatabaseTester` (вместе с реализующим его классом `org.dbunit.DataSourceDatabaseTester`, о котором речь пойдет далее) получается из контекста типа `TestContext`. Интерфейс `IDatabaseTester` предоставляется инструментальным средством DbUnit для поддержки операций, выполняемых в базе данных на основании имеющегося подключения к ней или источника данных. Во-вторых, экземпляр класса `XlsDataFileLoader` получается из контекста типа `TestContext`. Класс `XlsDataFileLoader` предоставляется в DbUnit для поддержки загрузки тестовой информации из файла Excel. Для чтения содержимого файлов формата Microsoft Office в этом классе подспудно используется библиотека Apache POI. В частности, для загрузки данных из файла вызывается метод `XlsDataFileLoader.load()`, возвращающий экземпляр класса, реализующего интерфейс `IDataSet` и представляющего набор загруженных данных. В-третьих, для установки тестовой информации вызывается метод `IDatabaseTester.setDataSet()`, а для того

чтобы начать заполнение базы данных тестовой информацией — метод `IDatabaseTester.onSetup()`. И, наконец, для очистки данных в методе `afterTestMethod()` вызывается метод `IDatabaseTester.onTearDown()`.

Реализация конфигурационного класса

Теперь реализуем конфигурационный класс для среды тестирования. Ниже приведен исходный код для конфигурирования на языке Java.

```
package com.apress.prospring5.ch13.config;

import javax.sql.DataSource;
import com.apress.prospring5.ch13.init.DBInitializer;
import org.dbunit.DataSourceDatabaseTester;
import org.dbunit.util.fileloader.XlsDataFileLoader;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.*;
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseType;

@Configuration
@ComponentScan(basePackages={"com.apress.prospring5.ch13"},
               excludeFilters = {@ComponentScan.Filter(
                      type = FilterType.ASSIGNABLE_TYPE,
                      value = DBInitializer.class)
})
@Profile("test")
public class ServiceTestConfig {
  private static Logger logger =
          LoggerFactory.getLogger(ServiceTestConfig.class);

  @Bean
  public DataSource dataSource() {
    try {
      EmbeddedDatabaseBuilder dbBuilder =
                      new EmbeddedDatabaseBuilder();
      return dbBuilder.setType(EmbeddedDatabaseType.H2)
                      .build();
    } catch (Exception e) {
      logger.error("Embedded DataSource bean cannot "
                + "be created!", e);
      return null;
    }
  }

  @Bean(name="databaseTester")
```

```
  public DataSourceDatabaseTester
         dataSourceDatabaseTester() {
    DataSourceDatabaseTester databaseTester =
              new DataSourceDatabaseTester(dataSource());
    return databaseTester;
  }

  @Bean(name="xlsDataFileLoader")
  public XlsDataFileLoader xlsDataFileLoader() {
    return new XlsDataFileLoader();
  }
}
```

В классе `ServiceTestConfig` определяется контекст типа `ApplicationCon text` для тестирования на уровне обслуживания. А в аннотации `@ComponentScan` каркасу Spring предписывается просмотреть на уровне обслуживания те компоненты Spring Beans, которые требуется протестировать. Атрибут `excludeFilters` этой аннотации служит для того, чтобы тестовая база данных не инициализировалась элементами разработки. В аннотации `@Profile` указывается, что компоненты Spring Beans, определенные в данном классе, относятся к профилю `test`.

В данном конфигурационном классе объявляется еще один компонент Spring Bean, называемый `dataSource` и выполняющий только сценарий из файла `schema.sql`, не заполняя при этом базу данных H2 тестовой информацией. Для загрузки данных из файла Excel в специальном приемнике событий, наступающих при выполнении тестов, применяются компоненты `databaseTester` и `xlsDataFileLoader`. Обратите внимание на то, что компонент `dataSourceDatabaseTester` сконструирован с помощью компонента `dataSource`, определенного для среды тестирования.

Модульное тестирование на уровне обслуживания

Начнем с модульного тестирования методов поиска, включая методы `SingerSer vice.findAll()` и `SingerService.findByFirstNameAndLastName()`. Прежде всего необходимо подготовить тестовую информацию в формате Excel. Общепринятая норма практики предусматривает размещение файла с тестовой информацией в той папке, где находится класс контрольного примера, и присвоение ему того же самого имени. Следовательно, в данном случае полностью уточненное имя этого файла будет следующим: `/src/test/java/com/apress/prospring5/ch13/SingerService ImplTest.xls`.

Тестовая информация подготовлена на рабочем листе электронной таблицы под именем, совпадающим с именем таблицы в базе данных (`SINGER`). В первой строке рабочего листа указаны имена столбцов этой таблицы, а начиная со второй строки введены сведения об имени, фамилии и дате рождения певца. Здесь указан столбец для идентификатора, но не его значение, поскольку идентификаторы будут запол-

няться самой базой данных. Пример такого файла Excel можно найти в исходном коде примеров, загружаемом по ссылке, указанной во введении.

Ниже приведен исходный код тестового класса с контрольными примерами для тестирования двух методов поиска.

```
package com.apress.prospring5.ch13;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertNull;
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.validation.ConstraintViolationException;

import com.apress.prospring5.ch13.entities.Singer;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context
          .ContextConfiguration;
import org.springframework.test.context
          .TestExecutionListeners;
import org.springframework.test.context.junit4.
AbstractTransactionalJUnit4SpringContextTests;
import org.springframework.test.context.junit4
          .SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {ServiceTestConfig.class,
                                 ServiceConfig.class,
                                 DataConfig.class})
@TestExecutionListeners(
              {ServiceTestExecutionListener.class})
@ActiveProfiles("test")
public class SingerServiceImplTest extends
       AbstractTransactionalJUnit4SpringContextTests {
  @Autowired
  SingerService singerService;

  @PersistenceContext
  private EntityManager em;

  @DataSets(setUpDataSet= "/com/apress/prospring5/ch13
                           /SingerServiceImplTest.xls")
  @Test
```

```
  public void testFindAll() throws Exception {
    List<Singer> result = singerService.findAll();

    assertNotNull(result);
    assertEquals(1, result.size());
  }

  @DataSets(setUpDataSet= "/com/apress/prospring5/ch13
                           /SingerServiceImplTest.xls")
  @Test
  public void testFindByFirstNameAndLastName_1()
         throws Exception {
    Singer result = singerService
           .findByFirstNameAndLastName("John", "Mayer");
    assertNotNull(result);
  }

  @DataSets(setUpDataSet= "/com/apress/prospring5/ch13
                           /SingerServiceImplTest.xls")

  @Test
  public void testFindByFirstNameAndLastName_2()
         throws Exception {
    Singer result = singerService
           .findByFirstNameAndLastName("BB", "King");
    assertNull(result);
  }
}
```

Здесь применяется та же аннотация `@RunWith`, что и при тестировании класса контроллера. Аннотация `@ContextConfiguration` указывает на то, что конфигурация контекста типа `ApplicationContext` должна быть загружена из классов `ServiceTestConfig` и `DataConfig`. И хотя классу `DataConfig` не следовало бы здесь быть, он все же используется лишь для того, чтобы сделать очевидным действие профилей в Spring. Аннотация `@TestExecutionListeners` обозначает, что класс `ServiceTestExecutionListener` должен применяться для перехвата событий, наступающих в жизненном цикле выполнения контрольных примеров. А в аннотации `@ActiveProfiles` задается применяемый профиль. Следовательно, в данном случае будет загружен компонент `dataSource`, определенный в классе `ServiceTestConfig`, а не компонент, определенный в файле конфигурации `datasource-tx-jpa.xml`, поскольку он относится к профилю `dev`.

Кроме того, класс `SingerServiceImplTest` расширяет класс `AbstractTransactionalJUnit4SpringContextTests`, с помощью которого в Spring поддерживается инструментальное средство модульного тестирования JUnit при наличии механизмов внедрения зависимостей и управления транзакциями. Обратите внимание на то, что в среде тестирования Spring будет производиться откат транзакции после вы-

полнения каждого тестового метода, а следовательно, и откат всех операций обновления базы данных. Для управления режимом отката можно применить аннотацию `@Rollback` на уровне методов.

В данном тестовом классе предусмотрен один контрольный пример для тестирования метода `findAll()` и два контрольных примера для тестирования метода `testFindByFirstNameAndLastName()` (в одном из них извлекается результат, а в другом — не извлекается). Ко всем методам поиска применяется аннотация `@DataSets` с указанным файлом Excel, содержащим тестовую информацию. Кроме того, служба типа `SingerService` автоматически связывается с контрольным примером из контекста типа `ApplicationContext`. Остальная часть рассматриваемого здесь исходного кода должна быть самоочевидной. В каждом контрольном примере применяются разные операторы утверждений, чтобы проверить, является ли получаемый результат предполагаемый.

Запустите описанный выше тест, чтобы удостовериться в его успешном прохождении. Перейдем далее к тестированию операции сохранения. В данном случае необходимо проверить две ситуации. Одной из них является нормальная ситуация, когда достоверные сведения о певце благополучно сохраняются в базе данных, а другая ситуация, когда сведения о певце содержат ошибку, которая приводит к генерированию правильного исключения. Ниже приведен дополнительный фрагмент кода контрольных примеров для этих двух ситуаций.

```
package com.apress.prospring5.ch13;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertNull;
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.validation.ConstraintViolationException;
import com.apress.prospring5.ch13.entities.Singer;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context
          .ContextConfiguration;
import org.springframework.test.context
          .TestExecutionListeners;
import org.springframework.test.context.junit4
          .AbstractTransactionalJUnit4SpringContextTests;
import org.springframework.test.context.junit4
          .SpringJUnit4ClassRunner;
```

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {ServiceTestConfig.class,
                                 ServiceConfig.class,
                                 DataConfig.class})
@TestExecutionListeners(
              {ServiceTestExecutionListener.class})
@ActiveProfiles("test")
public class SingerServiceImplTest extends
       AbstractTransactionalJUnit4SpringContextTests {
  @Autowired
  SingerService singerService;

  @PersistenceContext
  private EntityManager em;

  @Test
  public void testAddSinger() throws Exception {
    deleteFromTables("SINGER");

    Singer singer = new Singer();
    singer.setFirstName("Stevie");
    singer.setLastName("Vaughan ");

    singerService.save(singer);
    em.flush();

    List<Singer> singers = singerService.findAll();
    assertEquals(1, singers.size());
  }

  @Test(expected=ConstraintViolationException.class)
  public void testAddSingerWithJSR349Error()
         throws Exception {
    deleteFromTables("SINGER");

    Singer singer = new Singer();
    singerService.save(singer);
    em.flush();

    List<Singer> singers = singerService.findAll();
    assertEquals(0, singers.size());
  }
}
```

В приведенном выше тестовом коде обратите внимание на метод `testAddContact()`, в котором вызывается удобный метод `deleteFromTables()`, чтобы в таблице `SINGER` не осталось никаких данных. Этот метод специально предоставляется в классе `AbstractTransactionalJUnit4SpringContextTests` для очистки

таблиц базы данных. Однако после операции сохранения необходимо явным образом вызвать метод `EntityManager.flush()`, чтобы принудить библиотеку Hibernate сбросить контекст сохраняемости в базу данных, и тогда метод `findAll()` сможет правильно извлечь из нее информацию. В версии Spring 4.3 был внедрен псевдоним `SpringRunner.class` для литерала класса `SpringJUnit4ClassRunner.class`.

Во втором тестовом методе `testAddSingerWithJSR349Error()` проверяется операция сохранения сведений о певце, содержащих ошибку проверки достоверности. Обратите внимание на то, что аннотации `@Test` передается атрибут `expected`, обозначающий, что в контрольном примере ожидается генерирование исключения заданного типа (в данном случае — класса `ConstraintViolationException`). Выполните тестовый класс снова, чтобы убедиться в успешном прохождении теста.

Выше были рассмотрены только те классы, которые чаще всего применяются в среде тестирования Spring, где предоставляется немало других классов и аннотаций, обеспечивающих точный контроль на протяжении всего жизненного цикла контрольного примера. Например, аннотации `@BeforeTransaction` и `@AfterTransaction` позволяют выполнить определенную логику перед тем, как Spring инициирует транзакцию, или же после того, как транзакция будет завершена для контрольного примера. За более подробными сведениями о различных особенностях среды тестирования обращайтесь к справочной документации на Spring.

Отказ от услуг DbUnit

Пользоваться инструментальным средством DbUnit для модульного тестирования может оказаться нелегко, потому что для этого потребуются дополнительные зависимости и конфигурационные классы. Не лучше ли воспользоваться принятым в Spring подходом к модульному тестированию? Ведь, начиная с версии 4.0, в Spring появилось немало удобных аннотаций. К их числу относится аннотация `@Sql`, которая будет использована в рассматриваемом далее примере. Этой аннотацией можно снабдить тестовый класс или метод, чтобы сконфигурировать сценарии и операторы SQL для выполнения в конкретной базе данных при проведении комплексных тестов. Это означает, что тестовую информацию можно подготовить, не прибегая к услугам DbUnit. И благодаря этому не только упрощается тестовая конфигурация, но и отпадает потребность в дальнейшем расширении тестовых классов.

Кроме того, в этом разделе упоминается инструментальное средство JUnit 5, иначе называемое JUnit Jupiter (`http://junit.org/junit5/`). Его поддержка предоставлялась в Spring с версии 4.3, т.е. еще до выпуска первой устойчивой версии, а на момент написания данной книги текущей считалась версия 5.0.0-M4. JUnit 5 относится к новому поколению платформы JUnit для модульного тестирования и предназначается в качестве прочного основания для тестирования разработчиками приложений на виртуальной машине JVM, начиная с версии Java 8 и применяя самые разные виды тестирования, как поясняется в официальной документации.

Итак, выясним, как в таком случае изменяется конфигурирование. Конфигурационный класс тестового источника данных для тестового профиля значительно упрощается, как показано ниже, поскольку теперь достаточно и пустой базы данных.

```
package com.apress.prospring5.ch13.config;

import com.apress.prospring5.ch13.init.DBInitializer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.*;
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseType;
import javax.sql.DataSource;

@Configuration
@ComponentScan(basePackages={"com.apress.prospring5.ch13"},
excludeFilters = {@ComponentScan.Filter(
                  type = FilterType.ASSIGNABLE_TYPE,
                  value = DBInitializer.class)
})
@Profile("test")
public class SimpleTestConfig {

  private static Logger logger =
          LoggerFactory.getLogger(SimpleTestConfig.class);

  @Bean
  public DataSource dataSource() {
    try {
      EmbeddedDatabaseBuilder dbBuilder = new EmbeddedDatabaseBuilder();
      return dbBuilder.setType(EmbeddedDatabaseType.H2).build();
    } catch (Exception e) {
      logger.error("Embedded DataSource bean cannot "
                + "be created!", e);
      return null;
    }
  }
}
```

Данные и запросы, требующиеся для контрольного примера, будут предоставлены в файлах сценариев SQL. В частности, данные предоставляются в файле `test-data.sql`, содержимое которого приведено ниже.

```
insert into singer (first_name, last_name, birth_date,version)
values ('John', 'Mayer', '1977-10-16', 0);
```

Очистка тестовой базы данных будет выполняться по сценарию из файла `clean-up.sql`. Этот сценарий служит для очистки базы данных, чтобы данные, требующие-

ся для одного тестового метода, не испортили данные, необходимые для другого тестового метода. Ниже приведено содержимое упомянутого выше файла.

```
delete from singer;
```

В представленном ниже тестовом классе указан ряд аннотаций JUnit5 с целью продемонстрировать порядок их применения. Назначение каждой из этих аннотаций поясняется после исходного кода данного класса.

```
package com.apress.prospring5.ch13;

import com.apress.prospring5.ch13.config.DataConfig;
import com.apress.prospring5.ch13.config.ServiceConfig;
import com.apress.prospring5.ch13.config.SimpleTestConfig;

import com.apress.prospring5.ch13.entities.Singer;
import com.apress.prospring5.ch13.services.SingerService;
import org.junit.jupiter.api.*;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.jdbc.Sql;
import org.springframework.test.context.jdbc.SqlConfig;
import org.springframework.test.context.jdbc.SqlGroup;
import org.springframework.test.context.junit.jupiter
          .SpringJUnitConfig;

import java.util.List;
import static org.junit.jupiter.api.Assertions
                .assertEquals;
import static org.junit.jupiter.api.Assertions
                .assertNotNull;

@SpringJUnitConfig(classes = {SimpleTestConfig.class,
                              ServiceConfig.class,
                              DataConfig.class})
@DisplayName("Integration SingerService Test")
@ActiveProfiles("test")
public class SingerServiceTest {
  private static Logger logger =
          LoggerFactory.getLogger(SingerServiceTest.class);

  @Autowired
  SingerService singerService;

  @BeforeAll
  static void setUp() {
```

```
  logger.info("--> @BeforeAll - executes before "
          + "executing all test methods in this class");
}

@AfterAll
static void tearDown() {
  logger.info("--> @AfterAll - executes before "
          + "executing all test methods in this class");
}

@BeforeEach
void init() {
  logger.info("--> @BeforeEach - executes before "
           + "each test method in this class");
}

@AfterEach
void dispose() {
  logger.info("--> @AfterEach - executes before "
           + "each test method in this class");
}

@Test
@DisplayName("should return all singers")
@SqlGroup({
    @Sql(value = "classpath:db/test-data.sql",
         config = @SqlConfig(encoding = "utf-8",
         separator = ";", commentPrefix = "--"),
         executionPhase = Sql.ExecutionPhase
                          .BEFORE_TEST_METHOD),
    @Sql(value = "classpath:db/clean-up.sql",
         config = @SqlConfig(encoding = "utf-8",
         separator = ";", commentPrefix = "--"),
         executionPhase = Sql.ExecutionPhase
                         .AFTER_TEST_METHOD),
})
public void findAll() {
  List<Singer> result = singerService.findAll();
  assertNotNull(result);
  assertEquals(1, result.size());
}

@Test
@DisplayName("should return singer 'John Mayer'")
@SqlGroup({
    @Sql(value = "classpath:db/test-data.sql",
         config = @SqlConfig(encoding = "utf-8",
         separator = ";", commentPrefix = "--"),
         executionPhase = Sql.ExecutionPhase
```

```
                            .BEFORE_TEST_METHOD),
        @Sql(value = "classpath:db/clean-up.sql",
             config = @SqlConfig(encoding = "utf-8",
             separator = ";", commentPrefix = "--"),
             executionPhase = Sql.ExecutionPhase
                            .AFTER_TEST_METHOD),
    })
    public void testFindByFirstNameAndLastNameOne()
           throws Exception {
      Singer result = singerService
            .findByFirstNameAndLastName("John", "Mayer");
      assertNotNull(result);
    }
}
```

В приведенном выше тестовом коде контекст типа `ApplicationContext` создается с помощью аннотации `@SpringJUnitConfig`. Она состоит из аннотации `@ExtendWith(SpringExtension.class)` из JUnit Jupiter и аннотации `@Context Configuration` из каркаса Spring TestContext Framework.

Аннотация `@DisplayName` обычно применяется в JUnit Jupiter для объявления специально отображаемого значения из снабженного ею тестового класса или метода. В тех IDE, где поддерживается версия JUnit 5, это может выглядеть весьма привлекательно, как показано на рис. 13.1. Как видите, теперь намного проще обнаружить, что служба типа `SingerService` действует так, как и предполагалось.

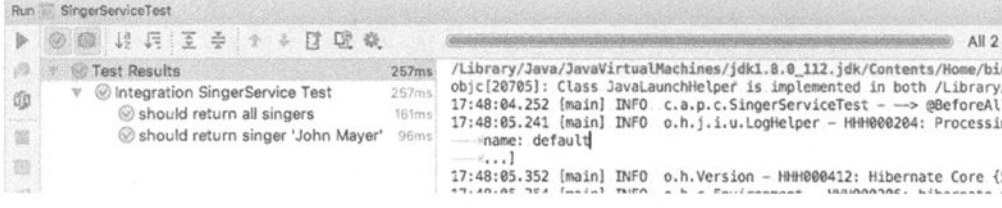


Рис. 13.1. Представление в IDE IntelliJ IDEA теста, выполняемого в JUnit 5

Имена аннотаций `@BeforeAll` и `@AfterAll` самоочевидны. Они заменяют собой аннотации `@BeforeClass` и `@AfterClass` из версии JUnit 4. Это же можно сказать и об аннотациях `@BeforeEach` и `@AfterEach`, которые заменяют собой аннотации `@Before` и `@After` из версии JUnit 4. И, наконец, аннотация `@SqlGroup` служит для группирования нескольких аннотаций `@Sql`.

Модульное тестирование клиентской части веб-приложений

Еще одной интересной разновидностью тестирования является проверка поведения клиентской части как единого целого после развертывания веб-приложения в веб-контейнере, подобном Apache Tomcat. Основная причина такого тестирования

состоит в потребности удостовериться в правильном поведении представлений по отношению к разным действиям пользователей, несмотря на то, что веб-приложение тестируется на каждом уровне.

Автоматизация тестирования клиентской части веб-приложений очень важна для экономии времени разработчиков и пользователей при наличии повторяющихся действий в клиентской части для выполнения контрольного примера. Тем не менее разработка контрольного примера для клиентской части — задача непростая, особенно в веб-приложениях с большим количеством интерактивных, функционально насыщенных и основанных на технологии Ajax компонентов.

Введение в Selenium

Selenium — это эффективное и комплексное инструментальное средство и каркас, которые совместно предназначены для автоматизации тестирования клиентской части веб-приложений. Главной особенностью Selenium является возможность "приводить в действие" браузеры, эмулируя взаимодействие пользователей с веб-приложением и в то же время контролируя состояние представлений.

В инструментальном средстве Selenium поддерживаются такие распространенные браузеры, как Firefox, IE и Chrome, а также языки программирования Java, C#, PHP, Perl, Ruby и Python. Оно разработано с учетом Ajax-приложений и насыщенных интернет-приложений (RIA), что позволяет тестировать современные веб-приложения. А на тот случай, если в веб-приложении имеется много внешних пользовательских интерфейсов, для проверки которых приходится проводить большое количество тестов, в модуле `selenium-server` предоставляются встроенные функциональные возможности для распределенных вычислений, позволяющие выполнять тесты клиентской части в группе компьютеров.

Selenium IDE — это модуль, подключаемый к браузеру Firefox и помогающий регистрировать взаимодействие пользователя с веб-приложением. В нем также поддерживается повторное воспроизведение и экспорт сценариев в разных форматах, чтобы упростить разработку тестовых сценариев.

Начиная с версии 2.0, в Selenium предусмотрена интеграция с WebDriver API, снимая тем самым многие ограничения и предоставляя альтернативный и более простой программный интерфейс. В итоге получается универсальный объектно-ориентированный прикладной интерфейс API, обеспечивающий дополнительную поддержку многочисленных браузеров наряду с усовершенствованными возможностями для разрешения затруднений, возникающих при тестировании сложных современных веб-приложений.

Тестирование клиентской части веб-приложений — обширная тема, подробное обсуждение которой выходит за рамки данной книги. Но даже из этого краткого введения ясно, что Selenium помогает автоматизировать взаимодействие пользователя с клиентской частью веб-приложения, совместимое с множеством браузеров. За дополнительными сведениями по данному вопросу обращайтесь к документации на Selenium, оперативно доступной по адресу `http://seleniumhq.org/docs`.

Резюме

В этой главе было показано, как проводить различные виды модульного тестирования в приложениях Spring с помощью общеупотребительных инструментальных средств, библиотек и каркасов, включая JUnit, DbUnit и Mockito. Во-первых, в общих чертах была описана среда тестирования корпоративных приложений, а также пояснено, какие именно тесты должны выполняться на каждой стадии жизненного цикла разработки приложений. Во-вторых, были разработаны два вида тестов: модульный тест логики и комплексно-модульный тест взаимодействия составных частей приложения. А в завершение главы была вкратце затронута тема тестирования клиентской части веб-приложений с помощью инструментального средства Selenium.

Тестирование корпоративных приложений — довольно обширная и сложная тема. Если у вас есть желание более подробно изучить платформу JUnit, рекомендуем прочитать книгу *JUnit in Action, Second Edition* (издательство Manning, 2010 г.). А если вас интересуют другие подходы к тестированию, принятые в Spring, подробнее об этом можно узнать из книги *Pivotal Certified Professional Spring Developer Exam* (издательство Apress, 2016 г.), где имеется отдельная глава, посвященная дополнительным библиотекам и модулю Spring Boot Test для тестирования приложений.

ГЛАВА 14

Поддержка сценариев в Spring

В предыдущих главах пояснялось, каким образом каркас Spring Framework помогает программирующим на Java разрабатывать приложения на платформе JEE. Благодаря механизму внедрения зависимостей в Spring Framework и его интеграции с каждым уровнем приложения (через библиотеки в собственных модулях Spring Framework или взаимодействие со сторонними библиотеками) можно упростить реализацию и сопровождение бизнес-логики.

Но для всей логики, реализованной в представленных до сих пор примерах, применялся язык Java. Несмотря на то что Java является одним из наиболее успешных языков за всю историю программирования, его по-прежнему критикуют за такие недостатки, как структура языка и отсутствие всесторонней поддержки в областях, подобных массовой параллельной обработке.

Например, одна из особенностей языка Java состоит в том, что все переменные являются статически типизированными. Иными словами, в программе на Java каждая объявленная переменная должна иметь связанный с ней статический тип (`String`, `int`, `Object`, `ArrayList` и т.д.). Но в некоторых случаях более предпочтительной может оказаться динамическая типизация, которая поддерживается в динамических языках вроде JavaScript.

Для решения подобных задач были разработаны многочисленные языки сценариев. К числу самых распространенных из них относятся JavaScript, Groovy, Scala, Ruby и Erlang. Почти во всех этих языках поддерживается динамическая типизация и предоставляются средства, недоступные в Java. Они предназначены и для других специальных целей. Например, в языке Scala (`www.scalalang.org`) шаблоны функционального программирования сочетаются с объектно-ориентированными шаблонами, а также поддерживается более развитая и масштабируемая модель параллельного программирования с помощью таких понятий и принципов, как исполнители

и передача сообщений. А в языке Groovy (`www.groovy-lang.org`) предлагается упрощенная модель программирования и поддерживается реализация предметно-ориентированных языков (DSL), которые делают прикладной код более удобным для чтения и сопровождения.

Еще одним важным понятием, которым в языках сценариев могут пользоваться разработчики приложений на Java, являются замыкания, более подробно рассматриваемые далее в этой главе. Проще говоря, *замыкание* — это фрагмент (или блок) кода, заключаемый в объект. Подобно методу в Java, замыкание является исполняемым и может получать параметры, а также возвращать объекты и значения. В то же время это обычный объект, который можно передавать по ссылке куда угодно в пределах приложения подобно любому простому объекту POJO в Java.

В этой главе будет рассмотрен ряд основных понятий языков сценариев с особым акцентом на Groovy. В ней будет показано, что каркас Spring Framework способен плавно взаимодействовать с языками сценариев, предоставляя специальные функциональные возможности для приложений, основанных на Spring. В этой главе будут, в частности, рассмотрены следующие вопросы.

- **Поддержка сценариев в Java.** В рамках JCP спецификация JSR-223 (Scripting for the Java Platform — Написание сценариев для платформы Java) регламентирует поддержку языков сценариев в Java. Она стала доступной в Java в версии SE 6. В этой части сделан краткий обзор поддержки написания сценариев в среде Java.
- **Язык Groovy.** В этой части представлено общее введение в Groovy — один из самых распространенных языков сценариев, применяемых вместе с Java.
- **Применение Groovy в Spring.** Каркас Spring Framework обеспечивает универсальную поддержку языков сценариев. Начиная с версии 3.1, в Spring обеспечивается готовая поддержка языков Groovy, JRuby и BeanShell.

Материал этой главы не предназначен служить подробным справочным пособием по применению языков сценариев. По каждому из языков сценариев имеется обширная литература, где подробно описана их структура и методики применения. Основная цель этой главы — продемонстрировать, каким образом языки сценариев поддерживаются в Spring Framework, а также предоставить наглядный пример тех дополнительных преимуществ, которые дает применение языка сценариев, помимо Java, при разработке приложений в Spring.

Как пользоваться поддержкой сценариев в Java

В версии Java 6 в комплект JDK встроена поддержка прикладного интерфейса API для написания сценариев на платформе Java (спецификация JSR-223). Цель состоит в том, предоставить стандартный механизм для выполнения в среде JVM кода, написанного на других языках сценариев. В состав JDK 6 входит библиотека Mozilla

Rhino, способная интерпретировать и выполнять программы на JavaScript. В этом разделе будет представлено введение в поддержку спецификации JSR-223 в JDK 6.

Классы поддержки сценариев в JDK 6 входят в состав пакета `javax.script`. Итак, начнем с разработки простой программы для извлечения списка механизмов выполнения сценариев. Соответствующий класс приведен ниже.

```
package com.apress.prospring5.ch14;

import javax.script.ScriptEngineManager;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class ListScriptEngines {
  private static Logger logger =
     LoggerFactory.getLogger(ListS criptEngines.class);

  public static void main(String... args) {
    ScriptEngineManager mgr = new ScriptEngineManager();
    mgr.getEngineFactories().forEach(factory -> {
      String engineName = factory.getEngineName();
      String languageName = factory.getLanguageName();
      String version = factory.getLanguageVersion();
      logger.info("Engine name: " + engineName
                + " Language: " + languageName
                + " Version: " + version);
    });
  }
}
```

Сначала в приведенном выше коде получается экземпляр класса `ScriptEngine Manager`, который будет обнаруживать и вести список механизмов (иными словами, классов, реализующих интерфейс `javax.script.ScriptEngine`) из пути к классам. Затем через вызов метода `ScriptEngineManager.getEngineFactories()` извлекается список экземпляров класса, реализующего интерфейс `ScriptEngine Factory`. Интерфейс `ScriptEngineFactory` служит для описания и получения экземпляров механизмов сценариев. Из каждого такого экземпляра можно извлечь сведения о поддерживаемом языке сценариев. В зависимости от настроек конкретной системы выполнение данной программы может дать разный результат, хотя он должен быть подобным приведенному ниже:

```
INFO: Engine name: AppleScriptEngine Language:
      AppleScript Version: 2.5
INFO: Engine name: Oracle Nashorn Language:
      ECMAScript Version: ECMA - 262 Edition 5.1
```

А теперь напишем простую программу для вычисления простого выражения на языке JavaScript. Исходный код этой программы приведен ниже.

```
package com.apress.prospring5.ch14.javascript;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class JavaScriptTest {
  private static Logger logger =
          LoggerFactory.getLogger(JavaScriptTest.class);

  public static void main(String... args) {
    ScriptEngineManager mgr = new ScriptEngineManager();
    ScriptEngine jsEngine =
                mgr.getEngineByName("JavaScript");
    try {
      jsEngine.eval("print('Hello JavaScript in Java')");
    } catch (ScriptException ex) {
       logger.error("JavaScript expression cannot "
                + "be evaluated!", ex);
    }
  }
}
```

Сначала в приведенном выше коде экземпляр интерфейса `ScriptEngine` извлекается по имени `JavaScript` из класса `ScriptEngineManager`. Затем вызывается метод `ScriptEngine.eval()`, которому передается аргумент типа `String`, содержащий выражение на языке JavaScript. Обратите внимание на то, что в качестве аргумента может быть также указан класс `java.io.Reader`, способный читать исходный код JavaScript из файла. Выполнение данной программы приводит к выводу на консоль следующего результата:

```
Hello JavaScript in Java
```

Приведенные выше примеры призваны дать основное представление о том, каким образом сценарии выполняются в Java. Но ведь организовать простой вывод информации из сценариев, написанных на других языках, не особенно интересно. Поэтому в следующем разделе представлено введение в Groovy — эффективный и универсальный язык сценариев.

Введение в Groovy

Проект Groovy был начат в 2003 году Джеймсом Стрэченом (James Strachan). Основная цель этого проекта — предоставить гибкий и динамический язык для виртуальной машины JVM с возможностями, характерными для других распространенных языков сценариев, включая Python, Ruby и Smalltalk. Язык Groovy служит надстройкой над Java, расширяет и устраняет некоторые его недостатки.

В последующих разделах описаны основные средства и понятия языка Groovy, а также показано, как он дополняет Java для удовлетворения особых потребностей в приложениях. Однако многие упоминаемые здесь языковые средства доступны и в других языках сценариев (например, в Scala, Erlang, Python и Clojure).

Динамическая типизация

Одно из главных отличий Groovy (и многих других языков сценариев) от Java состоит в поддержке динамической типизации переменных. В языке Java все свойства и переменные должны быть статически типизированными. Иными словами, в операторе объявления для них должен быть указан тип данных. А в Groovy поддерживается динамическая типизация переменных. Динамически типизированные переменные в Groovy объявляются с помощью ключевого слова `def`.

Рассмотрим действие динамической типизации на конкретном примере, разработав простой сценарий на языке Groovy. Исходный файл, содержащий класс или сценарий Groovy, имеет расширение `groovy`. Ниже приведен простой сценарий Groovy, в котором демонстрируется динамическая типизация.

```
package com.apress.prospring5.ch14

class Singer {
    def firstName
    def lastName
    def birthDate
    String toString() {
        "($firstName,$lastName,$birthDate)"
    }
}

Singer singer = new Singer(firstName: 'John',
       lastName: 'Mayer', birthDate: new Date(
           (new GregorianCalendar(1977, 9, 16))
           .getTime().getTime()))

Singer anotherSinger = new Singer(firstName: 39,
       lastName: 'Mayer', birthDate: new Date(
           (new GregorianCalendar(1977, 9, 16))
           .getTime().getTime()))

println singer
println anotherSinger

println singer.firstName + 39
println anotherSinger.firstName + 39
```

Этот сценарий Groovy может быть запущен на выполнение непосредственно в IDE, выполнен без компиляции (в Groovy предоставляется инструментальное сред-

ство командной строки, называемое `groovy` и способное выполнять сценарии Groovy непосредственно) или же скомпилирован в байт-код Java и выполнен подобно другим классам Java. Для выполнения сценариев Groovy не требуется обязательный метод `main()`. Кроме того, объявлять имя класса, которое должно совпадать с именем файла, совсем не обязательно.

В данном примере определяется класс `Singer` с тремя свойствами, динамически типизированными с помощью ключевого слова `def`. В этом классе переопределяется метод `toString()` с помощью замыкания, возвращающего символьную строку.

Далее конструируются два объекта типа `Singer` с помощью сокращенного синтаксиса, предоставляемого в Groovy для определения свойств. В свойстве `firstName` первого объекта типа `Singer` задается строковое значение типа `String`, тогда как в аналогичном свойстве второго объекта типа `Singer` — целочисленное значение. И, наконец, с помощью оператора `println` (его действие аналогично вызову метода `System.out.println()`) выводятся два объекта типа `Singer`. Чтобы продемонстрировать, каким образом в Groovy осуществляется динамическая типизация, в данном примере предусмотрены два оператора `println`, выводящие результат выполнения операции `firstName + 39`. Обратите внимание на то, что при передаче аргумента методу указывать круглые скобки в Groovy совсем не обязательно.

Выполнение сценария из данного примера дает следующий результат:

```
John,Mayer,Sun Oct 16 00:00:00 EET 1977
39,Mayer,Sun Oct 16 00:00:00 EET 1977
John39
78
```

Как следует из приведенного выше результата, объект успешно конструируется при передаче строкового или целочисленного значения, поскольку свойство `firstName` определено с динамической типизацией. Кроме того, в последних двух операторах `println` операция сложения была корректно выполнена над свойством `firstName` обоих полученных объектов. В первом случае к значению свойства `firstName` добавляется строка `"39"`, поскольку значение данного свойства относится к типу `String`. А во втором случае к значению свойства `firstName` добавляется целое число `39`, так как значение данного свойства относится к типу `int`. Поддержка динамической типизации в Groovy обеспечивает большую гибкость в манипулировании свойствами класса и переменными в прикладном коде.

Упрощенный синтаксис

В языке Groovy предоставляется также упрощенный синтаксис, поэтому для реализации одной и той же логики на Groovy потребуется меньший объем кода, чем на Java. Ниже перечислены некоторые особенности упрощенного синтаксиса Groovy.

- Для завершения оператора точка с запятой не требуется.
- Ключевое слово `return` в методах не является обязательным.

- Все методы и классы по умолчанию определены как открытые (`public`), поэтому указывать ключевое слово `public` в объявлениях методов не обязательно.
- В классе Groovy автоматически формируются методы получения и установки значений в объявляемых свойствах. Это означает, что в классе Groovy достаточно объявить тип и имя свойства (например, `String firstName` или `def firstName`), а для доступа к нему из любых других классов Groovy или Java будут автоматически вызываться методы получения и установки. Кроме того, к свойству можно обращаться без префикса **`get`** или **`set`** (например, `contact.firstName = 'John'`). Такое обращение будет интерпретировано в Groovy должным образом.

Помимо упрощенного синтаксиса, в языке Groovy предоставляется немало удобных методов для обращения к прикладному интерфейсу с API коллекций Java. В качестве примера ниже демонстрируется применение некоторых распространенных операций для манипулирования списками в Groovy.

```
def list = ['This', 'is', 'John Mayer']

println list
assert list.size() == 3

assert list.class == ArrayList

assert list.reverse() == ['John Mayer', 'is', 'This']

assert list.sort{it.size()} == ['is', 'This', 'John Mayer']

assert list[0..1] == ['is', 'This']
```

В приведенном выше коде представлена лишь очень малая доля возможностей языка Groovy. За более подробными сведениями обращайтесь к документации, оперативно доступной по адресу `http://www.groovy-lang.org/documentation.html`.

Замыкание

Одним из наиболее важных средств, которыми язык Groovy дополняет Java, является поддержка замыканий. Замыкание позволяет разместить фрагмент кода в объекте и свободно передавать его в пределах приложения. Замыкание является весьма эффективным языковым средством, благодаря которому становится возможным интеллектуальное и динамическое поведение. Внедрение поддержки замыканий в язык Java запрашивалось на протяжении длительного периода времени. Спецификация JSR-335 (Lambda Expressions for the Java Programming Language — Лямбда-выражения для языка программирования Java), направленная на поддержку программирования в многоядерной среде благодаря внедрению замыканий и связанных с ними язы-

ковых средств Java, включена в версию Java 8 и поддерживается в новой версии Spring Framework 4.

Ниже приведен простой пример применения замыканий в Groovy (из исходного файла `Runner.groovy`).

```
def names = ['John', 'Clayton', 'Mayer']

names.each {println 'Hello: ' + it}
```

Сначала здесь объявляется список, а затем с помощью удобного метода `each()` выполняется операция над каждым элементом в списке. В качестве аргумента методу `each()` передается замыкание, которое в Groovy заключается в фигурные скобки. В итоге логика из замыкания будет применена к каждому элементу списка. В самом замыкании специальная переменная `it` служит в Groovy для представления элемента списка в текущем контексте. Таким образом, замыкание сначала снабдит каждый элемент списка префиксом в виде символьной строки `"Hello: "`, а затем и его вывод на консоль. Выполнение этого простого сценария даст следующий результат:

```
Hello: John
Hello: Clayton
Hello: Mayer
```

Как упоминалось ранее, замыкание можно объявить как переменную и затем применять по мере надобности. Ниже приведен еще один пример применения замыкания.

```
def map = ['a': 10, 'b': 50]

Closure square = {key, value -> map[key] = value * value}

map.each square

println map
```

В данном примере определяется отображение `map` и объявляется переменная `square` типа `Closure`. Замыкание принимает в качестве аргументов ключ и значение элемента отображения, а логика выполнения вычисляет квадрат значения, связанного с ключом. Выполнение данного сценария дает следующий результат:

```
[a:100, b:2500]
```

В этом разделе представлено самое простое введение в замыкания. А в следующем разделе будет продемонстрировано построение простого механизма выполнения средствами Groovy и Spring, где будут также применяться замыкания. За более подробными сведениями о применении замыканий в Groovy обращайтесь к документации, оперативно доступной по адресу `http://www.groovy-lang.org/documentation.html`.

Применение Groovy в Spring

Главное преимущество, которое Groovy и другие языки сценариев привносят в приложения на Java, заключается в поддержке динамического поведения. Используя замыкания, можно упаковать бизнес-логику в объект и передавать ее в пределах приложения подобно любым другим переменным.

Еще одним важным языковым средством Groovy является поддержка разработки языков DSL с помощью упрощенного синтаксиса и замыканий. Как упоминалось ранее, DSL — это язык, ориентированный на конкретную предметную область с очень конкретными целями проектирования и реализации. Цель состоит в том, чтобы создать язык, который будет понятен не только разработчикам, но и бизнес-аналитикам и прочим пользователям. Чаще всего предметной областью для языков DSL служит сфера коммерческой деятельности. Например, языки DSL могут быть определены для классификации заказчиков, расчета налога с продаж, начисления заработной платы и т.д.

В этом разделе будет продемонстрировано применение Groovy для реализации простого механизма выполнения правил благодаря поддержке языков DSL в Groovy. Представленная здесь реализация повторяет пример из превосходной статьи на данную тему, доступной по адресу `www.pleus.net/articles/grules/grules.pdf`, хотя и с некоторыми изменениями. Кроме того, в этом разделе будет показано, каким образом поддержка обновляемых компонентов Spring Beans делает возможным оперативное обновление правил, не требуя компиляции, упаковки и развертывания приложения. В данном примере реализуется правило, применяемое для классификации конкретного певца по разным категориям, основанным на возрасте, который вычисляется на основании свойства даты рождения.

Разработка предметной области для певцов

Как упоминалось ранее, язык DSL ориентирован на конкретную предметную область, которая в большинстве случаев имеет отношение к некоторому виду коммерческих данных. Правило, которое мы собираемся реализовать, предназначено для применения в предметной области, связанной со сведениями о певцах.

Таким образом, первый шаг состоит в разработке объектной модели предметной области, в которой должно применяться правило. В данном примере модель предметной области очень проста и содержит единственный класс сущности `Singer`, исходный код которого приведен ниже. Обратите внимание на то, что это простой класс типа POJO, похожий на аналогичные классы, упоминавшиеся в предыдущих главах книги.

```
package com.apress.prospring5.ch14;

import org.joda.time.DateTime;

public class Singer {
```

```
  private Long id;
  private String firstName;
  private String lastName;
  private DateTime birthDate;
  private String ageCategory;

  ... // методы получения и установки

  @Override
  public String toString() {
    return "Singer - Id: " + id + ", First name: "
         + firstName + ", Last name: " + lastName + ",
           Birthday: " + birthDate + ", Age category: "
         + ageCategory;
  }
}
```

В классе `Singer` представляются простые сведения о певце. Для свойства `ageCategory` требуется разработать динамическое правило, которое можно применять для классификации. По этому правилу будет вычисляться возраст на основе свойства `birthDate` и затем устанавливаться свойство `ageCategory` в зависимости от получаемого результата: `kid` (ребенок), `youth` (юноша) или `adult` (взрослый).

Реализация механизма выполнения правил

Следующий шаг заключается в разработке простого процессора правил, предназначенного для применения правил к объекту предметной области. Для начала необходимо определить, какую информацию должно содержать правило. Ниже приведен класс `Rule` на языке Groovy (из исходного файла `Rule.groovy`).

```
package com.apress.prospring5.ch14

class Rule {
  private boolean singlehit = true
  private conditions = new ArrayList()
  private actions = new ArrayList()
  private parameters = new ArrayList()
}
```

У каждого правила имеется несколько свойств. В частности, свойство `conditions` определяет различные условия, которые механизм выполнения правил должен проверять для обработки объекта предметной области. Свойство `actions` определяет действия, которые должны выполняться, если условия удовлетворяются. А свойство `parameters` определяет поведение правила вследствие действий, выполняемых при разных условиях. И, наконец, свойство `singlehit` определяет, должно ли правило немедленно завершить свое выполнение, как только условия будут удовлетворены.

На следующем шаге строится механизм для выполнения правил. Ниже приведен интерфейс `RuleEngine` (обратите внимание на то, что это интерфейс Java).

```
package com.apress.prospring5.ch14;

public interface RuleEngine {
  void run(Rule rule, Object object);
}
```

В этом интерфейсе определяется единственный метод `run()`, предназначенный для применения правила к аргументу, обозначающему объект предметной области. Предоставим реализацию механизма выполнения правил на языке Groovy. Ниже приведен исходный код класса Groovy под именем `RuleEngineImpl` (из исходного файла `RuleEngineImpl.groovy`).

```
package com.apress.prospring5.ch14

import org.slf4j.Logger
import org.slf4j.LoggerFactory
import org.springframework.stereotype.Component

@Component("ruleEngine")
class RuleEngineImpl implements RuleEngine {
  Logger logger =
         LoggerFactory.getLogger(RuleEngineImpl.class);

  void run(Rule rule, Object object) {
    logger.info "Executing rule"

    def exit=false

    rule.parameters.each {ArrayList params ->
        def paramIndex=0
        def success=true
        if(!exit) {
          rule.conditions.each {
            logger.info "Condition Param index: "
                      + paramIndex
            success = success && it(object,
                                    paramsparamIndex)
            logger.info "Condition success: " + success
            paramIndex++
          }

          if(success && !exit) {
            rule.actions.each {
              logger.info "Action Param index: "
                        + paramIndex
              it(object,paramsparamIndex)
              paramIndex++
            }
```

```
            if (rule.singlehit) {
              exit=true
            }
          }
        }
      }
    }
}
```

Класс `RuleEngineImpl` реализует интерфейс `RuleEngine` на языке Java, и к нему применяется аннотация Spring, как и к любому другому простому объекту POJO. В методе `run()` параметры, определенные в правиле, по очереди передаются замыканию на обработку. Для каждого параметра (в виде списка значений) поочередно проверяется совпадение условий, каждое из которых является замыканием, с соответствующим элементом из списка параметров и объектом предметной области. Признак удачного исхода проверки (`success`) принимает логическое значение `true` только в том случае, если совпали все условия. В этом случае действия, каждое из которых также является замыканием и определено в правиле, будут выполнены над объектом с соответствующим значением из списка параметра. И, наконец, если условие совпадает со значением конкретного параметра, а переменная `singlehit` принимает логическое значение `true`, выполнение правила немедленно завершается.

Чтобы сделать способ извлечения правила более гибким, определим интерфейс `RuleFactory`, как показано ниже. Обратите внимание на то, что этот интерфейс определен на языке Java.

```
package com.apress.prospring5.ch14;

public interface RuleFactory {
  Rule getAgeCategoryRule();
}
```

Для классификации певцов по возрастным категориям имеется только одно правило, и поэтому в интерфейсе `RuleFactory` определен единственный метод для извлечения данного правила. Чтобы сделать механизм выполнения правил прозрачным для потребителя, разработаем простой уровень обслуживания и разместим на нем этот механизм. Ниже приведен исходный код интерфейса `SingerService` и класса `SingerServiceImpl` соответственно. Обратите внимание на то, что оба они написаны на языке Java.

```
// Исходный файл SingerService.java
package com.apress.prospring5.ch14;

public interface SingerService {
  void applyRule(Singer singer);
}

// Исходный файл SingerServiceImpl.java
```

```
package com.apress.prospring5.ch14;
import org.springframework.beans.factory.annotation
          .Autowired;

import org.springframework.context.ApplicationContext;
import org.springframework.stereotype.Service;

@Service("singerService")
public class SingerServiceImpl implements SingerService {
  @Autowired
  ApplicationContext ctx;

  @Autowired
  private RuleFactory ruleFactory;

  @Autowired
  private RuleEngine ruleEngine;
  public void applyRule(Singer singer) {
    Rule ageCategoryRule =
         ruleFactory.getAgeCategoryRule();
    ruleEngine.run(ageCategoryRule, singer);
  }
}
```

Как видите, требующиеся компоненты Spring Beans автоматически связываются с классом реализации службы. В методе `applyRule()` правило получается из фабрики правил и затем применяется к объекту типа `Singer`. В итоге свойство `ageCategory` будет выведено для объекта типа `Singer`, исходя из условий, действий и параметров, определенных в правиле.

Реализация фабрики правил в виде обновляемого компонента Spring Bean

Теперь можно реализовать фабрику правил и правило для классификации по возрастным категориям. В данном случае требуется иметь возможность обновлять правило оперативно, в Spring должны отслеживаться изменения в данном правиле, чтобы применить самую актуальную логику. В каркасе Spring Framework обеспечивается превосходная поддержка компонентов Spring Beans, которые написаны на языках сценариев и называются *обновляемыми компонентами*.

В этом разделе будет показано, как сконфигурировать сценарий Groovy в виде компонента SpringBean и указать каркасу Spring на необходимость его обновления через регулярные промежутки времени. Рассмотрим сначала реализацию фабрики правил в Groovy. Чтобы сделать возможным динамическое обновление, разместим исходный файл данного класса во внешней папке, чтобы в него можно было вносить коррективы. Назовем эту папку `rules`. В ней будет размещен исходный файл `Rule`

FactoryImpl.groovy, содержащий класс RuleFactoryImpl, написанный на языке Groovy, как показано ниже.

```
package com.apress.prospring5.ch14

import org.joda.time.DateTime
import org.joda.time.Years
import org.springframework.stereotype.Component;

@Component
class RuleFactoryImpl implements RuleFactory {
  Closure age = { birthDate -> return
                  Years.yearsBetween(birthDate,
                  new DateTime()).getYears() }
  Rule getAgeCategoryRule() {
    Rule rule = new Rule()

    rule.singlehit=true

    rule.conditions=[ {object, param ->
                       age(object.birthDate) >= param},
                       {object, param ->
                        age(object.birthDate) <= param}]

    rule.actions=[{object, param ->
                   object.ageCategory = param}]

    rule.parameters=[
        [0,10,'Kid'],
        [11,20,'Youth'],
        [21,40,'Adult'],
        [41,60,'Matured'],
        [61,80,'Middle-aged'],
        [81,120,'Old']
    ]

    return rule
  }
}
```

Класс RuleFactoryImpl реализует интерфейс RuleFactory, а метод getAge CategoryRule() предназначен для предоставления правила. В данном правиле определено замыкание (Closure), именуемое age и предназначенное для вычисления возраста на основе свойства birthDate (типа DateTime из библиотеки Joda-Time), определенного в объекте типа Singer.

В данном правиле определяются два условия. По первому условию проверяется, больше или равен возраст певца предоставленному значению параметра, а по второму условию — меньше или равен ему. Затем определяется одно действие для при-

сваивания значения предоставляемого параметра свойству ageCategory, определенному в объекте типа Singer.

Параметры определяют значения для проверки условий и действия. Например, первый параметр означает, что если возраст находится в пределах от 0 до 10, то свойству ageCategory в объекте типа Singer будет присвоено значение 'Kid', и т.д. Таким образом, первые два значения каждого параметра будут использованы в двух условиях для проверки возрастного диапазона, а последнее значение — для установки свойства ageCategory.

На следующем шаге необходимо определить контекст типа ApplicationContext. Ниже приведена соответствующая конфигурация из файла app-context.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi=
            "http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context=
            "http://www.springframework.org/schema/context"
       xmlns:lang=
            "http://www.springframework.org/schema/lang"
       xsi:schemaLocation=
            "http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans
             /spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context
             /spring-context.xsd
       http://www.springframework.org/schema/lang
       http://www.springframework.org/schema/lang
             /spring-lang.xsd">

    <context:component-scan
             base-package="com.apress.prospring5.ch14" />

    <lang:groovy id="ruleFactory"
          refresh-check-delay="5000"
       script-source="file:rules/RuleFactoryImpl.groovy"/>
</beans>
```

Приведенная выше конфигурация довольно проста. Для определения компонентов Spring Beans на языке сценариев необходимо использовать пространство имен lang. Дескриптор разметки <lang:groovy> применяется для объявления компонента Spring Bean вместе со сценарием Groovy. В атрибуте script-source указывается местоположение файла сценария Groovy, который будет загружаться в Spring. Для обновляемого компонента Spring Bean должен быть предоставлен атрибут refresh-check-delay. В данном случае в этом атрибуте задается значение 5000 миллисекунд, что вынуждает каркас Spring проверять изменения в файле сценария, если с момента последнего вызова прошло больше 5 секунд. Обратите внимание на

то, что Spring не будет проверять изменения в файле сценария каждые 5 секунд. Вместо этого проверка будет производиться только при обращении к соответствующему компоненту Spring Bean.

Проверка правила возрастной категории

Итак, все готово для проверки определенного выше правила. Ниже приведена соответствующая тестовая программа в виде класса Java.

```
package com.apress.prospring5.ch14;

import org.joda.time.DateTime;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.support
          .GenericXmlApplicationContext;

public class RuleEngineDemo {
  private static Logger logger =
          LoggerFactory.getLogger(RuleEngineTest.class);

  public static void main(String... args)
         throws Exception {
    GenericXmlApplicationContext ctx =
              new GenericXmlApplicationContext();
    ctx.load("classpath:spring/app-context.xml");
    ctx.refresh();

    SingerService singerService = ctx.getBean(
                  "singerService", SingerService.class);

    Singer singer = new Singer();
    singer.setId(1l);
    singer.setFirstName("John");
    singer.setLastName("Mayer");
    singer.setBirthDate(
           new DateTime(1977, 10, 16, 0, 0, 0, 0));
    singerService.applyRule(singer);
    logger.info("Singer: " + singer);

    System.in.read();

    singerService.applyRule(singer);
    logger.info("Singer: " + singer);

    ctx.close();
  }
}
```

После инициализации контекста типа `GenericXmlApplicationContext` в Spring конструируется экземпляр объекта типа `Singer`. Затем получается экземпляр типа `ContactService` для применения правила к объекту типа `Singer`, а результат выводится на консоль. Тестовая программа останавливается в ожидании ввода данных пользователем, после чего правило будет применено во второй раз. В течение этой паузы можно модифицировать класс `RuleFactoryImpl.groovy`, чтобы компонент Spring Bean был обновлен и можно было проверить измененное правило в действии.

Выполнение данной тестовой программы дает следующий результат:

```
00:34:24.814 [main] INFO c.a.p.c.RuleEngineImpl
   - Executing rule[1]
00:34:24.822 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition Param index:[2] 0
00:34:24.851 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition success: true[3]
00:34:24.858 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition Param index: 1
00:34:24.858 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition success: false[4]
00:34:24.858 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition Param index: 0
00:34:24.858 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition success: true
00:34:24.858 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition Param index: 1
00:34:24.858 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition success: false
00:34:24.859 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition Param index: 0
00:34:24.859 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition success: true
00:34:24.859 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition Param index: 1
00:34:24.859 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition success: true
00:34:24.860 [main] INFO c.a.p.c.RuleEngineImpl
   - Action Param index:[5] 2
00:34:24.870 [main] INFO c.a.p.c.RuleEngineDemo
   - Singer: Singer - Id: 1,
     First name: John,
     Last name: Mayer,
```

---

[1] Выполнение правила

[2] Индекс параметра условия:

[3] Условие выполнение успешно: верно

[4] Условие выполнение успешно: неверно

[5] Индекс параметра действия:

```
    Birthday: 1977-10-16T00:00:00.000+03:00,
    Age category: Adult[6]
```

Возраст певца составляет 39 лет, поэтому в проверяемом правиле обнаружено совпадение с третьим параметром (т.е. `[21, 40, 'Adult']`). В итоге устанавливается значение `'Adult'` в свойстве `ageCategory`. А теперь, когда выполнение тестовой программы приостановлено, изменим параметры в классе `RuleFactoryImpl.groovy`, как показано ниже.

```
rule.parameters=[
    [0,10,'Kid'],
    [11,20,'Youth'],
    [21,30,'Adult'],
    [31,60,'Middle-aged'],
    [61,120,'Old']
]
```

Внесите эти изменения и сохраните их в исходном файле `RuleFactoryImpl.groovy`. Затем нажмите клавишу <Enter> на консоли, чтобы инициировать второе применение правила к тому же самому объекту. В итоге на консоли появится следующий результат:

```
00:48:50.137 [main] INFO c.a.p.c.RuleEngineImpl
   - Executing rule
00:48:50.137 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition Param index: 0
00:48:50.137 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition success: true
00:48:50.138 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition Param index: 1
00:48:50.138 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition success: false
00:48:50.138 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition Param index: 0
00:48:50.138 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition success: true
00:48:50.138 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition Param index: 1
00:48:50.138 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition success: false
00:48:50.138 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition Param index: 0
00:48:50.139 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition success: true
00:48:50.139 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition Param index: 1
00:48:50.139 [main] INFO c.a.p.c.RuleEngineImpl
```

[6] Возрастная категория: Взрослый

```
   - Condition success: false
00:48:50.139 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition Param index: 0
00:48:50.139 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition success: true
00:48:50.139 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition Param index: 1
00:48:50.139 [main] INFO c.a.p.c.RuleEngineImpl
   - Condition success: true
00:48:50.139 [main] INFO c.a.p.c.RuleEngineImpl
    - Action Param index: 2
00:48:50.139 [main] INFO c.a.p.c.RuleEngineDemo
   - Singer: Singer - Id: 1,
     First name: John,
     Last name: Mayer,
     Birthday: 1977-10-16T00:00:00.000+03:00,
     Age category: Middle-aged[7]
```

Как видите, выполнение правила остановилось на четвертом параметре (т.е. `[31,60,'Middle-aged']`), в результате чего свойству `ageCategory` было присвоено значение `'Middle-aged'`. Если вы просмотрите статью, на которую мы ссылались при подготовке данного примера (`www.pleus.net/articles/grules/grules.pdf`), то найдете в ней пояснения, как вынести параметр правила во внешний файл формата Microsoft Excel, чтобы пользователи могли самостоятельно формировать и обновлять файл параметров.

Рассмотренное выше правило было, конечно, простым, но оно продемонстрировало, каким образом язык сценариев вроде Groovy может содействовать дополнению приложений Java EE, основанных на Spring, такими специальными функциональными средствами, как механизм выполнения правил с собственным языком DSL.

В связи с изложенным выше может возникнуть следующий вопрос: можно ли сохранить правило в базе данных, чтобы обновляемый компонент Spring Bean обнаружил изменения, внесенные в базу данных? В этом случае можно было бы еще больше упростить сопровождение правила, предоставив пользователям клиентской (или административной) части возможность оперативно обновлять правило в базе данных, а не загружать внешний файл.

В действительности в каркасе Spring Framework возникают трудности в связи с применением системы JIRA для отслеживания ошибок (подробнее они обсуждаются по адресу `https://jira.springsource.org/browse/SPR-5106`), поэтому следует проявлять осторожность, применяя описанный выше подход. Между тем предоставление пользовательского интерфейса для загрузки класса правила также является приемлемым решением. Но и в этом случае необходимо проявлять особую осторожность, поскольку вводимое пользователем правило должно быть тщательно проверено перед его загрузкой в рабочую среду.

---

[7] `Возрастная категория: Средний возраст`

Встраивание кода, написанного на динамическом языке

Код, написанный на динамическом языке, можно не только выполнять из внешних файлов, но и встраивать непосредственно в конфигурацию компонентов Spring Beans. И хотя подобный подход может оказаться удобным в таких случаях, как быстрая проверка концепций и т.д., его применение не рекомендуется для построения приложения в целом, поскольку затрудняет сопровождение такого приложения. Обратимся снова к механизму выполнения правил из предыдущего и удалим исходный файл `RuleEngineImpl.groovy`, предварительно встроив содержащийся в нем код в определение компонента Spring Bean, как показано в следующем фрагменте кода из файла конфигурации `app-context.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi=
            "http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context=
            "http://www.springframework.org/schema/context"
       xmlns:lang=
            "http://www.springframework.org/schema/lang"
       xsi:schemaLocation=
            "http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans
               /spring-beans.xsd
          http://www.springframework.org/schema/context
          http://www.springframework.org/schema/context
               /spring-context.xsd
          http://www.springframework.org/schema/lang
          http://www.springframework.org/schema/lang
               /spring-lang.xsd">

    <context:component-scan
             base-package="com.apress.prospring5.ch14"/>

    <lang:groovy id="ruleFactory" refresh-check-delay="5000">
         <lang:inline-script>
            <![CDATA[

       package com.apress.prospring5.ch14

       import org.joda.time.DateTime
       import org.joda.time.Years
       import org.springframework.stereotype.Component;

       @Component
       class RuleFactoryImpl implements RuleFactory {
```

```
        Closure age = { birthDate -> return
                        Years.yearsBetween(birthDate,
                        new DateTime()).getYears() }

        Rule getAgeCategoryRule() {
          Rule rule = new Rule()
          rule.singlehit = true
          rule.conditions = [{ object, param ->
               age(object.birthDate) >= param },
               { object, param ->
                 age(object.birthDate) <= param }]
          rule.actions = [{ object, param ->
                            object.ageCategory = param }]
          rule.parameters = [
              [0, 10, 'Kid'],
              [11, 20, 'Youth'],
              [21, 40, 'Adult'],
              [41, 60, 'Matured'],
              [61, 80, 'Middle-aged'],
              [81, 120, 'Old']
          ]
          return rule
        }
      }
           ]]>
      </lang:inline-script>
    </lang:groovy>
</beans>
```

Как следует из приведенного выше кода конфигурации, в него введен дескриптор `<lang:groovy>` с идентификатором `ruleEngine`, представляющим имя компонента Spring Bean. После этого был применен дескриптор `<lang:inline-script>` для инкапсуляции кода Groovy из исходного файла `RuleEngineImpl.groovy`. Код Groovy внедрен в дескриптор `<CDATA>`, что предотвращает его синтаксический разбор анализатором XML. Если выполнить исходный код из примера механизма выполнения правил еще раз, то можно обнаружить, что он работает, как и прежде, но в данном примере код Groovy был встроен непосредственно в определение компонента Spring Bean, а не сохранен во внешнем файле. Исходный код из файла `RuleEngine Impl.groovy` был намеренно использован в данном примере, чтобы продемонстрировать, насколько громоздким может стать приложение при встраивании больших объемов кода.

Резюме

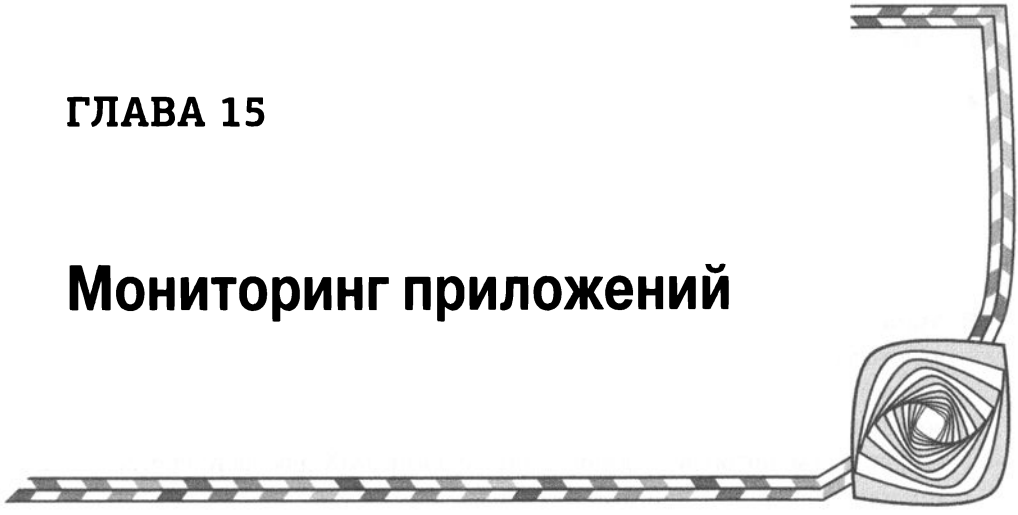
В этой главе было показано, как пользоваться языками сценариев в приложениях на Java, а также описана поддержка в каркасе Spring Framework языков сценариев,

помогающих внедрять динамическое поведение в приложение. Сначала в главе обсуждалась спецификация JSR-223, которая была внедрена в версии Java 6 и поддерживает выполнение сценариев на языке JavaScript. Затем было представлено введение в Groovy — язык сценариев, весьма распространенный в сообществе разработчиков приложений на Java. Здесь были также продемонстрированы некоторые из основных возможностей языка Groovy в сравнении с традиционным языком Java.

И, наконец, была рассмотрена поддержка языков сценариев в Spring Framework, продемонстрированная в действии на примере проектирования и реализации очень простого механизма выполнения правил, использующего поддержку языка DSL в Groovy. В этой главе было также пояснено, как видоизменить правило и принудить каркас Spring Framework автоматически обнаруживать изменения в этом правиле с помощью обновляемых компонентов Spring Beans, не прибегая к компиляции, упаковке и развертыванию приложения. И в конце этой главы было показано, каким образом код Groovy встраивается непосредственно в файл конфигурации для того, определить кода реализации компонента Spring Bean.

ГЛАВА 15

Мониторинг приложений



Типичное приложение на платформе JEE содержит несколько уровней и компонентов, в том числе уровень представления, уровень обслуживания, уровень сохраняемости и источник данных на сервере. На стадии разработки или после развертывания приложения в среде контроля качества (QA) или в производственной среде необходимо удостовериться, что приложение работоспособно и не имеет никаких потенциальных дефектов или узких мест.

В приложении на Java имеется немало участков, которые могут привести к снижению производительности или перегрузке таких серверных ресурсов, как центральный процессор, оперативная память и подсистема ввода-вывода. В качестве характерных примеров следует упомянуть неэффективный код Java, утечку памяти, когда в коде Java выделяются новые объекты без освобождения ссылок на них, что препятствует очистке оперативной памяти виртуальной машиной JVM в процессе сборки "мусора", параметры JVM, параметры пула потоков исполнения, конфигурация источников данных (например, количество разрешенных одновременных соединений с базой данных), настройка базы данных и наличие долго обрабатываемых запросов SQL.

Следовательно, необходимо понять поведение приложения во время выполнения и выявить любые потенциально узкие места или имеющиеся недостатки. В среде Java имеется немало инструментальных средств, которые помогают проводить мониторинг поведения приложений на платформе JEE во время выполнения. Большинство из них построены на основе технологии JMX (Java Management Extensions — управляющие расширения Java). В этой главе будет представлен ряд общих методик мониторинга приложений на платформе JEE, основанных на Spring. В частности, будут рассмотрены следующие вопросы:

- **Поддержка технологии JMX в Spring**. В этой части обсуждается комплексная поддержка технологии JMX в Spring и показано, как сделать компоненты Spring Beans открытыми для мониторинга с помощью инструментальных средств JMX и как пользоваться инструментальным средством VisualVM (`https://visualvm.github.io/?Java_VisualVM`) для мониторинга приложений.
- **Мониторинг статистики применения Hibernate.** Во многих библиотеках, включая и Hibernate, предоставляются поддерживающие классы и инфраструктура для доступа к рабочему состоянию и количественным показателям производительности с помощью технологии JMX. В этой части показано, как активизировать мониторинг с помощью технологии JMX для часто употребляемых компонентов приложений JEE в Spring.
- **Поддержка технологии JMX в модуле Spring Boot.** В этой части представлена стартовая библиотека, предоставляемая в модуле Spring Boot для поддержки технологии JMX с полностью готовой стандартной конфигурацией.

Однако эта глава не является введением в технологию JMX, поэтому предполагается наличие у вас элементарного представления о данной технологии. За подробными сведениями по данному вопросу обращайтесь к соответствующему ресурсу Oracle, оперативно доступному по адресу `www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html`.

Поддержка технологии JMX в Spring

В технологии JMX классы, доступные для мониторинга и управления, называются *управляемыми компонентами Spring Beans*, или *компонентами MBean*. В каркасе Spring Framework поддерживается несколько механизмов для доступа к компонентам MBean. Основное внимание в этой главе будет уделено доступу к компонентам Spring Beans, разработанным в виде простых объектов POJO, как к компонентам MBean для проведения мониторинга с помощью технологии JMX.

В последующих разделах обсуждается процедура доступа к компоненту Spring Bean, содержащему связанную с приложением статистику, в качестве компонента MBean для мониторинга средствами JMX, а также реализация компонента Spring Bean, доступ к компоненту Spring Bean как к компоненту MBean в контексте типа `ApplicationContext` и применение инструментального средства VisualVM для мониторинга компонентов MBean.

Экспорт компонентов Spring Beans в JMX

В качестве примера в этом разделе будет использоваться веб-служба REST, созданная в главе 12. Еще раз просмотрите исходный код соответствующего примера приложения, представленного в упомянутой главе, или код примеров, прилагаемый

к данной книге, где имеется исходный код, на который мы будем далее опираться. С помощью дополнений JMX в данном случае требуется выяснить количество записей о певцах в базе данных, чтобы провести мониторинг средствами JMX. С этой целью реализуем соответствующий интерфейс и класс, как показано ниже.

```
// Исходный файл AppStatistics.java
package com.apress.prospring5.ch15;

public interface AppStatistics {
  int getTotalSingerCount();
}
// Исходный файл AppStatisticsImpl.java
package com.apress.prospring5.ch15;

import com.apress.prospring5.ch12.services.SingerService;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.stereotype.Component;

public class AppStatisticsImpl implements AppStatistics {
  @Autowired
  private SingerService singerService;

  @Override
  public int getTotalSingerCount() {
    return singerService.findAll().size();
  }
}
```

В данном примере кода определяется метод для извлечения общего количества записей о певцах в базе данных. Чтобы получить доступ к компоненту Spring Bean как к средству JMX, придется внедрить конфигурацию в контекст типа `Application Context`. Это конфигурация защищенного в Spring веб-приложения, представленного в примере из главы 12. И теперь ее необходимо ввести в компоненты инфраструктуры Spring Beans типа `MBeanServer` и `MBeanExporter`, чтобы активизировать поддержку компонентов Spring Beans для управления JMX, как показано ниже.

```
package com.apress.prospring5.ch15.init;
...
import javax.management.MBeanServer;
import org.springframework.jmx.export.MBeanExporter;
import org.springframework.jmx.support
          .MBeanServerFactoryBean;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages =
               {"com.apress.prospring5.ch15"})
```

```
public class WebConfig implements WebMvcConfigurer {
  // другие компоненты Spring Beans Web, характерные
  // для инфраструктуры веб-приложений
  ...
  @Bean AppStatistics appStatisticsBean() {
    return new AppStatisticsImpl();
  }

  @Bean
  MBeanExporter jmxExporter() {
    MBeanExporter exporter = new MBeanExporter();
    Map<String, Object> beans = new HashMap<>();
    beans.put("bean:name=ProSpring5SingerApp",
              appStatisticsBean());
    exporter.setBeans(beans);
    return exporter;
  }
}
```

Сначала в приведенном выше коде объявляется компонент Spring Bean типа `AppStatisticsImpl`, к которому требуется получить доступ для получения статистики о простом объекте POJO, а затем компонент Spring Bean типа `MBeanExporter` — базового класса для поддержки технологии JMX в Spring Framework. Он отвечает за регистрацию компонентов Spring Beans на сервере компонентов JMX MBeans, где реализуется интерфейс `javax.management.MBeanServer` из комплекта JDK, присутствующий в большинстве распространенных веб- и JEE-контейнеров, в том числе Tomcat и WebSphere. При доступе к Spring Bean как к компоненту MBean каркас Spring попытается найти на сервере действующий экземпляр типа `MBeanServer` и с его помощью зарегистрировать данный компонент MBean. Например, в контейнере Tomcat экземпляр типа `MBeanServer` будет получен автоматически, поэтому никакого дополнительного конфигурирования не потребуется.

В компоненте `jmxExporter` свойство `beans` определяет те компоненты Spring Beans, к которым требуется доступ. Это свойство типа `Map`, поэтому в нем можно указать любое количество компонентов MBean. В данном случае требуется доступ к компоненту `appStatisticsBean`, содержащему сведения о приложении для певцов, которые нужно предоставить администраторам. Для определения компонента MBean будет использоваться ключ типа `ObjectName` (это класс `javax.management.ObjectName` из комплекта JDK) для компонента Spring Bean, на который ссылается значение из соответствующей записи. В приведенной выше конфигурации компонент `appStatisticsBean` будет доступен по ключу `bean:name=Prospring4ContactApp` типа `ObjectName`. По умолчанию все открытые свойства компонента Spring Bean доступны как атрибуты, а все открытые методы — как операции. А теперь, когда компонент MBean доступен для мониторинга средствами JMX, можно перейти к настройке инструментального средства VisualVM и применению его клиента JMX для мониторинга.

Настройка VisualVM для мониторинга средствами JMX

VisualVM — очень полезное инструментальное средство, помогающее проводить разнообразный мониторинг приложений на Java. Это инструментальное средство свободно доступно в папке `bin`, вложенной в папку установки комплекта JDK. На веб-сайте проекта для загрузки доступна также автономная версия VisualVM[1]. В примерах из этой главы применяется версия, доступная из установки комплекта JDK.

Для поддержки различных функций мониторинга в VisualVM применяется система подключаемых модулей. Для поддержки мониторинга компонентов MBean приложений на Java необходимо установить подключаемый модуль VisualVM-MBeans, выполнив следующие действия.

1. Выберите в меню VisualVM команду **Tools⇨Plugins** (Сервис⇨Подключаемые модули).
2. Перейдите на вкладку **Available Plugins** (Доступные подключаемые модули) в открывшемся диалоговом окне **Plugins**.
3. Щелкните на кнопке **Check for Newest** (Проверить наличие новейших подключаемых модулей).
4. Выберите подключаемый модуль VisualVM-MBeans, установив флажок слева от его наименования, а затем щелкните на кнопке **Install** (Установить).

Вид диалогового окна **Plugins** приведен на рис. 15.1. По завершении установки подключаемого модуля запустите на выполнение контейнер сервлетов Tomcat и пример приложения для мониторинга. И тогда на левой панели **Applications** (Приложения) диалогового окна **Plugins** будет показано, что процесс Tomcat выполняется.

По умолчанию VisualVM просматривает те приложения на Java, которые работают на платформе JDK. Дважды щелкнув на нужном узле древовидной структуры, можно отобразить соответствующий экран мониторинга.

После установки подключаемого модуля VisualVM-MBeans появится вкладка **MBeans**. Перейдя на эту вкладку, можно просмотреть доступные компоненты MBean. В древовидной структуре слева на вкладке **MBeans** должен присутствовать узел `bean`. Если развернуть его, появится компонент MBean типа `Prospring4ContactApp`, который был сделан доступным. А справа на вкладке **MBeans** появится метод, реализованный в компоненте Spring Bean, с атрибутом `TotalContactCount`, который был автоматически выведен методом `getTotal ContactCount()` в данном компоненте. Значение этого атрибута, равное 3, обозначает количество записей, введенных в базу данных при запуске на выполнение контролируемого приложения. В обычном приложении это число изменялось бы в зависимости от количества записей о певцах, введенных во время выполнения данного приложения. Диалоговое окно **Plugins** с открытой вкладкой **MBeans** показано на рис. 15.2.

---

[1] На момент написания данной книги текущей считалась версия Java VisualVM 1.3.9. Она доступна для загрузки по адресу `http://visualvm.java.net/download.html`.

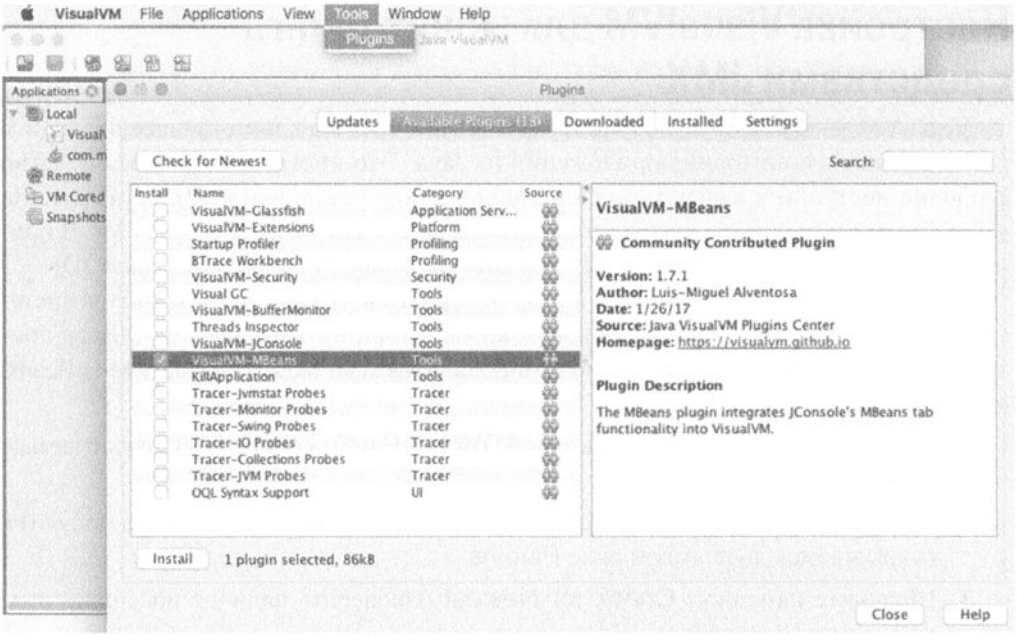


Рис. 15.1. Диалоговое окно Plugins в VisualVM

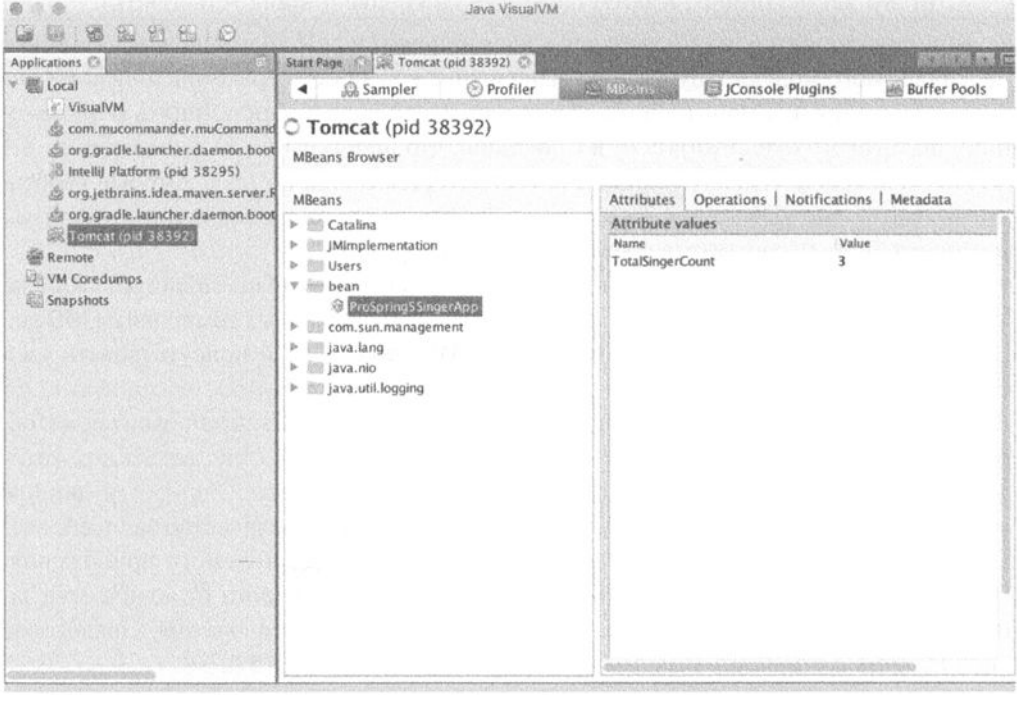


Рис. 15.2. Диалоговое окно Plugins с компонентом MBean типа **`Prospring5SingerApp`**, доступным в VisualVM

Мониторинг статистики применения Hibernate

В библиотеке Hibernate также поддерживается ведение количественных показателей сохраняемости и их доступность для технологии JMX. Чтобы сделать это возможным, придется добавить в конфигурацию прикладного интерфейса JPA два дополнительных свойства Hibernate, как показано ниже.

```
package com.apress.prospring5.ch12.config;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.data.jpa.repository.config
          .EnableJpaRepositories;
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseType;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa
          .LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor
          .HibernateJpaVendorAdapter;
import org.springframework.transaction
          .PlatformTransactionManager;
import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;
import java.util.Properties;

@Configuration
// применение компонентов, внедренных в проекте из главы 12
@EnableJpaRepositories(basePackages =
                       {"com.apress.prospring5.ch12.repos"})
@ComponentScan(basePackages =
               {"com.apress.prospring5.ch12"} )
public class DataServiceConfig {
  ...

  @Bean
  public Properties hibernateProperties() {
    Properties hibernateProp = new Properties();
    hibernateProp.put("hibernate.dialect",
                      "org.hibernate.dialect.H2Dialect");
```

```
    hibernateProp.put("hibernate.hbm2ddl.auto", "create-drop");
    hibernateProp.put("hibernate.show_sql", true);
    hibernateProp.put("hibernate.max_fetch_depth", 3);
    hibernateProp.put("hibernate.jdbc.batch_size", 10);
    hibernateProp.put("hibernate.jdbc.fetch_size", 50);

    hibernateProp.put("hibernate.jmx.enabled", true);
    hibernateProp.put("hibernate.generate_statistics",
                      true);
    hibernateProp.put("hibernate.session_factory_name",
                      "sessionFactory");
    return hibernateProp;
  }
  ...
}
```

Свойство `hibernate.jmx.enabled` служит для активизации режима работы JMX в библиотеке Hibernate. А свойство `hibernate.generate_statistics` предписывает библиотеке Hibernate сформировать статистику для своего поставщика услуг сохраняемости JPA, тогда как свойство `hibernate.session_factory_name` определяет имя фабрики сеансов, требующейся компоненту MBean, отвечающему за формирование статистики применения Hibernate.

И, наконец, этот компонент MBean необходимо ввести в конфигурацию класса `MBeanExporter` из каркаса Spring. В приведенном ниже фрагменте кода демонстрируется конфигурация компонента MBean, созданного ранее в классе `WebConfig`. Класс `CustomStatistics` служит заменой классу `org.hibernate.jmx.StatisticsService`, который больше не входит в состав версии Hibernate 5.[2]

```
package com.apress.prospring5.ch15.init;

...

@Configuration
@EnableWebMvc
@ComponentScan(basePackages =
               {"com.apress.prospring5.ch15"})
public class WebConfig implements WebMvcConfigurer {
  ...
  // Компоненты Spring Beans типа JMX
  @Bean AppStatistics appStatisticsBean() {
    return new AppStatisticsImpl();
  }
```

[2] Исходный код этого класса по-прежнему доступен в информационном хранилище GitHub по адресу `https://github.com/manuelbernhardt/hibernate-core/blob/master/hibernate-core/src/main/java/org/hibernate/jmx/StatisticsService.java`, если потребуется дополнить предоставляемую реализацию.

```
  @Bean CustomStatistics statisticsBean() {
    return new CustomStatistics();
  }

  @Autowired
  private EntityManagerFactory entityManagerFactory;
  @Bean SessionFactory sessionFactory() {
    return entityManagerFactory.unwrap(
                                SessionFactory.class);
  }

  @Bean
  MBeanExporter jmxExporter() {
    MBeanExporter exporter = new MBeanExporter();
    Map<String, Object> beans = new HashMap<>();
    beans.put("bean:name=ProSpring5SingerApp",
              appStatisticsBean());
    beans.put("bean:name=Prospring5SingerApp-hibernate",
              statisticsBean());
    exporter.setBeans(beans);
    return exporter;
  }
}
```

Метод `statisticsBean()` объявляется с помощью класса, реализующего интерфейс `org.hibernate.stat.Statistics` из библиотеки Hibernate в качестве базового компонента. Именно таким образом в библиотеке Hibernate поддерживается доступность статистки для технологии JMX.

Теперь статистика применения Hibernate активизирована и доступна через JMX. Перезагрузите контролируемое приложение и обновите VisualVM, чтобы появился компонент MBean, отвечающий за статистику в Hibernate. Если щелкнуть на узле `ProSpring3ContactApp-hibernate`, с правой стороны появится подробная статистика. Обратите внимание на то, что для получения статистических данных, не относящихся к примитивным типам данных Java, достаточно щелкнуть на соответствующем поле, чтобы расширить его для просмотра содержимого (например, статистических данных о типе `List`).

В инструментальном средстве VisualVM можно просмотреть и много других количественных показателей, в том числе `EntityNames` (Имена сущностей), `SessionOpenCount` (Подсчет открытых сеансов), `SecondCloseCount` (Подсчет вторичных сеансов) и `QueryExecutionMaxTime` (Максимальное время выполнения запроса). Значения этих количественных показателей полезны для лучшего понимания режима сохраняемости в приложениях и могут помочь при поиске и устранении неполадок, а также при настройке производительности.

Поддержка технологии JMX в модуле Spring Boot

Перенести приложение из рассмотренного выше примера в среду Spring Boot совсем не трудно, причем все его зависимости предоставляются и конфигурируются автоматически. Для поддержки технологии JMX в модуле Spring Boot специальная стартовая библиотека не предусмотрена, но в то же время можно внедрить библиотеку из архивного файла `spring-bootstarter-actuator.jar` в качестве зависимости. Она может оказать помощь в мониторинге приложения Spring непосредственно в IDE, чтобы отображать компоненты Spring Beans, рабочее состояние и преобразования в данном приложении, если в Spring применяется специальный подключаемый модуль.

Рассмотрим в качестве примера веб-приложение без интерфейса, поскольку этой теме посвящена глава 16, с базой данных, размещаемой в оперативной памяти, а также полноценной конфигурацией прикладного интерфейса JTA средствами Atomikos. Эта реализация уже была представлена в предыдущих главах, поэтому далее основное внимание будет уделено компонентам MBeans.

Обновим класс `AppStatisticsImpl` с помощью аннотации `@ManagedResource`, чтобы регистрировать экземпляры данного класса с помощью сервера JMX. Это практично потому, что в модуле Spring Boot по умолчанию создается компонент типа `MBeanServer` с идентификатором `mbeanServer` и становятся доступными любые компоненты Spring Beans, снабженные аннотациями JMX в Spring (`@Managed Resource`, `@ManagedAttribute`, `@ManagedOperation`). Ниже приведена обновленная версия класса `AppStatisticsImpl`, в которой применяются упомянутые здесь аннотации.

```
package com.apress.prospring5.ch15;

import com.apress.prospring5.ch15.entities.Singer;
import com.apress.prospring5.ch15.services.SingerService;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.jmx.export.annotation
          .ManagedAttribute;
import org.springframework.jmx.export.annotation
          .ManagedOperation;
import org.springframework.jmx.export.annotation
          .ManagedResource;
import org.springframework.stereotype.Component;
import java.util.List;

@Component
@ManagedResource(description = "JMX managed resource",
                 objectName = "jmxDemo:
                              name=ProSpring5SingerApp")
public class AppStatisticsImpl implements AppStatistics {
```

```
  @Autowired
  private SingerService singerService;

  @ManagedAttribute(description =
                    "Number of singers in the application")
  @Override
  public int getTotalSingerCount() {
    return singerService.findAll().size();
  }

  @ManagedOperation
  public String findJohn() {
    List<Singer> singers = singerService.
         findByFirstNameAndLastName("John", "Mayer");
    if (!singers.isEmpty()) {
      return singers.get(0).getFirstName() + " "
                  + singers.get(0).getLastName();
    }
    return "not found";
  }
}
```

По умолчанию модуль Spring Boot сделает доступными конечные точки управления в виде компонентов JMX MBeans в домене `org.springframework.boot`. В приведенном выше коде значение атрибута `objectName` в аннотации `@ManagedResource` обозначает домен и имя компонента MBean. В данном случае указан домен `jmxDemo`, поскольку требуется легко находить в VisualVM управляемые компоненты MBeans, созданные вручную. (Для внутреннего мониторинга в модуле Spring Boot предоставляются собственные автоматически сконфигурированные компоненты MBeans.)

Аннотация `@ManagedAttribute` служит для доступа к заданному свойству компонента Spring Bean как к атрибуту JMX, тогда как аннотация `@ManagedOperation` — для доступа к заданному методу как к операции JMX. Оба приведенных выше метода аннотированы по-разному, поэтому они появятся на разных вкладках в VisualVM. Так, результат вызова метода `getTotalSingerCount()` появится на вкладке **Attributes** (Атрибуты), а на вкладке **Operations** (Операции) оба метода появятся в виде экранных кнопок, на которых можно щелкнуть кнопкой мыши, чтобы вызвать их прямо на месте. И, наконец, на вкладке **Metadata** (Метаданные) можно просмотреть строку с описанием каждой аннотации.

На рис. 15.3 приведено диалоговое окно **Plugins** с компонентом MBean типа `Prospring5SingerApp`, доступным в домене `jmxDemo`. Ниже этого домена можно увидеть домен `org.springframework.boot`.

Но это еще не все! Необходимо активизировать поддержку технологии JMX. И для этой цели служит аннотация `@EnableMBeanExport`, активизирующая по умолчанию экспорт всех стандартных компонентов MBeans из контекста Spring, а

также всех компонентов Spring Beans, снабженных аннотацией `@ManagedResource`. По существу, эта аннотация предписывает модулю Spring Boot создать компонент Spring Bean типа `MBeanExporter` под именем `mbeanExporter`. Ниже приведен исходный код конфигурационного класса для рассматриваемого здесь примера приложения.

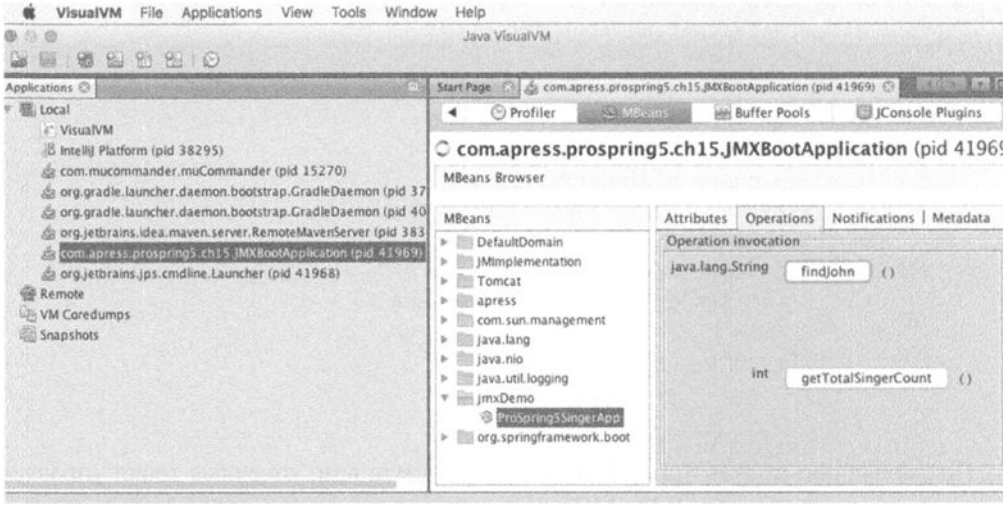


Рис. 15.3. Диалоговое окно Plugins с компонентом MBean типа **`Prospring5SingerApp`**, доступным в домене **`jmxDemo`**

```
package com.apress.prospring5.ch15;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation
          .EnableMBeanExport;
import java.io.IOException;

@EnableMBeanExport
@SpringBootApplication(scanBasePackages =
                       {"com.apress.prospring5.ch15"})
public class JMXBootApplication {
  private static Logger logger = LoggerFactory.
                getLogger(JMXBootApplication.class);
  public static void main(String args) throws IOException {
    ConfigurableApplicationContext ctx =
                SpringApplication.run(
                          JMXBootApplication.class, args);
    assert (ctx != null);
```

```
        logger.info("Started ...");
        System.in.read();
        ctx.close();
    }
}
```

Вот теперь можно сказать, что все, поскольку о применении технологии JMX в приложении Spring Boot больше нечего сказать!

Резюме

В этой главе в общих чертах были рассмотрены вопросы, связанные с мониторингом приложений JEE, построенных на основе Spring. Сначала была описана поддержка в Spring стандартной технологии JMX для мониторинга приложений на Java. Затем была продемонстрирована реализация специальных компонентов MBeans для доступа к статистическим данным контролируемого приложения, а также таких общих компонентов, как Hibernate. И в конце главы было показано, как пользоваться технологией JMX в приложении Spring Boot и чем в этом отношении отличается каркас Spring.

ГЛАВА 16

Разработка веб-приложений

Уровень представления в корпоративном приложении оказывает заметное влияние на восприятие этого приложения пользователями. Уровень представления — это своего рода "парадный вход" в приложение. Он дает пользователям возможность выполнять рабочие функции, предоставляемые приложением, а также обеспечивает наглядное представление информации, поддерживаемой в приложении. Эффективность работы пользовательского интерфейса в значительной степени способствует успешному введению в эксплуатацию всего приложения в целом.

Вследствие бурного роста Интернета (особенно теперь, когда появились самые разные типы мобильных устройств) разработка уровня представления в приложении становится все более сложной задачей. Ниже приведен ряд основных соображений, которые следует принимать во внимание при разработке веб-приложений.

- **Производительность.** Она всегда была главным требованием к веб-приложению. Если после выбора какой-нибудь функции или гиперссылки пользователи вынуждены слишком долго ожидать выполнения (три секунды ожидания в Интернете можно сравнить со столетием), то они будут явно не удовлетворены таким приложением.
- **Удобство для пользователей.** Приложение должно быть простым в эксплуатации и навигации с ясными инструкциями, не вводящими пользователя в заблуждение.
- **Интерактивность и расширенная функциональность.** Пользовательский интерфейс должен быть интерактивным и быстро реагирующим. Кроме того, уровень представления должен быть полнофункциональным с точки зрения наглядности представления информации, включая построение диаграммы, применение информационных панелей в пользовательском интерфейсе и т.п.

- **Доступность.** Пользователи требуют, чтобы приложение было доступно где угодно и на любом устройстве. На работе они будут пользоваться для доступа к приложению настольными системами, а в дороге — разнообразными мобильными устройствами, в том числе переносными и планшетными компьютерами или смартфонами.

Разработка веб-приложения, удовлетворяющего указанным выше требованиям, — не простая задача, но эти требования считаются пользователями обязательными. Правда, для удовлетворения этих требований за последнее время было разработано немало новых технологий и каркасов приложений. Во многих каркасах и библиотеках для разработки веб-приложений, в том числе Spring MVC (и Spring Web Flow), Struts, Tapestry, Java Server Faces (JSF), Google Web Toolkit (GWT), jQuery и Dojo, предоставляются инструментальные средства и развитые библиотеки компонентов, помогающие строить высокоинтерактивные пользовательские интерфейсы для веб-приложений. Кроме того, во многих каркасах предоставляются инструментальные средства или соответствующие библиотеки виджетов (графических компонентов), предназначенных для мобильных устройств, включая смартфоны и планшетные компьютеры. Развитие стандартов HTML5 и CSS3, а также их поддержка среди большинства веб-браузеров и производителей мобильных устройств также способствует упрощению разработки веб-приложений, которые должны быть доступными где угодно и на любом устройстве.

Для разработки веб-приложений в Spring обеспечивается обширная и всесторонняя поддержка. В частности, модуль Spring MVC предоставляет прочную инфраструктуру и архитектуру по проектному шаблону MVC (Model View Controller — модель–представление–контроллер), предназначенную для построения веб-приложений. Вместе с модулем Spring MVC можно применять различные технологии представления (например, JSP или Velocity). Кроме того, модуль Spring MVC вполне совместим со многими известными каркасами веб-приложений и наборами инструментальных средств, включая Struts и GWT. А другие проекты Spring помогают решать конкретные задачи при разработке веб-приложений. Например, проект Spring MVC в сочетании с проектом Spring Web Flow и его модулем Spring Faces обеспечивает всестороннюю поддержку для разработки веб-приложений со сложными потоками и применения JSF в качестве технологии представления. Проще говоря, для реализации уровня представления имеется немало вариантов выбора.

Основное внимание в этой главе уделяется модулю Spring MVC. В ней будет показано, как задействовать эффективные средства Spring MVC для разработки высокопроизводительных веб-приложений. В частности, будут рассмотрены следующие вопросы.

- **Модуль Spring MVC.** В этой части представлены основные понятия проектного шаблона MVC, а также введение в модуль Spring MVC. Здесь будут пояснены основные понятия Spring MVC, включая иерархию контекстов типа `Web ApplicationContext` и жизненный цикл обработки запросов.

- **Интернационализация, региональные настройки и тематическое оформление.** В модуле Spring MVC обеспечивается всесторонняя поддержка общих требований к веб-приложениям, включая интернационализацию (i18n), региональные настройки и тематическое оформление. В этой части будет обсуждаться применение модуля Spring MVC для разработки веб-приложений, удовлетворяющих этим требованиям.
- **Поддержка технологий представления и Ajax.** В модуле Spring MVC поддерживается немало технологий представления. В этой части основное внимание уделено применению технологий JavaServer Pages (JSP) и Tiles для реализации уровня представления в веб-приложениях, а для расширенных функциональных возможностей — язык JavaScript. Для этих целей имеется немало превосходных и широко распространенных библиотек JavaScript вроде jQuery и Dojo. В этой части будет показано, как пользоваться библиотекой jQuery вместе с ее подчиненной библиотекой jQuery UI, где поддерживается разработка высокоинтерактивных веб-приложений.
- **Поддержка разбиения на страницы и выгрузки файлов.** При обсуждении примеров в этой главе будет показано, как обеспечить средствами Spring Data JPA и клиентского компонента jQuery поддержку разбиения на страницы при просмотре данных с сеточной компоновкой. Здесь также поясняется, как реализовать в Spring MVC выгрузку файлов. Вместо интеграции с компонентом Apache Commons FileUpload будет описано применение модуля Spring MVC со встроенной в контейнер Servlet 3.1 многосторонней поддержкой выгрузки файлов.
- **Безопасность.** Тема безопасности веб-приложений довольно обширная. Поэтому в этой части показано, как пользоваться каркасом Spring Security для защиты веб-приложений и организации входа и выхода из них.

Реализация уровня обслуживания для примеров кода из этой главы

На уровне обслуживания для целей, преследуемых в этой главе, будет по-прежнему использоваться приложение для певцов. В этом разделе описывается модель данных и реализация уровня обслуживания для примеров кода, демонстрируемых далее в главе.

Модель данных для примеров кода

Модель данных для примеров кода из этой главы очень проста и содержит единственную таблицу `SINGER` для хранения сведений о певцах. Ниже приведен сценарий (из файла `schema.sql`) для создания схемы базы данных.

```
CREATE TABLE SINGER (
  ID INT NOT NULL AUTO_INCREMENT
```

```
  , FIRST_NAME VARCHAR(60) NOT NULL
  , LAST_NAME VARCHAR(40) NOT NULL
  , BIRTH_DATE DATE
  , DESCRIPTION VARCHAR(2000)
  , PHOTO BLOB
  , VERSION INT NOT NULL DEFAULT 0
  , UNIQUE UQ_SINGER_1 (FIRST_NAME, LAST_NAME)
  , PRIMARY KEY (ID)
);
```

Как видите, таблица `SINGER` содержит лишь несколько основных полей для хранения сведений о певцах. Следует лишь заметить, что в столбце `PHOTO` данные представлены в виде большого двоичного объекта BLOB, предназначенного для хранения фотографии певца из выгружаемого файла. Эту таблицу совсем не обязательно создавать с помощью упомянутого выше сценария. Вместо этого можно составить средствами Hibernate запрос SQL на создание таблицы, исходя из приведенной ниже конфигурации класса сущности `Singer`.

```
package com.apress.prospring5.ch16.entities;

import javax.persistence.*;
import javax.validation.constraints.NotEmpty;
import javax.validation.constraints.Size; import java
            .io.Serializable;
import java.text.SimpleDateFormat;
import java.util.Date;
import static javax.persistence.GenerationType.IDENTITY;

@Entity
@Table(name = "singer")
public class Singer implements Serializable {
  @Id
  @GeneratedValue(strategy = IDENTITY)
  @Column(name = "ID")
  private Long id;

  @Version
  @Column(name = "VERSION")
  private int version;

  @NotEmpty(message=
            "{validation.firstname.NotEmpty.message}")
  @Size(min=3, max=60, message="{validation.firstname.Size.message}")
  @Column(name = "FIRST_NAME")
  private String firstName;

  @NotEmpty(message=
            "{validation.lastname.NotEmpty.message}")
  @Size(min=1, max=40,
```

```
      message="{validation.lastname.Size.message}")
@Column(name = "LAST_NAME")
private String lastName;

@Temporal(TemporalType.DATE)
@Column(name = "BIRTH_DATE")
private Date birthDate;

@Column(name = "DESCRIPTION")
private String description;

@Basic(fetch= FetchType.LAZY)
@Lob
@Column(name = "PHOTO")
private byte photo;

public Long getId() {
  return id;
}

public int getVersion() {
  return version;
}

public String getFirstName() {
  return firstName;
}

public String getLastName() {
  return lastName;
}

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

public void setVersion(int version) {
  this.version = version;
}

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

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

public void setBirthDate(Date birthDate) {
```

```
    this.birthDate = birthDate;
  }

  public Date getBirthDate() {
    return birthDate;
  }

  public String getDescription() {
    return description;
  }

  public void setDescription(String description) {
    this.description = description;
  }

  public byte getPhoto() {
    return photo;
  }

  public void setPhoto(byte photo) {
    this.photo = photo;
  }

  @Transient
  public String getBirthDateString() {
    String birthDateString = "";
    if (birthDate != null) {
      SimpleDateFormat sdf =
            new SimpleDateFormat("yyyy-MM-dd");
      birthDateString = sdf.format(birthDate);
    }
    return birthDateString;
  }

  @Override
  public String toString() {
    return "Singer - Id: " + id + ",
           First name: " + firstName
         + ", Last name: " + lastName
         + ", Birthday: " + birthDate
         + ", Description: " + description;
  }
}
```

Если же заполнить таблицу `SINGER` по сценарию, то такой сценарий будет выглядеть следующим образом:

```
insert into singer first_name, last_name,
       birth_date values 'John', 'Mayer', '1977-10-16';
```

```
insert into singer first_name, last_name,
       birth_date values 'Eric', 'Clapton', '1945-03-30';
insert into singer first_name, last_name,
       birth_date values 'John', 'Butler', '1975-04-01';
insert into singer first_name, last_name,
       birth_date values 'B.B.', 'King', '1925-09-16';
insert into singer first_name, last_name,
       birth_date values 'Jimi', 'Hendrix', '1942-11-27';
insert into singer first_name, last_name,
       birth_date values 'Jimmy', 'Page', '1944-01-09';
insert into singer first_name, last_name,
       birth_date values 'Eddie', 'Van Halen',
       '1955-01-26';
insert into singer first_name, last_name,
       birth_date values 'Saul Slash', 'Hudson',
       '1965-07-23';
insert into singer first_name, last_name,
       birth_date values 'Stevie', 'Ray Vaughan',
       '1954-10-03';
insert into singer first_name, last_name,
       birth_date values 'David', 'Gilmour', '1946-03-06';
insert into singer first_name, last_name,
       birth_date values 'Kirk', 'Hammett', '1992-11-18';
insert into singer first_name, last_name,
       birth_date values 'Angus', 'Young', '1955-03-31';
insert into singer first_name, last_name,
       birth_date values 'Dimebag', 'Darrell',
       '1966-08-20';
insert into singer first_name, last_name,
       birth_date values 'Carlos', 'Santana', '1947-07-20';
```

Но, как следует из официально представленных примеров исходного кода, все данные, подобные приведенным выше, вводятся в таблицу с помощью класса `DB Initializer`. А в данном случае потребуются дополнительные тестовые данные, чтобы в дальнейшем можно было продемонстрировать поддержку разбиения на страницы.

Реализация уровня объектов доступа к базе данных

Класс сущности (информационного хранилища) и конфигурация базы данных образуют слой приложения `dao`, отвечающий за объекты доступа к базе данных. Этот класс был представлен выше, и в нем можно обнаружить типичные аннотации JPA, хотя имеются и две новые.

- Для представления клиентской части веб-приложения в последующих примерах введено новое свойство `birthDateString` с помощью аннотации `@Transient`, применяемой к методу получения.

- Для свойства `photo` в качестве типа данных, соответствующего типу данных BLOB в РСУБД, выбран байтовый массив. А соответствующий метод получения снабжен аннотациями `@Lob` и `@Basic(fetch=FetchType.LAZY)`. Первая аннотация указывает поставщику услуг сохраняемости JPA на столбец, содержащий крупный объект, тогда как вторая аннотация — на необходимость извлечь значение из данного свойства по требованию, чтобы исключить отрицательное влияние на производительность при загрузке класса, в котором не требуются сведения о фотографии певца.

Имеются также аннотации проверки достоверности, которые будут пояснены далее. В данном случае будет использована поддержка информационных хранилищ в проекте Spring Data JPA, и для этой цели придется реализовать приведенный ниже интерфейс `SingerRepository`.

```
package com.apress.prospring5.ch16.repo;

import org.springframework.data.repository
          .PagingAndSortingRepository;

public interface SingerRepository
       extends PagingAndSortingRepository<Singer, Long> {
}
```

В данном случае применяется интерфейс `PagingAndSortingRepository` как более совершенное расширение интерфейса `CrudRepository`, где предоставляются методы для извлечения сущностей путем разбиения на страницы и абстракции сортировки. Это очень удобно, поскольку по запросам возвращаются уже отсортированные данные, которые остается лишь отобразить в пользовательском интерфейсе без дополнительных изменений.

Реализация уровня обслуживания

Сначала в этом разделе будет рассмотрена реализация интерфейса `SingerService` с помощью прикладного интерфейса JPA 2, проекта Spring Data JPA и библиотеки Hibernate в качестве поставщика услуг сохраняемости, а затем конфигурация уровня обслуживания в проекте Spring. В следующем фрагменте кода демонстрируется определение интерфейса `SingerService` с методами, предоставляющими доступные услуги:

```
package com.apress.prospring5.ch16.services;

import java.util.List;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;

public interface SingerService {
  List<Singer> findAll();
```

```
  Singer findById(Long id);
  Singer save(Singer singer);
  Page<Singer> findAllByPage(Pageable pageable);
}
```

Методы этого интерфейса самоочевидны, и его совсем не трудно реализовать. В силу простоты рассматриваемого здесь примера веб-приложения никаких изменений в данных не требуется, поэтому основное назначение приведенного ниже класса `SingerServiceImpl` — направить все вызовы аналогичным методам обслуживания информационного хранилища.

```
package com.apress.prospring5.ch16.services;

import java.util.List;
import com.apress.prospring5.ch16.repos.SingerRepository;
import com.apress.prospring5.ch16.entitites.Singer;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation
          .Transactional;
import com.google.common.collect.Lists;

@Transactional
@Service("singerService")
public class SingerServiceImpl implements SingerService {
  private SingerRepository singerRepository;

  @Override
  @Transactional(readOnly=true)
  public List<Singer> findAll() {
    return Lists.newArrayList(singerRepository.findAll());
  }

  @Override
  @Transactional(readOnly=true)
  public Singer findById(Long id) {
    return singerRepository.findById(id).get();
  }

  @Override
  public Singer save(Singer singer) {
    return singerRepository.save(singer);
  }

  @Autowired
  public void setSingerRepository(
```

```
            SingerRepository singerRepository) {
    this.singerRepository = singerRepository;
  }

  @Override
  @Transactional(readOnly=true)
  public Page<Singer> findAllByPage(Pageable pageable) {
    return singerRepository.findAll(pageable);
  }
}
```

На этом реализация, по существу, завершается, а следующий шаг состоит в конфигурировании уровня обслуживания в контексте типа `ApplicationContext` текущего веб-проекта, разрабатываемого в Spring. Именно об этом и пойдет речь в следующем разделе.

Конфигурирование уровня обслуживания

Очевидно, что сконфигурировать уровень обслуживания можно двумя способами: в формате XML и на языке Java. Вариант конфигурирования рассматриваемого здесь веб-проекта можно найти в исходном коде примеров из этой главы, сопровождающем данную книгу. Если вас заинтересует именно этот вариант конфигурирования, можете проанализировать его самостоятельно. Но в этом разделе рассматривается вариант конфигурирования с помощью классов Java. В частности, сконфигурировать уровень обслуживания, доступ к базе данных и обработку транзакций можно в приведенном ниже классе, который должен быть вам уже знаком из главы 9. Итак, уровень обслуживания реализован и готов для доступа и применения удаленными клиентами.

```
package com.apress.prospring5.ch16.config;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseType;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;

import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;
```

```
import java.util.Properties;

@Configuration
@EnableJpaRepositories(basePackages =
                       {"com.apress.prospring5.ch16.repos"})
@ComponentScan(basePackages =
               {"com.apress.prospring5.ch16"} )
public class DataServiceConfig {

  private static Logger logger =
          LoggerFactory.getLogger(DataServiceConfig.class);

  @Bean
  public DataSource dataSource() {
    try {
      EmbeddedDatabaseBuilder dbBuilder =
              new EmbeddedDatabaseBuilder();
      return dbBuilder.setType(EmbeddedDatabaseType.H2)
                      .build();
    } catch (Exception e) {
      logger.error("Embedded DataSource bean cannot "
                + "be created!", e);
      return null;
    }
  }

  @Bean
  public Properties hibernateProperties() {
    Properties hibernateProp = new Properties();
    hibernateProp.put("hibernate.dialect",
                      "org.hibernate.dialect.H2Dialect");
    hibernateProp.put("hibernate.hbm2ddl.auto", "create-drop");
    hibernateProp.put("hibernate.show_sql", true);
    hibernateProp.put("hibernate.max_fetch_depth", 3);
    hibernateProp.put("hibernate.jdbc.batch_size", 10);
    hibernateProp.put("hibernate.jdbc.fetch_size", 50);
    return hibernateProp;
  }

  @Bean
  public PlatformTransactionManager transactionManager() {
    return new JpaTransactionManager(entityManagerFactory());
  }

  @Bean
  public JpaVendorAdapter jpaVendorAdapter() {
    return new HibernateJpaVendorAdapter();
  }
```

```
  @Bean
  public EntityManagerFactory entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean factoryBean =
         new LocalContainerEntityManagerFactoryBean();
    factoryBean.setPackagesToScan(
                "com.apress.prospring5.ch16.entities");
    factoryBean.setDataSource(dataSource());
    factoryBean.setJpaVendorAdapter(
                new HibernateJpaVendorAdapter());
    factoryBean.setJpaProperties(hibernateProperties());
    factoryBean.setJpaVendorAdapter(jpaVendorAdapter());
    factoryBean.afterPropertiesSet();
    return factoryBean.getNativeEntityManagerFactory();
  }
}
```

Введение в проектный шаблон MVC и модуль Spring MVC

Прежде чем перейти к реализации уровня представления в рассматриваемом здесь веб-приложении, рассмотрим ряд основных понятий проектного шаблона MVC для разработки веб-приложений и выясним, каким образом модуль Spring MVC обеспечивает всестороннюю поддержку в данной области.

Эти основные понятия будут поочередно описаны в последующих разделах. Во-первых, в этих разделах будет сделано краткое введение в проектный шаблон MVC. Во-вторых, в них будет дано общее представление о модуле Spring MVC и его иерархии контекстов типа `WebApplicationContext`. И, в-третьих, здесь будет описан жизненный цикл обработки запросов в Spring MVC.

Введение в проектный шаблон MVC

Проектный шаблон MVC обычно служит для реализации уровня представления в приложении. Главный принцип проектного шаблона MVC состоит в определении архитектуры с четкими обязанностями каждого компонента. Как следует из наименования проектного шаблона MVC, он состоит из следующих трех частей.

- **Модель.** Представляет данные из предметной области, а также состояние приложения в контексте отдельного пользователя. Например, на веб-сайте электронной коммерции модель будет включать в себя сведения о профиле пользователя, информацию из корзины для покупок и данные заказа, если пользователи приобретают товары на этом сайте.
- **Представление.** Отображает данные в удобном для пользователей формате, поддерживает взаимодействие с пользователями и выполняет проверку досто-

верности данных на стороне клиента, интернационализацию, стилевое оформление и т.д.

- **Контроллер.** Обрабатывает запросы на основании действий, выполняемых пользователями на стороне клиента, взаимодействуя с уровнем обслуживания, обновляя модель и направляя пользователей к соответствующему представлению в зависимости от результатов выполнения.

В связи с появлением веб-приложений, основанных на технологии Ajax, проектный шаблон MVC был расширен с целью сделать пользовательский интерфейс более удобным и оперативно реагирующим на действия пользователей. Так, если применяется сценарий JavaScript, представление может принимать события или действия, выполняемые пользователем, а затем отправлять серверу соответствующие запросы типа `XMLHttpRequest`. Вместо целого представления на стороне контроллера возвращаются необработанные данные (например, в формате XML или JSON), после чего сценарий JavaScript обновляет лишь нужную часть представления, используя полученные данные. На рис. 16.1 приведен распространенный шаблон веб-приложения, который можно трактовать как расширение традиционного проектного шаблона MVC. Обычный запрос на представление обрабатывается в следующем порядке.

1. **Запрос.** Направляется серверу. На стороне сервера в большинстве каркасов (например, Spring MVC или Struts) для обработки запроса предусмотрен диспетчер (в форме сервлета).
2. **Вызовы.** Диспетчер направляет запрос соответствующему контроллеру на основании информации из HTTP-запроса и конфигурации веб-приложения.
3. **Вызов службы.** Контроллер взаимодействует с уровнем обслуживания.
4. **Заполнение модели.** Сведения, полученные из уровня обслуживания, используются контроллером для заполнения модели.

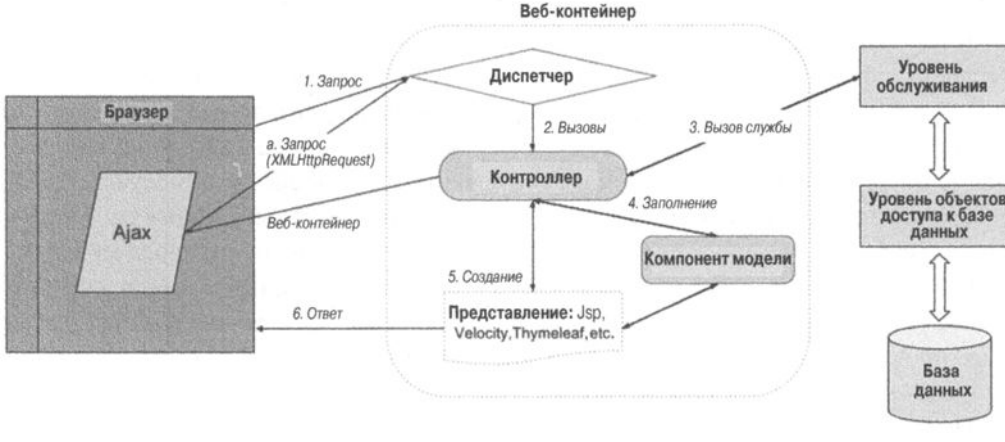


Рис. 16.1. Реализация проектного шаблона MVC в типичном веб-приложении

5. **Создание представления.** Конкретное представление создается на основании модели.
6. **Ответ.** Контроллер возвращает пользователю соответствующее представление.

Кроме того, в представлении происходят Ajax-вызовы. Допустим, пользователь просматривает данные в сеточной компоновке. Когда он щелкает на ссылке для перехода к следующей веб-странице, вместо обновления всей страницы будут инициированы следующие действия.

а) **Запрос.** Подготавливается запрос типа `XMLHttpRequest`, который далее отправляется серверу. Диспетчер направит запрос соответствующему контроллеру.

б) **Ответ.** Контроллер взаимодействует с уровнем обслуживания, чтобы отформатировать данные и отправить их браузеру. При этом ни одно из представлений не задействовано. Браузер получает данные и частично обновляет существующее представление.

Введение в Spring MVC

В каркасе Spring Framework модуль Spring MVC предоставляет всестороннюю поддержку проектного шаблона MVC, а также других функциональных средств, включая тематическое оформление, интернационализацию, проверку достоверности данных, преобразование типов данных и форматирование, которые упрощают реализацию уровня представления. В последующих разделах описываются основные понятия модуля Spring MVC, в том числе иерархия контекстов типа `WebApplicationContext` в модуле Spring MVC, типичный жизненный цикл обработки запросов и конфигурирование.

Иерархия контекстов типа `WebApplicationContext` в Spring MVC

Класс `DispatcherServlet` выполняет в модуле Spring MVC роль центрального сервлета, получающего запросы и направляющего их соответствующим контроллерам. В приложении Spring MVC может присутствовать произвольное количество экземпляров класса `DispatcherServlet`, предназначенных для разных целей (например, для обработки запросов пользовательского интерфейса и веб-служб REST). У каждого экземпляра класса `DispatcherServlet` имеется собственная конфигурация интерфейса `WebApplicationContext`, определяющая такие характеристики уровня сервлета, как контроллеры, поддерживающие сервлет, отображение обработчиков, распознавание представлений, интернационализация, тематическое оформление, проверка достоверности данных, преобразование типов и форматирование данных.

Помимо конфигураций интерфейса `WebApplicationContext` на уровне сервлетов, ниже по иерархии в модуле Spring MVC поддерживается конфигурация корнево-

го контекста типа `WebApplicationContext`, которая включает в себя конфигурацию уровня приложения, в том числе для источников данных на стороне сервера, безопасности, уровней обслуживания и сохраняемости. Конфигурация корневого контекста типа `WebApplicationContext` будет доступна всем его конфигурациям на уровне сервлетов.

Обратимся к конкретному примеру. Допустим, в приложении имеются два экземпляра сервлетов типа `DispatcherServlet`. Один сервлет предназначен для поддержки пользовательского интерфейса (он называется *сервлетом приложения*), а другой — для предоставления услуг другим приложениям в форме веб-служб REST (он называется *сервлетом REST*). Определим в модуле Spring MVC как конфигурацию корневого контекста типа `WebApplicationContext`, так и его конфигурации на уровне двух сервлетов типа `DispatcherServlet`. На рис. 16.2 приведена иерархия контекстов типа `WebApplicationContext`, которая будет поддерживаться в модуле Spring MVC для данного примера.

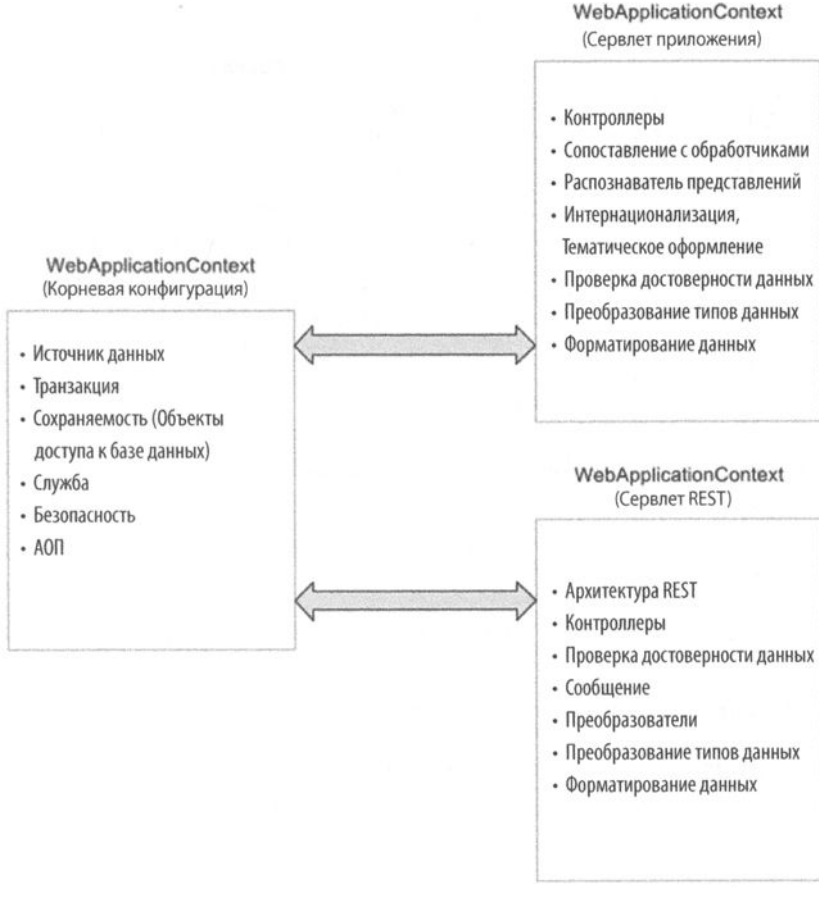


Рис. 16.2. Иерархия контекстов типа **`WebApplicationContext`** в модуле Spring MVC

Жизненный цикл обработки запросов в Spring MVC

Выясним, каким образом запросы обрабатываются в Spring MVC. На рис. 16.3 показаны основные компоненты Spring MVC, задействуемые в обработке запросов, а их краткое описание приведено ниже.

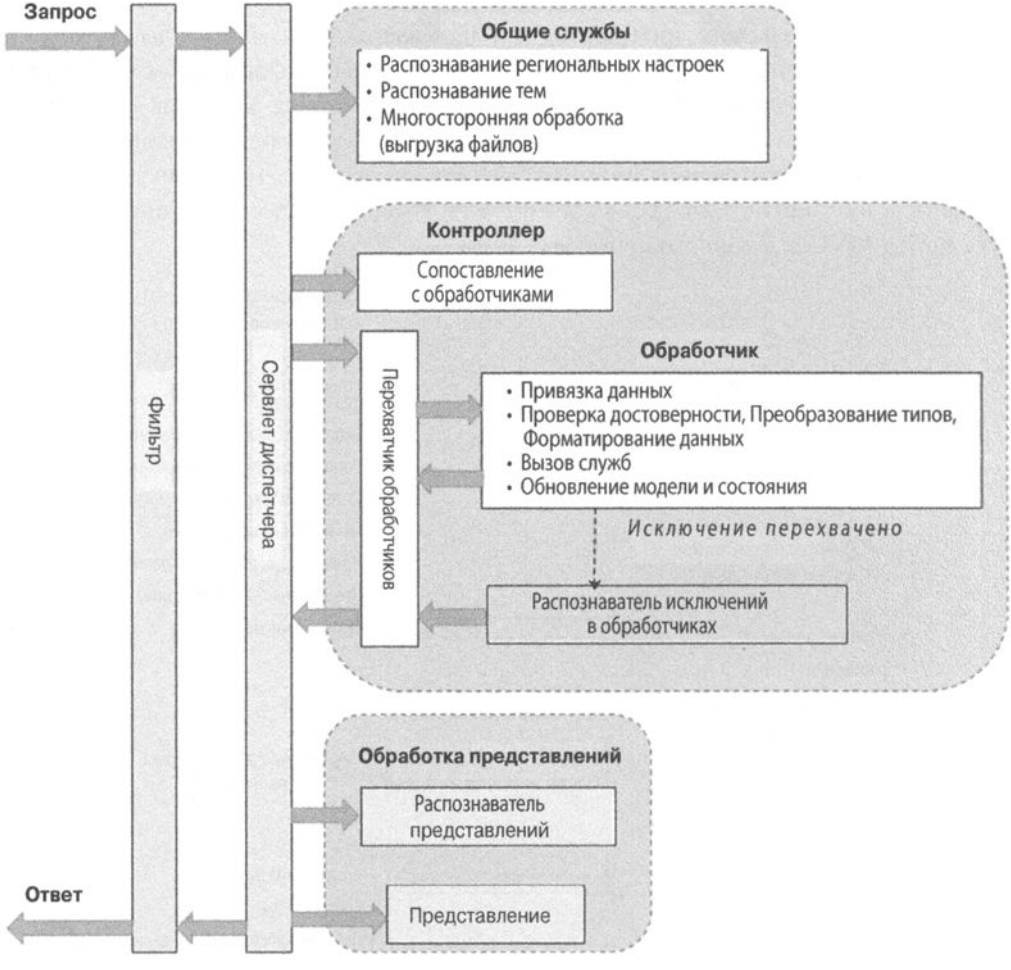


Рис. 16.3. Жизненный цикл обработки запросов в Spring MVC

- **Фильтр.** Применяется к каждому запросу. Наиболее употребительные фильтры и их назначение описаны в следующем разделе.

- **Сервлет диспетчера.** Анализирует запросы и направляет их на обработку соответствующему контроллеру[1].
- **Общие службы.** Применяются по каждому запросу для поддержки интернационализации, тематического оформления и загрузки файлов. Их конфигурация определяется в контексте типа `WebApplicationContext` сервлета диспетчера типа `DispatcherServlet`.
- **Сопоставление с обработчиками.** Сопоставляет запросы с обработчиками (методами из класса контроллера в модуле Spring MVC). Начиная с версии Spring 2.5, конфигурация для сопоставления с обработчиками обычно не требуется, поскольку модуль Spring MVC автоматически регистрирует готовую реализацию интерфейса `HandlerMapping`, сопоставляющую запросы с обработчиками по путям формата HTTP, выражаемым через аннотацию `@Request Mapping` на уровне типа данных или метода в классах контроллеров[2].
- **Перехватчик обработчиков.** В модуле Spring MVC можно зарегистрировать перехватчики обработчиков, чтобы реализовать общую проверку или логику. Например, перехватчик обработчиков может проверять, вызываются ли обработчики только в рабочее время.
- **Распознаватель исключений в обработчиках.** В модуле Spring MVC интерфейс `HandlerExceptionResolver` (из пакета `org.springframework.web.servlet`) предназначен для распознавания непредвиденных исключений, возникающих во время обработки запросов. По умолчанию сервлет типа `DispatcherServlet` регистрирует класс `DefaultHandlerExceptionResolver` (из пакета `org.springframework.web.servlet.mvc.support`), реализующий данный интерфейс. Этот стандартный распознаватель исключений в обработчиках обрабатывает некоторые стандартные исключения в Spring MVC, устанавливая специальный код состояния в ответе на запрос. Можно также реализовать собственный обработчик исключений, снабдив метод контроллера аннотацией `@ExceptionHandler` и указав в ее атрибуте тип исключения.
- **Распознаватель представлений.** Интерфейс `ViewResolver` (из пакета `org.springframework.web.servlet`) поддерживает в модуле Spring MVC распознавание представлений на основе логического имени, возвращаемого контроллером. Для поддержки различных механизмов распознавания представлений предусмотрено немало классов реализации данного интерфейса. Например,

---

[1] Если вы знакомы с проектными шаблонами, то сразу распознаете в классе `DispatcherServlet`, представляющем сервлет диспетчера, реализацию проектного шаблона "Единая точка входа" (Front Controller).

[2] В версии Spring 2.5 стандартной реализацией интерфейса `HandlerMapping` служил класс `DefaultAnnotationHandlerMapping`. А начиная с версии Spring 3.1, его стандартной реализацией стал класс `RequestMappingHandlerMapping`, где поддерживается также сопоставление запросов с обработчиками, определенными без аннотаций, но при условии, что соблюдаются принятые в Spring соглашения об именовании имен контроллеров и методов.

в классе `UrlBasedViewResolver` поддерживается прямое преобразование логических имен в URL. А класс `ContentNegotiatingViewResolver` динамически распознает представления в зависимости от типа среды передачи данных, поддерживаемого клиентом (XML, PDF и JSON). Существует также несколько реализаций для интеграции с различными технологиями представлений, в том числе FreeMarker (класс `FreeMarkerViewResolver`), Velocity (класс `VelocityViewResolver`) и JasperReports (класс `JasperReportsView Resolver`).

Приведенное выше описание касается лишь нескольких распространенных обработчиков и распознавателей. За полным их описанием обращайтесь к справочной документации на каркас Spring Framework и соответствующим документирующим комментариям в формате Javadoc.

Конфигурирование модуля Spring MVC

Чтобы сделать доступным модуль Spring MVC в веб-приложении, требуется начальная конфигурация, особенно для дескриптора веб-развертывания из файла `web.xml`. В версии Spring 3.1 стала доступной поддержка конфигурации на основе кода в веб-контейнере Servlet 3.0. Тем самым предоставляется альтернатива конфигурации в формате XML, требующейся в файле `web.xml` дескриптора веб-развертывания.

Чтобы настроить поддержку Spring MVC в веб-приложениях, нужно сконфигурировать следующие компоненты в дескрипторе веб-развертывания.

- Корневой контекст типа `WebApplicationContext`.
- Фильтры сервлетов, требующихся в Spring MVC.
- Сервлеты диспетчеров в веб-приложении.

Все эти виды конфигурирования выполняются лишь в нескольких строках кода из следующего конфигурационного класса.

```
package com.apress.prospring5.ch16.init;

import com.apress.prospring5.ch16.config.DataServiceConfig;
import com.apress.prospring5.ch16.config.SecurityConfig;
import com.apress.prospring5.ch16.config.WebConfig;
import org.springframework.web.filter
          .CharacterEncodingFilter;
import org.springframework.web.filter
          .HiddenHttpMethodFilter;
import org.springframework.web.servlet.support.
     AbstractAnnotationConfigDispatcherServletInitializer;
import javax.servlet.Filter;

public class WebInitializer extends
     AbstractAnnotationConfigDispatcherServletInitializer {
```

```
  @Override
  protected Class<?>[] getRootConfigClasses() {
    return new Class<?>[] {
      SecurityConfig.class, DataServiceConfig.class
    };
  }

  @Override
  protected Class<?>[] getServletConfigClasses() {
    return new Class<?>[] {
      WebConfig.class
    };
  }

  @Override
  protected String[] getServletMappings() {
    return new String[]{"/"};
  }

  @Override
  protected Filter[] getServletFilters() {
    CharacterEncodingFilter cef =
              new CharacterEncodingFilter();
    cef.setEncoding("UTF-8");
    cef.setForceEncoding(true);
    return new Filter[]{new HiddenHttpMethodFilter(), cef};
  }
}
```

Чтобы воспользоваться конфигурацией на основе кода, необходимо разработать класс, реализующий интерфейс `org.springframework.web.WebApplicationInitializer`. Ради простоты в приведенном выше примере был расширен класс `AbstractAnnotationConfigDispatcherServletInitializer`, реализующий в Spring интерфейс `WebApplicationInitializer`, поскольку он содержит конкретные реализации методов для конфигурирования веб-приложений в Spring, в которых применяется конфигурация, выполненная на языке Java.

Все классы, реализующие интерфейс `WebApplicationInitializer`, будут автоматически обнаружены классом `org.springframework.web.SpringServletContainerInitializer`, реализующим интерфейс `javax.servlet.ServletContainerInitializer` и автоматически запускающим любые контейнеры в версии Servlet 3.0. Для подключения специально настраиваемых конфигураций в приведенном выше примере переопределены следующие методы.

- Метод `getRootConfigClasses()`. Корневой контекст приложения типа `AnnotationConfigWebApplicationContext` будет создан с помощью конфигурационных классов, возвращаемых данным методом.

- Метод `getServletConfigClasses()`. Контекст веб-приложения типа `AnnotationConfigWebApplicationContext` будет создан с помощью конфигурационных классов, возвращаемых данным методом.
- Метод `getServletMappings()`. Сопоставления (контекста) с помощью сервлета диспетчера типа `DispatcherServelt` указываются в массиве символьных строк, возвращаемом этим методом.
- Метод `getServletFilters()`. Как подразумевает имя этого метода, он возвращает массив реализаций интерфейса `javax.servlet.Filter`, предназначенных для применения к каждому запросу.

Но ведь если проанализировать приведенный выше пример, то в нем вообще не упоминается о фильтрах безопасности! О фильтрах упоминалось в главе 12, но на тот случай, если вы пропустили материал этой главы, достаточно сказать, что для подобной цели в Spring имеется специальный класс:

```
package com.apress.prospring5.ch16.init;

import org.springframework.security.web.context.
       AbstractSecurityWebApplicationInitializer;
public class SecurityWebApplicationInitializer
  extends AbstractSecurityWebApplicationInitializer {
}
```

Предоставлением пустого класса, расширяющего класс `AbstractSecurityWebApplicationInitializer`, каркасу Spring, по существу, предписывается активизировать заместитель типа `DelegatingFilterProxy`, чтобы воспользоваться компонентом `springSecurityFilterChain`, прежде чем будет зарегистрирован любой другой фильтр, реализующий интерфейс `javax.servlet.Filter`.

Применяя такой подход вместе с конфигурированием на языке Java, можно реализовать конфигурацию веб-приложения в Spring исключительно в коде Java, вообще не прибегая к объявлению конфигурации в файле `web.xml` или других XML-файлах конфигурации в Spring. И сделать это совсем не трудно.

Если вернуться снова к фильтрам, то в табл. 16.1 перечислены те фильтры, которые возвращаются в массиве из метода `getServletFilters()`.

Таблица 16.1. Наиболее употребительные фильтры сервлетов в Spring MVC

Полное имя класса фильтра	Описание
`org.springframework.web.filter.CharacterEncodingFilter`	Служит для обозначения кодировки символов в запросе
`org.springframework.web.filter.HiddenHttpMethodFilter`	Поддерживает другие методы доступа по сетевому протоколу HTTP, кроме **GET** и **POST** (например, метода **PUT**)

На заметку Следует также упомянуть о реализации фильтра `org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter`, хотя он и не применяется в рассматриваемой здесь конфигурации. Эта реализация привязывает интерфейс `EntityManager` из прикладного интерфейса JPA к потоку исполнения, чтобы полностью обработать запрос. Она предназначена для реализации шаблона "Открытый диспетчер сущностей в представлении" (Open EntityManager in View), допуская загрузку данных по требованию в веб-представления, несмотря на то, что первоначальные транзакции уже завершены. Но, несмотря на всю практичность такого подхода, он довольно опасный, поскольку многие запросы способны в конечном счете исчерпать все ресурсы подключений к базе данных. Именно поэтому разработчики предпочитают не пользоваться данным фильтром, а вместо этого поручить отдельным обработчикам, вызываемым по Ajax-запросам, загрузить данные в отдельные объекты веб-представлений, а не в сущности.

Создание первого представления в Spring MVC

Настроив уровень обслуживания и подготовив конфигурацию веб-приложения в Spring MVC, можно приступить к реализации первого представления. В этом разделе будет продемонстрирована реализация простого представления для отображения всех певцов, сведения о которых были изначально занесены в базу данных компонентом Spring Bean типа `DBInitializer`.

Для реализации представления здесь будет использован формат JSPX, правильно построенный по нормам JSP форматом XML. Ниже перечислены основные преимущества формата JSPX по сравнению с JSP.

- Формат JPSX требует более строгого отделения кода от уровня представления. Например, в JSPX-документ нельзя внедрить скриптлеты Java.
- Доступны инструментальные средства для моментальной проверки достоверности (по синтаксису XML), поэтому ошибки могут быть обнаружены на самой ранней стадии.

Для рассматриваемых здесь целей в текущий проект придется внедрить зависимости, перечисленные в следующем фрагменте кода конфигурации:

```
// Файл конфигурации pro-spring-15\build.gradle
ext {
  // Библиотеки Spring
  springVersion = '5.0.0.RC2'
  springSecurityVersion = '5.0.0.M2'
  h2Version = '1.4.194'
  tilesVersion = '3.0.7'

  // Библиотеки сохраняемости
  hibernateVersion = '5.2.10.Final'
  hibernateJpaVersion = '1.0.0.Final'
  hibernateValidatorVersion = '5.4.1.Final'
  ...
```

```
  spring = [
    webmvc : "org.springframework:spring-webmvc:
             $springVersion",
    data : "org.springframework.data:spring-data-jpa:
           $springDataVersion",
    securityWeb : "org.springframework.security:
               spring-security-web:$springSecurityVersion",
    securityConfig : "org.springframework.security:
            spring-security-config:$springSecurityVersion",
    securityTaglibs: "org.springframework.security:
          spring-security-taglibs: $springSecurityVersion",
    ...
  ]

  hibernate = [
    validator : "org.hibernate:hibernate-validator:
                $hibernateValidatorVersion",
    em : "org.hibernate:hibernate-entitymanager:
         $hibernateVersion",
    jpaApi : "org.hibernate.javax.persistence:
             hibernate-jpa-2.1-api:$hibernateJpaVersion",
     ...
  ]

  misc = [
    validation : "javax.validation:validation-api:
                 $javaxValidationVersion",
    castor : "org.codehaus.castor:castor-xml:
             $castorVersion",
    io : "commons-io:commons-io:2.5",
    tiles : "org.apache.tiles:tiles-jsp:$tilesVersion",
    jstl : "jstl:jstl:1.2",
    ...
  ]

  db = [
    h2 : "com.h2database:h2:$h2Version"
  ]
}
...
// Файл конфигурации chapter-16\build.gradle
dependencies {
  // Здесь исключены общие транзитивные зависимости
  compile spring.contextSupport {
    exclude module: 'spring-context'
    exclude module: 'spring-beans'
    exclude module: 'spring-core'
  }
  compile spring.securityTaglibs {
```

```
    exclude module: 'spring-web'
    exclude module: 'spring-context'
    exclude module: 'spring-beans'
    exclude module: 'spring-core'
  }
  compile spring.securityConfig {
    exclude module: 'spring-security-core'
    exclude module: 'spring-context'
    exclude module: 'spring-beans'
    exclude module: 'spring-core'
  }
  Compile misc.slf4jJcl, misc.logback, misc.lang3,
          hibernate.em, hibernate.validator, misc.guava,
          db.h2, spring.data, spring.webmvc, misc.castor,
          misc.validation, misc.tiles,
          misc.jacksonDatabind, misc.servlet, misc.io,
          misc.jstl, spring.securityTaglibs
}
```

Большинство библиотек, перечисленных в приведенном выше фрагменте кода конфигурации, должно быть вам уже известно. А ниже вкратце поясняется назначение библиотек, впервые упоминаемых в данной книге.

- `spring-webmvc`. Служит для поддержки проектного шаблона MVC в модуле Spring MVC.
- `spring-security-web`. Служит в качестве веб-модуля Spring для поддержки проекта Spring Security. Непосредственная зависимость от библиотеки `spring-security-web` содержит определения дескрипторов разметки безопасности, употребляемых на JSP-страницах, а также класс `AbstractSecurityWebApplicationInitializer` и другие компоненты Spring для обеспечения безопасности веб-приложений.
- `spring-security-config`. Служит в качестве модуля Spring Security, содержащего классы, предназначенные для конфигурирования безопасности в приложении Spring.
- `tiles-jsp`. Служит в качестве модуля Apache Tiles, содержащего классы Java и определения дескрипторов разметки, предназначенных для создания на Java шаблонов веб-приложений[3].

Конфигурирование сервлета диспетчера

Следующий шаг — конфигурирование сервлета диспетчера типа `DispatcherServlet`. С этой целью необходимо создать конфигурационный класс, в котором определяются все компоненты инфраструктуры Spring Beans для разработки веб-

[3] Подробнее об этой библиотеке можно узнать по адресу `https://tiles.apache.org/`.

приложения в Spring. Ниже приведен исходный код класса Java, содержащего минимальную информацию, требующуюся для конфигурации.

```
package com.apress.prospring5.ch16.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.*;
import org.springframework.web.servlet.view.InternalResourceViewResolver;
...

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = {"com.apress.prospring5.ch16"})
public class WebConfig implements WebMvcConfigurer {

  // объявить статические ресурсы
  @Override
  public void addResourceHandlers(
         ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/resources/**")
           .addResourceLocations("/")
           .setCachePeriod(31556926);
  }

  @Bean
  InternalResourceViewResolver viewResolver() {
    InternalResourceViewResolver resolver =
            new InternalResourceViewResolver();
    resolver.setPrefix("/WEB-INF/views");
    resolver.setSuffix(".jspx" );
    resolver.setRequestContextAttribute("requestContext");
    return resolver;
  }

  // <=> <mvc:view-controller .../>
  @Override
  public void addViewControllers(ViewControllerRegistry registry) {
    registry.addViewController("/").setViewName("singers/list");
  }

  // <=> <mvc:default-servlet-handler/>
  @Override
  public void configureDefaultServletHandling(
              DefaultServletHandlerConfigurer configurer) {
    configurer.enable();
  }
}
```

В интерфейсе `WebMvcConfigurer` определяются методы обратного вызова для специальной настройки конфигурации, выполняемой в коде Java и активизируемой для модуля Spring MVC с помощью аннотации `@EnableWebMvc`. И хотя в приложении Spring может быть не один конфигурационный класс Java, лишь один из них допускается снабдить аннотацией `@EnableWebMvc`. Так, в приведенной выше конфигурации переопределяется несколько методов для специальной настройки конфигурации, как поясняется ниже.

- Метод `addResourceHandlers()`. Вводит обработчики, предназначенные для обслуживания таких статических ресурсов, как изображения, сценарии JavaScript и файлы стилевых таблиц CSS, извлекаемых из отдельных мест и находящихся по иерархии ниже корневого узла веб-приложения, по пути к классам и пр. В такой специализированной реализации любой запрос по URL, содержащему ресурсы, будет интерпретироваться специальным обработчиком в обход всех фильтров.
- Метод `configureDefaultServletHandling()`. Активизирует обработчик статических ресурсов.
- Метод `addViewControllers()`. Определяет простые автоматизированные контроллеры, предварительно сконфигурированные с ответным кодом состояния и/или представлением для воспроизведения тела ответа. Такие представления не содержат логику контроллера и служат для воспроизведения приветственной страницы, выполнения простых видов переадресации веб-сайтов по URL, возврата кода состояния `404` и прочих действий. В приведенной выше конфигурации этот метод применяется для переадресации к представлению по пути `singers/list`.
- Метод `viewResolver()`. Объявляет распознаватель представлений типа `InternalResourceViewResolver`, выполняющий сопоставление символических имен представлений с шаблонами `*.jspx` по пути `/WEB-INF/views`.

Реализация класса `SingerController`

Следующий шаг после конфигурирования контекста типа `WebApplicationContext` для сервлета диспетчера типа `DispatcherServlet` состоит в реализации класса контроллера. Ниже приведен исходный код класса `SingerController`.

```
package com.apress.prospring5.ch16.web;
...

@RequestMapping("/singers")
@Controller
public class SingerController {
  private final Logger logger =
          LoggerFactory.getLogger(SingerController.class);
  private SingerService singerService;
  private MessageSource messageSource;
```

```
  @RequestMapping(method = RequestMethod.GET)
  public String list(Model uiModel) {
    logger.info("Listing singers");
    List<Singer> singers = singerService.findAll();
    uiModel.addAttribute("singers", singers);
    logger.info("No. of singers: " + singers.size());
    return "singers/list";
  }

  @Autowired
  public void setSingerService(
              SingerService singerService) {
    this.singerService = singerService;
  }
}
```

В классе `SingerController` применяется аннотация `@Controller`, где указывается, что это контроллер Spring MVC. Аннотация `@RequestMapping` на уровне класса задает корневой URL, который будет обрабатываться контроллером. В данном случае все URL с префиксом `/singers` будут направляться этому контроллеру. В методе `list()` также применяется аннотация `@RequestMapping`, но на этот раз данный метод сопоставляется с методом доступа `GET` по сетевому протоколу HTTP. Это означает, что с помощью метода `list()` будет обработан URL с префиксом `/singers` и методом доступа `GET` по протоколу HTTP. В теле метода `list()` извлекается список певцов, который сохраняется в интерфейсе `Model`, экземпляр которого передается из Spring MVC. И, наконец, из данного метода возвращается логическое имя представления `singers/list`. В конфигурации сервлета диспетчера типа `DispatcherServlet` указан распознаватель представлений типа `InternalResourceViewResolver`, а файл имеет префикс `/WEB-INF/views/` и суффикс `.jspx`. В итоге модуль Spring MVC выберет для представления файл `/WEB-INF/views/singers/list.jspx`.

Реализация представления списка певцов

На следующем шаге необходимо реализовать страницу представления для отображения сведений о певцах в конкретном файле `/src/main/webapp/WEB-INF/views/singers/list.jspx`.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
     xmlns:c="http://java.sun.com/jsp/jstl/core"
     xmlns:spring="http://www.springframework.org/tags"
     xmlns:fmt=
          "http://java.sun.com/jsp/jstl/fmt" version="2.0">

    <h1>Singer Listing</h1>
```

```
    <c:if test="${not empty singers}">
      <table>
        <thead>
      <tr>
      <th>First Name</th>
      <th>Last Name</th>
      <th>Birth Date</th>
      </tr>
        </thead>
        <tbody>
    <c:forEach items="${singers}" var="singer">
      <tr>
        <td>${singer.firstName}</td>
        <td>${singer.lastName}</td>
        <td><fmt:formatDate
             value="${singer.birthDate}"/></td>
      </tr>
    </c:forEach>
       </tbody>
     </table>
    </c:if>
</div>
```

Если вы занимались раньше разработкой с помощью JSP, то приведенный выше код разметки должен показаться вам знакомым. Но поскольку это JSPX-страница, то ее содержимое встроено в дескриптор разметки `<div>`. Кроме того, применяемые библиотеки дескрипторов разметки объявлены в виде пространств имен XML.

Во-первых, в дескрипторе разметки `<jsp:directive.page>` определяются атрибуты, которые применяются ко всей JSPX-странице, тогда как дескриптор разметки `<jsp:output>` управляет свойствами вывода JSPX-документа. Во-вторых, в дескрипторе разметки `<c:if>` выясняется, пуст ли атрибут модели `singers`. Но поскольку мы уже заполнили базу данных тестовой информацией о певцах, то атрибут `singers` должен содержать данные. В итоге дескриптор разметки `<c:forEach>` воспроизводит на странице сведения о певцах из таблицы. Обратите внимание на то, что в дескрипторе разметки `<joda:format>` форматируется значение атрибута `birthDate`, относящегося к типу `java.utilDate`.

Тестирование представления списка певцов

Теперь все готово для того, чтобы протестировать представление списка певцов. Сначала постройте и разверните проверяемое веб-приложение. Чтобы протестировать представление списка певцов, откройте веб-браузер и перейдите на страницу по следующему URL: `http://localhost:8080/singers`. В итоге вы должны увидеть страницу со списком певцов.

Итак, нам удалось создать первое работающее представление. В последующих разделах мы расширим рассматриваемое здесь веб-приложение дополнительными

представлениями и активизируем поддержку интернационализации, тематического оформления и т.д.

Описание структуры проекта в Spring MVC

Прежде чем углубляться в реализацию различных аспектов веб-приложения, выясним, как выглядит структура проекта для примера веб-приложения, разработка которого представлена в этой главе. Как правило, для поддержки различных функциональных возможностей в веб-приложении требуется большое количество файлов. Например, имеется целый ряд файлов статических ресурсов, включая стилевые таблицы, сценарии JavaScript, изображения и библиотеки компонентов. Кроме того, имеются файлы для представления пользовательского интерфейса на разных языках. И, конечно, имеются страницы представлений, которые будут проанализированы и воспроизведены веб-контейнером, а также файлы компоновки и определений, применяемые шаблонизатором (например, Apache Tiles) для обеспечения согласованного внешнего вида и поведения веб-приложения.

В связи с изложенным выше на практике рекомендуется хранить файлы, которые служат разным целям, в хорошо структурированной иерархии папок. Такой подход дает ясное представление о применении различных ресурсов в веб-приложении и обеспечивает простоту его непрерывного сопровождения.

В табл. 16.2 описана структура и назначение папок для веб-приложения, пример разработки которого демонстрируется в этой главе. Обратите внимание на то, что представленная здесь структура не является обязательной, но она нередко применяется в сообществе разработчиков веб-приложений.

Таблица 16.2. Описание папок в проекте для примера веб-приложения

Имя папки	Назначение
`ckeditor`	CKEditor (`http://ckeditor.com`) — это библиотека компонентов JavaScript, предоставляющая редактор форматированного текста для формы ввода. Эта библиотека будет использована далее для редактирования текста, описывающего певца
`jqgrid`	jqGrid (`www.trirand.com/blog`) — это библиотека компонентов, построенная на основе библиотеки jQuery, предоставляющей различные компоненты с сеточной компоновкой для представления данных. Эта библиотека будет использована далее для реализации сеточной компоновки, предназначенной для отображения сведений о певцах, а также для поддержки разбиения на страницы в стиле Ajax
`scripts`	Папка для хранения всех общих файлов JavaScript. В примерах, представленных далее в этой главе, будут использоваться библиотеки jQuery (`http://jquery.org`) и jQuery UI (`http://jqueryui.com`) языка JavaScript для реализации пользовательского интерфейса с расширенными функциональными возможностями. Сценарии будут размещены в этой папке, где должны находиться и самостоятельно разработанные библиотеки JavaScript

Окончание табл. 16.2

Имя папки	Назначение
styles	Папка для хранения файлов стилевых таблиц и связанных со стилями файлов изображений
WEB-INF/i18n	Папка для хранения файлов для поддержки интернационализации. В файле **application*.properties** хранится текст, связанный с компоновкой (например, заголовки страниц, метки полей и заголовки меню). А файл **message*.properties** содержит различные сообщения (например, сообщения об удачном исходе операций, об ошибках и результатах проверки достоверности данных). В рассматриваемом здесь примере будут поддерживаться американский английский (US) и китайский (HK) языки
WEB-INF/views	Папка для хранения представлений (в данном случае — файлов формата JSPX), которые будут использоваться в рассматриваемом здесь веб-приложении

В последующих разделах для поддержки реализации нам понадобятся различные файлы (например, файлы CSS, файлы JavaScript и файлы изображений). Исходный код стилевых таблиц CSS и сценариев JavaScript здесь приводиться не будет, поэтому рекомендуется загрузить архив с исходным кодом для этой главы и распаковать его во временную папку, чтобы скопировать файлы, требующиеся непосредственно для рассматриваемого здесь проекта.

Интернационализация веб-приложений

При разработке веб-приложений их интернационализацию рекомендуется выполнять на самых ранних стадиях разработки. Основная работа связана с вынесением текста пользовательского интерфейса и сообщений во внешние файлы свойств. Даже если требования к интернационализации в настоящий момент отсутствуют, параметры настройки на конкретный язык все равно рекомендуется вынести за пределы приложения, чтобы впоследствии было проще поддерживать дополнительные языки.

В модуле Spring MVC интернационализация реализуется очень просто. Сначала следует вынести параметры настройки на конкретный язык в отдельные файлы свойств, поместив их в папку `/WEB-INF/i18n`, как пояснялось в табл. 16.2. В рассматриваемом здесь примере веб-приложения будет поддерживаться американский диалект английского (US) и китайский (HK) язык, поэтому для интернационализации данного приложения понадобятся четыре файла свойств. Так, в файлах `application.properties` и `message.properties` хранятся стандартные региональные настройки (в данном случае на американском диалекте английского языка (US)). А файлы `application_zh_HK.properties` и `message_zh_HK.properties` содержат региональные настройки на китайском языке (HK).

Настройка интернационализации в конфигурации сервлета диспетчера

Располагая региональными настройками на разные языки, можно настроить на интернационализацию контекст типа `WebApplicationContext` для сервлета диспетчера. В следующем фрагменте кода приведены компоненты Spring Beans и методы из конфигурационного класса `WebConfig`, которые объявляются для активизации и специальной настройки поддержки интернационализации:

```
package com.apress.prospring5.ch16.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.context.support
          .ReloadableResourceBundleMessageSource;
import org.springframework.web.servlet.config.annotation.*;
import org.springframework.web.servlet.i18n
          .CookieLocaleResolver;
import org.springframework.web.servlet.i18n
          .LocaleChangeInterceptor;
import org.springframework.web.servlet.mvc
          .WebContentInterceptor;
import java.util.Locale;
...

@Configuration
@EnableWebMvc
@ComponentScan(basePackages =
               {"com.apress.prospring5.ch16"})
public class WebConfig implements WebMvcConfigurer {

  // объявить статические ресурсы
  @Override
  public void addResourceHandlers(
              ResourceHandlerRegistry registry) {
   registry.addResourceHandler("/resources/**")
           .addResourceLocations("/")
           .setCachePeriod(31556926);
  }

  // <=> <mvc:default-servlet-handler/>
  @Override
  public void configureDefaultServletHandling(
         DefaultServletHandlerConfigurer configurer) {
    configurer.enable();
  }
```

```
  @Bean
  ReloadableResourceBundleMessageSource messageSource() {
      ReloadableResourceBundleMessageSource messageSource =
      new ReloadableResourceBundleMessageSource();
    messageSource.setBasenames(
      "WEB-INF/i18n/messages",
      "WEB-INF/i18n/application");
    messageSource.setDefaultEncoding("UTF-8");
    messageSource.setFallbackToSystemLocale(false);
    return messageSource;
  }

  @Override
  public void addInterceptors(
              InterceptorRegistry registry) {
    registry.addInterceptor(localeChangeInterceptor());
    ...
  }

  @Bean
  LocaleChangeInterceptor localeChangeInterceptor() {
    return new LocaleChangeInterceptor();
  }

  @Bean
  CookieLocaleResolver localeResolver() {
    CookieLocaleResolver cookieLocaleResolver =
                         new CookieLocaleResolver();
    cookieLocaleResolver.setDefaultLocale(Locale.ENGLISH);
    cookieLocaleResolver.setCookieMaxAge(3600);
    cookieLocaleResolver.setCookieName("locale");
    return cookieLocaleResolver;
  }
  ...
}
```

В приведенном выше коде конфигурации определение ресурсов исправлено с целью отразить новую структуру папок, как описано в табл. 16.2. Так, в методе `addResourceHandlers()` определяется местоположение файлов статических ресурсов, с помощью которых в Spring MVC можно эффективно обрабатывать файлы в этих папках. В дескрипторе разметки атрибут `location` определяет папки для хранения статических ресурсов. Так, местоположение ресурса / обозначает корневую папку для веб-приложения, а именно: `/src/main/webapp`. А по пути `/resources/**` к обработчикам ресурсов определяется URL для сопоставления со статическими ресурсами. В данном примере модуль Spring MVC извлечет файл `standard.css` из папки `src/main/webapp/styles` по следующему URL: `http://localhost:8080/resources/styles/standard.css`.

Метод `configureDefaultServletHandling()` разрешает сопоставление сервлета диспетчера типа `DispatcherServlet` с URL корневого контекста веб-приложения, допуская в то же время обработку запросов статических ресурсов в стандартном сервлете контейнера. Кроме того, в классе `LocaleChangeInterceptor` из модуля Spring MVC определяется перехватчик всех запросов к сервлету диспетчера типа `DispatcherServlet`. Этот перехватчик поддерживает смену региональных настроек с помощью настраиваемого параметра запроса. В конфигурации этого перехватчика определяется параметр URL под именем `lang` для смены региональных настроек веб-приложения.

В рассматриваемой здесь конфигурации определяется также компонент Spring Bean типа `ReloadableResourceBundleMessageSource`. Класс `ReloadableResourceBundleMessageSource` реализует интерфейс `MessageSource`, который служит для загрузки сообщений из определенных файлов, предназначенных для интернационализации (в данном случае это файлы `messages*.properties` и `application*.properties` из папки `/WEB-INF/i18n`). Обратите внимание на свойство `fallbackToSystemLocale`, которое предписывает модулю Spring MVC вернуться к региональным настройкам системы, в которой работает данное веб-приложение, если специальный комплект ресурсов для клиентских региональных настроек не обнаружен.

И, наконец, в данной конфигурации определяется компонент Spring Bean типа `CookieLocaleResolver`. В классе `CookieLocaleResolver` поддерживается хранение и извлечение региональных настроек из cookie-файла пользовательского браузера.

Модификация представления списка певцов для поддержки интернационализации

Теперь можно внести изменения, чтобы отображать на JSPX-странице интернационализированные сообщения. Ниже приведена исправленная разметка представления списка певцов[4].

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
     xmlns:c="http://java.sun.com/jsp/jstl/core"
     xmlns:fmt="http://java.sun.com/jsp/jstl/fmt"
     xmlns:spring="http://www.springframework.org/tags"
     version="2.0">
    <jsp:directive.page contentType=
        "text/html;charset=UTF-8"/>
    <jsp:output omit-xml-declaration="yes"/>

    <spring:message code="label_singer_list"
            var="label_SingerList"/>
```

[4] Функциональные средства Apache Tiles или JavaScript здесь пока еще не задействованы.

```
    <spring:message code="label_singer_first_name"
            var="labelSingerFirstName"/>
    <spring:message code="label_singer_last_name"
            var="labelSingerLastName"/>
    <spring:message code="label_singer_birth_date"
            var="labelSingerBirthDate"/>

    <h1>${label SingerList}</h1>

    <c:if test="${not empty singers}">
      <table>
        <thead>
     <tr>
        <th>${labelSingerFirstName}</th>
        <th>${labelSingerLastName}</th>
        <th>${labelSingerBirthDate}</th>
     </tr>
        </thead>
        <tbody>
    <c:forEach items="${singers}" var="singer">
      <tr>
        <td>${singer.firstName}</td>
        <td>${singer.lastName}</td>
        <td><fmt:formatDate value="${singer.birthDate}"/>
      </td>
      </tr>
    </c:forEach>
        </tbody>
      </table>
    </c:if>
</div>
```

Как следует из приведенной выше разметки, сначала на данной странице введено пространство имен `spring`. Затем с помощью дескриптора разметки `<spring:message>` в соответствующие переменные загружаются сообщения, требующиеся для представления. И, наконец, заголовок страницы и метки изменены для применения интернационализированных сообщений. А теперь постройте и повторно разверните проект, откройте браузер и перейдите на страницу по следующему URL: `http://localhost:8080/singers?lang=zh_HK`. В итоге вы увидите страницу с региональными настройками на китайском языке (HK).

В связи с тем что распознаватель региональных настроек `localeResolver` определен в контексте типа `WebApplicationContext` для сервлета диспетчера типа `DispatcherServlet`, модуль Spring MVC сохранит установленные региональные настройки в cookie-файле браузера (под именем `locale`), который по умолчанию будет храниться для поддержания сеанса связи с пользователем. Если же требуется хранить cookie-файл в течение более продолжительного периода времени, в определении компонента `localeResolver` достаточно переопределить свойство `cookieMax`

Age, наследуемое из класса org.springframework.web.util.CookieGenerator, вызвав метод setCookieMaxAge().

Чтобы переключиться на английский язык (US), достаточно изменить URL в браузере, указав ?lang=en_US, в результате чего появится страница на английском языке (US). И хотя мы не предусмотрели файл свойств application_en_US.properties, модуль Spring MVC возвратится к применению файла application.properties, в котором хранятся свойства на стандартном, т.е. английском, языке.

Тематическое оформление и шаблонизация

Помимо интернационализации, необходимо придать веб-приложению подходящий внешний вид (например, коммерческий веб-сайт должен выглядеть профессионально, тогда как социальный веб-сайт может иметь более свободный стиль), а также обеспечить согласованную компоновку, чтобы пользователи не запутались, работая с ним. А для того чтобы обеспечить согласованную компоновку, потребуется шаблонизатор. В этом разделе для поддержки шаблонизации представлений будет использован распространенный шаблонизатор Apache Tiles (http://tiles.apache.org).

Модуль Spring MVC тесно интегрирован с шаблонизатором Apache Tiles. Кроме того, в Spring стандартно поддерживаются более универсальные шаблонизаторы Velocity и FreeMarker, пригодные для применения за пределами веб-приложений, а также шаблоны электронной почты и пр. В последующих разделах поясняется, каким образом активизируется поддержка тематического оформления в Spring MVC и как пользоваться шаблонизатором Apache Tiles для определения компоновки страницы.

Поддержка тематического оформления

В модуле Spring MVC обеспечивается всесторонняя поддержка тематического оформления, которая легко активизируется в веб-приложениях. Например, в рассматриваемом здесь приложении для певцов требуется создать тему и назвать ее стандартной. С этой целью в папке /src/main/resources сначала создается файл свойств standard.properties со следующим содержимым:

```
styleSheet=resources/styles/standard.css
```

Этот файл содержит свойство styleSheet, где указывается стилевая таблица, предназначенная для применения в стандартной теме. Этот файл свойств служит в качестве комплекта ресурсов для темы, к которой можно добавить сколько угодно компонентов (например, местоположение изображения логотипа или фона).

Следующий шаг состоит в конфигурировании контекста типа WebApplicationContext для сервлета диспетчера DispatcherServlet, чтобы активизировать поддержку тематического оформления. И для этого придется внести коррективы в соответствующий конфигурационный класс. Во-первых, в методе addInterceptors() необходимо ввести еще один компонент перехватчика, как показано ниже.

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
  registry.addInterceptor(localeChangeInterceptor());
  registry.addInterceptor(themeChangeInterceptor());
}

@Bean
ThemeChangeInterceptor themeChangeInterceptor() {
  return new ThemeChangeInterceptor();
}
```

Здесь введен новый перехватчик типа `ThemeChangeInterceptor`, предназначенный для перехвата каждого запроса для изменения темы. Во-вторых, в конфигурационный класс необходимо ввести определения следующих компонентов Spring Beans:

```
@Bean
ResourceBundleThemeSource themeSource() {
  return new ResourceBundleThemeSource();
}

@Bean
CookieThemeResolver themeResolver() {
  CookieThemeResolver cookieThemeResolver =
                      new CookieThemeResolver();
  cookieThemeResolver.setDefaultThemeName("standard");
  cookieThemeResolver.setCookieMaxAge(3600);
  cookieThemeResolver.setCookieName("theme");
  return cookieThemeResolver;
}
```

Здесь определены два компонента Spring Beans. Первый компонент Spring Bean типа `ResourceBundleThemeSource` отвечает за загрузку комплекта ресурсов для активной темы. Так, если активная тема называется `standard`, то этот компонент Spring Bean будет искать файл свойств `standard.properties` как комплект ресурсов для данной темы. Второй компонент Spring Bean типа `CookieThemeResolver` распознает активную тему для пользователей. В частности, свойство `defaultThemeName` определяет используемую по умолчанию тему `standard`. Обратите внимание на то, что для сохранения темы специально для пользователя в классе `CookieThemeResolver` применяются cookie-файлы. Имеется также класс `SessionThemeResolver`, обеспечивающий сохранение атрибута *темы* в пользовательском сеансе.

Итак, тема `standard` сконфигурирована и готова к применению в представлениях рассматриваемого здесь веб-приложения. В приведенной ниже конфигурации JSPX-страницы демонстрируется исправленное представление списка певцов с поддержкой тематического оформления (`/WEB-INF/views/singers/list.jspx`).

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
```

```
     xmlns:c="http://java.sun.com/jsp/jstl/core"
     xmlns:fmt="http://java.sun.com/jsp/jstl/fmt"
     xmlns:spring="http://www.springframework.org/tags"
     version="2.0">
    <jsp:directive.page contentType="text/html;
     charset=UTF-8"/>
    <jsp:output omit-xml-declaration="yes"/>

    <spring:message code="label_singer_list"
            var="labelSingerList"/>
    <spring:message code="label_singer_first_name"
            var="labelSingerFirstName"/>
    <spring:message code="label_singer_last_name"
            var="labelSingerLastName"/>
    <spring:message code="label_singer_birth_date"
            var="labelSingerBirthDate"/>

    <head>
      <spring:theme code="styleSheet" var="app_css" />
      <spring:url value="/${app_css}" var="app_css_url" />
      <link rel="stylesheet" type="text/css" media="screen"
            href="${app_css_url}" />
    </head>

    <h1>${labelSingerList}</h1>

    <c:if test="${not empty singers}">
      <table>
        <thead>
        <tr>
    <th>${labelSingerFirstName}</th>
    <th>${labelSingerLastName}</th>
    <th>${labelSingerBirthDate}</th>
      </tr>
      </thead>
      <tbody>
      <c:forEach items="${singers}" var="singer">
    <tr>
      <td>${singer.firstName}</td>
      <td>${singer.lastName}</td>
      <td><fmt:formatDate value="${singer.birthDate}"/></td>
    </tr>
       </c:forEach>
        </tbody>
      </table>
    </c:if>
</div>
```

В конфигурацию данного представления добавлен раздел <head>, а дескриптор разметки <spring:theme> служит для извлечения из комплекта ресурсов тематического представления свойства styleSheet, в котором указан файл стилевой таблицы standard.css. И, наконец, в данное представление введена ссылка на эту стилевую таблицу.

Снова построив и развернув приложение на сервере, откройте браузер и перейдите на страницу с представлением списка певцов по следующему URL: http://localhost:8080/singers. В итоге вы увидите, что на данной странице был применен стиль тематического оформления, определенный в файле standard.css. Используя поддержку тематического оформления в Spring MVC, можно легко добавлять новые или изменять уже существующие темы в веб-приложении.

Шаблонизация представлений средствами Apache Tiles

Для шаблонизации представлений с помощью технологии JSP наиболее распространенной является шаблонизатор Apache Tiles (http://tiles.apache.org). Модуль Spring MVC тесно интегрирован с Apache Tiles. Чтобы воспользоваться шаблонизатором Apache Tiles и организовать проверку достоверности данных, в рассматриваемый здесь проект внедрены библиотеки tilesjsp, validation-api и hibernate-validator как отдельные зависимости. В последующих разделах поясняется, как реализовать шаблоны страниц, включая оформление компоновки страниц, определение и реализацию компонентов в этой компоновке.

Оформление компоновки шаблона

Прежде всего необходимо определить количество шаблонов, требующихся в приложении, а также компоновку для каждого шаблона. В рассматриваемом здесь примере приложения для певцов требуется лишь один шаблон. Как показано на рис. 16.4, этот шаблон довольно прост.

Как видите, в данном шаблоне требуются следующие страничные компоненты.

- **/WEB-INF/views/header.jspx**. Предоставляет область заголовка.
- **/WEB-INF/views/menu.jspx**. Предоставляет область левого меню, а также форму регистрации, которая будет реализована далее в главе.
- **/WEB-INF/views/footer.jspx**. Предоставляет область нижнего колонтитула.

Для определения шаблона далее будет использоваться шаблонизатор Apache Tiles. И для этой цели придется разработать перечисленные ниже файлы шаблона страницы и определений компоновки.

- **/WEB-INF/layouts/default.jspx**. Предоставляет полную компоновку для конкретного шаблона.

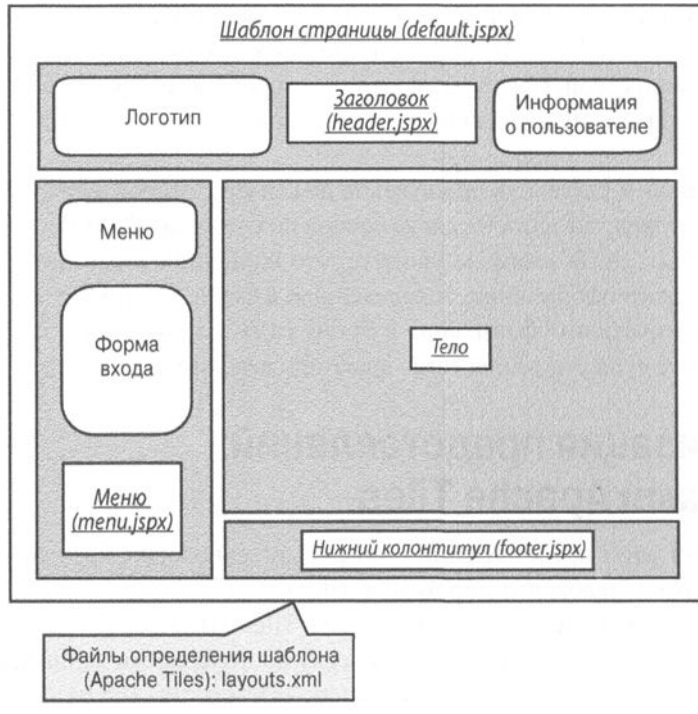


Рис. 16.4. Шаблон страницы с компонентами компоновки

- **`/WEB-INF/layouts/layouts.xml`**. Содержит определения компоновки, требующиеся в Apache Tiles.

Реализация компонентов компоновки страницы

Имея в своем распоряжении определение компоновки, можно реализовать ее компоненты для оформления страниц веб-приложения. Разработаем сначала файл шаблона страницы и файлы определений компоновки, требующиеся в Apache Tiles, как показано ниже.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE tiles-definitions PUBLIC
    "-//Apache Software Foundation
      //DTD Tiles Configuration 2.1//EN"
    "http://tiles.apache.org/dtds/tiles-config_3_0.dtd">

  <tiles-definitions>
    <definition name="default"
            template="/WEB-INF/layouts/default.jspx">
      <put-attribute name="header"
                  value="/WEB-INF/views/header.jspx" />
      <put-attribute name="menu"
```

```
                value="/WEB-INF/views/menu.jspx" />
    <put-attribute name="footer"
                value="/WEB-INF/views/footer.jspx" />
  </definition>
</tiles-definitions>
```

Код разметки в приведенном выше файле должен быть нетрудным для понимания. В нем содержится одно определение шаблона страницы под именем `default`, а код шаблона находится в файле `default.jspx`. На данной странице определяются три компонента: `header`, `menu` и `footer`. Содержимое этих компонентов будет загружаться из файлов, указанных в атрибутах `value`. За подробным описанием определения Apache Tiles обращайтесь к соответствующей документации (`http://tiles.apache.org/`).

В следующем фрагменте конфигурации JSPX-страницы демонстрируется содержимое файла шаблона `default.jspx`:

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:c="http://java.sun.com/jsp/jstl/core"
      xmlns:fn="http://java.sun.com/jsp/jstl/functions"
      xmlns:tiles="http://tiles.apache.org/tags-tiles"
      xmlns:spring="http://www.springframework.org/tags">

    <jsp:output doctype-root-element="HTML"
      doctype-system="about:legacy-compat" />

    <jsp:directive.page contentType="text/html;
        charset=UTF-8" />

    <jsp:directive.page pageEncoding="UTF-8" />
    <head>
      <meta http-equiv="Content-Type" content="text/html;
            charset=UTF-8" />
      <meta http-equiv="X-UA-Compatible" content="IE=8" />

      <spring:theme code="styleSheet" var="app_css" />
      <spring:url value="/${app_css}" var="app_css_url" />
      <link rel="stylesheet" type="text/css" media="screen"
            href="${app_css_url}" />

      <!-- Извлечь пользовательские региональные настройки
           из контекста страницы (они были установлены
           распознавателем региональных настроек
           Spring MVC) -->
      <c:set var="userLocale">
      <c:set var="plocale">${pageContext.response.locale}
      </c:set>
      <c:out value="${fn:replace(plocale, '_', '-')}"
           default="en" />
```

```
      </c:set>

      <spring:message code="application_name" var="app_name"
                htmlEscape="false"/>
      <title><spring:message code="welcome_h3"
                      arguments="${app_name}" />
      </title>
    </head>

    <body class="tundra spring">
    <div id="headerWrapper">
      <tiles:insertAttribute name="header" ignore="true" />
    </div>
    <div id="wrapper">
      <tiles:insertAttribute name="menu" ignore="true" />
    <div id="main">
      <tiles:insertAttribute name="body"/>
      <tiles:insertAttribute name="footer" ignore="true"/>
    </div>
  </div>
</body>
</html>
```

По существу, это JSP-страница. Ниже описаны основные положения, на которые следует обратить внимание в приведенном выше шаблоне.

- В шаблон введен дескриптор разметки `<spring:theme>`, чтобы поддержать тематическое оформление на уровне шаблона.
- Дескриптор разметки `<tiles:insertAttribute>` служит для обозначения страничных компонентов, которые должны загружаться из других файлов, как указано в файле `layouts.xml`.

А теперь реализуем компоненты заголовка (`header`), меню (`menu`) и нижнего колонтитула (`footer`). Код разметки этих компонентов приведен ниже. В частности, файл `header.jspx` довольно прост и содержит лишь следующий код разметки:

```
<div id="header" xmlns:jsp="http://java.sun.com/JSP/Page"
     xmlns:spring="http://www.springframework.org/tags"
     version="2.0">
    <jsp:directive.page contentType="text/html;
         charset=UTF-8" />
    <jsp:output omit-xml-declaration="yes" />

    <spring:message code="header_text" var="headerText"/>

    <div id="appname">
      <h1>${headerText}</h1>
    </div>
</div>
```

Не менее прост и файл `menu.jspx`, поскольку в данном веб-приложении предусмотрен минимальный пользовательский интерфейс, чтобы не отвлекать основное внимание от самого каркаса Spring.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div id="menu" xmlns:jsp="http://java.sun.com/JSP/Page"
     xmlns:spring="http://www.springframework.org/tags"
     version="2.0">
    <jsp:directive.page contentType="text/html;
         charset=UTF-8" />
    <jsp:output omit-xml-declaration="yes" />
    <spring:message code="menu_header_text"
                    var="menuHeaderText"/>
    <spring:message code="menu_add_singer"
                    var="menuAddsinger"/>
    <spring:url value="/singers?form" var="addsingerUrl"/>

      <h3>${menuHeaderText}</h3>
        <a href="${addsingerUrl}"><h3>${menuAddsinger}
      </h3></a>
</div>
```

Файл `footer.jspx` содержит URL для смены языка, на котором отображается пользовательский интерфейс.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div id="footer" xmlns:jsp="http://java.sun.com/JSP/Page"
     xmlns:spring="http://www.springframework.org/tags"
     version="2.0">
    <jsp:directive.page contentType="text/html;
        charset=UTF-8" />
    <jsp:output omit-xml-declaration="yes" />
    <spring:message code="home_text" var="homeText"/>
    <spring:message code="label_en_US" var="labelEnUs"/>
    <spring:message code="label_zh_HK" var="labelZhHk"/>
    <spring:url value="/singers" var="homeUrl"/>

    <a href="${homeUrl}">${homeText}</a> |
    <a href="${homeUrl}?lang=en_US">${labelEnUs}</a> |
    <a href="${homeUrl}?lang=zh_HK">${labelZhHk}</a>
</div>
```

Теперь можно изменить представление списка певцов, чтобы вписать его в шаблон. По существу, для этого достаточно удалить раздел `<head>`, поскольку он уже имеется на шаблонной странице (из файла `default.jspx`). Исправленное и усовершенствованное таким образом представление списка певцов приведено ниже.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
     xmlns:c="http://java.sun.com/jsp/jstl/core"
```

```
    xmlns:spring="http://www.springframework.org/tags"
    xmlns:fmt="http://java.sun.com/jsp/jstl/fmt"
    version="2.0">
  <jsp:directive.page contentType="text/html;
      charset=UTF-8"/>
  <jsp:output omit-xml-declaration="yes"/>

  <spring:message code="label_singer_list"
          var="labelSingerList"/>
  <spring:message code="label_singer_first_name"
          var="labelSingerFirstName"/>
  <spring:message code="label_singer_last_name"
          var="labelSingerLastName"/>
  <spring:message code="label_singer_birth_date"
          var="labelSingerBirthDate"/>

  <h1>${labelSingerList}</h1>

  <c:if test="${not empty singers}">
    <table>
      <thead>
      <tr>
  <th>${labelSingerFirstName}</th>
  <th>${labelSingerLastName}</th>
  <th>${labelSingerBirthDate}</th>
    </tr>
    </thead>
    <tbody>
    <c:forEach items="${singers}" var="singer">
  <tr>
    <td>${singer.firstName}</td>
    <td>${singer.lastName}</td>
    <td><fmt:formatDate value="${singer.birthDate}"/>
    </td>
  </tr>
     </c:forEach>
     </tbody>
    </table>
  </c:if>
</div>
```

Итак, шаблон, определение и компоненты готовы. Следующий шаг состоит в конфигурировании Spring MVC для интеграции с Apache Tiles.

Конфигурирование Apache Tiles в Spring MVC

Конфигурирование поддержки Apache Tiles в Spring MVC осуществляется довольно просто. В конфигурацию сервлета диспетчера типа `DispatcherServlet` (в классе `WebConfig`) необходимо внести коррективы, заменив класс `InternalResource`

ViewResolver классом UrlBasedViewResolver. В следующем фрагменте кода определяются только те компоненты Spring Beans, которые требуются для поддержки конфигурирования Apache Tiles:

```
package com.apress.prospring5.ch16.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.web.servlet.view
          .UrlBasedViewResolver;
import org.springframework.web.servlet.view.tiles3
          .TilesConfigurer;
import org.springframework.web.servlet.view.tiles3
          .TilesView;
...
@Configuration
@EnableWebMvc
@ComponentScan(basePackages =
               {"com.apress.prospring5.ch16"})
public class WebConfig implements WebMvcConfigurer {

  @Bean
  UrlBasedViewResolver tilesViewResolver() {
    UrlBasedViewResolver tilesViewResolver =
                         new UrlBasedViewResolver();
    tilesViewResolver.setViewClass(TilesView.class);
    return tilesViewResolver;
  }

  @Bean
  TilesConfigurer tilesConfigurer() {
    TilesConfigurer tilesConfigurer =
                    new TilesConfigurer();
    tilesConfigurer.setDefinitions(
                    "/WEB-INF/layouts/layouts.xml",
                    "/WEB-INF/views/**/views.xml"
    );
    tilesConfigurer.setCheckRefresh(true);
    return tilesConfigurer;
  }
  ...
}
```

В приведенном выше конфигурационном классе определяется компонент Spring Bean типа UrlBasedViewResolver вместо типа ViewResolver со свойством viewClass, в котором задается класс TilesView, обеспечивающий поддержку Apache

Tiles в Spring MVC. Кроме того, здесь определен компонент `tilesConfigurer`, в котором предоставляются конфигурации компоновки, требующиеся для Apache Tiles.

Необходимо также подготовить еще один конфигурационный файл, `/WEB-INF/views/singers/views.xml`, где определяются представления, требующиеся в веб-приложении для певцов. Содержимое этого файла приведено ниже.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE tiles-definitions PUBLIC "-//Apache Software
    Foundation//DTD Tiles Configuration 3.0//EN"
    "http://tiles.apache.org/dtds/tiles-config_3_0.dtd">
<tiles-definitions>
    <definition extends="default" name="singers/list">
      <put-attribute name="body"
           value="/WEB-INF/views/singers/list.jspx" />
   </definition>
</tiles-definitions>
```

Как показано выше, логическое имя представления сопоставляется с соответствующим атрибутом `body` отображаемого представления. Как и в упомянутом выше классе `SingerController`, метод `list()` возвращает логическое имя представления `contacts/list`, чтобы шаблонизатор Apache Tiles смог сопоставить имя представления с нужным шаблоном и отобразить тело представления.

Теперь можно протестировать скомпонованную страницу. С этой целью убедитесь, что данный проект повторно построен и развернут на сервере. Снова загрузите представление списка певцов (`http://localhost:8080/singers`), чтобы отобразить представление, построенное по шаблону.

Реализация представлений для показа сведений о певцах

Продолжим реализацию представлений, которые позволят пользователям просматривать подробные сведения о певцах, создавать новые записи о них и обновлять сведения о существующих певцах. В последующих разделах будет показано, каким образом URL сопоставляются с различными представлениями, как реализуются эти представления и активизируется поддержка проверки достоверности по спецификации JSR-349 в Spring MVC для представления процесса редактирования.

Сопоставление URL с представлениями

Прежде всего необходимо придумать способ сопоставления URL с соответствующими представлениями. В модуле Spring MVC чаще всего принято выполнять сопоставление URL с представлениями в стиле REST. В табл. 16.3 представлено такое сопоставление в рассматриваемом здесь веб-приложении наряду с именами методов контроллера, выполняющих соответствующие действия.

Таблица 16.3. Сопоставление URL с представлениями в примере веб-приложения

URL	Метод доступа по протоколу HTTP	Метод контроллера	Описание
`/singers`	`GET`	`list()`	Выводит список певцов
`/singers/id`	`GET`	`show()`	Отображает сведения об одном певце
`/singers/id?form`	`GET`	`updateForm()`	Отображает форму редактирования для обновления сведений о существующем певце
`/singers/id?form`	`POST`	`update()`	В этом случае пользователи обновляют сведения о певце и отправляют форму на обработку
`/singers?form`	`GET`	`createForm()`	Отображает форму редактирования для создания записи о новом певце
`/singers?form`	`POST`	`create()`	В этом случае пользователи вводят сведения о певце и отправляют форму на обработку
`/singers/photo/id`	`GET`	`downloadPhoto()`	Загружает фотографию певца

Реализация представления для показа сведений о певцах

А теперь реализуем представление для показа сведений о певцах. Реализация этого представления осуществляется в три этапа.

1. Реализация метода контроллера.
2. Реализация представления для показа сведений о певцах (`/views/singers/show.jspx`).
3. Изменение файла определения представлений (`/views/singers/views.xml`) для данного представления.

Ниже приведена реализация метода `show()` из класса `SingerController`, предназначенного для отображения сведений о певце.

```
package com.apress.prospring5.ch16.web;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation
        .Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
```

```
import org.springframework.web.bind.annotation
          .PathVariable;
import org.springframework.web.bind.annotation
          .RequestMapping;
import org.springframework.web.bind.annotation
          .RequestMethod;
...

@RequestMapping("/singers")
@Controller
public class SingerController {
  private final Logger logger =
          LoggerFactory.getLogger(SingerController.class);

  private SingerService singerService;
  private MessageSource messageSource;

  @RequestMapping(value = "/{id}",
                  method = RequestMethod.GET)
  public String show(@PathVariable("id") Long id,
                     Model uiModel) {
    Singer singer = singerService.findById(id);
    uiModel.addAttribute("singer", singer);

    return "singers/show";
  }

  @Autowired
  public void setSingerService(SingerService singerService) {
    this.singerService = singerService;
  }
  ...
}
```

Аннотация `@RequestMapping`, применяемая в методе `show()`, указывает на то, что этот метод предназначен для обработки URL типа `/singers/{id}` с методом `GET` доступа по сетевому протоколу HTTP. В данном методе к аргументу `id` применяется аннотация `@PathVariable`, которая вынуждает модуль Spring MVC извлекать идентификатор из URL и присваивать его аргументу `id`. Затем сведения о певце извлекаются и вводятся в модель данных, а возвращается логическое имя представления `singers/show`. Следующий шаг состоит в реализации представления для показа сведений о певцах (`/views/singers/show.jspx`):

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
     xmlns:c="http://java.sun.com/jsp/jstl/core"
     xmlns:spring="http://www.springframework.org/tags"
     xmlns:fmt="http://java.sun.com/jsp/jstl/fmt"
```

```
 version="2.0">
<jsp:directive.page contentType="text/html;
     charset=UTF-8"/>
<jsp:output omit-xml-declaration="yes"/>

<spring:message code="label_singer_info"
        var="labelSingerInfo"/>
<spring:message code="label_singer_first_name"
        var="labelSingerFirstName"/>
<spring:message code="label_singer_last_name"
        var="labelSingerLastName"/>
<spring:message code="label_singer_birth_date"
        var="labelSingerBirthDate"/>
<spring:message code="label_singer_description"
                 var="labelSingerDescription"/>
<spring:message code="label_singer_update"
                 var="labelSingerUpdate"/>
<spring:message code="date_format_pattern"
                 var="dateFormatPattern"/>
<spring:message code="label_singer_photo"
                 var="labelSingerPhoto"/>

<spring:url value="/singers/photo"
              var="singerPhotoUrl"/>
<spring:url value="/singers" var="editSingerUrl"/>

<h1>${labelSingerInfo}</h1>

<div id="singerInfo">

<c:if test="${not empty message}">
  <div id="message" class="${message.type}">
    ${message.message}</div>
</c:if>

<table>
  <tr>
<td>${labelSingerFirstName}</td>
<td>${singer.firstName}</td>
  </tr>
  <tr>
<td>${labelSingerLastName}</td>
<td>${singer.lastName}</td>
  </tr>
  <tr>
<td>${labelSingerBirthDate}</td>
<td><fmt:formatDate value="${singer.birthDate}"/></td>
  </tr>
  <tr>
```

```
    <td>${labelSingerDescription}</td>
    <td>${singer.description}</td>
      </tr>
      <tr>
    <td>${labelSingerPhoto}</td>
    <td><img src="${singerPhotoUrl}/${singer.id}"></img>
    </td>
      </tr>
    </table>

      <a href="${editSingerUrl}/${singer.id}?form">
         ${labelSingerUpdate}</a>
    </div>
</div>
```

Размеченная выше страница довольно проста: на ней лишь отображается атрибут `singer` модели данных. И последний шаг состоит в модификации файла определения представлений (`/views/singers/views.xml`) для сопоставления логического имени с представлением `singers/show`:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE tiles-definitions PUBLIC "-//Apache Software
     Foundation//DTD Tiles Configuration 3.0//EN"
     "http://tiles.apache.org/dtds/tiles-config_3_0.dtd">
    <tiles-definitions>
      <definition extends="default" name="singers/list">
      <put-attribute name="body"
           value="/WEB-INF/views/singers/list.jspx" />
      </definition>
      <definition extends="default" name="singers/show">
        <put-attribute name="body"
             value="/WEB-INF/views/singers/show.jspx" />
      </definition>
</tiles-definitions>
```

На этом реализация представления для показа сведений о певцах завершается. Остается лишь добавить ссылку на это представление (`/views/singers/list.jspx`) для отображения сведений о каждом певце. Ниже приведено исправленное содержимое файла `list.jspx`.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
     xmlns:c="http://java.sun.com/jsp/jstl/core"
     xmlns:spring="http://www.springframework.org/tags"
     xmlns:fmt="http://java.sun.com/jsp/jstl/fmt"
     version="2.0">
    <jsp:directive.page contentType="text/html;
         charset=UTF-8"/>
    <jsp:output omit-xml-declaration="yes"/>
```

```
    <spring:message code="label_singer_list"
                    var="labelSingerList"/>
    <spring:message code="label_singer_first_name"
                    var="labelSingerFirstName"/>
    <spring:message code="label_singer_last_name"
                    var="labelSingerLastName"/>
    <spring:message code="label_singer_birth_date"
                    var="labelSingerBirthDate"/>

    <h1>${labelSingerList}</h1>

    <spring:url value="/singers" var="showSingerUrl"/>

    <c:if test="${not empty singers}">
      <table>
        <thead>
          <tr>
        <th>${labelSingerFirstName}</th>
        <th>${labelSingerLastName}</th>
        <th>${labelSingerBirthDate}</th>
          </tr>
          </thead>
          <tbody>
          <c:forEach items="${singers}" var="singer">
        <tr>
          <td>
          <a href="${showSingerUrl}/${singer.id}">
                  ${singer.firstName}</a>
          </td>
          <td>${singer.lastName}</td>
          <td><fmt:formatDate value="${singer.birthDate}"/>
          </td>
        </tr>
          </c:forEach>
         </tbody>
      </table>
    </c:if>
</div>
```

Как показано выше, с помощью дескриптора разметки `<spring:url>` объявляется переменная URL и добавляется ссылка на атрибут `firstName`. Чтобы протестировать представление для отображения сведений о певцах, снова постройте, разверните и откройте представление списка певцов. Этот список должен теперь включать в себя гиперссылки на представление сведений о певцах. После щелчка на любой из этих ссылок воспроизводится представление сведений о певцах.

Реализация представления для редактирования сведений о певцах

Теперь реализуем представление для редактирования сведений о певцах. Это то же самое представление, что и для показа сведений о певцах. С этой целью введем сначала в класс `SingerController` методы `updateForm()` и `update()`. Ниже приведен модифицированный код контроллера с двумя новыми методами.

```
package com.apress.prospring5.ch16.web;
...
@RequestMapping("/singers")
@Controller
public class SingerController {
  private final Logger logger =
          LoggerFactory.getLogger(SingerController.class);
  private SingerService singerService;
  private MessageSource messageSource;

  @RequestMapping(value = "/{id}", params = "form",
                  method = RequestMethod.POST)
  public String update(@Valid Singer singer,
                BindingResult bindingResult, Model uiModel,
                HttpServletRequest httpServletRequest,
                RedirectAttributes redirectAttributes,
                Locale locale) {
    logger.info("Updating singer");
    if (bindingResult.hasErrors()) {
      uiModel.addAttribute("message", new Message("error",
              messageSource.getMessage("singer_save_fail",
                            new Object[]{}, locale)));
      uiModel.addAttribute("singer", singer);
      return "singers/update";
    }
    uiModel.asMap().clear();
    redirectAttributes.addFlashAttribute("message",
           new Message("success", messageSource.getMessage(
                       "singer_save_success",
                       new Object[]{}, locale)));
    singerService.save(singer);
    return "redirect:/singers/"
          + UrlUtil.encodeUrlPathSegment(singer.getId()
                   .toString(), httpServletRequest);
  }

  @RequestMapping(value = "/{id}", params = "form",
                  method = RequestMethod.GET)
  public String updateForm(@PathVariable("id") Long id,
                           Model uiModel) {
```

```
    uiModel.addAttribute("singer",
                         singerService.findById(id));
    return "singers/update";
  }

  @Autowired
  public void setSingerService(
              SingerService singerService) {
    this.singerService = singerService;
  }

  @Autowired
  public void setMessageSource(
              MessageSource messageSource) {
    this.messageSource = messageSource;
  }
  ...
}
```

Ниже описаны основные положения, на которые следует обратить внимание в приведенном выше классе контроллера.

- Интерфейс `MessageSource` автоматически связан с контроллером для извлечения сообщений с поддержкой интернационализации.
- В методе `updateForm()` сведения о певце извлекаются и сохраняются в модели данных, после чего возвращается логическое имя представления `singers/update`, по которому воспроизводится представление для редактирования сведений о певцах.
- Метод `update()` будет вызываться в тот момент, когда пользователь обновляет сведения о певце и щелкает на кнопке **Save** (Сохранить). Этот метод требует некоторых пояснений. Сначала модуль Spring MVC попытается привязать отправленные данные к объекту предметной области типа `Singer` и автоматически выполнит преобразование типов и форматирование данных. Если обнаружатся ошибки привязки (например, дата рождения, введенная в неверном формате), эти ошибки будут сохранены в интерфейсе `BindingResult` (из пакета `org.springframework.validation`), а сообщения об ошибках — в модели данных, что приведет к повторному отображению представления для редактирования сведений о певцах. Если же привязка завершится удачно, данные сохранятся и возвратится логическое имя представления сведений о певцах, снабженное префиксом **`redirect:`**. Обратите внимание на то, что в данном случае требуется отобразить сообщение после переадресации, и поэтому необходимо вызвать метод `RedirectAttributes.addFlashAttribute()` (из интерфейса, входящего в состав пакета `org.springframework.web.servlet.mvc.support`), чтобы отобразить сообщение об удачном завершении данной операции в представлении сведений о певцах. Перед переадресацией в

Spring MVC временно сохраняются (обычно в сеансе связи) специальные атрибуты, которые становятся доступными по запросу после переадресации, а затем сразу же удаляются.

- Специальный класс `Message` служит для хранения отдельного сообщения, извлеченного из интерфейса `MessageSource`, а также типа сообщения (об удачном завершении данной операции или ошибке) для последующего воспроизведения с помощью соответствующего представления в области сообщений. Ниже приведен исходный код этого класса.

```
package com.apress.prospring5.ch16.util;

public class Message {
  private String type;
  private String message;

  public Message(String type, String message) {
    this.type = type;
    this.message = message;
  }

  public String getType() {
    return type;
  }

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

- Служебный класс `UrlUtil` предназначен для кодирования URL с целью переадресации. Ниже приведен исходный код этого класса.

```
package com.apress.prospring5.ch16.util;

import java.io.UnsupportedEncodingException;
import javax.servlet.http.HttpServletRequest;
import org.springframework.web.util.UriUtils;
import org.springframework.web.util.WebUtils;

public class UrlUtil {
  public static String encodeUrlPathSegment(
                String pathSegment,
                HttpServletRequest httpServletRequest) {
    String enc = httpServletRequest.getCharacterEncoding();
    if (enc == null) {
      enc = WebUtils.DEFAULT_CHARACTER_ENCODING;
    }

    try {
```

```
            pathSegment =
                UriUtils.encodePathSegment(pathSegment, enc);
          } catch (UnsupportedEncodingException uee) {
            // обработать исключение
          }
          return pathSegment;
        }
      }
```

А теперь перейдем непосредственно к представлению для редактирования сведений о певцах (`/views/singers/edit.jspx`), которое будет использоваться как для обновления сведений о существующем певце, так и для создания записи о новом певце. Код разметки этого представления приведен ниже.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<div xmlns:jsp="http://java.sun.com/JSP/Page"
     xmlns:c="http://java.sun.com/jsp/jstl/core"
     xmlns:spring="http://www.springframework.org/tags"
     xmlns:form="http://www.springframework.org/tags/form"
     version="2.0">

    <jsp:directive.page contentType="text/html;
         charset=UTF-8"/>
    <jsp:output omit-xml-declaration="yes"/>

    <spring:message code="label_singer_new"
                    var="labelSingerNew"/>
    <spring:message code="label_singer_update"
                    var="labelSingerUpdate"/>
    <spring:message code="label_singer_first_name"
                    var="labelSingerFirstName"/>
    <spring:message code="label_singer_last_name"
                    var="labelSingerLastName"/>
    <spring:message code="label_singer_birth_date"
                    var="labelSingerBirthDate"/>
    <spring:message code="label_singer_description"
                    var="labelSingerDescription"/>
    <spring:message code="label_singer_photo"
                    var="labelSingerPhoto"/>

    <spring:eval expression="singer.id == null ? labelSingerNew:
                 labelSingerUpdate" var="formTitle"/>

    <h1>${formTitle}</h1>
      <div id="singerUpdate">
    <form:form modelAttribute="singer"
          id="singerUpdateForm" method="post">
```

```
<c:if test="${not empty message}">
  <div id="message" class="${message.type}">
          ${message.message}</div>
</c:if>

<form:label path="firstName">
     ${labelSingerFirstName}*
</form:label>
<form:input path="firstName" />
<div>
  <form:errors path="firstName" cssClass="error" />
</div>
<p/>

<form:label path="lastName">
     ${labelSingerLastName}*
</form:label>
<form:input path="lastName" />
<div>
  <form:errors path="lastName" cssClass="error" />
</div>
<p/>

<form:label path="birthDate">
     ${labelSingerBirthDate}
</form:label>
<form:input path="birthDate" id="birthDate"/>
<div>
  <form:errors path="birthDate" cssClass="error" />
</div>
<p/>

<form:label path="description">
     ${labelSingerDescription}
</form:label>
<form:textarea cols="60" rows="8" path="description"
  id="singerDescription"/>
<div>
  <form:errors path="description" cssClass="error" />
</div>
<p/>

<label for="file">
  ${labelSingerPhoto}
</label>
<input name="file" type="file"/>
<p/>

<form:hidden path="version" />
```

```
    <button type="submit">Save</button>
    <button type="reset">Reset</button>
    </form:form>
      </div>
</div>
```

Ниже описаны основные положения, на которые следует обратить внимание в приведенном выше коде разметки.

- В дескрипторе разметки `<spring:eval>` используется язык SpEL (Spring Expression Language — язык выражений Spring), чтобы проверить, равен ли идентификатор певца пустому значению `null`. Если это действительно так, значит, это новый певец, а иначе обновляются сведения о существующем певце. В итоге будет отображен соответствующий заголовок формы.
- В заполняемой форме с помощью различных дескрипторов разметки `<form>` из Spring MVC отображаются метка, поле ввода и сообщения об ошибках, если при передаче формы на обработку привязка завершилась неудачно.

Введем далее сопоставление с представлением в файл определения представлений (`/views/singers/views.xml`). Соответствующий код разметки приведен ниже.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE tiles-definitions PUBLIC "-//Apache Software
    Foundation//DTD Tiles Configuration 3.0//EN"
    "http://tiles.apache.org/dtds/tiles-config_3_0.dtd">
<tiles-definitions>
  <definition extends="default" name="singers/update">
    <put-attribute name="body"
         value="/WEB-INF/views/singers/edit.jspx" />
  </definition>
  ...
</tiles-definitions>
```

На этом реализация представления для редактирования сведений о певцах завершается. Снова постройте и разверните данный проект. После щелчка на ссылке для редактирования появится представление редактирования сведений о певцах. Обновите сведения о певце и щелкните на кнопке Save. Если привязка завершилась удачно, вы увидите сообщение об успешном сохранении сведений о певце, а затем появится представление для показа сведений о певцах.

Реализация представления для ввода сведений о певце

Реализация представления для ввода сведений о певце очень похожа на реализацию представления для редактирования этих сведений. В данном случае будет повторно использована страница из файла `edit.jspx`, поэтому останется лишь ввести нужные методы в класс `SingerController` и определение этого представления.

В следующем фрагменте кода из класса `SingerController` представлены методы, введенные с целью реализовать операцию сохранения сведений о новом певце:

```
package com.apress.prospring5.ch16.web;
...

@RequestMapping("/singers")
@Controller
public class SingerController {
  private final Logger logger =
          LoggerFactory.getLogger(SingerController.class);
  private SingerService singerService;
  private MessageSource messageSource;

  @RequestMapping(method = RequestMethod.POST)
  public String create(@Valid Singer singer,
         BindingResult bindingResult, Model uiModel,
         HttpServletRequest httpServletRequest,
         RedirectAttributes redirectAttributes,
         Locale locale) {
    logger.info("Creating singer");
    if (bindingResult.hasErrors()) {
       uiModel.addAttribute("message", new Message("error",
              messageSource.getMessage("singer_save_fail",
                                  new Object{}, locale)));
       uiModel.addAttribute("singer", singer);
       return "singers/create";
    }
    uiModel.asMap().clear();
    redirectAttributes.addFlashAttribute("message",
            new Message("success",
            messageSource.getMessage("singer_save_success",
                                new Object[]{}, locale)));
    logger.info("Singer id: " + singer.getId());
    singerService.save(singer);
    return "redirect:/singers/";
  }

  @RequestMapping(params = "form",
                  method = RequestMethod.GET)
  public String createForm(Model uiModel) {
    Singer singer = new Singer();
    uiModel.addAttribute("singer", singer);
    return "singers/create";
  }

  @Autowired
  public void setSingerService(
              SingerService singerService) {
```

```
    this.singerService = singerService;
  }
  ...
}
```

В следующем фрагменте кода разметки сопоставление с представлением вводится в файл определения представлений (`/views/singers/views.xml`):

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE tiles-definitions PUBLIC "-//Apache Software
    Foundation//DTD Tiles Configuration 3.0//EN"
    "http://tiles.apache.org/dtds/tiles-config_3_0.dtd">
<tiles-definitions>
  <definition extends="default" name="singers/create">
    <put-attribute name="body"
         value="/WEB-INF/views/singers/edit.jspx" />
  </definition>
  ...
</tiles-definitions>
```

На этом реализация представления для ввода сведений о певце завершается. Построив и развернув данный проект снова, щелкните на ссылке для создания новой записи о певце в области меню. В итоге появится представление, в котором можно ввести подобные сведения о новом певце.

Активизация проверки достоверности по спецификации JSR-349

Сконфигурируем поддержку проверки достоверности по спецификации JSR-349 (Bean Validation — проверка достоверности компонентов Spring Beans) для операций создания и обновления сведений о певцах. Прежде всего наложим ограничения проверки достоверности на объект предметной области типа `Singer`. В рассматриваемом здесь примере эти ограничения накладываются только на свойства `firstName` и `lastName` данного объекта. Класс `Singer` был уже представлен в начале этой главы с аннотациями, накладывающими ограничения проверки достоверности, а в этом разделе поясняется их назначение. Ниже приведен фрагмент исходного кода этого класса с интересующими нас аннотированными полями.

```
package com.apress.prospring5.ch16.entities;

import org.hibernate.validator.constraints.NotBlank;
import javax.persistence.*;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
...

@Entity
@Table(name = "singer")
```

```
public class Singer implements Serializable {
  ...
  @NotBlank(message=
            "{validation.firstname.NotBlank.message}")
  @Size(min=2, max=60,
        message="{validation.firstname.Size.message}")
  @Column(name = "FIRST_NAME")
  private String firstName;

  @NotBlank(message=
            "{validation.lastname.NotBlank.message}")
  @Size(min=1, max=40,
        message="{validation.lastname.Size.message}")
  @Column(name = "LAST_NAME")
  private String lastName;
  ...
}
```

Ограничения накладываются на соответствующие поля. Обратите внимание на то, что для сообщения о результатах проверки достоверности ключ к сообщению указывается в фигурных скобках. Благодаря этому соответствующие сообщения извлекаются из комплекта ресурсов, а следовательно, обеспечивается поддержка интернационализации.

Чтобы активизировать проверку достоверности по спецификации JSR-349 во время привязки данных, достаточно применить аннотацию `@Valid` к соответствующему аргументу методов `create()` и `update()` из класса `SingerController`. В следующем фрагменте кода приведены сигнатуры этих методов:

```
package com.apress.prospring5.ch16.web;

@RequestMapping("/singers")
@Controller
public class SingerController {
  ...
  @RequestMapping(value = "/{id}", params = "form",
                  method = RequestMethod.POST)
  public String update(@Valid Singer singer,
                       BindingResult bindingResult, ...
  @RequestMapping(method = RequestMethod.POST)
  public String create(@Valid Singer singer,
                       BindingResult bindingResult, ...
  ...
}
```

Кроме того, требуется, чтобы для сообщения о результатах проверки достоверности по спецификации JSR-349 использовался тот же комплект ресурсов, что и для представлений. Для этого необходимо настроить средство проверки достоверности

в конфигурации сервлета диспетчера типа `DispatcherServlet`, реализуемой в классе `WebConfig`, как показано ниже.

```
package com.apress.prospring5.ch16.config;

import com.apress.prospring5.ch16.util.DateFormatter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.context.support
          .ReloadableResourceBundleMessageSource;
import org.springframework.validation.Validator;
import org.springframework.validation.beanvalidation
          .LocalValidatorFactoryBean;
import org.springframework.web.servlet.config.annotation.*;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages =
               {"com.apress.prospring5.ch16"})
public class WebConfig implements WebMvcConfigurer {
  @Bean
  public Validator validator() {
    final LocalValidatorFactoryBean validator =
          new LocalValidatorFactoryBean();
    validator.setValidationMessageSource(messageSource());
    return validator;
  }

  // <=> <mvc:annotation-driven validator="validator"/>
  @Override
  public Validator getValidator() {
    return validator();
  }

  @Bean
  ReloadableResourceBundleMessageSource messageSource() {
    ReloadableResourceBundleMessageSource messageSource =
              new ReloadableResourceBundleMessageSource();
    messageSource.setBasenames("WEB-INF/i18n/messages",
                               "WEB-INF/i18n/application");
    messageSource.setDefaultEncoding("UTF-8");
    messageSource.setFallbackToSystemLocale(false);
    return messageSource;
  }
  ...
}
```

Сначала в приведенном выше коде определяется компонент Spring Bean типа `LocalValidatorFactoryBean` под именем `validator` для поддержки проверки достоверности по спецификации JSR-349. Обратите внимание на то, что в свойстве `validationMessageSource` устанавливается ссылка на определяемый компонент `messageSource`, который предписывает средству проверки достоверности по спецификации JSR-349 искать сообщения по коду, возвращаемому из компонента `messageSource`. Затем в данном коде определяется метод `getValidator()` для возврата определяемого здесь компонента `validator`.

Вот, собственно, и все. А теперь можно протестировать проверку достоверности. Перейдите к представлению для ввода сведений о певцах и щелкните на кнопке **Save**. На возвращаемой в ответ странице появится сообщение об ошибке при проверке достоверности. Смените язык на китайский (HK) и сделайте то же самое. На этот раз сообщения будут выводиться на китайском языке.

На этом реализация представлений в основном завершена, за исключением представления для удаления сведений о певцах. Оставляем его реализацию вам в качестве упражнения. А далее перейдем к расширению функциональных возможностей пользовательского интерфейса рассматриваемого здесь веб-приложения.

Применение библиотек jQuery и jQuery UI

Несмотря на то что представления в рассматриваемом здесь веб-приложении для певцов функционируют вполне удовлетворительно, его пользовательский интерфейс несколько сыроват. Например, было бы неплохо предусмотреть для поля с датой рождения средство выбора даты, а не заставлять пользователя вводить строку даты вручную.

Чтобы предоставить пользователям веб-приложения более развитый функционально интерфейс, требующиеся для этого средства придется реализовать на языке JavaScript, если, конечно, не применять технологии RIA (Rich Internet Application — насыщенное интернет-приложение), которые требуют наличия в клиентском веб-браузере специальных исполняющих сред (например, для Adobe Flex требуется среда Flash, для JavaFX — JRE, а для Microsoft Silverlight — Silverlight).

Но разрабатывать пользовательские интерфейсы для веб-приложений исключительно на языке JavaScript непросто. Синтаксис языка JavaScript заметно отличается от синтаксиса Java, а кроме того, приходится решать вопросы кросс-браузерной совместимости. Тем не менее имеется немало библиотек JavaScript с открытым кодом, которые значительно упрощают дело. К их числу относятся библиотеки jQuery, jQuery UI и Dojo Toolkit.

В последующих разделах будет показано, как пользоваться библиотеками jQuery и jQuery UI для разработки более оперативно реагирующих и интерактивных пользовательских интерфейсов. Кроме того, здесь будут рассмотрены некоторые из наиболее употребительных модулей, подключаемых к библиотеке jQuery для таких специаль-

ных целей, как поддержка редактирования форматированного текста, а также представлены компоненты с сеточной компоновкой, применяемые для показа данных.

Введение в библиотеки jQuery и jQuery UI

jQuery (`http://jquery.org`) — это одна из библиотек JavaScript, наиболее широко применяемых при разработке пользовательских интерфейсов для веб-приложений. Библиотека jQuery обеспечивает всестороннюю поддержку основных функциональных возможностей, включая надежный синтаксис "селекторов" для выбора элементов модели DOM в документе, развитую модель событий и мощную поддержку технологии Ajax.

А библиотека jQuery UI (`http://jqueryui.com`), построенная на основании jQuery, предоставляет богатый набор виджетов (графических элементов) и визуальных эффектов. К ее основным средствам относятся виджеты для часто употребляемых компонентов пользовательского интерфейса (средств выбора даты и автозавершения вводимой информации, раскрывающихся списков и проч.), перетаскивания, визуальных эффектов и анимации, тематического оформления и многих других функциональных возможностей пользовательского интерфейса.

Имеются также многообразные подключаемые модули, созданные в сообществе разработчиков библиотеки jQuery для специальных целей. Два из них будут рассмотрены далее в этой главе. Мы лишь слегка коснемся здесь особенностей библиотеки jQuery, а за более подробными сведениями о ней обращайтесь к следующей литературе: *jQuery Recipes: A Problem-Solution* (B. M. Harwani, издательство Apress; 2010 г.), *Approach by and jQuery in Action* (Bear Bibeault and Yehuda Katz; издательство Manning, 2010 г.), а также *jQuery 2.0 для профессионалов* (Адам Фримен; ИД "Вильямс", 2014 г.).

Активизация библиотек jQuery и jQuery UI в представлении

Чтобы воспользоваться компонентами jQuery и jQuery UI в представлении, необходимо включить в рассматриваемый здесь проект обязательные файлы стилевых таблиц и исходного кода JavaScript. Если вы проработали материал раздела "Описание структуры проекта в Spring MVC" ранее в этой главе, то обязательные файлы уже должны быть скопированы в данный проект. Ниже перечислены основные файлы, которые должны быть включены в представление.

- Файл `/src/main/webapp/scripts/jquery-1.12.4.js`. Это базовая библиотека JavaScript из ядра jQuery. В рассматриваемом здесь примере веб-приложения применяется ее версия 1.12.4. Обратите внимание на то, что это полная исходная версия. В производственной среде должна использоваться уменьшенная версия библиотеки (т.е. файл `jquery-1.12.4.min.js`), которая оптимизирована и сжата с целью сократить время загрузки и улучшить производительность во время выполнения.

- Файл `/src/main/webapp/scripts/jquery-ui.min.js`. Это библиотека jQuery UI, которая упакована вместе со стилевой таблицей темы и может быть специально настроена и загружена со страницы jQuery UI Themeroller (`http://jqueryui.com/themeroller`). В рассматриваемом здесь примере веб-приложения применяется версия jQuery UI 1.12.1. Обратите внимание на то, что это уменьшенная версия JavaScript.
- Файл `/src/main/webapp/styles/custom-theme/jquery-ui.theme.min.css`. Это стилевая таблица для специальной темы, которая будет использоваться в jQuery UI для поддержки тематического оформления.

Перечисленные выше файлы должны быть включены только в шаблонную страницу (т.е. в файл `/layouts/default.jspx`). В следующем фрагменте кода разметки показано, как это делается:

```
<html xmlns:jsp="http://java.sun.com/JSP/Page" ..>
...
<head>
  <meta http-equiv="Content-Type" content="text/html;
        charset=UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=8" />

    <spring:theme code="styleSheet" var="app_css" />
    <spring:url value="/${app_css}" var="app_css_url" />
    <link rel="stylesheet" type="text/css"
        media="screen" href="${app_css_url}" />
    <spring:url value="/resources/scripts/jquery-1.12.4.js"
                 var="jquery_url" />
    <spring:url value="/resources/scripts/jquery-ui.min.js"
                 var="jquery_ui_url" />
    <spring:url value="/resources/styles/custom-theme
                       /jquery-ui.theme.min.css"
                 var="jquery_ui_theme_css" />
    <link rel="stylesheet" type="text/css" media="screen"
         href="${jquery_ui_theme_css}" />
    <script src="${jquery_url}" type="text/javascript">
    <jsp:text/></script>
    <script src="${jquery_ui_url}" type="text/javascript">
    <jsp:text/></script>
    ...
</head>
...
</html>
```

Сначала URL для файлов определяются и сохраняются в переменных с помощью дескриптора разметки `<spring:url>`. Затем в разделе `<head>` вводится ссылка на файлы стилевых таблиц CSS и сценариев JavaScript. Обратите внимание на применение дескриптора разметки `<jsp:text/>` в дескрипторе `<script>`. Это объясняется

тем, что дескрипторы, не имеющие тела, будут автоматически сворачиваться в JSPX. Таким образом, дескриптор разметки `<script ...></script>` из данного файла превратится в дескриптор `<script .../>` непосредственно в браузере, что приведет к неопределенному поведению страницы. Добавление дескриптора разметки `<jsp:text/>` гарантирует, что дескриптор `<script>` не будет воспроизведен на странице, поскольку это позволит избежать неожиданных осложнений.

Благодаря включению упомянутых выше файлов сценариев можно дополнить представление некоторыми любопытными особенностями. В частности, улучшим немного внешний вид экранных кнопок в представлении для редактирования сведений о певцах и дополним поле ввода даты рождения компонентом выбора даты. Ниже приведены те изменения, которые необходимо внести в представление из файла `/views/singers/edit.jspx`.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<div xmlns:jsp="http://java.sun.com/JSP/Page"
     xmlns:c="http://java.sun.com/jsp/jstl/core"
     xmlns:spring="http://www.springframework.org/tags"
     xmlns:form="http://www.springframework.org/tags/form"
     version="2.0">

    <script type="text/javascript">
      $(function() {
        $('#birthDate').datepicker({
          dateFormat: 'yy-mm-dd',
          changeYear: true
        });
      });
    </script>
    ...
    <form:form modelAttribute="singer"
          id="singerUpdateForm" method="post">
    ...
    <button type="submit" class="ui-button ui-widget
       ui-state-default ui-corner-all ui-button-text-only">
       <span class="ui-button-text">Save</span>
    </button>
    <button type="reset" class="ui-button ui-widget
       ui-state-default ui-corner-all ui-button-text-only">
       <span class="ui-button-text">Reset</span>
    </button>
    </form:form>
      </div>
</div>
```

Синтаксис `$(function(){})` предписывает библиотеке jQuery запускать сценарий, когда документ готов. В теле функции, определяемой с помощью этого синтак-

сиса, поле ввода даты рождения (с идентификатором `birthDate`) декорируется посредством функции `datepicker()` из библиотеки jQuery UI. Затем экранные кнопки дополняются различными классами стилей. Снова собрав и развернув данный проект, вы обнаружите новый стиль оформления экранных кнопок, а щелкнув на поле ввода даты рождения — компонент выбора даты.

Редактирование форматированного текста средствами CKEditor

Для определения поля с описанием сведений о певце используется дескриптор разметки `<form:textarea>` из модуля Spring MVC, поддерживающий многострочный ввод данных. Допустим, требуется разрешить редактирование форматированного текста, что является весьма распространенным требованием для ввода длинных текстов вроде пользовательских комментариев.

Для поддержки такой возможности в рассматриваемом здесь примере веб-приложения будет применяться библиотека CKEditor (`http://ckeditor.com`), предоставляющая типичный компонент JavaScript для редактирования форматированного текста и поддерживающая интеграцию с библиотекой jQuery UI. Ее файлы находятся в папке `/src/main/webapp/ckeditor` исходного кода данного примера.

Сначала необходимо включить обязательные файлы исходного кода JavaScript в шаблонную страницу (из файла `default.jspx`). Ниже приведен фрагмент кода, который следует ввести на данной странице.

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:c="http://java.sun.com/jsp/jstl/core"
      xmlns:fn="http://java.sun.com/jsp/jstl/functions"
      xmlns:tiles="http://tiles.apache.org/tags-tiles"
      xmlns:spring="http://www.springframework.org/tags">

    <head>
    <!-- Библиотека CKEditor -->
    <spring:url value="/resources/ckeditor/ckeditor.js"
                var="ckeditor_url" />
    <spring:url value="/resources/ckeditor/adapters
                       /jquery.js"
                var="ckeditor_jquery_url" />
    <script type="text/javascript" src="${ckeditor_url}">
    <jsp:text/></script>
    <script type="text/javascript"
            src="${ckeditor_jquery_url}">
    <jsp:text/></script>
    ...
    </head>
    ...
</html>
```

В приведенный выше фрагмент кода разметки включены два сценария: сценарий ядра CKEditor и адаптер jQuery. Следующий шаг состоит в том, чтобы ввести CKEditor в представление для редактирования сведений о певцах. Ниже приведены изменения, которые следует внести в код разметки страницы из файла `edit.jspx`.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<div xmlns:jsp="http://java.sun.com/JSP/Page"
     xmlns:c="http://java.sun.com/jsp/jstl/core"
     xmlns:spring="http://www.springframework.org/tags"
     xmlns:form="http://www.springframework.org/tags/form"
        version="2.0">

    <script type="text/javascript">
      $(function() {
        $("#singerDescription").ckeditor({
           toolbar : 'Basic',
           uiColor : '#CCCCCC'
        });
      });
    </script>
    ...
</div>
```

Поле для ввода сведений о певце декорируется средствами CKEditor, когда документ готов. Снова развернув данный проект и перейдя на страницу для ввода сведений о певцах, вы увидите, что в данном поле стала доступной поддержка редактирования форматированного текста. За подробными сведениями о применении и конфигурировании CKEditor обращайтесь к соответствующей документации, оперативно доступной по адресу `https://docs-old.ckeditor.com/Main_Page`.

Применение jqGrid для построения сетки данных с разбиением на страницы

Текущее представление списка певцов вполне работоспособно только в том случае, если в системе существует относительно небольшое число записей о певцах. Но с увеличением количества подобных записей до нескольких тысяч и больше возникают осложнения, связанные с производительностью.

Общее решение предусматривает реализацию компонента сетки данных с поддержкой разбиения на страницы, где пользователь просматривает только определенное количество записей. Благодаря этому уменьшается объем данных, которыми обмениваются браузер и веб-контейнер. В этом разделе будет продемонстрирована реализация сетки данных с помощью jqGrid (`www.trirand.com/blog`) — весьма распространенного компонента сетки данных, построенного на JavaScript. В рассматриваемом здесь примере веб-приложения применяется версия 4.6.0 данного компонента.

Кроме того, в данном примере будет использована встроенная в jqGrid поддержка разбиения на страницы средствами Ajax, которая инициирует запрос типа `XMLHttpRequest` для каждой страницы и воспринимает формат JSON для данных на странице.

В последующих разделах будет показано, как реализовать поддержку разбиения на страницы на стороне сервера и клиента. Сначала будет рассмотрена реализация компонента jqGrid в представлении списка певцов, а затем — реализация разбиения на страницы на стороне сервера благодаря поддержке такого разбиения в модуле Spring Data Commons.

Активизация jqGrid в представлении списка певцов

Чтобы сделать доступным компонент jqGrid в представлениях, необходимо включить требующиеся файлы исходного кода JavaScript и стилевых таблиц в разметку шаблонной страницы (из файла `default.jspx`), как показано ниже.

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
...
    <head>
    ...
    <!-- Компонент jqGrid -->
      <spring:url value="/resources/jqgrid/css
                         /ui.jqgrid.css"
                  var="jqgrid_css" />
      <spring:url value="/resources/jqgrid/js/i18n
                         /grid.locale-en.js"
                  var="jqgrid_locale_url" />
      <spring:url value="/resources/jqgrid/js
                         /jquery.jqGrid.min.js"
                  var="jqgrid_url" />
      <link rel="stylesheet" type="text/css" media="screen"
           href="${jqgrid_css}" />
      <script type="text/javascript" src="${jqgrid_locale_url}">
      <jsp:text/></script>
      <script type="text/javascript" src="${jqgrid_url}">
      <jsp:text/></script>
      ...
    </head>
    ...
</html>
```

Прежде всего в приведенном выше коде разметки загружается нужный CSS-файл, затем два файла исходного кода JavaScript. Первый из них содержит сценарий региональных настроек (в данном случае на английском языке), а второй — базовую библиотеку jqGrid (из файла `jquery.jqGrid.min.js`). Следующий шаг состоит в том, чтобы изменить представление списка певцов (из файла `list.jspx`) для применения jqGrid. Соответственно измененный код разметки страницы приведен ниже.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
     xmlns:c="http://java.sun.com/jsp/jstl/core"
     xmlns:spring="http://www.springframework.org/tags"
     version="2.0">
    <jsp:directive.page
         contentType="text/html;charset=UTF-8"/>
    <jsp:output omit-xml-declaration="yes"/>

    <spring:message code="label_singer_list"
                    var="labelSingerList"/>
    <spring:message code="label_singer_first_name"
                    var="labelSingerFirstName"/>
    <spring:message code="label_singer_last_name"
                    var="labelSingerLastName"/>
    <spring:message code="label_singer_birth_date"
                    var="labelSingerBirthDate"/>
    <spring:url value="/singers/" var="showSingerUrl"/>

    <script type="text/javascript">
      $(function() {
         $("#list").jqGrid({
           url:'${showSingerUrl}/listgrid',
           datatype: 'json',
           mtype: 'GET',

          colNames:['${labelSingerFirstName}',
                    '${labelSingerLastName}',
                    '${labelSingerBirthDate}'],
          colModel :[
             {name:'firstName', index:'firstName',
              width:150},
             {name:'lastName', index:'lastName',
              width:100},
             {name:'birthDateString', index:'birthDate',
              width:100}
          ],
          jsonReader : {
              root:"singerData",
              page: "currentPage",
              total: "totalPages",
              records: "totalRecords",
              repeatitems: false,
              id: "id"
          },
          pager: '#pager', rowNum:10, rowList:[10,20,30],
          sortname: 'firstName',
          sortorder: 'asc',
          viewrecords: true,
```

```
            gridview: true,
            height: 250,
            width: 500,
            caption: '${labelSingerList}',
                   onSelectRow: function(id) {
                          document.location.href =
                          "${showSingerUrl}/" + id;
            }
          });
        });

    </script>

    <c:if test="${not empty message}">
    <div id="message" class="${message.type}">
       ${message.message}
    </div>
    </c:if>

    <h2>${labelSingerList}</h2>

    <div>
      <table id="list"><tr><td/></tr></table>
    </div>
    <div id="pager"></div>
</div>
```

Для отображения данных сетки в приведенном выше коде разметки объявляется дескриптор `<table>` с идентификатором `list`. После этого дескриптора следует дескриптор `<div>` с идентификатором `pager`, обозначающим ту часть компонента jqGrid, которая отвечает за разбиение на страницы. Если документ готов, то в сценарии JavaScript компоненту jqGrid предписывается оформить таблицу с идентификатором `list` в виде сетки, а также предоставить сведения о конфигурации. Ниже перечислены некоторые важные особенности данного сценария.

- В атрибуте `url` указывается ссылка для отправки запроса типа `XMLHttpRequest`, по которому получаются данные для текущей страницы.
- В атрибуте `datatype` указывается формат данных (в данном случае формат JSON). В компоненте jqGrid поддерживается также формат XML.
- В атрибуте `mtype` указывается метод доступа по сетевому протоколу HTTP (в данном случае `GET`).
- В атрибуте `colNames` задается заголовок столбца для отображения данных в виде сетки, тогда как в атрибуте `colModel` — подробные сведения для каждого столбца данных.
- В атрибуте `jsonReader` задается формат данных JSON, который будет возвращаться сервером.

- В атрибуте `pager` активизируется поддержка разбиения на страницы.
- В атрибуте `onSelectRow` определяется действие, которое должно выполняться при выборе строки. В данном случае пользователь направляется к представлению, отображающему сведения о певце с заданным идентификатором.

За подробным описанием конфигурирования и применения компонента jqGrid обращайтесь к документации, оперативно доступной по адресу `www.trirand.com/jqgridwiki/doku.php?id=wiki:jqgriddocs`.

Активизация разбиения на страницы на стороне сервера

На стороне сервера для реализации разбиения на страницы придется предпринять ряд шагов. Прежде всего необходимо воспользоваться поддержкой разбиения на страницы в модуле Spring Data Commons. Для ее активизации достаточно внести коррективы в интерфейс `ContactRepository`, чтобы он расширял интерфейс `PagingAndSortingRepository<T, ID extends Serializable>` вместо интерфейса `CrudRepository<T, ID extends Serializable>`. Ниже приведен соответственно измененный вариант данного интерфейса.

```
package com.apress.prospring5.ch16.repos;

import com.apress.prospring5.ch16.entities.Singer;
import org.springframework.data.repository
          .PagingAndSortingRepository;

public interface SingerRepository extends
PagingAndSortingRepository<Singer, Long> {
}
```

Далее в интерфейс `SingerService` необходимо ввести новый метод для поддержки постраничного извлечения данных. Измененный вариант данного интерфейса приведен ниже.

```
package com.apress.prospring5.ch16.services;

import java.util.List;
import com.apress.prospring5.ch16.entities.Singer;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;

public interface SingerService {
  List<Singer> findAll();
  Singer findById(Long id);
  Singer save(Singer singer);
  Page<Singer> findAllByPage(Pageable pageable);
}
```

Как следует из приведенного выше кода, в интерфейс `SingerService` был введен метод `findAllByPage()`, принимающий экземпляр интерфейса `Pageable` в качестве аргумента. В приведенном ниже коде представлена реализация метода `findAllByPage()` в классе `SingerServiceImpl`. Этот метод возвращает экземпляр интерфейса `Page<T>` (который относится к модулю Spring Data Commons и входит в состав пакета `org.springframework.data.domain`). Как и следовало ожидать от столь простого примера, в этом служебном методе просто вызывается метод `findAll()`, предоставляемый в интерфейсе `PagingAndSortingRepository<T,ID extends Serializable>` для поиска записей обо всех певцах в информационном хранилище.

```
package com.apress.prospring5.ch16.services;

import java.util.List;
import com.apress.prospring5.ch16.repos.SingerRepository;
import com.apress.prospring5.ch16.entities.Singer;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation
          .Transactional;
import com.google.common.collect.Lists;

@Repository
@Transactional
@Service("singerService")
public class SingerServiceImpl implements SingerService {
  private SingerRepository singerRepository;
  ...

  @Autowired
  public void setSingerRepository(
              SingerRepository singerRepository) {
    this.singerRepository = singerRepository;
  }

  @Override
  @Transactional(readOnly=true)
  public Page<Singer> findAllByPage(Pageable pageable) {
    return singerRepository.findAll(pageable);
  }
}
```

Следующий, самый сложный шаг состоит в том, чтобы реализовать в классе `SingerController` метод, получающий из компонента jqGrid Ajax-запрос на страничные данные. Реализация этого метода приведена ниже.

```
package com.apress.prospring5.ch16.web;
....

@RequestMapping("/singers")
@Controller
public class SingerController {
  private final Logger logger =
          LoggerFactory.getLogger(SingerController.class);
  private SingerService singerService;

  @ResponseBody
  @RequestMapping(value = "/listgrid",
                  method = RequestMethod.GET,
                  produces="application/json")
  public SingerGrid listGrid(@RequestParam(value = "page",
         required = false) Integer page,
         @RequestParam(value = "rows", required = false)
                       Integer rows,
         @RequestParam(value = "sidx", required = false)
                       String sortBy,
         @RequestParam(value = "sord", required = false)
                       String order) {
    logger.info("Listing singers for grid with page: {},
                rows: {}", page, rows);
    logger.info("Listing singers for grid with sort: {},
                order: {}", sortBy, order);

    // обработать по порядку
    Sort sort = null;
    String orderBy = sortBy;
    if (orderBy != null &&
        orderBy.equals("birthDateString"))
    orderBy = "birthDate";
    if (orderBy != null && order != null) {
      if (order.equals("desc")) {
        sort = new Sort(Sort.Direction.DESC, orderBy);
      } else
        sort = new Sort(Sort.Direction.ASC, orderBy);
    }

    // составить запрос текущей страницы
    // Примечание: номер страницы в Spring Data JPA
    // начинается с 0, тогда как в jqGrid - с 1
    PageRequest pageRequest = null;
    if (sort != null) {
```

```
      pageRequest = PageRequest.of(page - 1, rows, sort);
    } else {
      pageRequest = PageRequest.of(page - 1, rows);
    }

    Page<Singer> singerPage =
                singerService.findAllByPage(pageRequest);

    // сформировать данные сетки, чтобы возвратить их
    // в формате JSON
    SingerGrid singerGrid = new SingerGrid();
    singerGrid.setCurrentPage(singerPage.getNumber() + 1);
    singerGrid.setTotalPages(singerPage.getTotalPages());
    singerGrid.setTotalRecords(
               singerPage.getTotalElements());
    singerGrid.setSingerData(Lists.newArrayList(
                              singerPage.iterator()));
    return singerGrid;
  }

  @Autowired
  public void setSingerService(
              SingerService singerService) {
    this.singerService = singerService;
  }
  ...
}
```

В методе `listGrid()` обрабатывается Ajax-запрос, читаются параметры (номер страницы, количество записей, приходящихся на каждую страницу, поле, по которому производится сортировка, порядок сортировки) из запроса (имена параметров в данном примере кода соответствуют принятым в jqGrid по умолчанию), конструируется экземпляр класса `PageRequest`, реализующего интерфейс `Pageable`, а затем вызывается метод `SingerService.findAllByPage()` для получения страничных данных. После этого получается экземпляр класса `SingerGrid`, который возвращается компоненту jqGrid в формате JSON. Исходный код класса `SingerGrid` приведен ниже.

```
package com.apress.prospring5.ch16.util;

import com.apress.prospring5.ch16.entities.Singer;
import java.util.List;

public class SingerGrid {
  private int totalPages;
  private int currentPage;
  private long totalRecords;
  private List<Singer> singerData;
```

```
  public int getTotalPages() {
    return totalPages;
  }

  public void setTotalPages(int totalPages) {
    this.totalPages = totalPages;
  }

  // другие методы получения и установки
  ...
}
```

Теперь все готово для тестирования нового представления списка певцов. Постройте и разверните данный проект снова, а затем загрузите представление списка певцов. В итоге вы должны увидеть расширенное представление с сеточной компоновкой списка певцов.

Можете поэкспериментировать с сеточной компоновкой, перемещаясь по страницам, устанавливая разное количество записей на странице, изменяя порядок сортировки щелчком кнопкой мыши на заголовках столбцов и т.д. Здесь также поддерживается интернационализация, поэтому при желании можете отобразить сетку с метками на китайском языке.

В компоненте jqGrid поддерживается и фильтрация данных. Например, данные можно отфильтровать по имени `John` или по дате рождения в заданных пределах.

Организация выгрузки файлов

В записи о певце имеется поле типа BLOB, предназначенное для хранения фотографии, которая может быть выгружена из клиента. В этом разделе будет показано, как реализовать выгрузку файлов в модуле Spring MVC.

Долгое время в стандартной спецификации сервлетов не поддерживалась выгрузка файлов. В итоге для этой цели в модуле Spring MVC применялись другие библиотеки, и чаще всего — библиотека Apache Commons FileUpload (`http://commons.apache.org/proper/commons-fileupload/`). В модуле Spring MVC имеется встроенная поддержка библиотеки Apache Commons FileUpload. Но в версии Servlet 3.0 выгрузка файлов стала встроенной функциональной возможностью веб-контейнера. Версия Servlet 3.0 поддерживается как в контейнере сервлетов Tomcat 7, так и в каркасе Spring 3.1. В последующих разделах будет показано, как реализовать выгрузку файлов средствами Spring MVC и Servlet 3.0.

Конфигурирование поддержки выгрузки файлов

Прежде всего в конфигурационный класс Java, где определяется все, что требуется для создания сервлета диспетчера типа `DispatcherServlet`, необходимо ввести компонент Spring Bean типа `StandardServletMultipartResolver`. Это стандарт-

ная реализация интерфейса `MultipartResolver` на основе прикладного интерфейса `javax.servlet.http.Part` из компонента Servlet 3.0. В следующем фрагменте кода приведено объявление данного компонент Spring Bean, которое требуется ввести в конфигурационный класс `WebConfig`.

Далее необходимо активизировать многосторонний синтаксический анализ в среде Servlet 3.0. Это означает, что в реализацию класса `WebInitializer` придется внести некоторые коррективы. В абстрактном классе `AbstractDispatcherServletInitializer`, который расширяется классом `AbstractAnnotationConfigDispatcherServletInitializer`, определен метод `customizeRegistration()`. Именно этот метод и должен быть реализован для регистрации экземпляра класса `javax.servlet.MultipartConfigElement`. Ниже приведена усовершенствованная версия класса `WebInitializer`.

```
package com.apress.prospring5.ch16.init;

import com.apress.prospring5.ch16.config.DataServiceConfig;
import com.apress.prospring5.ch16.config.SecurityConfig;
import com.apress.prospring5.ch16.config.WebConfig;
import org.springframework.web.filter
          .CharacterEncodingFilter;
import org.springframework.web.filter
          .HiddenHttpMethodFilter;
import org.springframework.web.servlet.support.
     AbstractAnnotationConfigDispatcherServletInitializer;

import javax.servlet.Filter;
import javax.servlet.MultipartConfigElement;
import javax.servlet.ServletRegistration;

public class WebInitializer extends
  AbstractAnnotationConfigDispatcherServletInitializer {

  @Override
  protected Class<?>[] getRootConfigClasses() {
    return new Class<?>[] {
      SecurityConfig.class, DataServiceConfig.class
    };
  }

  @Override
  protected Class<?>[] getServletConfigClasses() {
    return new Class<?>[] {
      WebConfig.class
    };
  }

  @Override
  protected String[] getServletMappings() {
```

```
    return new String[]{"/"};
  }

  @Override
  protected Filter[] getServletFilters() {
    CharacterEncodingFilter cef =
              new CharacterEncodingFilter();
    cef.setEncoding("UTF-8");
    cef.setForceEncoding(true);
    return new Filter[]{new HiddenHttpMethodFilter(), cef};
  }

  // <=> <multipart-config>

  protected void customizeRegistration(
            ServletRegistration.Dynamic registration) {
    registration.setMultipartConfig(
                getMultipartConfigElement());
  }

  @Bean
  private MultipartConfigElement
          getMultipartConfigElement() {
    return new MultipartConfigElement(
               null, 5000000, 5000000, 0);
  }
}
```

В первом параметре компонента Spring Bean типа `MultipartConfigElement` задается место для временного хранения файлов, а во втором параметре — максимально допустимый размер выгружаемого файла (в данном случае 5 Мбайт). В третьем параметре указывается длина запроса, которая в данном случае также составляет 5 Мбайт. И, наконец, в четвертом параметре задается порог, при превышении которого файлы должны записываться на диск.

Видоизменение представлений для поддержки выгрузки файлов

Для поддержки выгрузки файлов следует видоизменить два представления. Первым из них является представление для редактирования (из файла `edit.jspx`), где необходимо поддерживать загрузку фотографий певцов, а вторым — представление для показа (из файла `show.jspx`), где необходимо воспроизводить эти фотографии.

В следующем фрагменте кода разметки JSPX-страницы приведены изменения, которые требуется внести в представление редактирования из файла `edit.jspx`:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
...
```

```
   <form:form modelAttribute="singer"
         id="singerUpdateForm" method="post"
         enctype="multipart/form-data">
   ...
   <form:label path="description">
         ${labelSingerDescription}
   </form:label>
   <form:textarea cols="60" rows="8" path="description"
         id="singerDescription"/>
   <div>
   <form:errors path="description" cssClass="error" />
</div>
   <p/>
     <label for="file">
          ${labelSingerPhoto}
     </label>
     <input name="file" type="file"/>
   <p/>
   ...
   </form:form>
</div>
```

Сначала в приведенном выше фрагменте кода разметки активизируется многосторонняя поддержка выгрузки файлов с помощью атрибута `enctype` в дескрипторе `<form:form>`, а затем в форму вводится поле для указания выгружаемого файла. Необходимо также видоизменить представление для показа сведений о певцах, чтобы воспроизводить в нем фотографии певцов. Изменения, которые должны быть внесены в это представление (из файла `show.jspx`), приведены ниже.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
...
   <spring:message code="label_singer_photo"
           var="labelSingerPhoto"/>
   <spring:url value="/singers/photo" var="singerPhotoUrl"/>
   ...
   <tr>
     <td>${labelSingerDescription}</td>
     <td>${singer.description}</td>
   </tr>
   <tr>
     <td>${labelSingerPhoto}</td>
     <td><img src="${singerPhotoUrl}/${singer.id}"></img>
     </td>
   </tr>
   ...
</div>
```

В приведенном выше фрагменте кода разметки в таблицу введена новая строка для воспроизведения фотографии. В этой строке указывается URL ресурса для загрузки фотографии.

Видоизменение контроллера для поддержки выгрузки файлов

И последний шаг — видоизменение контроллера. Для этого необходимо внести два изменения. Первое изменение вносится в метод `create()`, чтобы воспринимать выгружаемый файл в качестве обязательного параметра. А второе вносится с целью реализовать новый метод для выгрузки фотографии по указанному идентификатору певца. Соответствующие исправления в исходном коде класса `SingerController` приведены ниже.

```
package com.apress.prospring5.ch16.web;
...

@RequestMapping("/singers")
@Controller
public class SingerController {
  private final Logger logger =
          LoggerFactory.getLogger(SingerController.class);
  private SingerService singerService;

  @RequestMapping(method = RequestMethod.POST)
  public String create(@Valid Singer singer,
         BindingResult bindingResult, Model uiModel,
         HttpServletRequest httpServletRequest,
         RedirectAttributes redirectAttributes,
         Locale locale, @RequestParam(value="file",
                        required=false) Part file) {
    logger.info("Creating singer");
    if (bindingResult.hasErrors()) {
      uiModel.addAttribute("message", new Message("error",
              messageSource.getMessage("singer_save_fail",
                            new Object[]{}, locale)));
      uiModel.addAttribute("singer", singer);
      return "singers/create";
    }
    uiModel.asMap().clear();
    redirectAttributes.addFlashAttribute("message",
            new Message("success",
            messageSource.getMessage("singer_save_success",
                          new Object[]{}, locale)));
    logger.info("Singer id: " + singer.getId());
    // обработать выгружаемый файл
    if(file != null) {
      logger.info("File name: " + file.getName());
```

```
      logger.info("File size: " + file.getSize());
      logger.info("File content type: "
                + file.getContentType());
      byte[] fileContent = null;
      try {
        InputStream inputStream = file.getInputStream();
        if (inputStream == null)
          logger.info("File inputstream is null");
        fileContent = IOUtils.toByteArray(inputStream);
        singer.setPhoto(fileContent);
      } catch (IOException ex) {
        logger.error("Error saving uploaded file");
      }
      singer.setPhoto(fileContent);
    }
    singerService.save(singer);
    return "redirect:/singers/";
  }

  @RequestMapping(value = "/photo/{id}",
                  method = RequestMethod.GET)
  @ResponseBody
  public byte[] downloadPhoto
              (@PathVariable("id") Long id) {
    Singer singer = singerService.findById(id);
    if (singer.getPhoto() != null) {
      logger.info("Downloading photo for id: {} with "
                + "size: {}", singer.getId(),
                  singer.getPhoto().length);
    }
    return singer.getPhoto();
  }
  ...
}
```

В первую очередь в объявление метода `create()` добавлен новый параметр запроса относящийся к типу интерфейса `javax.servlet.http.Part`, который предоставляется модулем Spring MVC, исходя из запрашиваемого выгружаемого содержимого. И тогда данный метод получит содержимое, сохраненное в свойстве `photo` объекта типа `Singer`.

Затем в приведенном выше фрагменте кода объявляется новый метод `downloadPhoto()`, выгружающий указанный файл. Этот метод извлекает содержимое поля `photo` из объекта типа `Singer` и направляет его непосредственно в поток вывода ответа, что соответствует дескриптору разметки `<img>` в разметке представления показа.

Чтобы протестировать функции выгрузки файлов, снова разверните веб-приложение из рассматриваемого здесь примера и введите сведения о новом певце вместе с фотографией. В итоге вы сможете увидеть эту фотографию в представлении для

показа сведений о певцах. Необходимо также внести изменения в функцию редактирования для смены фотографий певцов, но мы оставляем это вам в качестве упражнения.

Защита веб-приложения средствами Spring Security

Допустим, теперь требуется защитить рассматриваемое здесь веб-приложение для певцов. Только тем пользователям, которые вошли в приложение с достоверным идентификатором, должно быть разрешено вводить сведения о новых певцах или обновлять сведения об уже существующих певцах. А другие пользователи, называемые *анонимными*, могут только просматривать сведения о певцах.

Для защиты приложений, построенных на основе Spring, лучше всего подходит каркас Spring Security. Несмотря на то что каркас Spring Security применяется в основном на уровне представления, он может оказать помощь в защите и всех остальных уровней приложения, включая уровень обслуживания. В последующих разделах будет продемонстрировано применение каркаса Spring Security для защиты рассматриваемого здесь примера веб-приложения для певцов.

Каркас и одноименный проект Spring Security были представлены в главе 12, когда демонстрировалась защита веб-службы REST. Для защиты веб-приложений имеются самые разные возможности. Так, для защиты только определенных элементов разметки веб-страниц в Spring имеется отдельная библиотека дескрипторов, которая входит в состав модуля `spring-security-web`.

Конфигурирование защиты в Spring Security

Как упоминалось в предыдущем разделе, для конфигурирования веб-приложения, рассматриваемого в этой главе, применяются только аннотации, объявляемые в конфигурационном классе Java. До версии Spring 3.*x* защита веб-приложений в Spring активизировалась путем настройки фильтра в дескрипторе разворачивания веб-приложений (из файла `web.xml`). Этот фильтр назывался `springSecurityFilterChain` и применялся к любым запросам, кроме запроса статических компонентов. А в версии Spring 4.0 был внедрен класс `AbstractSecurityWebApplicationInitializer`, который может быть расширен для активизации Spring Security, как показано ниже.

```
package com.apress.prospring5.ch16.init;

import org.springframework.security.web.context
          .AbstractSecurityWebApplicationInitializer;

public class SecurityWebApplicationInitializer
       extends AbstractSecurityWebApplicationInitializer {
}
```

Предоставив пустой класс, расширяющий класс `AbstractSecurityWebApplicationInitializer`, можно, по существу, предписать Spring активизировать заместитель типа `DelegatingFilterProxy`, чтобы воспользоваться компонентом `springSecurityFilterChain`, прежде чем будет зарегистрирован любой другой фильтр, реализующий интерфейс `javax.servlet.Filter`. Помимо этого, необходимо настроить контекст Spring Security, используя конфигурационный класс, который должен быть введен в конфигурацию корневого контекста типа `WebApplicationContext`. Для этого ниже приведен исходный код конфигурационного класса `SecurityConfig`.

```
package com.apress.prospring5.ch16.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation
          .authentication.builders
          .AuthenticationManagerBuilder;
import org.springframework.security.config.annotation
          .method.configuration.EnableGlobalMethodSecurity;
import org.springframework.security.config.annotation.web
          .builders.HttpSecurity;
import org.springframework.security.config.annotation.web
          .configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web
          .configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.web.csrf
          .CsrfTokenRepository;
import org.springframework.security.web.csrf
          .HttpSessionCsrfTokenRepository;

@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig
       extends WebSecurityConfigurerAdapter {

  @Autowired
  public void configureGlobal(
              AuthenticationManagerBuilder auth) {
    try {
      auth.inMemoryAuthentication().withUser("user")
          .password("user").roles("USER");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  @Override
```

```
  protected void configure(HttpSecurity http)
                throws Exception {
    http
    .authorizeRequests()
    .antMatchers("/*").permitAll()
    .and()
    .formLogin()
    .usernameParameter("username")
    .passwordParameter("password")
    .loginProcessingUrl("/login")
    .loginPage("/singers")
    .failureUrl("/security/loginfail")
    .defaultSuccessUrl("/singers")
    .permitAll()
    .and()
    .logout()
    .logoutUrl("/logout")
    .logoutSuccessUrl("/singers")
    .and()
    .csrf().disable();
  }
}
```

В методе `configure(HttpSecurity http)` определяется конфигурация защиты для обработки HTTP-запросов. В цепочке вызовов `.antMatchers("/*").permitAll()` указывается на то, что всем пользователям разрешается войти в данное веб-приложение. Далее будет показано, как защитить функцию входа в веб-приложение, скрыв параметры редактирования в представлении с помощью доступной в Spring Security библиотеки дескрипторов и защиты отдельных методов контроллера. Затем в вызове метода `.formLogin()` определяется поддержка регистрации через заполняемую форму. А во всех последующих вызовах вплоть до метода `.and()` конфигурируется форма регистрации. Как пояснялось ранее при обсуждении компоновки рассматриваемого здесь веб-приложения, форма регистрации будет отображаться слева. И, наконец, через вызов метода `.logout()` предоставляется ссылка на выход из данного веб-приложения.

В версии Spring Security 4 была внедрена возможность пользоваться ключами CSFR в формах Spring, чтобы предотвратить атаки типа межсайтовой подделки запросов (CSFR).[5] Ради простоты примера в данном веб-приложении применение ключей CSFR запрещается через вызов `.csrf().disable()`. По умолчанию конфигурация без элементов защиты от атак типа межсайтовой подделки запросов считается

[5] Этот тип атаки состоит в подделке существующего сеанса связи с целью выполнить несанкционированные команды в веб-приложении. Подробнее об этом типе атаки можно узнать по адресу `https://en.wikipedia.org/wiki/Cross-site_request_forgery`.

недостоверной, и поэтому любой запрос на регистрацию будет направлен на страницу с кодом состояния ошибки 403, где сообщается следующее[6]:

```
Invalid CSRF Token 'null' was found on the request parameter
'_csrf' or header 'X-CSRF-TOKEN'.
```

В методе configureGlobal(AuthenticationManagerBuilder auth) определяется механизм аутентификации. В данной упрощенной конфигурации жестко закодирован единственный пользователь с присвоенной ролью USER. Но в производственной среде пользователь должен проходить аутентификацию через базу данных, протокол LDAP или механизм SSO.

На заметку До версии Spring 3, в URL для регистрации по умолчанию указывается значение **/j_spring_security_check**, а также стандартные имена ключей аутентификации **j_username** и **j_password**. А начиная с версии Spring 4 в URL для регистрации по умолчанию указывается значение **/login**, а также стандартные имена ключей аутентификации **username** и **password**.

На заметку В приведенной выше конфигурации имя пользователя, пароль и URL для регистрации задаются явным образом. Но если в представлении употребляются стандартные имена, то из конфигурации необходимо исключить следующую часть:

```
.usernameParameter("username")
.passwordParameter("password")
.loginProcessingUrl("/login")|
```

Несмотря на то что содержимое класса WebInitializer было описано выше, конфигурация данного веб-приложения еще раз представлена ниже, чтобы подчеркнуть, где именно применяется конфигурационный класс SecurityConfig.

```
package com.apress.prospring5.ch16.init;
...
public class WebInitializer extends
     AbstractAnnotationConfigDispatcherServletInitializer {

  @Override
  protected Class<?>[] getRootConfigClasses() {
    return new Class<?>[] {
    SecurityConfig.class, DataServiceConfig.class
  };

  @Override
  protected Class<?>[] getServletConfigClasses() {
    return new Class<?>[] {
```

---

[6] В параметре запроса '_csrf' или заголовке 'X-CSRF-TOKEN' обнаружен недостоверный "пустой" ключ CSRF

```
      WebConfig.class
    };
  }
  ...
}
```

На момент написания данной книги версии Spring 5 и Spring Boot 2 все еще находились на стадии построения, и поэтому после обновления соответствующих библиотек представленные здесь примеры кода могут перестать нормально работать. Что же касается защиты веб-приложений с помощью модуля Spring Security, то версия Security 5.0.0.RC1 вышла со 150 дополнительными исправлениями, многие из которых имеют непосредственное отношение к защите с помощью паролей. И в связи с этим потребуются соответствующие реализации интерфейса `PasswordEncoder`. До этой версии пароли хранились в исходном виде без всякого шифрования при конфигурировании аутентификации, осуществляемой в оперативной памяти. Такое поведение может поддерживаться и теперь, особенно в учебных приложениях, где основное внимание уделяется другим вопросам. В двух приведенных ниже примера показано, как это делается.

```
@Autowired
public void
       configureGlobal(AuthenticationManagerBuilder auth)
       throws Exception {
  auth
    .inMemoryAuthentication()
    .withUser("user").password("{noop}user").roles("USER");
}

// или

import org.springframework.security.crypto
          .password.NoOpPasswordEncoder;
...
@Autowired
public void
       configureGlobal(AuthenticationManagerBuilder auth)
       throws Exception {
  auth
    .inMemoryAuthentication()
    .passwordEncoder(NoOpPasswordEncoder.getInstance())
    .withUser("user").password("user").roles("USER");
}
```

Но если все же требуется разработать приложение, где аутентификация осуществляется в оперативной памяти, а пароли хранятся непосредственно в прикладном коде, что обычно делается в тестовых средах и в учебных целях, в таком случае пароли придется зашифровать, чтобы не раскрывать их в декомпилированном архиве фор-

мата JAR. В модуле Spring Security поддерживается немало готовых реализаций шифраторов паролей, хотя можно реализовать и собственный шифратор. В следующем примере кода демонстрируется простая реализация такого шифратора в Spring (в данном случае это реализация интерфейса `org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder`):

```
@Autowired
public void
       configureGlobal(AuthenticationManagerBuilder auth)
       throws Exception {
  PasswordEncoder passwordEncoder =
          new BCryptPasswordEncoder();
  auth
    .inMemoryAuthentication()
    .passwordEncoder(passwordEncoder)
    .withUser("user").password(passwordEncoder
    .encode("user")).roles("USER");
}
```

Внедрение в веб-приложение функций регистрации

Чтобы внедрить форму регистрации в рассматриваемое здесь веб-приложение, придется внести изменения в два представления. Ниже приведено содержимое файла `header.jspx` представления, предназначенного для отображения информации для пользователя при его регистрации.

```
<div id="header" xmlns:jsp="http://java.sun.com/JSP/Page"
     xmlns:spring="http://www.springframework.org/tags"
     xmlns:sec="http://www.springframework.org
                      /security/tags"
    version="2.0">
    <jsp:directive.page
         contentType="text/html;charset=UTF-8" />
    <jsp:output omit-xml-declaration="yes" />

    <spring:message code="header_text" var="headerText"/>
    <spring:message code="label_logout" var="labelLogout"/>
    <spring:message code="label_welcome"
                    var="labelWelcome"/>
    <spring:url var="logoutUrl" value="/logout" />

    <div id="appname">
      <h1>${headerText}</h1>
    </div>

    <div id="userinfo">
      <sec:authorize access="isAuthenticated()">
```

```
            ${labelWelcome}
        <sec:authentication property="principal.username" />
        <br/>
        <a href="${logoutUrl}">${labelLogout}</a>
        </sec:authorize>
    </div>
</div>
```

Сначала в приведенной выше разметке внедряется библиотека дескрипторов с префиксом **sec** для Spring Security. Затем вводится раздел `<div>` с дескриптором `<sec:authorize>` с целью выявить, вошел ли пользователь в веб-приложение. Если это так (т.е. из вызова метода `isAuthenticated()` возвращается логическое значение `true`), то отобразится имя пользователя и ссылка для выхода из приложения.

Второе изменяемое представление содержится в файле `menu.jspx`, где введена форма регистрации. Если пользователь войдет в веб-приложение, то в меню должен появиться пункт **New Singer** (Новый певец).

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div id="menu" xmlns:jsp="http://java.sun.com/JSP/Page"
     xmlns:spring="http://www.springframework.org/tags"
     xmlns:sec="http://www.springframework.org/security/tags"
     version="2.0">

    <jsp:directive.page
         contentType="text/html;charset=UTF-8" />
    <jsp:output omit-xml-declaration="yes" />
    <spring:message code="menu_header_text"
            var="menuHeaderText"/>
    <spring:message code="menu_add_singer"
            var="menuAddSinger"/>
    <spring:url value="/singers?form" var="addSingerUrl"/>
    <spring:message code="label_login" var="labelLogin"/>
    <spring:url var="loginUrl" value="/login" />

    <h3>${menuHeaderText}</h3>
    <sec:authorize access="hasRole('ROLE_USER')">
      <a href="${addSingerUrl}"><h3>${menuAddSinger}</h3>
      </a>
    </sec:authorize>

    <sec:authorize access="isAnonymous()">
    <div id="login">
    <form name="loginForm" action="${loginUrl}"
        method="post">
      <table>
        <caption align="left">Login:</caption>
        <tr>
        <td>User Name:</td>
```

```
        <td><input type="text" name="username"/></td>
       </tr>
       <tr>
       <td>Password:</td>
       <td><input type="password" name="password"/></td>
       </tr>
       <tr>
         <td colspan="2" align="center">
           <input name="submit" type="submit" value="Login"/>
           </td>
          </tr>
        </table>
      </form>
      </div>
    </sec:authorize>
</div>
```

Пункт Add Singer (Ввести нового певца) появится в меню лишь в том случае, если пользователь войдет в веб-приложение с назначенной ролью USER (как указано в первом дескрипторе <sec:authorize>). А если пользователь не войдет в веб-приложение (т.е. из вызова метода isAuthenticated() во втором дескрипторе <sec:authorize> возвращается логическое значение true), то отобразится форма регистрации.

Снова развернув данное веб-приложение, вы обнаружите, что оно отображает форму регистрации. Обратите внимание на то, что пункт меню New Singer не виден. Введите **user** в полях для имени пользователя и пароля и щелкните на кнопке Login (Войти). В области заголовка появятся сведения о пользователе. Кроме того, появится пункт меню New Singer.

Необходимо также изменить представление для показа (из файла show.jspx), чтобы отображать ссылку Edit Singer (Редактировать сведения о певце) только для пользователей, вошедших в веб-приложение, но мы оставляем это вам в качестве упражнения. Если введенные регистрационные данные окажутся неверными, это состояние должно быть обработано по следующему URL: /security/loginfail. Следовательно, на этот случай необходимо реализовать соответствующий контроллер, как показано ниже во фрагменте исходного кода из класса SecurityController.

```
package com.apress.prospring5.ch16.web;

import com.apress.prospring5.ch16.util.Message;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.context.MessageSource;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation
```

```
          .RequestMapping;
import java.util.Locale;

@Controller
@RequestMapping("/security")
public class SecurityController {
  private final Logger logger =
         LoggerFactory.getLogger(SecurityController.class);
  private MessageSource messageSource;

  @RequestMapping("/loginfail")
  public String loginFail(Model uiModel, Locale locale) {
    logger.info("Login failed detected");
    uiModel.addAttribute("message", new Message("error",
          messageSource.getMessage("message_login_fail",
                                   new Object[], locale)));
    return "singers/list";
  }

  @Autowired
  public void setMessageSource(
              MessageSource messageSource) {
    this.messageSource = messageSource;
  }
}
```

В приведенном выше классе контроллера будут обрабатываться все URL с префиксом **security**, а в методе `loginFail()` — случай неверной регистрации. В данном методе сообщение о неверной регистрации сохраняется в модели данные, а затем направляется на начальную страницу для отображения пользователю. Попробуйте снова развернуть данное веб-приложение и введите неверные регистрационные данные пользователя. В итоге снова появится начальная страница с сообщением о неверной регистрации.

Применение аннотаций для защиты методов контроллера

Сокрытия пункта **New Singer** в меню недостаточно. Так, если ввести в окне браузера следующий URL: `http://localhost:8080/users?form`, то по-прежнему будет видна страница для ввода сведений о певцах, несмотря на то, что входа в данное веб-приложение не было. Дело в том, что данное веб-приложение не защищено на уровне URL. Один из способов защиты страницы веб-приложения предусматривает конфигурирование цепочки фильтров Spring Security (в файле `securitycontext.xml`) для перехвата URL только для аутентифицированных пользователей. Но в этом случае остальные пользователи не смогут видеть представление списка певцов.

Другой способ решения данной задачи состоит в том, чтобы организовать защиту на уровне методов контроллера, используя поддержку аннотаций в Spring Security. Защита на уровне методов активизируется с помощью аннотации `@EnableGlobalMethodSecurity(prePostEnabled = true)`, объявляемой в конфигурационном классе `SecurityConfig`, причем в атрибуте `prePostEnabled` активизируется поддержка применения аннотаций как до, так и после вызова методов. После этого можно воспользоваться аннотацией `@PreAuthorize` для защиты нужного метода контроллера. Так, в следующем фрагменте кода демонстрируется пример защиты метода `createForm()`:

```
import org.springframework.security.access.prepost.PreAuthorize;
...

  @PreAuthorize("isAuthenticated()")
  @RequestMapping(params = "form",
                  method = RequestMethod.GET)
  public String createForm(Model uiModel) {
    Singer singer = new Singer();
    uiModel.addAttribute("singer", singer);

    return "singers/create";
  }
```

Здесь применяется аннотация `@PreAuthorize` (из пакета `org.springframework.security.access.prepost`) для защиты метода `createForm()` с аргументом, в котором указывается выражение для требований безопасности. Можете теперь попробовать ввести вручную URL страницы, предназначенной для ввода сведений о певцах. И если вы не вошли в данное веб-приложение, то модуль Spring Security автоматически направит вас на страницу регистрации с представлением списка певцов, как было настроено в конфигурационном классе `SecurityConfig`.

Разработка веб-приложений средствами Spring Boot

Проект Spring Boot и одноименный модуль были представлены в самом начале этой книги потому, что они приносят практическую пользу, ускоряя разработку приложений. В этом разделе будет показано, как создать полноценное веб-приложение с защитой и веб-страницами, используя Thymeleaf — шаблонизатор в форматах XML/XHTML/HTML5, предназначенный для применения как в веб, так и в других средах и легко интегрируемый с каркасом Spring. Этот шаблонизатор лучше всего подходит для Spring потому, что его создатель и руководитель одноименного проекта Даниэль Фернандес (Daniel Fernandez) поставил перед собой цель снабдить модуль Spring

MVC подходящим шаблонизатором[7]. Thymeleaf служит практической альтернативой JSP или Apache Tiles и весьма благосклонно воспринят командой разработчиков из компании SpringSource, поэтому и вам будет полезно на будущее знать, как конфигурировать и применять этот шаблонизатор.

Первая версия Thymeleaf была выпущена в апреле 2011 года. На момент написания данной книги уже была выпущена версия Thymeleaf 3.0.7, включая обновление нового модуля интеграции для версии Spring 5. Имеется немало расширений Thymeleaf, написанных и сопровождаемых официальной командой разработчиков Thymeleaf (например, расширение Thymeleaf Spring Security[8] и модуль Thymeleaf для совместимости с прикладным интерфейсом Java 8 Time API[9]).

Как бы там ни было, перейдем от истории развития Thymeleaf непосредственно к созданию веб-приложения Spring Boot. Для создания полноценного веб-приложения в Spring это означает потребность в наличии уровней объектов DAO и обслуживания, а следовательно, потребность в специальной стартовой библиотеке Spring Boot для поддержки сохраняемости и транзакций. Ниже приведен фрагмент кода конфигурации Gradle, в котором определяются библиотеки, требующиеся для создания рассматриваемого здесь веб-приложения. Каждая из них будет подробнее рассмотрена далее в соответствующем месте.

```
// Файл конфигурации pro-spring-15/build.gradle
ext {
  // Библиотеки Spring
  bootVersion = '2.0.0.M3'

  bootstrapVersion = '3.3.7-1'
  thymeSecurityVersion = '3.0.2.RELEASE'
  jQueryVersion = '3.2.1'
  ...

  spring = [
    ...
    springSecurityTest: "org.springframework.security:
          spring-security-test:$springSecurityVersion"
  ]
```

---

[7] Всю дискуссию по данному вопросу можно найти на официальном форуме Thymeleaf по адресу `http://forum.thymeleaf.org/why-Thymeleaf-td3412902.html`.

[8] В библиотеке Thymeleaf Extras Spring Security предоставляется диалект, позволяющий интегрировать ряд аспектов авторизации и аутентификации из модуля Spring Security (версиях 3.*x* и 4.*x*) в приложения, основанные на Thymeleaf. Подробнее об этом см. по адресу `https://github.com/thymeleaf/thymeleaf-extras-springsecurity`.

[9] Это модуль Thymeleaf Extras, а не часть ядра Thymeleaf, а следовательно, в нем употребляется своя схема контроля версий, хотя он и полностью поддерживается командой разработчиков Thymeleaf. Подробнее об этом см. по адресу `https://github.com/thymeleaf/thymeleaf-extras-java8time`.

```
  boot = [
    ...
    starterThyme :" org.springframework.boot:
           spring-boot-starter-thymeleaf:$bootVersion",
    starterSecurity : "org.springframework.boot:
           spring-boot-starter-security:$bootVersion"
  ]

  web = [
    bootstrap : "org.webjars:bootstrap:$bootstrapVersion",
    jQuery    : "org.webjars:jquery:$jQueryVersion",
    thymeSecurity: "org.thymeleaf.extras:
                    thymeleaf-extras-springsecurity4:
                    $thymeSecurityVersion"
  ]

  db = [
    ...
    h2 : "com.h2database:h2:$h2Version"
  ]
}

// Файл конфигурации chapter16/build.gradle
...
apply plugin: 'org.springframework.boot'

dependencies {
  compile boot.starterJpa, boot.starterJta, db.h2,
  boot.starterWeb, boot.starterThyme, boot.starterSecurity,
  web.thymeSecurity, web.bootstrap, web.jQuery
  testCompile boot.starterTest, spring.springSecurityTest
}
```

Установка уровня объектов DAO

В состав библиотеки `spring-boot-starter-data-jpa` для запуска прикладного интерфейса JPA в модуле Spring Boot входят автоматически конфигурируемые компоненты Spring Beans, с помощью которых можно установить и сформировать базу данных H2, если ее библиотека указана в пути к классам. А разработчикам веб-приложений останется лишь создать класс сущности и информационное хранилище. Но ради простоты в рассматриваемом здесь примере будет использоваться приведенная ниже упрощенная версия класса `Singer`.

```
package com.apress.prospring5.ch16.entities;

import org.hibernate.validator.constraints.NotBlank;
import javax.persistence.*;
import javax.validation.constraints.NotNull;
```

```
import javax.validation.constraints.Size;
import java.io.Serializable;
import java.util.Date;
import static javax.persistence.GenerationType.IDENTITY;

@Entity
@Table(name = "singer")
public class Singer implements Serializable {

  @Id
  @GeneratedValue(strategy = IDENTITY)
  @Column(name = "ID")
  private Long id;

  @Version
  @Column(name = "VERSION")
  private int version;

  @NotBlank(message =
            "{validation.firstname.NotBlank.message}")
  @Size(min = 2, max = 60,
        message = "{validation.firstname.Size.message}")
  @Column(name = "FIRST_NAME")
  private String firstName;

  @NotBlank(message =
            "{validation.lastname.NotBlank.message}")
  @Size(min = 1, max = 40,
        message = "{validation.lastname.Size.message}")
  @Column(name = "LAST_NAME")
  private String lastName;

  @NotNull
 @Temporal(TemporalType.DATE)
  @Column(name = "BIRTH_DATE")
  private Date birthDate;

  @Column(name = "DESCRIPTION")
  private String description;
  // Методы установки и получения
  ...
}
```

Кроме того, ради простоты в данном примере будет использоваться упрощенное расширение экземпляра приведенного выше класса `CrudRepository`.

```
package com.apress.prospring5.ch16.repos;

import com.apress.prospring5.ch16.entities.Singer;
import org.springframework.data.repository.CrudRepository;
```

```
public interface SingerRepository
       extends CrudRepository<Singer, Long> {
}
```

Установка уровня обслуживания

Уровень обслуживания рассматриваемого здесь веб-приложения столь же прост. Он состоит из класса `SingerServiceImpl`, а также из класса `DBInitializer`, предназначенного для инициализации и заполнения базы данных записями о певцах. Ниже приведен исходный код класса `SingerServiceImpl`, тогда как класс `DBInitializer` был рассмотрен в предыдущих разделах, и поэтому его исходный код не повторяется здесь снова.

```
package com.apress.prospring5.ch16.services;

import com.apress.prospring5.ch16.entities.Singer;
import com.apress.prospring5.ch16.repos.SingerRepository;
import com.google.common.collect.Lists;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class SingerServiceImpl implements SingerService {

  private SingerRepository singerRepository;

  @Override
  public List<Singer> findAll() {
    return Lists.newArrayList(singerRepository.findAll());
  }

  @Override
  public Singer findById(Long id) {
    return singerRepository.findById(id).get();
  }

  @Override
  public Singer save(Singer singer) {
   return singerRepository.save(singer);
  }

  @Autowired
  public void setSingerRepository(
              SingerRepository singerRepository) {
    this.singerRepository = singerRepository;
  }
}
```

Установка веб-уровня

Веб-уровень состоит только из простейшей версии класса `SingerController`, где аннотации `@RequestMapping` были заменены равнозначными аннотациями, чтобы больше не указывать метод доступа по сетевому протоколу HTTP. Исходный код данного класса приведен ниже.

```
package com.apress.prospring5.ch16.web;

import com.apress.prospring5.ch16.entities.Singer;
import com.apress.prospring5.ch16.services.SingerService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation
        .Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

import javax.validation.Valid;
import java.util.List;

@Controller
@RequestMapping(value = "/singers")
public class SingerController {

  private final Logger logger =
          LoggerFactory.getLogger(SingerController.class);
  @Autowired SingerService singerService;

  @GetMapping
  public String list(Model uiModel) {
   logger.info("Listing singers");
   List<Singer> singers = singerService.findAll();
   uiModel.addAttribute("singers", singers);
   logger.info("No. of singers: " + singers.size());
   return "singers";
  }

  @GetMapping(value = "/{id}")
  public String show(@PathVariable("id") Long id,
                     Model uiModel) {
    Singer singer = singerService.findById(id);
    uiModel.addAttribute("singer", singer);
    return "show";
  }

  @GetMapping(value = "/edit/{id}")
  public String updateForm(@PathVariable Long id, Model model) {
```

```
    model.addAttribute("singer", singerService.findById(id));
    return "update";
  }

  @GetMapping(value = "/new")
  public String createForm(Model uiModel) {
    Singer singer = new Singer();
    uiModel.addAttribute("singer", singer);
    return "update";
  }

  @PostMapping
  public String saveSinger(@Valid Singer singer) {
    singerService.save(singer);
    return "redirect:/singers/" + singer.getId();
  }
}
```

Методы `updateForm()` и `createForm()` возвращают одно и то же представление, а отличаются они лишь тем, что метод `updateForm()` принимает в качестве аргумента идентификатор существующего экземпляра типа `Singer`, употребляемого в качестве объекта модели для представления `update`, содержащего экранную кнопку, после щелчка на которой вызывается метод `createForm()`. Если экземпляр класса `Singer` содержит идентификатор певца, то в базе данных выполняется обновление этого объекта, а иначе создается экземпляр типа `Singer`, который затем сохраняется в базе данных. В данном примере проверка достоверности и надлежащая обработка ее ошибок не рассматриваются. Реализацию этих операций мы оставляем вам в качестве упражнения. А задачи обнаружения и создания компонентов Spring Beans для реализации контроллера поддерживаются в пусковой библиотеке `spring-boot-starter-web` из модуля Spring Boot.

Установка защиты средствами Spring Security

Для защиты средствами Spring Security в модуле Spring Boot предоставляется стартовая библиотека `spring-boot-starter-security`. Если эта библиотека указана в пути к классам, модуль Spring Boot автоматически защитит все конечные точки подключения по сетевому протоколу HTTP с помощью элементарной аутентификации. Тем не менее устанавливаемые по умолчанию параметры защиты могут быть специально настроены в дальнейшем. Но ради простоты допустим, что только главная (т.е. начальная) страница рассматриваемого здесь веб-приложения доступна из корневого (/) контекста (по адресу `http:\localhost:8080/`). Ниже приведена конфигурация, специально настроенная для этой цели.

```
package com.apress.prospring5.ch16;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
```

```
import org.springframework.security.config.annotation
          .authentication.builders
          .AuthenticationManagerBuilder;
import org.springframework.security.config.annotation
          .web.builders.HttpSecurity;
import org.springframework.security.config.annotation
          .web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation
          .web.configuration.WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity
public class WebSecurityConfig
       extends WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http)
            throws Exception {
    http
       .authorizeRequests()
       .antMatchers("/", "/home").permitAll()
       .and()
       .authorizeRequests().antMatchers("/singers/**")
       .authenticated()
       .and()
       .formLogin()
       .loginPage("/login")
       .permitAll()
       .and()
       .logout()
       .permitAll();
  }

  @Autowired
  public void configureGlobal(
              AuthenticationManagerBuilder auth)
              throws Exception {
    auth
       .inMemoryAuthentication()
       .withUser("user").password("user").roles("USER");
  }
}
```

Создание представлений в Thymeleaf

Прежде чем перейти к созданию представлений с помощью шаблонизатора Thymeleaf, рассмотрим элементарную структуру веб-приложения Spring Boot. В этом приложении каталог `resources` служит для хранения веб-ресурсов, а следователь-

но, отпадает потребность в каталоге `webapp`. Если же содержимое каталога `resources` организовано в соответствии с требованиями к стандартной структуре в Spring Boot, то не потребуется также писать немало кода конфигурации, поскольку в стартовых библиотеках Spring Boot предоставляются предварительно сконфигурированные компоненты Spring Beans. Чтобы воспользоваться шаблонизатором Thymeleaf со стандартной конфигурацией, библиотеку `spring-bootstarter-thymeleaf` для запуска Thymeleaf из модуля Spring Boot необходимо указать в пути к классам данного проекта.

На рис. 16.5 приведены транзитивные зависимости библиотеки `spring-boot starter-thymeleaf` наряду с другими стартовыми библиотеками, которые могут представлять определенный интерес.

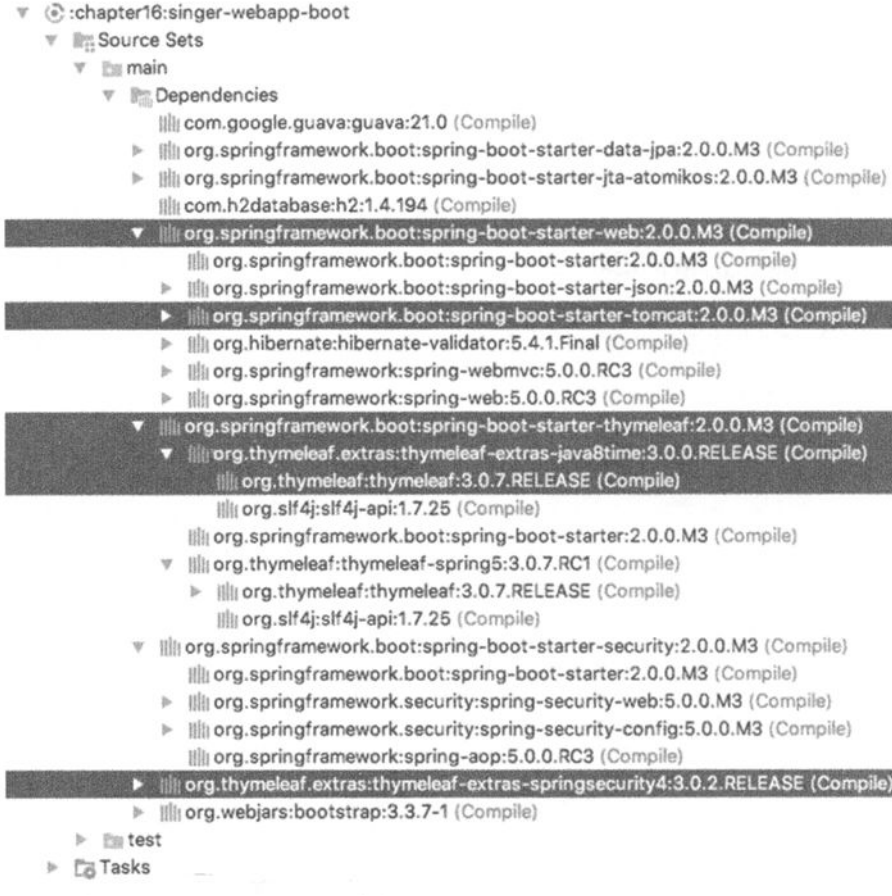


Рис. 16.5. Стартовые библиотеки Spring Boot и их зависимости

Как видите, версии 2.0.0.M3 библиотеки запуска Thymeleaf в Spring Boot сопутствует самая последняя (на момент написания данной книги) версия Thymeleaf 3.0.7. Обратите внимание на то, что в библиотеку `spring-boot-starter-web` внедрен

в качестве зависимости сервер Tomcat, применяемый для выполнения данного веб-приложения.

А теперь, когда все требующиеся библиотеки указаны в пути к классам, проанализируем структуру каталога `resources`, приведенную на рис. 16.6. По умолчанию шаблонизатор Thymeleaf настраивается в модуле Spring Boot на чтение файлов шаблонов из каталога `templates`. Если же модуль Spring MVC применяется самостоя-

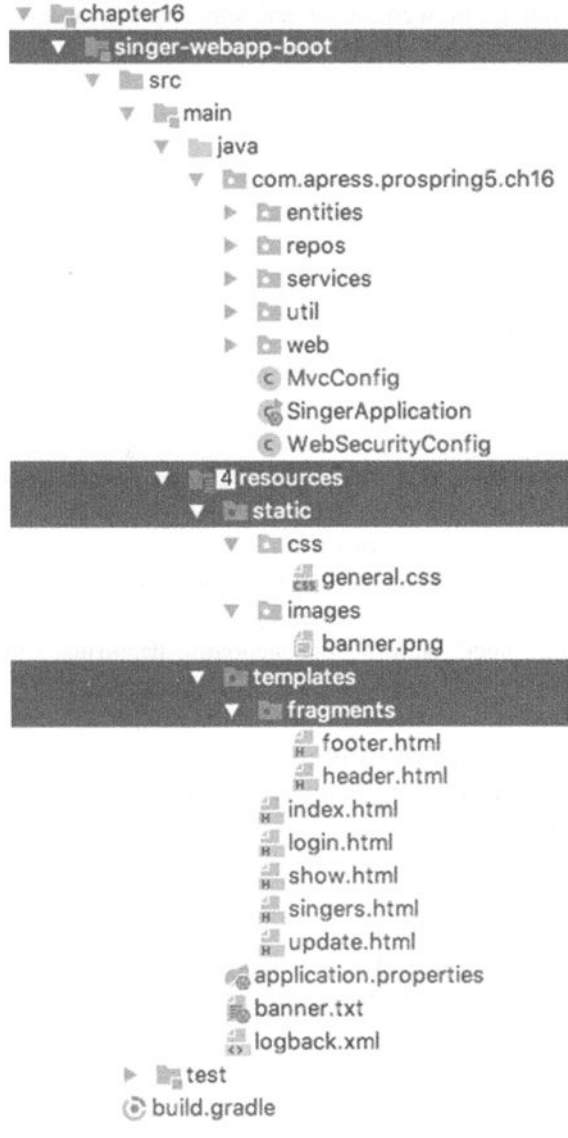


Рис. 16.6. Стандартная структура каталога **`resources`** для веб-приложения, построенного с помощью модуля Spring Boot и шаблонизатора Thymeleaf

тельно, т.е. без модуля Spring Boot, то шаблонизатор Thymeleaf придется сконфигурировать вручную, определив три компонента Spring Beans типа `SpringResourceTemplateResolver`, `SpringTemplateEngine` и `ThymeleafViewResolver` соответственно[10]. Следовательно, разработчику, использующему модуль Spring Boot, остается лишь приступить к созданию шаблонов и разместить их в каталоге `/resources/templates`.

Создать шаблон средствами Thymeleaf совсем не трудно, используя простой синтаксис HTML-разметки. Начнем создание шаблона со следующего простого примера:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head lang="en">
  <title>Spring Boot Thymeleaf Sample</title>
  <meta http-equiv="Content-Type" content="text/html;
   charset=UTF-8"/>
</head>
<body>
<h1>Hello World</h1>

<h2>Just another simple text here</h2>
</body>
</html>
```

В строке кода разметки `<html xmlns:th="http://www.thymeleaf.org">` объявляется пространство имен Thymeleaf. И это очень важно, поскольку без этого шаблон превратится в простой статический HTML-файл. Шаблонизатор Thymeleaf позволяет определить и специально настроить порядок обработки шаблонов до мельчайших подробностей. В самом шаблонизаторе Thymeleaf отдельные шаблоны обрабатываются одним или несколькими процессорами, распознающими определенные элементы, написанные на стандартном диалекте Thymeleaf. Этот диалект состоит в основном из процессоров атрибутов, с помощью которых браузер может правильно отображать шаблоны, даже не обрабатывая их файлы, поскольку неизвестные атрибуты просто игнорируются. В качестве примера ниже приведена разметка поля ввода текста в заполняемой форме:

```
<input type="text" name="singerName" value="John Mayer" />
```

В приведенной выше строке кода объявляется элемент статической HTML-разметки, который браузер в состоянии распознать и правильно интерпретировать. Если же объявить тот же самый элемент разметки с помощью элементов формы в Spring, то он будет выглядеть следующим образом:

```
<form:inputText name="singerName" value="${singer.name}" />
```

---

[10] На официальном веб-сайте Thymeleaf имеется хороший учебный материал для работы с Thymeleaf и Spring. Подробнее об этом см. по адресу `http://thymeleaf.org/doc/tutorials/3.0/thymeleafspring.html#spring-mvc-configuration`.

Но браузер не сумеет отобразить объявленный таким образом элемент, поэтому в процессе написания шаблонов с помощью Apache Tiles приходится постоянно просматривать размечаемую форму в браузере, предварительно скомпилировав разрабатываемый проект. Этот недостаток устраняется в Thymeleaf с помощью обычных элементов HTML-разметки, которые могут быть сконфигурированы с помощью атрибутов Thymeleaf. Ниже показано, каким образом приведенный выше элемент разметки можно написать в Thymeleaf.

```
<input type="text" name="singerName" value="John Mayer"
   th:value="${singer.name}" />
```

Благодаря приведенному выше объявлению браузер сумеет отобразить элемент разметки, а разработчик — установить значение этого элемента, чтобы увидеть расположение элементов на странице. Это значение заменяется результатом вычисления заданного выражения `${singer.name}` в процессе обработки шаблона.

Как упоминалось в предыдущих разделах, при разработке пользовательского интерфейса для веб-приложения может возникнуть потребность выделить общие части и включить их в другие шаблоны, чтобы исключить дублирование. Например, заголовки, подзаголовки и меню являются общими частями для всех страниц веб-приложения. И эти общие части представлены специальными шаблонами Thymeleaf, называемыми *фрагментами*. По умолчанию эти частичные шаблоны должны быть определены в каталоге `/templates/fragments`, хотя имя данного каталога может быть другим, если того требует конкретная ситуация[11]. Ниже приведен синтаксис для определения фрагмента подзаголовка в файле шаблона `footer.html`.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <meta http-equiv="Content-Type" content="text/html;
        charset=UTF-8"/>
  <title>
     ProSpring5 Singer Boot Application
  </title>
</head>
<body>
<div th:fragment="footer" th:align="center">
  <p>Copyright (c) 2017 by Iuliana Cosmina and Apress.
     All rights reserved.</p>
</div>
</body>
</html>
```

Обратите внимание на то, что в атрибуте `th:fragment` объявляется имя фрагмента, который может быть далее вставлен или заменен в других шаблонах определен-

[11] У веб-приложения может быть несколько тем, причем каждая из них может быть представлена рядом фрагментов.

ными элементами. Ниже демонстрируется синтаксис для применения такого фрагмента в файле шаблона index.html.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
head>
  <meta http-equiv="Content-Type" content="text/html;
        charset=UTF-8"/>
  <title>
     ProSpring5 Singer Boot Application
  </title>
</head>

<body>
<div th:replace="~{fragments/header :: header}">Header</div>
<div class="container">
...
</div>
</body>
</html>
```

Фрагментом можно воспользоваться следующими способами.

- Атрибут th:replace заменяет дескриптор своего узла указанным фрагментом.
- Атрибут th:insert вставляет указанный фрагмент в тело дескриптора своего узла.
- Атрибут th:include вставляет содержимое данного фрагмента[12].

Чтобы узнать подробнее о шаблонах Thymeleaf, лучше всего обратиться к ресурсам, доступным на официальном веб-сайте Thymeleaf по адресу http://thymeleaf.org/documentation.html. И, наконец, необходимо упомянуть о каталоге static, который содержит статические ресурсы для шаблонов приложения, включая файлы стилевых таблиц CSS и изображений. Чтобы включить специальные классы CSS в шаблон Thymeleaf, можно воспользоваться следующим синтаксисом:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <meta http-equiv="Content-Type" content="text/html;
        charset=UTF-8"/>
<link href="../../public/css/general.css"
  th:href="@{css/general.css}" rel="stylesheet"
    media="screen"/>
<title>
  ProSpring5 Singer Boot Application
</title>
</head>
```

[12] Начиная с версии 3.0, делать это не рекомендуется.

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

Атрибут `href` содержит путь относительно шаблона и применяется при открытии шаблона в браузере для загрузки стилей из файла `general.css`. Этот атрибут применяется при обработке шаблона, а из выражения `{css/general.css}` определяется URL ресурса для веб-приложения.

Применение расширений Thymeleaf

В веб-приложении, рассматриваемом в этом разделе, применяются два расширения Thymeleaf. В частности, расширение `thymeleaf-extrasspringsecurity4` требуется для того, чтобы построить защищенное веб-приложение с помощью шаблонов Thymeleaf. Так, если пользователь не вошел в веб-приложение, то должен появиться вариант выбора **Log In** (Войти), а если пользователь вошел в веб-приложение, — вариант выбора **Log Out** (Выйти).

Данное расширение реализовано в виде модуля Thymeleaf Extras, а не входит в ядро Thymeleaf, хотя оно полностью поддерживается командой разработчиков Thymeleaf. Несмотря на то что в имени данного расширения указан номер 4, в нем требуется лишь поддержка некоторых служебных объектов, специально предназначенных для защиты. Следовательно, оно совместимо с версией Spring Security 5.0.*x*, т.е. с транзитивной зависимостью от стартовой библиотеки защиты из модуля Spring Boot. А теперь рассмотрим более расширенный файл шаблона `header.html`, содержащий навигационное меню с разными пунктами для зарегистрированных и незарегистрированных пользователей.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <meta http-equiv="Content-Type" content="text/html;
        charset=UTF-8"/>
  <link href="../../public/css/bootstrap.min.css"
    th:href=
      "@{/webjars/bootstrap/3.3.7-1/css/bootstrap.min.css}"
    rel="stylesheet" media="screen"/>
  <script src="http://cdn.jsdelivr.net/webjars/jquery
                    /3.2.1/jquery.min.js"
    th:src="@{/webjars/jquery/3.2.1/jquery.min.js}"></script>
  <title>
    ProSpring5 Singer Boot Application
  </title>
</head>
<body>
<div class="container">
```

```
  <div th:fragment="header">
    <nav class="navbar navbar-inverse">
  <div class="container-fluid">
  <div class="navbar-header">
    <a class="navbar-brand"
        href="#" th:href="@{/}">ProSpring 5</a>
  </div>
  <ul class="nav navbar-nav">
    <li><a href="#" th:href="@{/singers}">Singers</a></li>
    <li><a href="#" th:href="@{/singers/new}">Add Singer</a>
    </li>
  </ul>
  <ul class="nav navbar-nav navbar-right" >
    <li th:if=
      "${#authorization.expression('!isAuthenticated()')}">
    <a href="/login" th:href="@{/login}">
    <span class="glyphicon glyphicon-log-in"></span> 
          Log in
    </a>
    </li>
    <li th:if=
       "${#authorization.expression('isAuthenticated()')}">
    <a href="/logout" th:href="@{#}"
       onclick="$('#form').submit();">
    <span class="glyphicon glyphicon-log-out"></span> 
          Logout
    </a>
    <form style="visibility: hidden" id="form"
         method="post" action="#"
      th:action="@{/logout}"></form>
    </li>
      </ul>

  </div>
  </nav>
   </div>
</div>
</body>
</html>
```

В данном модуле расширения предоставляется новый диалект `org.thymeleaf.extras.springsecurity4.dialect.SpringSecurityDialect`, включающий в себя объект `#authorization`, использовавшийся в предыдущем примере в качестве служебного объекта с методами для проверки авторизации на основании выражений,

URL и списков управления доступом[13]. Этот диалект изначально конфигурируется в модуле Spring Boot.

Второе расширение, применяемое в рассматриваемом здесь приложении, называется `thymeleaf-extras-java8time`. Зависимость от этой библиотеки вводится вручную в путь к классам в данном проекте и обеспечивает поддержку прикладного интерфейса Java 8 Time API. Это расширение полностью поддерживается также официальной командой разработчиков Thymeleaf. В нем вводится служебный объект `#temporals` (и другие аналогичные объекты) в контекст типа `ApplicationContext` в качестве процессора во время вычисления выражений. Это означает, что выражения вычисляются на языках OGNL (Object-Graph Navigation Language — язык перемещения по графам объектов) и SpringEL (Spring Expression Language — язык выражений Spring). Рассмотрим далее шаблон из файла `show.html`, требующийся для отображения подробных сведений о певцах:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">

<head>
  <meta http-equiv="Content-Type" content="text/html;
        charset=UTF-8"/>
  <link href="../../public/css/bootstrap.min.css"
    th:href=
      "@{/webjars/bootstrap/3.3.7-1/css/bootstrap.min.css}"
    rel="stylesheet" media="screen"/>
  <script src="http://cdn.jsdelivr.net/webjars/jquery
                    /3.2.1/jquery.min.js"
    th:src="@{/webjars/jquery/3.2.1/jquery.min.js}">
  </script>

  <title>
    ProSpring5 Singer Boot Application
  </title>

  </head>
  <body>
  <div th:replace="~{fragments/header :: header}">Header</div>
  <div class="container">

  <h1>Singer Details</h1>

  <div>
    <form class="form-horizontal">
    <div class="form-group">
```

[13] Рассмотрение этого вопроса выходит за рамки данной книги, но если он вас все же заинтересует, то обратитесь за дополнительными сведениями на официальный веб-сайт по адресу `https://github.com/thymeleaf/thymeleaf-extras-springsecurity`.

```
      <label class="col-sm-2 control-label">First Name:
      </label>
    <div class="col-sm-10">
      <p class="form-control-static"
         th:text="${singer.firstName}">
         Singer First Name
      </p>
    </div>
    </div>
    <div class="form-group">
      <label class="col-sm-2 control-label">Last Name:
      </label>
    <div class="col-sm-10">
      <p class="form-control-static"
         th:text="${singer.lastName}">
         Singer Last Name
      </p>
    </div>
    </div>
    <div class="form-group">
      <label class="col-sm-2 control-label">Description:
      </label>
    <div class="col-sm-10">
      <p class="form-control-static"
         th:text="${singer.description}">
         Singer Description
      </p>
    </div>
    </div>
    <div class="form-group">
      <label class="col-sm-2 control-label">BirthDate:
      </label>
    <div class="col-sm-10">
      <p class="form-control-static"
         th:text="${#dates.format(singer.birthDate,
                  'dd-MMM-yyyy')}">
         Singer BirthDate
      </p>
    </div>
    </div>
  </form>
  </div>
  <div th:insert="~{fragments/footer :: footer}">
       (c) 2017 Iuliana Cosmina & Apress</div>
  </div>
</body>
</html>
```

В предыдущем примере значение свойства `singer.birthDate` типа `java.util.Date` форматируется с помощью служебного объекта `#dates`. В таком случае специальный объект для форматирования дат не требуется, а принятый образец форматирования дат жестко кодируется в шаблоне. Но вряд ли такой подход можно считать практичным. Допустим, имеется класс `DateFormatter`, определяемый следующим образом:

```
package com.apress.prospring5.ch16.util;

import org.springframework.format.Formatter;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Locale;

public class DateFormatter implements Formatter<Date> {
  public static final SimpleDateFormat formatter =
         new SimpleDateFormat("yyyy-MM-dd");

  @Override
  public Date parse(String s, Locale locale)
              throws ParseException {
    return formatter.parse(s);
  }

  @Override
  public String print(Date date, Locale locale) {
    return formatter.format(date);
  }
}
```

Воспользуемся минимально возможным конфигурационным классом, чтобы внедрить реализованное выше средство форматирования дат в путь к классам рассматриваемого здесь веб-приложения.

```
package com.apress.prospring5.ch16;

import com.apress.prospring5.ch16.util.DateFormatter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.format.FormatterRegistry;
import org.springframework.web.servlet.config.annotation
          .ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation
          .WebMvcConfigurer;

@Configuration
public class MvcConfig implements WebMvcConfigurer {
```

```
  @Override
  public void addViewControllers(
              ViewControllerRegistry registry) {
    registry.addViewController("/index")
            .setViewName("index");
    registry.addViewController("/")
            .setViewName("index");
    registry.addViewController("/login")
            .setViewName("login");
  }

  @Override
  public void addFormatters(
              FormatterRegistry formatterRegistry) {
    formatterRegistry.addFormatter(dateFormatter());
  }

  @Bean
  public DateFormatter dateFormatter() {
    return new DateFormatter();
  }
}
```

Как только средство форматирования дат будет зарегистрировано в данном веб-приложении, оно может быть использовано в шаблонах Thymeleaf с помощью синтаксиса двойных фигурных скобок. Таким образом, следующий фрагмент разметки:

```
<p class="form-control-static"
  th:text=
     "${#dates.format(singer.birthDate, 'dd-MMM-yyyy')}">
  Singer BirthDate
</p>
```

можно переписать так:

```
<p class="form-control-static"
   th:text="${{singer.birthDate}}">
   Singer BirthDate
</p>
```

Следовательно, жесткое кодирование образца форматирования дат в данном примере было исключено, а класс, реализующий средство формирования дат, можно видоизменить внешним образом, не внося вообще никаких изменений в шаблон Thymeleaf. Согласитесь, что такой подход более практичный.

Применение архивов Webjars

Возможно, в самом последнем примере применения шаблона Thymeleaf вы заметили ряд необычных ссылок, указанных в элементах разметки `<head>`.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <meta http-equiv="Content-Type"
        content="text/html; charset=UTF-8"/>
  <link href="../../public/css/bootstrap.min.css"
    th:href=
      "@{/webjars/bootstrap/3.3.7-1/css/bootstrap.min.css}"
    rel="stylesheet" media="screen"/>
  <script src="http://cdn.jsdelivr.net/webjars/jquery
                    /3.2.1/jquery.min.js"
    th:src="@{/webjars/jquery/3.2.1/jquery.min.js}">
  </script>
  <title>
    ProSpring5 Singer Boot Application
  </title>
</head>
<body>
  ...
</body>
</html>
```

Создание привлекательной HTML-страницы еще больше упрощается после внедрения Bootstrap[14] — самой распространенной библиотеки HTML, CSS и JS для разработки оперативно реагирующих веб-приложений, рассчитанных в первую очередь на мобильные устройства. Чтобы воспользоваться библиотекой Bootstrap в шаблонах, достаточно сделать ссылку на файл Bootstrap CSS в своем шаблоне. Долгое время стили CSS и сценарии JavaScript были составной частью веб-приложения и вручную копировались разработчиками в специальный каталог. Но в последнее время библиотеки вроде jQuery и Bootstrap, чаще всего применяемые для создания веб-страниц, могут по-разному употребляться в веб-приложениях через внедрение зависимостей как упакованные в архивы Java, иначе называемые Webjars[15]. Архивы Webjars развертываются в центральном хранилище Maven, загружаются и вводятся в веб-приложение автоматически избранным инструментальным средством построения проектов (Maven, Gradle и т.д.), как только они будут объявлены в качестве зависимостей в разрабатываемом проекте.

В приведенном выше примере шаблона с помощью атрибутов `th:href` объявляются архивы Webjars библиотек JQuery и Bootstrap. Введя ссылки на них в файл шаблона, можно гарантировать, что все классы Bootstrap будут применяться к элементам разметки в шаблоне (например, `class="container-fluid"`), а все функции — доступны из библиотеки jQuery (`onclick="$('#form').submit();`) при обработке шаблона и развертывании веб-приложения на сервере.

---

[14] Официальный веб-сайт Bootstrap доступен по адресу `http://getbootstrap.com/`.

[15] См. по адресу `https://www.webjars.org/`.

Если выполнить приведенный ниже исходный код класса `SingerApplication` и перейти по адресу `http://localhost:8080/`, то в конечном итоге появится обработанный шаблон `index.html`.

```
package com.apress.prospring5.ch16;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure
          .SpringBootApplication;
import org.springframework.context
          .ConfigurableApplicationContext;

@SpringBootApplication
public class SingerApplication {

  private static Logger logger =
          LoggerFactory.getLogger(SingerApplication.class);

  public static void main(String... args)
         throws Exception {
    ConfigurableApplicationContext ctx =
                SpringApplication.run(
                      SingerApplication.class, args);
    assert (ctx != null);
    logger.info("Application started...");
    System.in.read();
    ctx.close();
  }
}
```

Обратите внимание на то, что в данном классе не требуется больше никаких других аннотаций, кроме `@SpringBootApplication`. Для автоматического выбора сущностей, контроллеров и хранилищ достаточно внедрить прикладной интерфейс JPA и библиотеки запуска веб-приложений из модуля Spring Boot в путь к классам. И если теперь перейти в окне браузера по следующему URL: `http://localhost:8080/`, то появится начальная страница веб-приложения. Если все пройдет нормально и архив Webjars библиотеки Bootstrap будет обработан правильно, то в конечном итоге начальная страница будет выглядеть так, как показано на рис. 16.7.

Рис. 16.7. Начальная страница веб-приложения Spring Boot, построенного в этом разделе

Резюме

В этой главе были раскрыты многие темы, связанные с разработкой веб-приложений с помощью Spring MVC. Сначала в ней были рассмотрены общие понятия проектного шаблона MVC, а затем модуль Spring MVC, включая иерархию контекста типа `WebApplicationContext`, жизненный цикл обработки запросов и конфигурирование.

Далее в главе был продемонстрирован пример разработки веб-приложения для певцов с помощью модуля Spring MVC и технологии представлений JSPX. Между тем обсуждались различные вопросы, в том числе интернационализация, тематическое оформление и поддержка шаблонов в шаблонизаторе Apache Tiles. Кроме того, было показано, как применять jQuery, jQuery UI и другие библиотеки JavaScript для совершенствования пользовательского интерфейса на конкретных примерах средства выбора даты, редактора форматированного текста и сетки данных с поддержкой разбиения на страницы. Здесь также обсуждались вопросы защиты веб-приложений средствами Spring Security.

В главе были также продемонстрированы функциональные средства веб-контейнеров, совместимых с Servlet 3.0, в том числе конфигурирование в исходном коде вместо применения файла `web.xml`. Далее было показано, как организовать выгрузку файлов в среде Servlet 3.0.

Команда разработчиков Spring предоставила замечательное средство в качестве модуля Spring Boot для построения полноценных веб-приложений, и в конце этой главы было показано, как им пользоваться в подобных целях. Безусловно, для этого

придется также воспользоваться наиболее подходящим шаблонизатором Thymeleaf, также обсуждавшимся в данной главе.

В следующей главе будут рассмотрены дополнительные функциональные возможности, предоставляемые в Spring для разработки веб-приложений. В частности, будет представлено введение в сетевой протокол WebSocket.

ГЛАВА 17

Протокол WebSocket

Для обеспечения связи клиента с сервером в веб-приложениях традиционно применялись стандартные функциональные возможности запросов и ответов по сетевому протоколу HTTP. По мере развития Интернета возникла потребность в более интерактивных возможностях, включая продвижение и извлечение обновлений из сервера или выполнение обновлений в реальном времени на сервере. С этой целью со временем были внедрены различные методики, в том числе непрерывный и длительный опросы, а также технология Comet. Каждой из этих методик были присущи свои достоинства и недостатки, и сетевой протокол WebSocket появился в результате изучения насущных потребностей и недостатков с целью выработать более простой и надежный способ построения интерактивных веб-приложений. В спецификации HTML5 WebSocket определяется прикладной интерфейс API, допускающий применение сетевого протокола WebSocket на веб-страницах для организации двухсторонней связи с удаленным хостом (т.е. сетевым узлом).

В этой главе представлен общий обзор сетевого протокола WebSocket и основных функциональных средств, предоставляемых для его поддержки в Spring Framework. В ней будут, в частности, рассмотрены следующие вопросы.

- **Введение в WebSocket.** В этой части представлено общее введение в сетевой протокол WebSocket. Эта часть главы не призвана служить подробным справочным пособием по протоколу WebSocket и представляет собой лишь общий обзор[1].

[1] Подробнее об этом сетевом протоколе см. в стандартном документе RFC-6455, доступный по адресу `http://tools.ietf.org/html/rfc6455` или `https://www.websocket.org/`.

- **Применение сетевого протокола WebSocket вместе с каркасом Spring.** В этой части подробно рассмотрены некоторые особенности применения сетевого протокола WebSocket вместе с каркасом Spring Framework. Здесь описано применение прикладного интерфейса WebSocket API в Spring, SockJS в качестве запасного варианта для браузеров, не поддерживающих сетевой протокол WebSocket, а также отправка сообщений по сетевому протоколу STOMP (Simple (или Streaming) Text Oriented Message Protocol — простой (или потоковый) протокол обмена текстовыми сообщениями) через протоколы SockJS/WebSocket.

Введение в сетевой протокол WebSocket

Сетевой протокол WebSocket представляет собой спецификацию, разработанную как часть инициативы HTML5 и допускающую полнодуплексное подключение через единственный сокет для обмена сообщениями между клиентом и сервером. В прошлом веб-приложения, которым требовались функциональные возможности для обновления данных в реальном времени, должны были периодически опрашивать соответствующий компонент на стороне сервера, устанавливая целый ряд соединений или производя длительный опрос для получения нужных данных.

Применение сетевого протокола WebSocket для двухстороннего обмена данными между клиентом и сервером позволяет исключить потребность в опросе по сетевому протоколу HTTP для установления двухсторонней связи между клиентом (например, веб-браузером) и HTTP-сервером. Протокол WebSocket предназначен для замены всех существующих способов двухстороннего обмена данными, применяющих HTTP в качестве транспортного протокола. Модель с единственным сокетом, принятая в сетевом протоколе WebSocket, обеспечивает в конечном счете более простое решение, исключая потребность устанавливать многочисленные соединения для каждого клиента и снижая связанные с этим издержки (например, потребность отправлять HTTP-заголовок вместе с каждым сообщением).

Сетевой протокол HTTP применяется в WebSocket при первоначальном установлении связи, что, в свою очередь, делает возможным его применение для обмена данными через стандартные порты HTTP (`80`) и HTTPS (`443`). Для обозначения незащищенных и защищенных соединений в спецификации WebSocket определены схемы **`ws://`** и **`wss://`**.

Сетевой протокол WebSocket состоит из двух частей: сначала подтверждения связи, установливаемой между клиентом и сервером, а затем обмена данными. Соединение по протоколу WebSocket устанавливается путем отправки из протокола HTTP запроса на обновление протоколу WebSocket на стадии первоначального подтверждения связи, установливаемой между клиентом и сервером через одно и то же базовое соединение по сетевому протоколу TCP/IP. На стадии передачи данных клиент и сервер могут одновременно обмениваться сообщениями, что, как нетрудно догадаться,

открывает возможность внедрять в приложения функциональные средства для более надежной связи в реальном времени.

Применение протокола WebSocket вместе с каркасом Spring

Начиная с версии 4.0 в каркасе Spring Framework поддерживается обмен сообщениями по сетевому протоколу WebSocket, а также STOMP в качестве подпротокола на прикладном уровне. В самом каркасе Spring поддержка сетевого протокола WebSocket находится в модуле `spring-websocket`, совместимом со стандартом JSR-356 (Java WebSocket)[2].

Разработчики приложений должны также осознавать, что сетевой протокол WebSocket поддерживается далеко не всеми браузерами, хотя он и сулит новые захватывающие возможности. Принимая во внимание это обстоятельство, приложение должно поддерживать взаимодействие с пользователями, задействуя какую-нибудь запасную технологию, чтобы смоделировать как можно больше требующихся функциональных возможностей. На этот случай в Spring Framework предоставляются прозрачные запасные варианты через протокол SockJS, как поясняется далее в этой главе.

В отличие от приложений, основанных на технологии REST, где службы представлены разными URL, в сетевом протоколе WebSocket применяется единственный URL для первоначального подтверждения установленной связи, а данные перемещаются потоком через одно и то же соединение. Такой тип функциональных возможностей для передачи сообщений в большей степени присущ традиционным системам обмена сообщениями. В версии Spring Framework 4 основные интерфейсы для обмена сообщениями (например, `Message`) были перенесены из проекта Spring Integration в новый модуль `spring-messaging`, предназначенный для поддержки приложений обмена сообщениями по сетевому протоколу WebSocket.

Когда STOMP упоминается как подпротокол, применяемый на прикладном уровне, то имеется в виду протокол, который транспортируется через WebSocket — низкоуровневый протокол, который просто преобразует байты в сообщения. В приложении должно быть известно, что именно передается по сети, но именно здесь и вступает в действие такой подпротокол, как STOMP. При первоначальном подтверждении установленной связи для определения применяемого подпротокола клиент и сервер могут использовать заголовок `Sec-WebSocket-Protocol`. И хотя в Spring Framework обеспечивается поддержка протокола STOMP, в протоколе WebSocket ничего особенного в этом отношении не регламентируется.

А теперь, когда стало понятно, что собой представляет сетевой протокол WebSocket и какую поддержку он получает в Spring, выясним, где можно пользоваться этой технологией. Учитывая односокетный характер протокола WebSocket и его способность

---

[2] Подробнее см. по адресу `www.oracle.com/technetwork/articles/java/jsr356-1937161.html`.

обеспечивать непрерывный двунаправленный поток данных, WebSocket идеально подходит для тех приложений, которые характеризуются высокой частотой передачи сообщений и требуют связи с малой задержкой. К числу приложений, которые подходят для применения протокола WebSocket, относятся игры, инструментальные средства для совместной работы в группе и реальном времени, системы оперативного обмена сообщениями с информацией о ценах, положении на финансовых рынках и т.д. Проектируя приложение, в котором предполагается применять сетевой протокол WebSocket, следует принимать во внимание как частоту появления сообщений, так и требования к сетевой задержке. Это поможет принять решение, следует ли применять протокол WebSocket или, например, длительный опрос по протоколу HTTP.

Применение прикладного интерфейса WebSocket API

Как упоминалось ранее в этой главе, сетевой протокол WebSocket преобразует байты в сообщения и транспортирует их между клиентом и сервером. Эти сообщения по-прежнему должны распознаваться самим приложением, и именно здесь вступают в действие подпротоколы вроде STOMP. На тот случай, если вы предпочитаете работать с низкоуровневым прикладным интерфейсом WebSocket API напрямую, в Spring Framework предоставляется удобный для взаимодействия прикладной интерфейс API. Имея дело с прикладным интерфейсом WebSocket API в Spring, обычно приходится реализовывать интерфейс `WebSocketHandler` или применять такие удобные подклассы, как `BinaryWebSocketHandler` — для обмена двоичными сообщениями, `SockJsWebSocketHandler` — для обмена сообщениями по протоколу SockJS или `TextWebSocketHandler` — для обмена символьных сообщений в формате `String`. В рассматриваемом далее примере ради простоты будет использоваться класс `TextWebSocketHandler` для обмена сообщениями в формате `String` по протоколу WebSocket. Итак, выясним сначала, каким образом можно получать и обрабатывать сообщения по протоколу WebSocket на низком уровне, задействовав прикладной интерфейс WebSocket API из каркаса Spring.

При желании каждый пример в этой главе также может быть сконфигурирован на языке Java. Но, на наш взгляд, пространство имен XML представляет аспекты конфигурации в сжатой форме, поэтому именно оно и будет применяться повсюду в примерах из этой главы. За дополнительными сведениями о конфигурировании на языке Java обращайтесь к справочному руководству. Итак, начнем с внедрения требующихся зависимостей. В следующем фрагменте кода конфигурации Gradle перечислены зависимости от требующихся библиотек:

```
// Файл конфигурации pro-spring-15/build.gradle
ext {
  springVersion = '5.0.0.RC3'
  twsVersion = '9.0.0.M22'
  ...
```

```
  spring = [
    ...

    context : "org.springframework:spring-context:
               $springVersion",
    webmvc : "org.springframework:spring-webmvc:
              $springVersion",
    webSocket : "org.springframework:spring-websocket:
                 $springVersion",
    messaging : "org.springframework:spring-messaging:
                 $springVersion"
  ]
  ...
  web = [
    ...
    jacksonDatabind: "com.fasterxml.jackson.core:
                      jackson-databind:$jacksonVersion",
    tomcatWsApi : "org.apache.tomcat:tomcat-websocket-api:
                   $twsVersion",
    tomcatWsEmbed: "org.apache.tomcat.embed:
                    tomcat-embed-websocket:$twsVersion",
    httpclient : "org.apache.httpcomponents:httpclient:
                  $httpclientVersion",
    websocket : "javax.websocket:javax.websocket-api:1.1"
  ]
}
...
// Файл конфигурации pro-spring-15/chapter17/build.gradle
compile (web.tomcatWsApi) {
    exclude module: 'tomcat-embed-core'
  }
  compile (web.tomcatWsEmbed) {
    exclude module: 'tomcat-embed-core'
  }
  compile spring.context, spring.webSocket,
          spring.messaging, spring.webmvc,
          web.websocket, misc.slf4jJcl,
          misc.logback, misc.lang3, web.jacksonDatabind
```

Ниже приведено содержимое файла конфигурации `WEB-INF/web.xml`, которое позволяет использовать протокол WebSocket вместе со стандартным сервлетом диспетчера из Spring MVC.

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi=
               "http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
             "http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
```

```
        version="3.0">

    <display-name>Spring WebSocket API Sample</display-name>

    <servlet>
      <servlet-name>websocket</servlet-name>
      <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
      </servlet-class>
      <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/root-context.xml
        </param-value>
      </init-param>
      <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
      <servlet-name>websocket</servlet-name>
      <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Сначала в приведенной выше конфигурации определяется сервлет с помощью класса `DispatcherServlet` из Spring, для которого предоставляется файл конфигурации (по пути `/WEB-INF/spring/root-context.xml`). Затем определяется сопоставление с сервлетом, где указывается, что все запросы должны проходить через данный сервлет диспетчера.

А теперь создадим файл `root-context.xml`, содержащий конфигурацию протокола WebSocket, как показано ниже.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi=
            "http://www.w3.org/2001/XMLSchema-instance"
       xmlns:websocket=
          "http://www.springframework.org/schema/websocket"
       xmlns:mvc=
            "http://www.springframework.org/schema/mvc"
       xsi:schemaLocation=
          "http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans
                 /spring-beans.xsd
           http://www.springframework.org/schema/websocket
           http://www.springframework.org/schema/websocket
                 /spring-websocket.xsd
           http://www.springframework.org/schema/mvc
           http://www.springframework.org/schema/mvc
                 /spring-mvc.xsd">
```

```
    <websocket:handlers>
      <websocket:mapping path="/echoHandler"
                handler="echoHandler"/>
    </websocket:handlers>

    <mvc:default-servlet-handler/>

    <mvc:view-controller path= "/"
         view-name="/static/index.html" />

    <bean id="echoHandler"
       class="com.apress.prospring5.ch17.EchoHandler"/>
</beans>
```

Сначала в приведенной выше конфигурации определяется статический ресурс `index.html`. Этот файл содержит статическую HTML-разметку и сценарий JavaScript, предназначенный для взаимодействия со службой WebSocket на стороне сервера. Затем в пространстве имен `websocket` определяются обработчики и соответствующий компонент Spring Bean для обработки запросов. В данном примере определяется единственное сопоставление с обработчиком, получающим запросы по пути `/echoHandler` и применяющим компонент Spring Bean с идентификатором `echoHandler` для получения отдельного сообщения и отправки в ответ этого же сообщения обратно клиенту.

Приведенная выше конфигурация может показаться едва ли знакомой вам, поскольку конфигурирование в формате XML редко употреблялось в примерах из данной книги. Поэтому перейдем к конфигурированию на языке Java. Ниже приведена соответствующая конфигурация для модуля Spring MVC.

```
package com.apress.prospring5.ch17.config;

import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.web.servlet.config.annotation.*;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = {"com.apress.prospring5.ch17"})
public class WebConfig implements WebMvcConfigurer {

  // <=> <mvc:default-servlet-handler/>
  @Override
  public void configureDefaultServletHandling(
              DefaultServletHandlerConfigurer configurer) {
    configurer.enable();
  }
```

```
  // <=> <mvc:view-controller .../>
  @Override
  public void addViewControllers(
              ViewControllerRegistry registry) {
   registry.addViewController("/")
           .setViewName("/static/index.html");
  }
}
```

А теперь перейдем к определению класса, заменяющего файл конфигурации `web.xml` для конфигурирования сервлета диспетчера типа `DispatcherServlet`.

```
package com.apress.prospring5.ch17.config;

import org.springframework.web.servlet.support
    .AbstractAnnotationConfigDispatcherServletInitializer;

public class WebInitializer extends
    AbstractAnnotationConfigDispatcherServletInitializer {

  @Override
  protected Class<?>[] getRootConfigClasses() {
    return new Class<?>[] {
      WebSocketConfig.class
    };
  }

  @Override
  protected Class<?>[] getServletConfigClasses() {
    return new Class<?>[] {
      WebConfig.class
    };
  }

  @Override
  protected String[] getServletMappings() {
    return new String[]{"/"};
  }
}
```

Класс `WebConfig` содержит инфраструктуру приложения Spring MVC, а поскольку, выполняя конфигурирование на языке Java, приходится соблюдать принцип разделения обязанностей, то для поддержки обмена данными по сетевому протоколу WebSocket следует определить другой конфигурационный класс. Этот класс реализует интерфейс `WebSocketConfigurer`, в котором определяются методы обратного вызова для конфигурирования обработки запросов по сетевому протоколу WebSocket.

```
package com.apress.prospring5.ch17.config;

import com.apress.prospring5.ch17.EchoHandler;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.web.socket.config.annotation
          .EnableWebSocket;
import org.springframework.web.socket.config.annotation
          .WebSocketConfigurer;
import org.springframework.web.socket.config.annotation
          .WebSocketHandlerRegistry;

@Configuration
@EnableWebSocket
public class WebSocketConfig
       implements WebSocketConfigurer {
  @Override
  public void registerWebSocketHandlers(
              WebSocketHandlerRegistry registry) {
    registry.addHandler(echoHandler(), "/echoHandler");
  }

  @Bean
  public EchoHandler echoHandler() {
    return new EchoHandler();
  }
}
```

Аннотацию `@EnableWebSocket` пришлось ввести в приведенный выше класс для конфигурирования обработки запросов по сетевому протоколу WebSocket. Теперь все готово для реализации подкласса, производного от класса `TextWebSocketHandler` (см. исходный файл `src/main/java/com/apress/prospring5/ch17/EchoHandler.java`), чтобы сделать удобной обработку символьных сообщений в формате `String`, как показано ниже.

```
package com.apress.prospring5.ch17;

import org.springframework.web.socket.TextMessage;
import org.springframework.web.socket.WebSocketSession;
import org.springframework.web.socket.handler
          .TextWebSocketHandler;
import java.io.IOException;

public class EchoHandler extends TextWebSocketHandler {
  @Override
  public void handleTextMessage(WebSocketSession session,
              TextMessage textMessage) throws IOException {
    session.sendMessage(new TextMessage(
                        textMessage.getPayload()));
  }
}
```

Как видите, здесь определен элементарный обработчик, принимающий сообщение и отправляющий его обратно клиенту без изменений. Содержимое сообщения, полученного по сетевому протоколу WebSocket, находится в методе `getPayload()`.

Мы сделали практически все, что требуется на стороне сервера. Учитывая, что класс `EchoHandler` реализует типичный компонент Spring Bean, теперь можно сделать все, что допускается в обычном приложении Spring, в том числе внедрить службы, чтобы выполнить любые функции, которые могут потребоваться в данном обработчике.

Теперь создадим простой клиент для взаимодействия со службой WebSocket на стороне сервера. Этот клиент будет представлять собой обычную HTML-страницу, дополненную небольшим сценарием JavaScript, устанавливающим соединение со службой WebSocket через прикладной интерфейс API браузера, а также функциями из библиотеки jQuery для обработки событий от щелчков на экранных кнопках и отображения данных. Клиентское приложение будет иметь возможность устанавливать и разрывать соединение со службой WebSocket, отправлять сообщение и выводить обновленное состояние на экран. Код разметки клиентской страницы (из файла `src/main/webapp/static/index.html`) приведен ниже.

```
<html>
<head>
    <meta charset="UTF-8">
    <title>WebSocket Tester</title>
    <script language="javascript" type="text/javascript"
          src="http://code.jquery.com/jquery-2.1.1.min.js">
    </script>
    <script language="javascript" type="text/javascript">
      var ping;
      var websocket;

      jQuery(function ($) {
        function writePing(message) {
          $('#pingOutput').append(message + '\n');
        }

        function writeStatus(message) {
          $("#statusOutput").val($("#statusOutput")
                              .val() + message + '\n');
        }

        function writeMessage(message) {
          $('#messageOutput').append(message + '\n')
        }

        $('#connect').click(function doConnect() {
          websocket = new WebSocket($("#target").val());
          websocket.onopen = function (evt) {
            writeStatus("CONNECTED");
```

```
        var ping = setInterval(function () {
          if (websocket != "undefined") {
            websocket.send("ping");
          }
        }, 3000);
      };

      websocket.onclose = function (evt) {
        writeStatus("DISCONNECTED");
      };

      websocket.onmessage = function (evt) {
        if (evt.data === "ping") {
          writePing(evt.data);
        } else {
          writeMessage('ECHO: ' + evt.data);
        }
      };

      websocket.onerror = function (evt) {
        onError(writeStatus('ERROR:' + evt.data))
      };
    });

    $('#disconnect').click(function () {
      if (typeof websocket != 'undefined') {
        websocket.close();
        websocket = undefined;
      } else {
        alert("Not connected.");
      }
    });

    $('#send').click(function () {
      (typeof websocket != 'undefined') {
        websocket.send($('#message').val());
      } else {
        alert("Not connected.");
      }
    });
  });
  </script>
</head>

<body>
  <h2>WebSocket Tester</h2> Target:
  <input id="target" size="40"
    value="ws://localhost:8080/websocket-api
              /echoHandler"/>
```

```
    <br/>
    <button id="connect">Connect</button>
    <button id="disconnect">Disconnect</button>
    <br/>
    <br/>Message:
      <input id="message" value=""/>
      <button id="send">Send</button>
    <br/>
    <p>Status output:</p>
    <pre><textarea id="statusOutput" rows="10"
                cols="50"></textarea></pre>
    <p>Message output:</p>
    <pre><textarea id="messageOutput" rows="10"
          cols="50"></textarea></pre>
    <p>Ping output:</p>
    <pre><textarea id="pingOutput" rows="10" cols="50">
    </textarea></pre>
</body>
</html>
```

В приведенном выше коде представлен пользовательский интерфейс, который позволяет производить обратные вызовы прикладного интерфейса WebSocket API и наблюдать в реальном времени получаемые результаты на экране. Постройте сначала рассматриваемый здесь проект и разверните его в веб-контейнере. Затем перейдите по ссылке `http://localhost:8080/websocket-api/index.html`, чтобы отобразить пользовательский интерфейс. Щелкнув на экранной кнопке **Connect** (Подключиться) в текстовой области **Status Output** (Вывод состояния), вы обнаружите сообщение `"CONNECTED"` (Подключено), а в текстовой области **Ping Output** (Вывод результатов тестового опроса) каждые 3 секунды будет отображаться сообщение тестового опроса. Введите сообщение в текстовом поле **Message** (Сообщение) и щелкните на экранной кнопке **Send** (Отправить). Это сообщение будет отправлено службе WebSocket на стороне сервера и отображено в текстовой области **Message Output** (Вывод сообщений). Завершив отправлять сообщения, щелкните на экранной кнопке **Disconnect** (Отключиться), после чего в текстовой области **Status Output** появится сообщение `"DISCONNECTED"` (Отключено). Но вам не удастся отправить больше никаких других сообщений или отключиться до тех пор, пока вы не подключитесь снова к службе WebSocket.

Несмотря на то что в данном примере задействована абстракция Spring, определенная над низкоуровневым прикладным интерфейсом WebSocket API, это обстоятельство не мешает ясно видеть те превосходные возможности, которые данная технология способна привнести в веб-приложения. А теперь выясним, как предоставить такие же функциональные возможности, когда браузер не поддерживает сетевой протокол WebSocket, а следовательно, потребуется запасной вариант. Проверить свой браузер на совместимость с сетевым протоколом WebSocket можно, например, с помощью веб-сайта, доступного по адресу `www.websocket.org/echo.html`.

Применение протокола SockJS

Вследствие того, что не все браузеры поддерживают сетевой протокол WebSocket, но веб-приложение должно правильно взаимодействовать с любым конечным пользователем, в Spring Framework предоставляется запасной вариант, предусматривающий применение сетевого протокола SockJS. Благодаря этому обеспечивается максимально сходное с протоколом WebSocket поведение во время выполнения без необходимости вносить коррективы в прикладной код.

Протокол SockJS применяется на стороне клиента с помощью специальных библиотек JavaScript, а модуль `spring-websocket` из каркаса Spring Framework содержит соответствующие компоненты SockJS для применения на стороне сервера. Когда в качестве удобного запасного варианта применяется протокол SockJS, клиент сначала отправит серверу запрос по методу доступа `GET` и по пути `/info`, чтобы получить сведения о транспорте от сервера. Сначала протокол SockJS попробует воспользоваться протоколом WebSocket, а затем режимом потоковой передачи по сетевому протоколу HTTP и, наконец, длительным опросом по тому же самому протоколу как крайним средством. Дополнительные сведения о сетевом протоколе SockJS и связанных с ним проектах можно найти по адресу `https://github.com/sockjs`.

Активизация поддержки протокола SockJS через пространство имен `websocket` осуществляется очень просто и требует только дополнительной директивы в блоке дескрипторов разметки `<websocket:handlers>`. Построим в качестве примера веб-приложение, подобное рассмотренному ранее, но обращающемуся к сетевому протоколу SockJS для обмена сообщениями вместо низкоуровневого прикладного интерфейса WebSocket API. В данном случае содержимое файла конфигурации `src/main/webapp/WEB-INF/spring/root-context.xml` выглядит следующим образом:

```
<beans ...>

    <websocket:handlers>
      <websocket:mapping path="/echoHandler"
                 handler="echoHandler"/>
      <websocket:sockjs/>
    </websocket:handlers>

    <mvc:default-servlet-handler/>

    <mvc:view-controller path= "/"
         view-name="/static/index.html" />

   <bean id="echoHandler"
      class="com.apress.prospring5.ch17.EchoHandler"/>
</beans>
```

Обратите внимание на дополнение приведенной выше разметки дескриптором `<websocket:sockjs>`. На самом элементарном уровне это все, что требуется для активизации сетевого протокола SockJS. В рассматриваемом здесь примере можно

снова воспользоваться классом `EchoHandler` из предыдущего примера применения прикладного интерфейса WebSocket API, чтобы предоставить те же самые функциональные возможности.

В дескрипторе разметки пространства имен `<websocket:sockjs/>` поддерживаются и другие атрибуты, предназначенные для установки таких параметров конфигурации, как обработка сеансовых cookie-файлов (по умолчанию она включена), местоположение для загрузки специальных клиентских библиотек (на момент написания данной книги стандартным считалось следующее местоположение: `https://d1fxtkz8shb9d2.cloudfront.net/sockjs-0.3.4.min.js`), конфигурация опроса состояния, предельные размеры сообщений и т.д. Эти параметры придется просмотреть и настроить должным образом в зависимости от конкретных потребностей веб-приложения и видов транспорта данных. Чтобы отразить наличие сервлета SockJS, в файл конфигурации достаточно ввести немного дополнений, как показано ниже.

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi=
              "http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
            "http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0">

    <display-name>Spring SockJS API Sample</display-name>

    <servlet>
      <servlet-name>sockjs</servlet-name>
      <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
      </servlet-class>
      <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/root-context.xml
        </param-value>
      </init-param>
      <load-on-startup>1</load-on-startup>
      <async-supported>true</async-supported>
    </servlet>

    <servlet-mapping>
      <servlet-name>sockjs</servlet-name>
      <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Как и следовало ожидать, далее требуется выполнить конфигурирование на языке Java. Для поддержки обмена данными по сетевому протоколу WebSocket с помощью протокола SockJS необходимо внести в конфигурацию две коррективы. Во-первых,

поддержать асинхронный обмен сообщениями, активизированный в приведенной выше конфигурации с помощью дескриптора разметки <async-supported>true </asyncsupported>. Для этого достаточно снабдить аннотацией @EnableAsync конфигурационный класс Java, уже снабженный аннотацией @Configuration. Если обратиться к официальной документации на Spring в формате Javadoc, то в ней можно обнаружить, что она активизирует режим выполнения асинхронных методов, а следовательно, и асинхронную обработку под управлением аннотаций для контекста приложения Spring в целом[3].

```
package com.apress.prospring5.ch17.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.annotation
          .EnableAsync;
import org.springframework.web.servlet.config.annotation
          .DefaultServletHandlerConfigurer;
import org.springframework.web.servlet.config.annotation
          .EnableWebMvc;
import org.springframework.web.servlet.config.annotation
          .ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation
          .WebMvcConfigurer;

@Configuration
@EnableWebMvc
@EnableAsync
@ComponentScan(basePackages =
               {"com.apress.prospring5.ch17"})
public class WebConfig implements WebMvcConfigurer {

  @Override
  public void configureDefaultServletHandling(
    DefaultServletHandlerConfigurer configurer) {
    configurer.enable();
  }

  @Override
  public void addViewControllers(
             ViewControllerRegistry registry) {
    registry.addViewController("/")
            .setViewName("/static/index.html");
  }
}
```

[3] Подробнее об этом см. раздел официальной документации на Spring в формате Javadoc, доступный по адресу https://docs.spring.io/spring/docs/current/javadocapi/org/springframework/scheduling/annotation/EnableAsync.html.

И, во-вторых, в классе `WebSocketConfig` необходимо активизировать поддержку протокола SockJS для обработчика запросов, как показано ниже.

```
package com.apress.prospring5.ch17.config;

import com.apress.prospring5.ch17.EchoHandler;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.messaging.simp.config
          .MessageBrokerRegistry;
import org.springframework.web.socket.config.annotation.*;

@Configuration
@EnableWebSocket
public class WebSocketConfig
       implements WebSocketConfigurer {

  @Override
  public void registerWebSocketHandlers(
              WebSocketHandlerRegistry registry) {
    registry.addHandler(echoHandler(), "/echoHandler")
            .withSockJS();
  }

  @Bean
  public EchoHandler echoHandler() {
    return new EchoHandler();
  }
}
```

Далее необходимо создать HTML-страницу, как и в предыдущем примере применения прикладного интерфейса WebSocket API, но на этот раз задействовать сетевой протокол SockJS, чтобы согласовать порядок транспорта данных. Наиболее заметные отличия состоят в применении библиотеки SockJS вместо непосредственного обращения к протоколу WebSocket и употреблении типичной схемы **`http://`** вместо **`ws://`** для подключения к конечной точке, как демонстрируется в приведенном ниже простом коде HTML-разметки клиентской страницы.

```
html>
<head>
  <meta charset="UTF-8">
  <title>SockJS Tester</title>
  <script language="javascript" type="text/javascript"
    src="https://d1fxtkz8shb9d2.cloudfront.net
                /sockjs-0.3.4.min.js">
  </script>
  <script language="javascript" type="text/javascript"
    src="http://code.jquery.com/jquery-2.1.1.min.js">
```

```
</script>
<script language="javascript" type="text/javascript">
  var ping;
  var sockjs;

  jQuery(function ($) {
    function writePing(message) {
      $('#pingOutput').append(message + '\n');
    }

    function writeStatus(message) {
      $("#statusOutput").val($("#statusOutput").val()
        + message + '\n');
    }

    function writeMessage(message) {
      $('#messageOutput').append(message + '\n')
    }

    $('#connect').click(function doConnect() {
      sockjs = new SockJS($("#target").val());
      sockjs.onopen = function (evt) {
        writeStatus("CONNECTED");
        var ping = setInterval(function () {
          if (sockjs != "undefined") {
            sockjs.send("ping");
          }
        }, 3000);
      };

      sockjs.onclose = function (evt) {
        writeStatus("DISCONNECTED");
      };

      sockjs.onmessage = function (evt) {
        if (evt.data === "ping") {
          writePing(evt.data);
        } else {
          writeMessage('ECHO: ' + evt.data);
        }
      };

      sockjs.onerror = function (evt) {
        onError(writeStatus('ERROR:' + evt.data))
      };
    });

    $('#disconnect').click(function () {
      if(typeof sockjs != 'undefined') {
```

```
          sockjs.close();
          sockjs = undefined;
        } else {
          alert("Not connected.");
        }
      });

      $('#send').click(function () {
        if(typeof sockjs != 'undefined') {
          sockjs.send($('#message').val());
        } else {
          alert("Not connected.");
        }
      });
    });
  </script>
</head>
<body>
<h2>SockJS Tester</h2>
    Target:
    <input id="target" size="40"
      value="http://localhost:8080/sockjs/echoHandler"/>
    <br/>
    <button id="connect">Connect</button>
    <button id="disconnect">Disconnect</button>
    <br/>
    <br/>Message:
    <input id="message" value=""/>
    <button id="send">Send</button>
    <br/>
    <p>Status output:</p>
    <pre><textarea id="statusOutput" rows="10" cols="50">
    </textarea></pre>
    <p>Message output:</p>
    <pre><textarea id="messageOutput" rows="10" cols="50">
    </textarea></pre>
    <p>Ping output:</p>
    <pre><textarea id="pingOutput" rows="10" cols="50">
    </textarea></pre>
</body>
</html>
```

Реализовав новый код для применения протокола SockJS, постройте данный проект снова и разверните его в веб-контейнере, а затем передите к пользовательскому интерфейсу, доступному по ссылке `http://localhost:8080/sockjs/index.html` и обладающему всеми теми же характеристиками и функциональными возможностями, что и пользовательский интерфейс из предыдущего примера применения протокола WebSocket. Чтобы проверить запасные функциональные возможности примене-

ния протокола SockJS, отключите поддержку сетевого протокола WebSocket в своем браузере. Так, в браузере Firefox перейдите на страницу `about:config` и найдите настройку `network.websocket.enabled`. Переключите ее на логическое значение `false`, перезагрузите пользовательский интерфейс из данного примера и снова установите соединение с сервером. С помощью такого инструментального средства, как Live HTTP Headers, можно проанализировать сетевой трафик между браузером и сервером. После проверки поведения переключите настройку `network.websocket.enabled` в браузере Firefox обратно на логическое значение `true`, перезагрузите клиентскую страницу и снова установите соединение с сервером. Понаблюдав за сетевым трафиком с помощью инструментального средства Live HTTP Headers, вы должны обнаружить подтверждение установленной связи по сетевому протоколу WebSocket. В данном простом примере все должно работать так, как если бы применялся прикладной интерфейс WebSocket API.

Отправка сообщений по протоколу STOMP

При обращении к сетевому протоколу WebSocket в качестве общего формата для обмена данными между клиентом и сервером обычно применяется такой подпротокол, как STOMP, поэтому обеим сторонам соединения должно быть известно, чего следует ожидать и каким образом реагировать. Подпротокол STOMP изначально поддерживается в Spring Framework, и он будет использован в рассматриваемом далее примере.

Простой протокол обмена сообщениями STOMP, опирающийся на кадры и моделируемый на основе сетевого протокола HTTP, может применяться через любой надежный двунаправленный потоковый сетевой протокол вроде WebSocket. У протокола STOMP имеется стандартный формат и поддержка сценариев JavaScript на стороне клиента для отправки и получения сообщений в браузере, а также для подключения к традиционным брокерам сообщений, поддерживающим протокол STOMP (например, RabbitMQ и ActiveMQ). Изначально в Spring Framework поддерживается простой брокер сообщений, обрабатывающий запросы на подписку и широковещательную рассылку сообщений подключенным клиентам в оперативной памяти. В рассматриваемом далее примере мы задействуем простой брокер сообщений, оставив вам настройку полнофункционального брокера сообщений в качестве упражнения[4].

На заметку Полное описание протокола STOMP доступно по адресу **`http://stomp.github.io/stomp-specification-1.2.html`**.

В рассматриваемом здесь примере применения сетевого протокола STOMP будет создано простое веб-приложение для биржевых котировок, отображающее ряд зара-

---

[4] Подробнее об этом см. по адресу `http://docs.spring.io/spring/docs/current/spring-framework-reference/html/websocket.html#websocket-stomp-handle-broker-relay-configure`.

нее определенных символов акций, текущие цены на них и отметки времени изменения цен. В пользовательском интерфейсе данного веб-приложения можно будет вводить новые символы акций и начальные цены на них. Любые подключающиеся клиенты (т.е. другие браузеры в сети или совершенно новые клиенты из других сетей) будут видеть одни и те же данные при условии, что они подписаны на широковещательные рассылки сообщений. Ежесекундно цена на каждую акцию будет обновляться новым случайным значением вместе с обновлением соответствующей отметки времени.

Чтобы обеспечить возможность взаимодействия клиентов с веб-приложением биржевых котировок, даже если их браузеры не поддерживают сетевой протокол WebSocket, мы снова прибегнем к протоколу SockJS для прозрачного переключения на любой другой вид транспорта данных. Прежде чем обратиться непосредственно к прикладному коду, стоит заметить, что поддержка обмена сообщениями по сетевому протоколу STOMP обеспечивается в библиотеке `spring-messaging`.

А теперь создадим объект предметной области типа `Stock`, в котором будут храниться сведения об акции (ее код и цена), как показано ниже.

```
package com.apress.prospring5.ch17;

import java.util.Date;
import java.io.Serializable;
import java.text.DateFormat;
import java.text.SimpleDateFormat;

public class Stock implements Serializable {
  private static final long serialVersionUID = 1L;
  private static final String DATE_FORMAT =
          "MMM dd yyyy HH:mm:ss";

  private String code;
  private double price;
  private Date date = new Date();
  private DateFormat dateFormat =
          new SimpleDateFormat(DATE_FORMAT);

  public Stock() { }

  public Stock(String code, double price) {
    this.code = code;
    this.price = price;
  }

  // Методы получения и установки
  ...
}
```

Далее необходимо внедрить контроллер MVC для обработки входящих запросов:

```
package com.apress.prospring5.ch17;

import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.messaging.handler.annotation
          .MessageMapping;
import org.springframework.messaging.simp
          .SimpMessagingTemplate;
import org.springframework.scheduling.TaskScheduler;
import org.springframework.stereotype.Controller;

import javax.annotation.PostConstruct;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.Random;

@Controller
public class StockController {
  private TaskScheduler taskScheduler;
  private SimpMessagingTemplate simpMessagingTemplate;

  private List<Stock> stocks = new ArrayList<Stock>();
  private Random random =
          new Random(System.currentTimeMillis());

  public StockController() {
    stocks.add(new Stock("VMW", 1.00d));
    stocks.add(new Stock("EMC", 1.00d));
    stocks.add(new Stock("GOOG", 1.00d));
    stocks.add(new Stock("IBM", 1.00d));
  }

  @MessageMapping("/addStock")
  public void addStock(Stock stock) throws Exception {
    stocks.add(stock);
    broadcastUpdatedPrices();
  }

  @Autowired
  public void setSimpMessagingTemplate(
         SimpMessagingTemplate simpMessagingTemplate) {
    this.simpMessagingTemplate = simpMessagingTemplate;
  }

  @Autowired
  public void setTaskScheduler(TaskScheduler
              taskScheduler) {
    this.taskScheduler = taskScheduler;
```

```
  }

  private void broadcastUpdatedPrices() {
    for(Stock stock : stocks) {
      stock.setPrice(stock.getPrice()
            + (getUpdatedStockPrice() * stock.getPrice()));
      stock.setDate(new Date());
    }
    simpMessagingTemplate.convertAndSend(
                          "/topic/price", stocks);
  }

  private double getUpdatedStockPrice() {
    double priceChange = random.nextDouble() * 5.0;
    if (random.nextInt(2) == 1) {
      priceChange = -priceChange;
    }
    return priceChange / 100.0;
  }

  @PostConstruct
  private void broadcastTimePeriodically() {
    taskScheduler.scheduleAtFixedRate(new Runnable() {
      @Override
      public void run() {
        broadcastUpdatedPrices();
      }
    }, 1000);
  }
}
```

В приведенном выше контроллере выполняется ряд действий. Сначала в целях демонстрации в список вводится несколько заранее определенных символов акций, а также их начальные цены. Затем определяется метод `addStock()`, принимающий объект типа `Stock`, который вводится в список символов акций и производится широковещательная рассылка котировок акций на бирже всем подписчикам. В ходе широковещательной рассылки котировок перебираются все введенные в список символы акций, обновляются цены на каждую из них и сведения о них отправляются всем подписчикам по следующему URL: `/topic/price`, используя привязанный объект типа `SimpMessagingTemplate`. Кроме того, для непрерывной широковещательной рассылки подписанным клиентам обновленного списка цен на акции каждую секунду здесь применяется экземпляр типа `TaskExecutor`.

Располагая контроллером, можно теперь создать HTML-разметку (в файле `src/main/webapp/static/index.html`) пользовательского интерфейса, предназначенного для отображения клиентам, как показано ниже.

```
<html>
<head>
    <title>Stock Ticker</title>
    <script src="https://d1fxtkz8shb9d2.cloudfront.net
                        /sockjs-0.3.4.min.js"/>
    <script src="http://cdnjs.cloudflare.com/ajax/libs
                       /stomp.js/2.3.2/stomp.min.js"/>
    <script src="http://code.jquery.com/jquery-2.1.1.min.js"/>
    <script>
      var stomp = Stomp.over(new SockJS("/stomp/ws"));

      function displayStockPrice(frame) {
        var prices = JSON.parse(frame.body);

        $('#price').empty();

        for (var i in prices) {
          var price = pricesi;

          $('#price').append(
                  $('<tr>').append(
                    $('<td>').html(price.code),
                    $('<td>').html(price.price.toFixed(2)),
                    $('<td>').html(price.dateFormatted)
                  )
          );
        }
      }

      var connectCallback = function () {
        stomp.subscribe('/topic/price', displayStockPrice);
      };

      var errorCallback = function (error) {
        alert(error.headers.message);
      };

      stomp.connect("guest", "guest",
                    connectCallback, errorCallback);
      $(document).ready(function () {
      $('.addStockButton').click(function (e) {
        e.preventDefault();
        var jsonstr = JSON.stringify({
            'code': $('.addStock .code').val(),
            'price': Number($('.addStock .price').val()) });

            stomp.send("/app/addStock", {}, jsonstr);
            return false;
         });
      });
```

```
    </script>
</head>
<body>
<h1><b>Stock Ticker</b></h1>
<table border="1">
    <thead>
    <tr>
      <th>Code</th>
      <th>Price</th>
      <th>Time</th>
    </tr>
    </thead>
<tbody id="price"></tbody>
</table>
<p class="addStock">
     Code: <input class="code"/><br/>
     Price: <input class="price"/><br/>
    <button class="addStockButton">Add Stock</button>
</p>
</body>
</html>
```

Как и в предыдущих примерах, в данном примере HTML-разметка сочетается с исходным кодом JavaScript для обновления информации, отображаемой на экране[5]. В частности, для обновления данных в формате HTML здесь применяется библиотека jQuery, для выбора вида транспорта — протокол SockJS, а для взаимодействия с сервером — библиотека `stomp.js`, обеспечивающая поддержку протокола STOMP на языке JavaScript. Данные, отправляемые в сообщениях по протоколу STOMP, кодируются в формате JSON и затем извлекаются по событиям. После соединения по протоколу STOMP производится подписка на веб-ресурс `/topic/price` для получения обновленных цен на акции.

Сконфигурируем далее встроенный брокер сообщений по протоколу STOMP в файле `src/main/webapp/WEB-INF/spring/root-context.xml`, содержимое которого приведено ниже.

```
<beans ...">

    <mvc:annotation-driven />

    <mvc:default-servlet-handler/>

    <mvc:view-controller path= "/"
         view-name="/static/index.html" />
```

[5] Сочетание HTML-разметки со сценарием JavaScript объясняется стремлением сделать конфигурацию в Spring MVC как можно более простой, хотя это и не отвечает разумным нормам практики программирования.

```
    <context:component-scan
            base-package="com.apress.prospring5.ch17" />

    <websocket:message-broker
              application-destination-prefix="/app">
      <websocket:stomp-endpoint path="/ws">
          <websocket:sockjs/>
      </websocket:stomp-endpoint>
      <websocket:simple-broker prefix="/topic"/>
    </websocket:message-broker>

    <bean id="taskExecutor"
      class="org.springframework.core.task
                .SimpleAsyncTaskExecutor"/>
</beans>
```

Большая часть приведенной выше конфигурации должна быть вам уже знакома. В данном примере конфигурируется брокер сообщений `message-broker` в пространстве имен `websocket`, определяется конечная точка соединения по протоколу STOMP и активизируется протокол SockJS. Здесь также задается префикс, которым подписчики будут пользоваться для получения сообщений. Сконфигурированный компонент `taskExecutor` служит для предоставления биржевых котировок через заданные промежутки времени в классе контроллера. Когда применяется поддержка пространства имен, объект типа `SimpMessagingTemplate` создается автоматически и доступен для внедрения в компоненты Spring Beans.

Осталось лишь выполнить конфигурацию в файле `src/main/webapp/WEB-INF/web.xml`, как показано ниже.

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi=
              "http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
            "http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0">

    <display-name>Spring STOMP Sample</display-name>

    <servlet>
        <servlet-name>stomp</servlet-name>
        <servlet-class>
          org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <init-param>
           <param-name>contextConfigLocation</param-name>
           <param-value>/WEB-INF/spring/root-context.xml
           </param-value>
        </init-param>
```

```
        <load-on-startup>1</load-on-startup>
        <async-supported>true</async-supported>
    </servlet>
    <servlet-mapping>
    <servlet-name>stomp</servlet-name>
       <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Представив конфигурацию рассматриваемого здесь веб-приложения в формате XML, перейдем к нетрадиционному конфигурированию с помощью классов Java. Чтобы активизировать асинхронные вызовы Spring и выполнение задач, необходимо снабдить конфигурационный класс `WebConfig` аннотацией `@EnableAsync`, а также объявить компонент Spring Bean типа `org.springframework.core.task.TaskExecutor`, как показано ниже.

```
package com.apress.prospring5.ch17.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.task
          .SimpleAsyncTaskExecutor;
import org.springframework.core.task.TaskExecutor;
import org.springframework.scheduling.annotation
          .EnableAsync;
import org.springframework.web.servlet.config.annotation
          .DefaultServletHandlerConfigurer;
import org.springframework.web.servlet.config.annotation
          .EnableWebMvc;
import org.springframework.web.servlet.config.annotation
          .ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation
          .WebMvcConfigurer;

@Configuration
@EnableWebMvc
@EnableAsync
@ComponentScan(basePackages = {"com.apress.prospring5.ch17"})
public class WebConfig implements WebMvcConfigurer {

  // <=> <mvc:default-servlet-handler/>
  @Override
  public void configureDefaultServletHandling(
         DefaultServletHandlerConfigurer configurer) {
    configurer.enable();
  }

  // <=> <mvc:view-controller .../>
```

```
  @Override
  public void addViewControllers(
              ViewControllerRegistry registry) {
    registry.addViewController("/")
            .setViewName("/static/index.html");
  }

  @Bean TaskExecutor taskExecutor() {
    return new SimpleAsyncTaskExecutor();
  }
}
```

Более серьезные коррективы следует внести в класс `WebSocketConfig`. В этом классе необходимо реализовать соответствующий интерфейс из пакета `org.springframework.web.servlet` или же расширить абстрактный класс `AbstractWebSocketMessageBrokerConfigurer`, что поможет решить, какие именно методы фактически требуется реализовать. В данном классе теперь определяются методы для конфигурирования обработки сообщений от клиентов WebSocket с помощью таких простых протоколов обмена сообщениями, как STOMP. Кроме того, данный класс необходимо снабдить другой аннотацией, называемой `@EnableWebSocketMessageBroker`, чтобы активизировать поддерживаемый брокером обмен сообщениями по сетевому протоколу WebSocket, используя для этой цели подпротокол более высокого уровня.

```
package com.apress.prospring5.ch17.config;

import org.springframework.context.annotation
          .Configuration;
import org.springframework.messaging.simp.config
          .MessageBrokerRegistry;
import org.springframework.web.socket.config.annotation
          .AbstractWebSocketMessageBrokerConf
import org.springframework.web.socket.config.annotation
          .EnableWebSocketMessageBroker;
import org.springframework.web.socket.config.annotation
          .StompEndpointRegistry;

@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig extends
             AbstractWebSocketMessageBrokerConfigurer {

  // <=> <websocket:stomp-endpoint ... />
  @Override
  public void registerStompEndpoints(
              StompEndpointRegistry registry) {
    registry.addEndpoint("/ws").withSockJS();
  }
```

```
  // <=> websocket:message-broker../>
  @Override
  public void configureMessageBroker(
              MessageBrokerRegistry config) {
    config.setApplicationDestinationPrefixes("/app");
    config.enableSimpleBroker("/topic");
  }
}
```

Методы, переопределяемые в приведенном выше конфигурационном классе, равнозначны закомментированным элементам XML-разметки. Они служат для конфигурирования конечной точки соединения по протоколу STOMP и брокера сообщений.

Резюме

В этой главе были описаны общие принципы действия сетевого протокола Web Socket, рассмотрена поддержка в Spring Framework низкоуровневого прикладного интерфейса WebSocket API, а также применение сетевого протокола SockJS в качестве запасного варианта для выбора подходящего транспорта данных в зависимости от применяемого клиентом браузера. И, наконец, был представлен сетевой протокол STOMP в качестве подпротокола WebSocket, предназначенного для обмена сообщениями между клиентом и сервером. Все примеры веб-приложений, представленные в этой главе, были снабжены конфигурацией в формате XML и конфигурационными классами Java, поскольку общая тенденция в данной области состоит в том, чтобы полностью отказаться от конфигурирования в формате XML.

В следующей главе будут рассмотрены подпроекты Spring, которые можно использовать в своих приложениях для реализации еще более надежных функциональных возможностей.

ГЛАВА 18

Проекты Spring Batch, Spring Integration, Spring XD и прочие

В этой главе представлен общий обзор нескольких проектов, входящих в состав каркаса Spring, в особенности проектов Spring Batch, Spring Integration, Spring XD, а также наиболее примечательных функциональных средств, внедренных в версии 5 каркаса Spring Framework. Эта глава не призвана служить справочным руководством по всем проектам Spring, а дает лишь минимально необходимые сведения и примеры, позволяющие эффективно приступить к работе. В состав Spring входит намного больше проектов, чем упоминается в этой главе, хотя эти проекты применяются наиболее широко, тогда как другие появились лишь недавно или их начало планируется в будущем. Полный перечень проектов Spring доступен по адресу `http://spring.io/projects`. В этой главе будут, в частности, рассмотрены следующие вопросы.

- **Проект Spring Batch.** В этой части описаны основные принципы пакетной обработки в Spring, включая и то, что она может предложить разработчикам, а также представлена вкратце дополнительная поддержка спецификации JSR-352, начиная с версии Spring Batch 3.0.
- **Проект Spring Integration.** Шаблоны интеграции применяются во многих корпоративных приложениях, и в проекте Spring Integration предоставляется надежная инфраструктура для реализации таких шаблонов. На основе примера пакета в этой части продемонстрировано применение проекта Spring Integration как части рабочего потока для инициирования пакетного задания.
- **Проект Spring XD.** Этот проект связывает вместе многие существующие в Spring проекты, чтобы предоставить единообразную и расширяемую систему для приложений, оперирующих большими массивами данными. В проекте Spring XD предоставляется распределенная система, предназначенная для по-

лучения данных, анализа в реальном времени, пакетной обработки и экспорта данных. На конкретных примерах в этой части показано, как внедрить приложения, разработанные на основе проектов Spring Batch и Spring Integration, используя проект Spring XD и простой код на языке DSL в интерфейсе оболочки.

- **Наиболее примечательные функциональные средства, внедренные в версии Spring 5.** По поводу новых функциональных средств в версии Spring Framework 5 велась обширная дискуссия, но по мере приближения даты официального выпуска этой версии их перечень утвердился окончательно. Помимо таких внутренних возможностей, как перенос кодовой базы в версию Java 8[1], внедрение мостового модуля Commons Logging, названного `spring-jcl` вместо стандартного обозначения модуля Commons Logging[2], для упорядочения процесса протоколирования и автоматического обнаружения Log4j 2.*x*, SLF4J и JUL без всяких дополнительных мостов, внедрения *индекса подходящих компонентов* в качестве альтернативы просмотру путей к классам и многого другого[3], версия Spring 5 отличается рядом других примечательных усовершенствований. Три из них рассматриваются в этой части главы.

  — **Функциональный каркас веб-приложений.** Модуль `spring-webflux` служит дополнением модуля `spring-mvc` и построен на основе реактивного программирования. А поскольку прикладной интерфейс Reactive Streams API официально входит в состав версии Java 9, поддержка потоковой передачи данных в версии Spring Framework 5 построена на основе библиотеки Project Reactor (`http://projectreactor.io/`), реализующей спецификацию прикладного интерфейса Reactive Streams API.

  — **Полная совместимость с версией Java 9.** Версия Spring Framework RC3 стала доступной в июле 2017 года и представлена как полностью протестированная по последней предварительной версии JDK 9. В версии Java 9 появилось немало интересных средств, в том числе модули на основе проекта Jigsaw, новый HTTP-клиент, поддерживающий сетевой протокол HTTP 2 и подтверждение установленной связи по протоколу WebSocket, усовершенствованный прикладной интерфейс API для процессов и синтаксис для таких языковых средств, как оператор `try` с ресурсами, закрытые интерфейс-

---

[1] Разработчики Spring собирались добиться полной совместимости версии Spring 5 с версией Java 9, но поскольку первая из них должна была быть выпущена на 18 месяцев раньше второй, они решили ограничиться переходом на версию Java 8. Впрочем, очередную версию Spring 5.*x* планируется сделать совместимой с Java 9.

[2] Напомним, что в данном проекте сначала не предполагалось применение Commons Logging из-за проблематичного алгоритма обнаружения во время выполнения. Подробнее об этом можно узнать на странице официального справочного руководства по Spring, доступной по адресу `https://docs.spring.io/spring/docs/current/spring-framework-reference/`.

[3] Полный перечень нововведений в версии Spring 5 см. по адресу `https://github.com/spring-projects/spring-framework/wiki/What%27s-New-in-Spring-Framework-5.x`.

ные методы, модель "издатель–подписчик" для реактивного программирования, а также целый ряд новых прикладных интерфейсов API. Перечень изменений и нововведений в версии Java 9 доступен на официальном вебсайте компании Oracle[4], но непосредственное отношение к Spring имеют лишь два из них: модульные функциональные средства JDK и каркас для реактивного программирования.

— **Полная поддержка версии JUnit 5**[5]. Модели программирования и расширения Jupiter из версии JUnit 5 полностью поддерживаются в версии Spring Framework 5, включая параллельное выполнение модульных тестов в среде Spring TestContext Framework.

Любая из перечисленных выше тем заслуживает написания отдельной главы и даже целой книги, поэтому подробно описать каждый упоминаемый здесь проект и предоставляемые им функциональные средства в одной главе просто невозможно. Надеемся, однако, что представленный в этой главе вводный материал и простые примеры применения вызовут у вас интерес к дополнительному изучению всех обсуждаемых здесь тем.

Проект Spring Batch

Проект Spring Batch представляет собой каркас для пакетной обработки и входит в число проектов, реализованных в Spring. Этот легковесный и гибкий каркас предназначен на предоставления разработчикам возможности создавать надежные пакетные приложения, прилагая минимальные усилия. В состав проекта Spring Batch входит целый ряд готовых компонентов для самых разных технологий, и зачастую требующееся пакетное приложение удается построить с помощью только тех компонентов, которые предоставляются в данном проекте.

К типичным функциям пакетных приложений относится ежедневное формирование счетов-фактур, расчета заработной платы и выполнение процессов ETL (Extract, Transform, Load — извлечение, преобразование, загрузка). Несмотря на то что это элементарные примеры, Spring Batch можно применять для организации любого процесса, который нуждается в автоматическом запуске, а не только в упомянутых выше случаях. Как и все другие проекты, Spring Batch построен на основе ядра Spring, и все его функциональные возможности доступны разработчикам приложений.

В общем, пакетное задание выполняется в течение одной или большего количества стадий. На каждой стадии предоставляется возможность выполнить единицу работы или же принять участие в так называемой *порционной обработке*. При такой обработке на каждой стадии применяется интерфейс `ItemReader` для чтения данных в определенной форме, а дополнительно — интерфейс `ItemProcessor` для вы-

[4] См. по адресу `https://docs.oracle.com/javase/9/whatsnew/toc.htm`.

[5] Официальный веб-сайт с документацией на JUnit 5 можно найти по адресу `http://junit.org/junit5/docs/current/user-guide/`.

полнения над данными любых требующихся преобразований и, наконец, интерфейс `ItemWriter` для записи обработанных данных. Для конфигурирования каждой стадии имеются различные атрибуты, в которых задается, например, величина порции (объем обрабатываемых данных в течение транзакции), активизация многопоточного выполнения, пределы пропуска данных и т.д. На уровне отдельных стадий и задания в целом могут использоваться приемники уведомлений о различных событиях, которые происходят в течение всего жизненного цикла пакетного задания (например, перед началом стадии, по завершении стадии, на протяжения сценария порционной обработки и т.д.).

Несмотря на то что большинство заданий можно вполне выполнять в единственном потоке и процессе, в Spring Batch предоставляются также варианты для масштабирования и параллельной обработки заданий. В настоящее время проект Spring Batch предоставляет следующие стандартные возможности для масштабирования.

- **Многопоточные стадии.** Это простейший способ сделать стадию многопоточной. Для этого достаточно вести по своему выбору экземпляр типа `TaskExecutor` в конфигурацию отдельной стадии, после чего каждая порция элементов, указанных при установке порционной обработки, будет обрабатываться в отдельном потоке исполнения.
- **Параллельные стадии.** Допустим, в начале задания необходимо прочитать два крупных файла с разными данными. На первый взгляд, для этого можно было бы сформировать две стадии и выполнять их по очереди. Но если данные, загружаемые из обоих файлов, не зависят друг от друга, то почему бы не обработать их одновременно? В таком случае в Spring Batch можно определить разделение на элементы потока с инкапсуляцией параллельно выполняемых задач.
- **Удаленное разделение на порции.** Такая возможность для масштабирования позволяет удаленно распределить работу среди нескольких рабочих процессов и организовать их взаимодействие с помощью какого-нибудь надежного промежуточного программного обеспечения вроде AMQP или JMS. Удаленное разделение на порции обычно применяется в тех случаях, когда узким местом в процессе оказывается не чтение, а запись и дополнительная обработка порции данных. С помощью промежуточного программного обеспечения порции данных отправляются на обработку подчиненным узлам, которые затем сообщают главному узлу состояние обработки отдельной порции.
- **Разбиение на части.** Такая возможность для масштабирования, как правило, используется в тех случаях, когда требуется обработать данные в определенных пределах, для каждого из которых предусмотрен отдельный поток исполнения. Типичным тому примером может служить таблица базы данных со столбцом, содержащим числовой идентификатор. Разбиение на части позволяет распределить обработку данных по отдельным потокам исполнения с определенным количеством записей. Проект Spring Batch предоставляет разработчикам возможность вмешиваться в такую схему разбиения на части, поскольку

она существенно зависит от конкретного варианта использования. Разбиение на части можно произвести в локальных потоках исполнения или же поручить это удаленным рабочим процессам (что напоминает вариант удаленного разделения на порции).

Один из распространенных вариантов использования пакетной обработки предусматривает чтение файла определенного вида (как правило, плоского файла в текстовом формате с разделителями, например, запятыми), который в дальнейшем необходимо загрузить в базу данных, предварительно обработав каждую запись по мере надобности. Выясним теперь, как реализовать такой вариант в Spring Batch. Для этого придется сначала внедрить обязательные зависимости, как показано в следующем фрагменте кода конфигурации:

```
// Файл конфигурации pro-spring-15/build.gradle
ext {
    // Библиотеки Spring
    ...
    springBatchVersion = '4.0.0.M3'
    ...
    spring = [
       context : "org.springframework:spring-context:
                  $springVersion",
       jdbc : "org.springframework:spring-jdbc:
                $springVersion",
       batchCore : "org.springframework.batch:
                    spring-batch-core:$springBatchVersion"
       ...
    ]
    misc = [
       io : "commons-io:commons-io:2.5",
       ...
    ]
    db = [
       ...
       dbcp2 : "org.apache.commons:commons-dbcp2:
                $dbcpVersion",
       h2 : "com.h2database:h2:$h2Version",

       // требуется для модуля Batch JSR-352
       hsqldb: "org.hsqldb:hsqldb:2.4.0"
       dbcp : "commons-dbcp:commons-dbcp:1.4",
    ]
}
...
// Файл конфигурации pro-spring-15/chapter18/build.gradle
dependencies {
    if (!project.name.contains("boot")) {
       compile(spring.jdbc) {
```

```
            // исключить перечисленные ниже модули,
            // поскольку они будут задействованы в batchCore
            // по транзитивным зависимостям
            exclude module: 'spring-core'
            exclude module: 'spring-beans'
            exclude module: 'spring-tx'
        }
        compile spring.batchCore, db.dbcp2, db.h2, misc.io,
                misc.slf4jJcl, misc.logback
    }
}
```

В приведенной выше конфигурации определяются основные зависимости, требующиеся для внедрения в проект Spring Batch, но не Spring Boot. Именно поэтому здесь проверяется условие `if (!project.name.contains("boot"))`, не позволяющее спутать библиотеки, где явно указана версия, с зависимостями из проектов Spring Boot, рассматриваемых далее в этой главе.

Внедрив зависимости, можно приступить к написанию кода. Создадим сначала объект предметной области, представляющий певца, основываясь на данных из читаемого файла, как показано ниже.

```
package com.apress.prospring5.ch18;

public class Singer {
  private String firstName;
  private String lastName;
  private String song;

  ... // Методы установки и получения
  @Override
  public String toString() {
    return "firstName: " + firstName + ", lastName: "
         + lastName + ", song: " + song;
  }
}
```

Создадим далее реализацию интерфейса `ItemProcessor`, предназначенную для преобразования в верхний регистр Ф.И.О. и песни каждого певца, представленного объектом типа `Singer`:

```
package com.apress.prospring5.ch18;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.batch.item.ItemProcessor;
import org.springframework.stereotype.Component;

@Component("itemProcessor")
public class SingerItemProcessor implements
```

```
      ItemProcessor<Singer, Singer> {
  private static Logger logger =
       LoggerFactory.getLogger(SingerItemProcessor.class);

  @Override
  public Singer process(Singer singer) throws Exception {
    String firstName = singer.getFirstName().toUpperCase();
    String lastName = singer.getLastName().toUpperCase();
    String song = singer.getSong().toUpperCase();

    Singer transformedSinger = new Singer();
    transformedSinger.setFirstName(firstName);
    transformedSinger.setLastName(lastName);
    transformedSinger.setSong(song);

    logger.info("Transformed singer: " + singer + " Into: "
              + transformedSinger);

    return transformedSinger;
  }
}
```

Обратите внимание на то, что для порционной обработки требуются лишь экземпляры типа `ItemReader` и `ItemWriter`, но не `ItemProcessor`. Но в данном примере экземпляр типа `ItemProcessor` служит лишь для того, чтобы показать, как можно обработать данные перед их записью.

После этого создадим реализацию интерфейса `StepExecutionListener`, которая находится на уровне стадии и сообщает, сколько записей было сохранено по завершении стадии, как показано в следующем фрагменте кода:

```
package com.apress.prospring5.ch18;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.batch.core.ExitStatus;
import org.springframework.batch.core.StepExecution;
import org.springframework.batch.core.listener
          .StepExecutionListenerSupport;
import org.springframework.stereotype.Component;

@Component
public class StepExecutionStatsListener extends
             StepExecutionListenerSupport {
  public static Logger logger = LoggerFactory.
  getLogger(StepExecutionStatsListener.class);

  @Override
  public ExitStatus afterStep(StepExecution stepExecution) {
    logger.info("--> Wrote: "
```

```
                + stepExecution.getWriteCount()
                + " items in step: "
                + stepExecution.getStepName());
        return null;
    }
}
```

Кроме того, класс `StepExecutionListener` позволяет видоизменить, если потребуется, возвращаемое значение типа `ExitStatus`. В противном случае следует просто возвратить пустое значение `null`, чтобы оставить данное значение без изменения. Итак, нам удалось связать вместе основные компоненты, но прежде чем перейти к конфигурированию и вызову кода, рассмотрим как модель данных, так и сами данные. Модель данных для рассматриваемого здесь примера задания очень проста (см. файл `src/main/resources/support/test-data.sql`):

```
-- Файл singer.sql
DROP TABLE singer IF EXISTS;

CREATE TABLE singer (
    singer_id BIGINT IDENTITY NOT NULL PRIMARY KEY,
    first_name VARCHAR(20),
    last_name VARCHAR(20),
    song VARCHAR(100)
);

-- Файл test-data.sql
John,Mayer,Helpless
Eric,Clapton,Change The World
John,Butler,Ocean
BB,King,Chains And Things
```

Теперь необходимо создать файл конфигурации Spring Batch, определить задание и установить встроенную базу данных и связанные с ней компоненты задания. В связи с тем что конфигурировать в формате XML в данном случае затруднительно, как пояснялось в предыдущем издании данной книги, основное внимание в этой главе будет уделено только конфигурационным классам Java. В духе благоразумных норм программирования конфигурирование пакетного приложения следует отделить от конфигурирования источника данных, создав для каждого из них отдельный конфигурационный класс. Так, ниже приведен исходный код класса `DataSourceConfig` для конфигурирования источника данных.

```
package com.apress.prospring5.ch18.config;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .Configuration;
```

```
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded
          .EmbeddedDatabaseType;
import javax.sql.DataSource;

@Configuration
public class DataSourceConfig {

  private static Logger logger =
          LoggerFactory.getLogger(DataSourceConfig.class);

  @Bean
  public DataSource dataSource() {
    try {
      EmbeddedDatabaseBuilder dbBuilder =
                      new EmbeddedDatabaseBuilder();
      return dbBuilder.setType(EmbeddedDatabaseType.H2)
                      .addScripts("classpath:"
         + "/org/springframework/batch/core/schema-h2.sql",
           "classpath:support/singer.sql" ).build();
    } catch (Exception e) {
      logger.error("Embedded DataSource bean cannot "
                + "be created!", e);
      return null;
    }
  }
}
```

Приведенная выше конфигурация должна быть вам уже знакома, поэтому поясним лишь содержимое файла `schema-h2.sql`. Этот файл входит в состав библиотеки `spring-batch-core` и содержит операторы языка DML, требующиеся для создания служебных таблиц в Spring Batch. Класс `DataSourceConfig` будет импортирован в класс `BatchConfig`, как показано ниже.

```
package com.apress.prospring5.ch18.config;

import com.apress.prospring5.ch18.Singer;
import com.apress.prospring5.ch18
                .StepExecutionStatsListener;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import org.springframework.batch.core.configuration
          .annotation.EnableBatchProcessing;
import org.springframework.batch.core.configuration
          .annotation.JobBuilderFactory;
import org.springframework.batch.core.configuration
          .annotation.StepBuilderFactory;
import org.springframework.batch.item.ItemProcessor;
```

```
import org.springframework.batch.item.ItemReader;
import org.springframework.batch.item.ItemWriter;
import org.springframework.batch.item.database
          .BeanPropertyItemSqlParameterSourceProvider;
import org.springframework.batch.item.database
          .JdbcBatchItemWriter;
import org.springframework.batch.item.file
          .FlatFileItemReader;
import org.springframework.batch.item.file.mapping
          .BeanWrapperFieldSetMapper;
import org.springframework.batch.item.file.mapping
          .DefaultLineMapper;
import org.springframework.batch.item.file.transform
          .DelimitedLineTokenizer;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.beans.factory.annotation
          .Qualifier;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.core.io.ResourceLoader;
import javax.sql.DataSource;

@Configuration
@EnableBatchProcessing
@Import(DataSourceConfig.class)
@ComponentScan("com.apress.prospring5.ch18")
public class BatchConfig {

  @Autowired
  private JobBuilderFactory jobs;

  @Autowired
  private StepBuilderFactory steps;

  @Autowired DataSource dataSource;

  @Autowired ResourceLoader resourceLoader;
  @Autowired StepExecutionStatsListener
             executionStatsListener;

  @Bean
  public Job job(@Qualifier("step1") Step step1) {
    return jobs.get("singerJob").start(step1).build();
  }
```

```
  @Bean
  protected Step step1(ItemReader<Singer> reader,
            ItemProcessor<Singer,Singer> itemProcessor,
            ItemWriter<Singer> writer) {
    return steps.get("step1")
                .listener(executionStatsListener)
                .<Singer, Singer>chunk(10)
                .reader(reader)
                .processor(itemProcessor)
                .writer(writer)
                .build();
  }

  @Bean
  public ItemReader itemReader() {
    FlatFileItemReader itemReader =
                        new FlatFileItemReader();
    itemReader.setResource(resourceLoader.getResource(
              "classpath:support/test-data.csv"));
    DefaultLineMapper lineMapper = new DefaultLineMapper();

    DelimitedLineTokenizer tokenizer =
                           new DelimitedLineTokenizer();
    tokenizer.setNames("firstName","lastName","song");
    tokenizer.setDelimiter(",");
    lineMapper.setLineTokenizer(tokenizer);

    BeanWrapperFieldSetMapper<Singer> fieldSetMapper =
                        new BeanWrapperFieldSetMapper<>();
    fieldSetMapper.setTargetType(Singer.class);
    lineMapper.setFieldSetMapper(fieldSetMapper);
    itemReader.setLineMapper(lineMapper);
    return itemReader;
  }

  @Bean
  public ItemWriter itemWriter() {
    JdbcBatchItemWriter<Singer> itemWriter =
                         new JdbcBatchItemWriter<>();
    itemWriter.setItemSqlParameterSourceProvider(
       new BeanPropertyItemSqlParameterSourceProvider<>());
    itemWriter.setSql(
          "INSERT INTO singer (first_name, last_name, song)
          VALUES (:firstName, :lastName, :song)");
    itemWriter.setDataSource(dataSource);
    return itemWriter;
  }
}
```

На первый взгляд конфигурационный класс `BatchConfig` кажется довольно крупным, но на самом деле он не настолько велик, как конфигурация, которую пришлось бы составить в формате XML. А теперь поясним каждый компонент Spring Bean, определенный в данном классе.

- Аннотация `@EnableBatchProcessing` действует таким же образом, как и все остальные аннотации типа `@Enable*` в Spring. В этой аннотации предоставляется базовая конфигурация для составления пакетных заданий. Если снабдить конфигурационный класс этой аннотацией, то произойдет следующее.
  - — Получается экземпляр класса `org.springframework.batch.core.scope.StepScope`. Объекты в данной области видимости пользуются контейнером Spring как фабрикой объектов, чтобы на каждую выполняемую стадию приходился лишь один экземпляр соответствующего компонента Spring Bean.
  - — Для автосвязывания становятся доступными следующие компоненты Spring Beans, специально предназначенные для пакетной инфрастуктуры: `jobRepository` (типа `JobRepository`), `jobLauncher` (типа `jobLauncher`), `jobBuilders` (типа `JobBuilderFactory`), `stepBuilders` (типа `StepBuilderFactory`). Это означает, что их совсем не обязательно объявлять явным образом, как это обычно делается в формате XML.
- Компонент `job` представляет пакетное задание `singerJob`, составляемое в результате вызова метода `JobBuilderFactory.get(...)`.
- Компонент `step1` создается в результате вызова метода `StepBuilderFactory.get()` и конфигурируется для порционной обработки. Контейнер Spring автоматически внедрит экземпляры типа `ItemReader`, `ItemProcessor` и `ItemWriter`, обнаруженные в текущем контексте. Но компонент Spring Bean типа `StepExecutionStatsListener` должен быть установлен вручную.
- Компонент `database` служит для объявления компонента типа `ItemWriter`, предназначенного для записи экземпляров типа `Singer` во встроенную базу данных.

И, наконец, для запуска задания потребуется тестовая программа, исходный код которой приведен ниже.

```
package com.apress.prospring5.ch18;

import com.apress.prospring5.ch18.config.BatchConfig;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.context.annotation
          .AnnotationConfigApplicationContext;
import org.springframework.context.support.GenericApplicationContext;
```

```
import java.util.Date;

public class SingerJobDemo {
  public static void main(String... args)
        throws Exception {
    GenericApplicationContext ctx =
          new AnnotationConfigApplicationContext(
                              BatchConfig.class);
    Job job = ctx.getBean(Job.class);
    JobLauncher jobLauncher =
                ctx.getBean(JobLauncher.class);
    JobParameters jobParameters =
                  new JobParametersBuilder()
                 .addDate("date", new Date())
                 .toJobParameters();
    jobLauncher.run(job, jobParameters);

    System.in.read();
    ctx.close();
  }
}
```

Этот код должен быть вам уже знаком, поскольку в нем создается нужный контекст, получается ряд компонентов Spring Beans и вызываются их методы. Здесь следует обратить внимание на объект типа `JobParameters`. Он инкапсулирует параметры, которые служат для различения экземпляров задания. Такое различение заданий требуется для определения последнего состояния задания, которое, среди прочего, имеет значение и для реализации таких возможностей, как, например, перезапуск задания. В данном примере в качестве параметра задания используется текущая дата. В объектах типа `JobParameters` поддерживаются самые разные типы параметров, доступных в задании в виде справочных данных.

Итак, все готово для проверки нового задания. Скомпилируйте и запустите на выполнение исходный код класса `SingerJobDemo`. В итоге на экране появятся протокольные сообщения, самые интересные из которых приведены ниже.

```
o.s.b.c.l.s.SimpleJobLauncher - Job: [SimpleJob:
  [name=singerJob]] launched with the following parameters:[6]
  [{date=1501418591075}]
o.s.b.c.j.SimpleStepHandler - Executing step: [step1][7]
c.a.p.c.SingerItemProcessor - Transformed singer:
  firstName: John, lastName: Mayer, song: Helpless Into:[8]
  firstName: JOHN, lastName: MAYER, song: HELPLESS
c.a.p.c.SingerItemProcessor - Transformed singer:
```

[6] Задание: [SimpleJob: [name=singerJob]] запущено со следующими параметрами:

[7] Выполнение стадии: [*имя_стадии*]

[8] Преобразование сведений о певце:
Имя:... Фамилия: ... Песня: ... в верхний регистр букв

```
  firstName: Eric, lastName: Clapton,
  song: Change The World Into:
  firstName: ERIC, lastName: CLAPTON,
  song: CHANGE THE WORLD
c.a.p.c.SingerItemProcessor - Transformed singer:
  firstName: John, lastName: Butler, song: Ocean Into:
  firstName: JOHN, lastName: BUTLER, song: OCEAN
c.a.p.c.SingerItemProcessor - Transformed singer:
  firstName: BB, lastName: King,
  song: Chains And Things Into:
  firstName: BB, lastName: KING, song: CHAINS AND THINGS
c.a.p.c.StepExecutionStatsListener - -->
  Wrote: 4 items in step: step1[9]
o.s.b.c.l.s.SimpleJobLauncher - Job: [SimpleJob:
  [name=singerJob]] completed with the following parameters:
  [{date=1501418591075}] and the following status: [COMPLETED][10]
```

Вот, собственно, и все, что следовало сделать в данном примере пакетного задания. Используя проект Spring Batch, мы создали простое пакетное задание для чтения данных из файла формата CSV, их преобразования через интерфейс `ItemProcessor` (в частности, Ф.И.О. певца и его песни в верхний регистр букв) и последующей записи результатов в базу данных. А с помощью интерфейса `StepExecutionListener` был организован вывод количества элементов, записанных на отдельной стадии выполнения этого пакетного задания. За дополнительными сведениями о проекте Spring Batch обращайтесь на соответствующую страницу, доступную по адресу `https://spring.io/projects/spring-batch`.

Спецификация JSR-352

Проект Spring Batch оказал значительное влияние на спецификацию JSR-352 (Batch Applications for the Java Platform — пакетные приложения для платформы Java). Если вы решили употребить спецификацию JSR-352 в своей работе, то заметите в ней все больше сходных черт с данным проектом, а следовательно, вам будет удобнее работать с ней, если вы раньше пользовались проектом Spring Batch. Проект Spring Batch и спецификация JSR-352 обладают в основном сходными конструкциями, а, начиная с версии Spring Batch 3.0, спецификация JSR-352 полностью поддерживается в данном проекте. Как и в Spring Batch, задания по спецификации JSR-352 конфигурируются по схеме XML на так называемом языке спецификации заданий (Job Specification Language — JSL). А поскольку в стандарте JSR-352 определяется как спецификация, так и прикладной интерфейс API, то в нем никаких готовых компонентов инфраструктуры не предоставляется, в отличие от ситуации, когда приме-

[9] Записано 4 элемента на стадии: step1

[10] Задание: [SimpleJob: [name=singerJob]] завершено со следующими параметрами и состоянием:

няется проект Spring Batch. Если вы строго придерживаетесь прикладного интерфейса API по спецификации JSR-352, то вам придется самостоятельно реализовать соответствующие интерфейсы и написать все компоненты инфраструктуры, в том числе реализовать интерфейсы `ItemReader` и `ItemWriter`.

Чтобы продемонстрировать применение языка JSL по спецификации JSR-352, переделаем предыдущий пример пакетного задания, но вместо разработки собственных компонентов инфраструктуры задействуем те же самые компоненты типа `ItemReader`, `ItemProcessor` и `ItemWriter`, а также воспользуемся Spring для внедрения зависимостей и т.д. Самостоятельную реализацию задания, которое полностью соответствует спецификации JSR-352, мы оставляем вам в качестве упражнения.

Как упоминалось выше, в данном примере мы воспользуемся большей частью исходного кода из предыдущего примера, разработанного исключительно с помощью проекта Spring Batch, внеся в него небольшие изменения. Если вы этого еще не сделали, то самое время привести в рабочее состояние предыдущий пример пакетного задания, составленного средствами Spring Batch, чтобы внести в него изменения, представленные в этом разделе.

Для целей данного примера встроенная база данных H2 будет заменена на HSQLDB, а компонент DBCP 2 — на DBCP, поскольку в них не поддерживаются более новые версии. Кроме того, согласно нормам JSR, файл конфигурации `singerJob.xml` должен быть объявлен в каталоге `src/main/resources/META-INF/batch-jobs/`, и поэтому для запуска задания на выполнение достаточно указать имя этого файла без расширения **`.xml`**. Таким образом, действуя по спецификации JSR-352, просто невозможно обойтись без конфигурирования в формате XML. Ниже приведено содержимое данного XML-файла.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi=
            "http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jdbc=
            "http://www.springframework.org/schema/jdbc"
       xsi:schemaLocation="
          http://www.springframework.org/schema/jdbc
       http://www.springframework.org/schema/jdbc
             /spring-jdbc.xsd
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans
             /spring-beans.xsd
       http://xmlns.jcp.org/xml/ns/javaee
       http://xmlns.jcp.org/xml/ns/javaee/jobXML_1_0.xsd">

    <job id="singerJob"
         xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         version="1.0">
```

```
      <step id="step1">
        <listeners>
          <listener ref="stepExecutionStatsListener"/>
        </listeners>
        <chunk item-count="10">
          <reader ref="itemReader"/>
          <processor ref="itemProcessor"/>
          <writer ref="itemWriter"/>
        </chunk>
        <fail on="FAILED"/>
        <end on="*"/>
      </step>
    </job>

    <jdbc:embedded-database id="dataSource" type="HSQL">
      <jdbc:script location="classpath:support/singer.sql"/>
    </jdbc:embedded-database>

    <!-- диспетчер транзакций не требуется -->
    <bean id="stepExecutionStatsListener" ../>
    <bean id="itemReader" ../>
    <bean id="itemProcessor" ../>
    <bean id="itemWriter" ../>
</beans>
```

Компоненты Spring Beans, не приведенные выше, определяются таким же образом, как и в предыдущем примере применения проекта Spring Batch. Полное их определение в формате XML можно найти в исходном коде примеров, прилагаемом к данной книге. Но в данном случае без конфигурирования пакетного задания в формате XML просто не обойтись, и поэтому ниже приведена конфигурация предыдущего задания средствами Spring Batch.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi=
            "http://www.w3.org/2001/XMLSchema-instance"
       xmlns:batch=
            "http://www.springframework.org/schema/batch"
       xmlns:jdbc=
            "http://www.springframework.org/schema/jdbc"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="
          http://www.springframework.org/schema/batch
          http://www.springframework.org/schema/batch
                /spring-batch.xsd
          http://www.springframework.org/schema/jdbc
          http://www.springframework.org/schema/jdbc
                /spring-jdbc.xsd
```

```
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans
              /spring-beans.xsd">

    <batch:job id="singerJob">
       <batch:step id="step1">
          <batch:tasklet>
             <batch:chunk reader="itemReader"
                    processor="itemProcessor"
                    writer="itemWriter"
                    commit-interval="10"/>
             <batch:listeners>
               <batch:listener
                      ref="stepExecutionStatsListener"/>
             </batch:listeners>
             </batch:tasklet>
             <batch:fail on="FAILED"/>
             <batch:end on="*"/>
          </batch:step>
    </batch:job>

    <jdbc:embedded-database id="dataSource" type="H2">
    <jdbc:script location="classpath:
          /org/springframework/batch/core/schema-h2.sql"/>
       <jdbc:script location="classpath:support/singer.sql"/>
    </jdbc:embedded-database>

    <batch:job-repository id="jobRepository"/>

    <bean id="jobLauncher"
       class="org.springframework.batch.core.launch
                 .support.SimpleJobLauncher"
       p:jobRepository-ref="jobRepository"/>
    <bean id="transactionManager"
       class="org.springframework.jdbc.datasource
                .DataSourceTransactionManager"
       p:dataSource-ref="dataSource"/>
    <bean id="stepExecutionStatsListener" ../>
    <bean id="itemReader" ../>
    <bean id="itemProcessor" ../>
    <bean id="itemWriter" ../>
</beans>
```

Таким образом, обе приведенные выше конфигурации весьма похожи, за исключением того, что в определении пакетного задания используется язык JSL по спецификации JSR-352, а кроме того, можно удалить несколько компонентов Spring Beans (`transactionManager`, `jobRepository`, `jobLauncher`), поскольку они так или иначе уже предоставлены. Обратите также внимание на определение дополнительной схемы `jobXML_1.0.xsd`. Поддержка этой схемы достигается с помощью архив-

ного JAR-файла прикладного интерфейса API по спецификаци JSR-352 и внедряется автоматически, если применяется инструментальное средство построения вроде Gradle. Если же зависимость необходимо получить вручную, зайдите на страницу проекта, указанную в конце этого раздела. Необходимо также внести изменения в класс `SingerJobDemo`, как показано ниже. Ведь теперь для запуска пакетного задания используется код, соответствующий спецификации JSR-352.

```
package com.apress.prospring5.ch18;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support
          .ClassPathXmlApplicationContext;
import java.util.Date;

public class SingerJobDemo {
  public static void main(String... args)
         throws Exception {
    ApplicationContext applicationContext =
              new ClassPathXmlApplicationContext(
                  "/spring/singerJob.xml");
    Job job = applicationContext.getBean(Job.class);
    JobLauncher jobLauncher =
             applicationContext.getBean(JobLauncher.class);
    JobParameters jobParameters =
                  new JobParametersBuilder()
                 .addDate("date", new Date())
                 .toJobParameters();
    jobLauncher.run(job, jobParameters);
  }
}
```

Приведенный выше код несколько отличается от кода из других примеров, где приходилось работать с контекстом типа `ApplicationContext` и компонентами Spring Beans напрямую. Для запуска пакетного задания по спецификации JSR-352 и управления им в данном случае используется класс `JsrJobOperator`, а для предоставления параметров пакетному заданию — объект типа `Properties` вместо объекта типа `JobParameters`. Применяемый объект типа `Properties` относится к стандартному классу `java.util.Properties`, а параметры пакетного задания должны быть созданы по ключу типа `String` и с соответствующим значением. Еще одно интересное изменение связано с методом `waitForJob()`. В спецификации JSR-352 по умолчанию предусматривается асинхронный запуск всех заданий. Таким образом, прежде чем завершить автономную тестовую программу, придется подождать до тех пор, пока задание не перейдет в приемлемое состояние. Если же прикладной

код выполняется в контейнере вроде сервера приложений, то приведенный выше код может и не понадобиться. Если теперь скомпилировать и запустить на выполнение исходный код класса `SingerJobDemo`, то на консоль будут выведены следующие протокольные сообщения:

```
o.s.b.c.r.s.JobRepositoryFactoryBean -
  No database type set, using meta data indicating: HSQL[11]
o.s.b.c.j.c.x.JsrXmlApplicationContext -
  Refreshing org.springframework.batch.core.jsr.configuration
               .xml.JsrXmlApplicationContext@48c76607
o.s.b.c.j.SimpleStepHandler - Executing step: [step1]
c.a.p.c.SingerItemProcessor -
  Transformed singer: firstName: John, lastName: Mayer,
  song: Helpless Into: firstName: JOHN, lastName: MAYER,
  song: HELPLESS
c.a.p.c.SingerItemProcessor -
  Transformed singer: firstName: Eric, lastName: Clapton,
  song: Change The World Into: firstName: ERIC,
  lastName: CLAPTON, song: CHANGE THE WORLD
c.a.p.c.SingerItemProcessor -
  Transformed singer: firstName: John, lastName: Butler,
  song: Ocean Into: firstName: JOHN, lastName: BUTLER,
  song: OCEAN
c.a.p.c.SingerItemProcessor -
  Transformed singer: firstName: BB, lastName: King,
  song: Chains And Things Into: firstName: BB,
  lastName: KING, song: CHAINS AND THINGS
c.a.p.c.StepExecutionStatsListener - --> Wrote:
  4 items in step: step1
o.s.b.c.j.c.x.JsrXmlApplicationContext -
  Closing ... JsrXmlApplicationContext
```

Этот вывод протокольных сообщений выглядит практически так же, как и прежде, но на этот раз для определения и запуска пакетного задания были использована спецификация JSR-352, а для внедрения зависимостей — функциональные средства Spring. Кроме того, вместо написания собственных компонентов инфраструктуры в данном случае были применены компоненты из проекта Spring Batch. За дополнительными сведениями о спецификации JSR-352 обращайтесь на страницу соответствующего проекта, доступного по адресу `https://jcp.org/en/jsr/detail?id=352`.

Библиотека Spring Boot для запуска Spring Batch

Как и следовало ожидать, в состав модуля Spring Boot входит специальная библиотека для запуска проекта Spring Batch, предназначенная для того, чтобы еще больше упростить конфигурирование приложений. Достоинство такого подхода заключается

[11] Тип базы данных не установлен, с помощью метаданных указывается база данных: HSQL

в том, что достаточно указать стартовую библиотеку Spring Boot для запуска проекта Spring Batch в пути к классам, чтобы избавиться от хлопот, связанных с внедрением зависимостей, а недостаток — в том, что конфигурирование Spring Batch все равно не удается существенно упростить.

Тем не менее попробуем внести изменения в рассматриваемый здесь пример пакетного задания, заменив класс `StepExecutionStatsListener` на `JobExecutionStatsListener`. Этот последний класс расширяет класс `JobExecutionListenerSupport`, представляющий собой абстрактную реализацию интерфейса `JobExecutionListener`, где предоставляются обратные вызовы в отдельных точках жизненного цикла задания. В данном случае из класса `JobExecutionStatsListener` делается запрос базы данных с целью проверить, действительно ли в ней сохранены записи о певцах.

Ниже приведен исходный код класса `JobExecutionStatsListener`.

```
package com.apress.prospring5.ch18;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.batch.core.BatchStatus;
import org.springframework.batch.core.ExitStatus;
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.StepExecution;
import org.springframework.batch.core.listener.JobExecutionListenerSupport;
import org.springframework.batch.core.listener
          .StepExecutionListenerSupport;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Component;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

@Component
public class JobExecutionStatsListener
       extends JobExecutionListenerSupport {
  public static Logger logger = LoggerFactory.getLogger(
                       JobExecutionStatsListener.class);
  private final JdbcTemplate jdbcTemplate;

  @Autowired
  public JobExecutionStatsListener(
         JdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
  }
```

```
  @Override
  public void afterJob(JobExecution jobExecution) {
    if(jobExecution.getStatus() == BatchStatus.COMPLETED) {
      logger.info(" --> Singers were saved to the "
                + "database. Printing results ...");
      jdbcTemplate.query("SELECT first_name, last_name,
                    song FROM SINGER",
                    (rs, row) -> new Singer(rs.getString(1),
                    rs.getString(2),
                    rs.getString(3))).forEach(singer ->
                          logger.info(singer.toString())
      );
    }
  }
}
```

Как видите, в данном классе интенсивно применяются лямбда-выражения ради большей наглядности примера, хотя и совершенно очевидно, что именно происходит в теле метода обратного вызова `afterJob()`. А ниже показано, какие коррективы следует внести в конфигурационный класс `BatchConfig`.

```
package com.apress.prospring5.ch18;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import org.springframework.batch.core.configuration
          .annotation.EnableBatchProcessing;
import org.springframework.batch.core.configuration
          .annotation.JobBuilderFactory;
import org.springframework.batch.core.configuration
          .annotation.StepBuilderFactory;
import org.springframework.batch.item.ItemProcessor;
import org.springframework.batch.item.ItemReader;
import org.springframework.batch.item.ItemWriter;
import org.springframework.batch.item.database
          .BeanPropertyItemSqlParameterSourceProvider;
import org.springframework.batch.item.database
          .JdbcBatchItemWriter;
import org.springframework.batch.item.file
          .FlatFileItemReader;
import org.springframework.batch.item.file.mapping
          .BeanWrapperFieldSetMapper;
import org.springframework.batch.item.file.mapping
          .DefaultLineMapper;
import org.springframework.batch.item.file.transform
          .DelimitedLineTokenizer;
import org.springframework.beans.factory
          .annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .ComponentScan;
import org.springframework.context.annotation
          .Configuration;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.ResourceLoader;
import javax.sql.DataSource;

Configuration
@EnableBatchProcessing
public class BatchConfig {

  @Autowired
  private JobBuilderFactory jobs;
  @Autowired
  private StepBuilderFactory steps;

  @Autowired DataSource dataSource;

  @Autowired SingerItemProcessor itemProcessor;

  @Bean
  public Job job(JobExecutionStatsListener listener) {
    return jobs.get("singerJob")
               .listener(listener)
               .flow(step1())
               .end()
               .build();
  }

  @Bean
  protected Step step1() {
    return steps.get("step1")
                .<Singer, Singer>chunk(10)
                .reader(itemReader())
                .processor(itemProcessor)
                .writer(itemWriter())
                .build();
  }

  // ввести здесь лямбда-выражения
  @Bean
  public ItemReader itemReader() {
    FlatFileItemReader<Singer> itemReader = new FlatFileItemReader<>();
    itemReader.setResource(new ClassPathResource(
                           "support/test-data.csv"));
    itemReader.setLineMapper(
               new DefaultLineMapper<Singer>() {{
```

```
        setLineTokenizer(new DelimitedLineTokenizer() {{
              setNames(new String { "firstName", "lastName", "song" });
        }});
        setFieldSetMapper(
                new BeanWrapperFieldSetMapper<Singer>() {{
           setTargetType(Singer.class);
        }});
    }});
    return itemReader;
  }

  @Bean
  public ItemWriter itemWriter() {
... // то же, что и прежде
  }
}
```

Помимо интенсивного употребления лямбда-выражений в объявлении компонента `itemReader`, существенные изменения претерпел и компонент `job`. В частности, исполняемая стадия пакетного задания больше не привязывается автоматически контейнером Spring на основании описателя, а метод `flow()` вызывается для создания нового составителя заданий, который и будет выполнять одну стадию или последовательный ряд стадий пакетного задания.

Большая часть описанных выше действий реализуется средствами Spring Boot. Остается лишь запустить пакетное задание на выполнение с помощью типичного прикладного класса, исходный код которого приведен ниже.

```
package com.apress.prospring5.ch18;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure
          .SpringBootApplication;
import org.springframework.context
          .ConfigurableApplicationContext;

@SpringBootApplication
public class Application {

  private static Logger logger =
          LoggerFactory.getLogger(Application.class);
  public static void main(String... args)
                    throws Exception {
    ConfigurableApplicationContext ctx =
         SpringApplication.run(Application.class, args);
    assert (ctx != null);
    logger.info("Application started...");
```

```
    System.in.read();
    ctx.close();
  }
}
```

В конечном счете можно проверить журналы регистрации на наличие в них предполагаемых результатов, выводимых из класса `JobExecutionStatsListener`:

```
o.s.b.c.l.s.SimpleJobLauncher - Job: [FlowJob:
  [name=singerJob]] launched with the following
  parameters: [{}]
o.s.b.c.j.SimpleStepHandler - Executing step: [step1]
c.a.p.c.SingerItemProcessor - Transformed singer:
  firstName: John, lastName: Mayer, song: Helpless Into:
  firstName: JOHN, lastName: MAYER, song: HELPLESS
c.a.p.c.SingerItemProcessor - Transformed singer:
  firstName: Eric, lastName: Clapton,
  song: Change The World Into: firstName: ERIC,
  lastName: CLAPTON, song: CHANGE THE WORLD
c.a.p.c.SingerItemProcessor - Transformed singer:
  firstName: John, lastName: Butler, song: Ocean Into:
  firstName: JOHN, lastName: BUTLER, song: OCEAN
c.a.p.c.SingerItemProcessor - Transformed singer:
  firstName: BB, lastName: King,
  song: Chains And Things Into: firstName: BB,
  lastName: KING, song: CHAINS AND THINGS
c.a.p.c.JobExecutionStatsListener - -->
  Singers were saved to the database. Printing results ...
c.a.p.c.JobExecutionStatsListener - firstName: JOHN,
  lastName: MAYER, song: HELPLESS
c.a.p.c.JobExecutionStatsListener - firstName: ERIC,
  lastName: CLAPTON, song: CHANGE THE WORLD
c.a.p.c.JobExecutionStatsListener - firstName: JOHN,
  lastName: BUTLER, song: OCEAN
c.a.p.c.JobExecutionStatsListener - firstName: BB,
  lastName: KING, song: CHAINS AND THINGS
o.s.b.c.l.s.SimpleJobLauncher - Job: [FlowJob:
  [name=singerJob]] completed with the following parameters:
  [{}] and the following status: [COMPLETED]
```

Проект Spring Integration

В проекте Spring Integration предоставляются готовые реализации хорошо известных проектных шаблонов интеграции корпоративных приложений (Enterprise Integration Pattern — EIP). Этот проект опирается на архитектуры, управляемые сообщениями, предоставляет простую модель для решений интеграции, возможности асинхронного выполнения и слабо связные компоненты. Кроме того, данный проект разработан с учетом расширяемости и тестируемости.

По существу, главная роль в каркасе обмена сообщениями принадлежит интерфейсу `Message`, служащему в качестве оболочки для сообщений. Эта обобщенная оболочка для объекта Java сочетается с метаданными (полезной информацией и заголовками), применяемыми для определения способа обработки данного объекта.

Канал сообщений типа `Message` выполняет функции *канала* в архитектуре каналов и фильтров, где поставщики отправляют сообщения в канал, откуда их получают потребители. С другой стороны, конечные точки сообщений представляют фильтр в архитектуре каналов и фильтров, а также соединяют прикладной код с каркасом обмена сообщениями. В проекте Spring Integration предоставляется ряд готовых конечных точек сообщений, к числу которых относятся преобразователи, фильтры, маршрутизаторы и разделители. У каждой конечной точки сообщений свои роли и обязанности.

В проекте Spring Integration имеется также немало конечных точек интеграции (свыше 20 на момент написания данной книги), которые описаны в разделе "Endpoint Quick Reference Table" (Краткий справочник по конечным точкам) документации на проект Spring Integration, доступном по адресу `http://docs.spring.io/spring-integration/reference/htmlsingle/#endpoint-summary`. Эти конечные точки позволяют подключать различнык ресурсы, в том числе AMQP, файлы, HTTP, JMX, Syslog и Twitter. Кроме стандартных возможностей проекта Spring Integration, имеется еще один проект под названием Spring Integration Extensions (`https://github.com/spring-projects/spring-batch-extensions`) в виде модели коллективных дополнений для интеграции с такими ресурсами, как AWS (Amazon Web Services — веб-службы Amazon), Apache Kafka, SMPP (Short Message Peer-to-Peer — одноранговый обмен короткими сообщениями) и Voldemort. Помимо стандартных и расширяющих компонентов, в проекте Spring Integration предоставляется немало готовых компонентов, значительно снижающих потребность в написании собственных компонентов.

В рассматриваемом здесь примере будут использованы предыдущие примеры пакетных заданий, но на этот раз будет задействован проект Spring Integration, чтобы продемонстрировать его возможности для текущего контроля содержимого каталога через заданные промежутки времени. Когда в каталог поступает файл, он обнаруживается и для его обработки запускается пакетное задание.

Данный пример также опирается на пример проекта, построенного в начале этой главы только средствами Spring Batch. Поэтому, прежде чем продолжить дальше, просмотрите этот пример еще раз, поскольку в дальнейшем будут поясняться только новые классы и изменения в конфигурации.

Очевидно, что в текущий проект необходимо внедрить новые зависимости, как показано ниже.

```
// Файл конфигурации pro-spring-15/build.gradle
ext {
    // Библиотеки Spring
    ...
```

```
    springBatchVersion = '4.0.0.M3'
    springIntegrationVersion = '5.0.0.M6'
    springBatchIntegrationVersion = '4.0.0.M3'
    ...
    spring = [
       context : "org.springframework:spring-context:
                  $springVersion",
       jdbc : "org.springframework:spring-jdbc:
              $springVersion",
       batchCore : "org.springframework.batch:
                    spring-batch-core:$springBatchVersion"
       batchIntegration : "org.springframework.batch:
                           spring-batch-integration:
                           $springBatchIntegrationVersion",
       integrationFile : "org.springframework.integration:
                         spring-integration-file:
                         $springIntegrationVersion"
    ...
    ]
    misc = [
       io : "commons-io:commons-io:2.5",
       ...
    ]
    db = [
       ...
       dbcp2 : "org.apache.commons:commons-dbcp2:
               $dbcpVersion",
       h2 : "com.h2database:h2:$h2Version",
       // требуется для модуля Batch JSR-352
       hsqldb: "org.hsqldb:hsqldb:2.4.0"
       dbcp : "commons-dbcp:commons-dbcp:1.4",
    ]
}
...
// Файл конфигурации pro-spring-15/chapter18/build.gradle
dependencies {
    if (!project.name.contains("boot")) {
       compile(spring.jdbc) {
       // исключить перечисленные ниже модули, поскольку они будут
       // задействованы в batchCore по транзитивным зависимостям
      exclude module: 'spring-core'
      exclude module: 'spring-beans'
      exclude module: 'spring-tx'
    }
    compile spring.batchCore, db.dbcp2, db.h2, misc.io,
            spring.batchIntegration, spring.integrationFile,
            misc.slf4jJcl, misc.logback
    }
}
```

Внедрив все требующиеся зависимости, создадим класс, действующий в качестве преобразователя в Spring Integration. Этот преобразователь в виде экземпляра типа `Transformer` будет получать из входящего канала сообщение в виде экземпляра типа `Message`, представляющего найденный файл, и запускать с ним пакетное задание, как показано ниже.

```
package com.apress.prospring5.ch18;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.integration.launch.JobLaunchRequest;
import org.springframework.messaging.Message;
import java.io.File;

public class MessageToJobLauncher {
  private Job job;
  private String fileNameKey;

  public MessageToJobLauncher(Job job, String fileNameKey) {
    this.job = job;
    this.fileNameKey = fileNameKey;
  }

  public JobLaunchRequest toRequest(Message<File> message) {
    JobParametersBuilder jobParametersBuilder =
                         new JobParametersBuilder();
    jobParametersBuilder.addString(fileNameKey,
                 message.getPayload().getAbsolutePath());
    return new JobLaunchRequest(job,
                jobParametersBuilder.toJobParameters());
  }
}
```

Теперь внесем коррективы в конфигурационный класс `BatchConfig`, чтобы поддержать пакетную интеграцию, а точнее — в компонент `itemReader`, создаваемый всякий раз, когда требуется обработать новый файл. Это означает, что путь к этому файлу необходимо ввести в компонент `itemReader` и что у данного компонента больше не может быть одиночной области видимости.

```
package com.apress.prospring5.ch18.config;
...
@Configuration
@EnableBatchProcessing
@Import(DataSourceConfig.class)
@ComponentScan("com.apress.prospring5.ch18")
public class BatchConfig {
  ... // автоматически привязываемые компоненты Spring Beans
```

```
  @Bean
  public Job singerJob() {
    return jobs.get("singerJob").start(step1()).build();
  }

  @Bean
  protected Step step1() {
    ... // без изменений
  }

  @Bean
  @StepScope
  public FlatFileItemReader itemReader(
    @Value("file://#{jobParameters['file.name']}")
           String filePath) {
    FlatFileItemReader itemReader =
                       new FlatFileItemReader();
    itemReader.setResource(
              resourceLoader.getResource(filePath));
    DefaultLineMapper lineMapper = new DefaultLineMapper();
    DelimitedLineTokenizer tokenizer =
                           new DelimitedLineTokenizer();
    tokenizer.setNames("firstName", "lastName", "song");
    tokenizer.setDelimiter(",");
    lineMapper.setLineTokenizer(tokenizer);
    BeanWrapperFieldSetMapper<Singer> fieldSetMapper =
                       new BeanWrapperFieldSetMapper<>();
    fieldSetMapper.setTargetType(Singer.class);
    lineMapper.setFieldSetMapper(fieldSetMapper);
    itemReader.setLineMapper(lineMapper);
    return itemReader;
  }

  @Bean
  public ItemWriter<Singer> itemWriter() {
    ... // без изменений
  }
}
```

Аннотация `@StepScope` удобна тем, что она равнозначна аннотации `@Scope (value="step", proxyMode=TARGET_CLASS)`, а следовательно, для компонента `itemReader` задается область видимости заместителя под названием `step`, чтобы стало совершенно ясно, как пользоваться этим компонентом. Теперь, когда конфигурация составлена, перейдем к конфигурации, типичной для самой интеграции. Эта конфигурация будет составлена в формате XML, поскольку на момент написания данной книги такой способ конфигурирования оказался более практичным.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:batch-int="http://www.springframework.org
                          /schema/batch-integration"
       xmlns:int=
            "http://www.springframework.org/schema/integration"
       xmlns:int-file="http://www.springframework.org/schema
                         /integration/file"
       xmlns:context=
            "http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org
                             /schema/batch-integration
       http://www.springframework.org/schema
             /batch-integration/spring-batch-integration.xsd
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans
             /spring-beans.xsd
       http://www.springframework.org/schema/integration
       http://www.springframework.org/schema/integration
             /spring-integration.xsd
       http://www.springframework.org/schema
             /integration/file
       http://www.springframework.org/schema/integration
             /file/spring-integration-file.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context
             /spring-context.xsd">

    <bean name="/BatchConfig"
         class="com.apress.prospring5
                   .ch18.config.BatchConfig"/>

    <context:annotation-config/>

    <int:channel id="inbound"/>
    <int:channel id="outbound"/>
    <int:channel id="loggingChannel"/>
    <int-file:inbound-channel-adapter
         id="inboundFileChannelAdapter"
         channel="inbound"
         directory="file:/tmp/" filename-pattern="*.csv">
    <int:poller fixed-rate="1000"/>
    </int-file:inbound-channel-adapter>
    <int:transformer input-channel="inbound"
         output-channel="outbound">
      <bean class="com.apress.prospring5.ch18
                      .MessageToJobLauncher">
```

```
        <constructor-arg ref="singerJob"/>
        <constructor-arg value="file.name"/>
      </bean>
    </int:transformer>

    <batch-int:job-launching-gateway
          request-channel="outbound"
          reply-channel="loggingChannel"/>
    <int:logging-channel-adapter channel="loggingChannel"/>
</beans>
```

Основные дополнения в данной конфигурации касаются разделов с префиксами пространств имен **int:** и **batch-int:**. Сначала здесь создается несколько именованных каналов для передачи по ним данных, затем адаптер входящего канала `inbound-channel-adapter` специально настраивается для наблюдения за указанным каталогом через каждую секунду. Далее настраивается компонент преобразователя, получающий файлы в виде стандартных объектов типа `java.io.File`, заключенных в оболочку экземпляров сообщений типа `Message`. После этого настраивается шлюз для запуска заданий `job-launching-gateway`, принимающий запросы на запуск заданий от преобразователя типа `Trasformer` для фактического вызова пакетного задания. И, наконец, создается адаптер протоколирования канала `logging-channel-adapter`, предназначенный для вывода информационных уведомлений по завершении задания. Анализируя атрибуты конфигурации канала, можно заметить, что каждый компонент потребляет или поставляет сообщения через экземпляры канала типа `Channel` или же делает и то и другое.

А теперь создадим простой тестовый класс, загружающий файл конфигурации. Основное назначение приведенного ниже тестового класса — загрузить контекст приложения вместе с его конфигурацией. Он продолжает действовать до тех пор, пока не будет уничтожен его процесс, поскольку он непрерывно опрашивает указанный каталог.

```
package com.apress.prospring5.ch18;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.support
      .GenericXmlApplicationContext;

public class FileWatcherDemo {

  private static Logger logger =
          LoggerFactory.getLogger(FileWatcherDemo.class);

  public static void main(String... args)
                throws Exception {
    GenericXmlApplicationContext ctx =
          new GenericXmlApplicationContext(
```

```
                "classpath:spring/integration-config.xml");
    assert (ctx != null);
    logger.info("Application started...");
    System.in.read();
    ctx.close();
  }
}
```

Скомпилируйте приведенный выше код и запустите класс `FileWatcherDemo`. Как только приложение начнет работать, вы обнаружите, что на консоль выводятся протокольные сообщения, как показано ниже, но больше ничего не происходит. Дело в том, что адаптер файлов Spring Integration ожидает размещения файлов в сконфигурированном месте в течение интервала опроса, а до тех пор, пока он не обнаружит там файл, то ничего происходить не будет. В каталоге `src/main/resources/support/` вы обнаружите четыре файла формата CSV под именами от `singer1.csv` до `singer4.csv`. Каждый из них содержит одну строку с Ф.И.О. певца и еще одну строку с названием его песни. Скопируйте эти файлы по очереди в каталог `tmp` и понаблюдайте за тем, что будет выводиться на консоль.

```
o.s.i.f.FileReadingMessageSource - Created message:
  [GenericMessage [payload=/tmp/singers1.csv,
   headers={file_originalFile=/tmp/singers1.csv,
   id=ca0ec15e-b9b6-5dc2-2fc6-44fdb4b433f4,
   file_name=singers1.csv, file_relativePath=singers1.csv,
   timestamp=1501442201624}]]
o.s.b.c.l.s.SimpleJobLauncher - Job: [SimpleJob:
  [name=singerJob]] launched with the following parameters:
  [{file.name=/tmp/singers1.csv}]
o.s.b.c.j.SimpleStepHandler - Executing step: [step1]
c.a.p.c.SingerItemProcessor - Transformed singer: firstName:
  John, lastName: Mayer, song: Helpless Into:
  firstName: JOHN, lastName: MAYER, song: HELPLESS
INFO c.a.p.c.StepExecutionStatsListener - -->
  Wrote: 1 items in step: step1
o.s.b.c.l.s.SimpleJobLauncher - Job: [SimpleJob:
  [name=singerJob]] completed with the following parameters:
  [{file.name=/tmp/singers1.csv}] and the following status:
  [COMPLETED]
o.s.i.h.LoggingHandler - JobExecution: id=1, version=2,
  startTime=Sun Jul 30 22:16:41 EEST 2017,
  endTime=Sun Jul 30 22:16:41 EEST 2017,
  lastUpdated=Sun Jul 30 22:16:41 EEST 2017,
  status=COMPLETED, exitStatus=exitCode=COMPLETED;
  exitDescription=, job=[JobInstance: id=1,
  version=0, Job=[singerJob]],
  jobParameters=[{file.name=/tmp/singers1.csv}]
```

Из приведенных выше протокольных сообщений можно сделать вывод, что в Spring Integration удалось обнаружить файл, создать сообщение, запустить пакетное задание, преобразующее содержимое файла формата CSV, а затем записать это содержимое в базу данных, действующую в оперативной памяти. Несмотря на то что представленный выше пример довольно прост, он все же наглядно демонстрирует порядок построения сложных и несвязанных вместе рабочих потоков между разными типами приложений с помощью Spring Integration. За дополнительными сведениями о Spring Integration обращайтесь на страницу данного проекта по адресу `https://spring.io/projects/spring-integration`.

Проект Spring XD

Проект Spring XD — это расширяемая служба среды выполнения, предназначенная для получения распределенных данных, анализа в реальном времени, пакетной обработки и экспортирования данных. Проект Spring XD построен на основе многих существующих проектов, входящих в состав Spring, наиболее значимыми из которых являются сам каркас Spring Framework, а также проекты Spring Batch и Spring Integration. Назначение проекта Spring XD — обеспечить единообразный способ интеграции многих систем в связное решение больших данных, помогающее преодолеть сложность распространенных вариантов использования.

Служба Spring XD может функционировать в автономном режиме с единственным узлом, который обычно предназначен для разработки и тестирования, а также в полностью распределенном режиме, предоставляя возможность иметь высокодоступные ведущие узлы наряду с несколькими рабочими узлам. Проект Spring XD позволяет управлять этими узлами через интерфейс оболочки (с помощью оболочки Spring), а также графический веб-интерфейс. С помощью этих интерфейсов можно определить порядок сборки различных компонентов для удовлетворения потребностей в обработке данных с помощью синтаксиса языка DSL в приложении оболочки или путем ввода данных в веб-интерфейсе, где определения нужных компонентов будут созданы автоматически.

Язык DSL, применяемый в Spring XD, опирается на такие понятия, как потоки, модули, источники, процессоры, приемники данных и задания. Объединяя упомянутые выше компоненты с помощью лаконичного синтаксиса, легко создавать потоки для соединения разных технологий, которые позволяют получать данные, обрабатывать их и в конечном итоге выводить во внешний источник или даже запускать пакетное задание для дальнейшей обработки данных. Рассмотрим вкратце эти понятия.

- *Поток* определяет порядок протекания данных от источника к приемнику и возможного их прохождения через любое количество процессоров. Для определения потока применяется язык DSL; например, простое определение "источник–приемник" может выглядеть следующим образом: `http | file`.

- *Модуль* инкапсулирует неоднократно используемые единицы работы, из которых состоят потоки. Модули объединяются в категории по типам, основанным на их ролях. На момент написания данной книги проект Spring XD содержал модули типа источника, процессора, приемника и задания.
- *Источник* в Spring XD опрашивает внешний ресурс или активизируется каким-то событием. Источники предоставляют выводимые данные следующим в потоке компонентам, причем первым модулем в потоке должен быть источник.
- *Процессор* по своему характеру подобен тому, что демонстрировалось ранее в проекте Spring Batch. Роль процессора состоит в получении вводимых данных, выполнения преобразований или бизнес-логики относительно предоставленного объекта и возврате выводимого результата.
- На другой от источника стороне передачи данных в потоке находится *приемник*, вводящий данные от источника и выводящий данные в свой целевой ресурс. Приемник является конечной точкой доставки данных в потоке.
- *Задания* — это модули, определяющие задание Spring Batch. Подобные задания определяются так, как пояснялось в начале этой главы, и разворачиваются в Spring XD для обеспечения возможностей пакетной обработки.
- *Отводы* принимают данные, проходящие через поток, а также позволяют обрабатывать отводимые данные в отдельном потоке. Понятие отвода подобно проектному шаблону интеграции корпоративных приложений WireTap (Ответвление).

Как и следовало ожидать, в проекте Spring XD предоставляется ряд готовых источников, процессоров, заданий, приемников и отводов. Но, помимо того, что предлагается в готовом виде, разработчики вольны построить собственные модули и компоненты. За дополнительными сведениями по поводу создания собственных модулей и компонентов обращайтесь к следующим разделам оперативно доступного справочного руководства по Spring XD.

- **Модули:** `http://docs.spring.io/spring-xd/docs/current/reference/html/#modules`.
- **Процессоры:** `http://docs.spring.io/spring-xd/docs/current/reference/html/#creating-a-processor-module`.
- **Приемники данных:** `http://docs.spring.io/spring-xd/docs/current/reference/html/#creating-a-sink-module`.
- **Задания:** `http://docs.spring.io/spring-xd/docs/current/reference/html/#creating-a-job-module`.

В рассматриваемом далее примере показано, как применять готовые компоненты Spring XD для воспроизведения того, что было создано в примерах применения проектов Spring Batch и Spring Integration. И все это будет сделано с помощью простой

конфигурации, составленной средствами оболочки Spring XD и языка DSL для работы в режиме командной строки.

Прежде всего необходимо установить Spring XD. Различные способы установки Spring XD описаны в разделе "Getting Started" (Начало работы) руководства пользователя, оперативно доступного по адресу `http://docs.spring.io/spring-xd/docs/1.0.0.RC1/reference/html/#getting-started`. Выбор способа установки зависит от личных предпочтений и не имеет особого значения для рассматриваемого здесь примера. Установив Spring XD, запустите исполняющую среду в режиме одиночного узла, как описано в документации.

Чтобы воспроизвести в Spring XD то, что было создано в примерах применения проектов Spring Batch и Spring Integration, достаточно выполнить несколько простых действий. Сначала создайте импортируемый файл формата CSV с приведенным ниже содержимым, разместив его по пути `/tmp/people.csv`.

```
John,Mayer,Helpless
Eric,Clapton,Change The World
John,Butler,Ocean
BB,King,Chains And Things
```

Затем введите с консоли оболочки Spring XD следующую команду:

```
job create singerjob --definition "filejdbc
    --resources=file:///tmp/singers.csv
    --names=firstname,lastname,song --tableName=singer
    --initializeDatabase=true" --deploy
```

Как только вы нажмете клавишу <Enter>, появится сообщение `"Successfully created and deployed job 'singerjob'"` (Задание `singerjob` успешно создано и развернуто). Если такое сообщение не появится, просмотрите консольный вывод на тот терминал, где запускался контейнер Spring XD с единственным узлом, чтобы получить дополнительные сведения об этом задании.

Итак, новое определение задания создано, но пока еще ничего особенного не произошло, поскольку задание не было запущено на выполнение. Для его запуска введите в оболочке приведенную ниже команду, на которую оболочка должна отреагировать сообщением, что запрос на запуск задания `singerjob` успешно отправлен на обработку.

```
job launch singerjob
```

В результате анализа предоставленного кода DSL в Spring XD становится известно, что нам требуется создать пакетное задание для чтения данных из файла и последующей их записи в базу данных через интерфейс JDBC с помощью оператора `job create` и указанного источника `filejdbc`. Кроме того, в Spring XD автоматически создается таблица под именем, заданным в параметре `tableName`, а также именами столбцов, указанными в параметре `names`. Импортированные данные читаются из файла формата CSV по пути, указанному в параметре `resources`.

Чтобы проверить импортированные данные, подключитесь к базе данных, указанной во время установки (встроенной или настоящей СУРБД), воспользовавшись избранным вами инструментальным средством, и выберите для просмотра записи из таблицы `Singer`. Если вы не увидите никаких данных, просмотрите протокольные сообщения, выводимые на консоль, где функционирует контейнер Spring XD с единственным узлом.

Итак, мы ввели в оболочке две команды, и хотя нам не пришлось писать никакого кода или составлять сложную конфигурацию, мы все же импортировали содержимое файла формата CSV в базу данных, приложив для этого минимальные усилия. И этого нам удалось добиться с помощью пакетного задания, предварительно составленного в Spring XD с помощью простого синтаксиса командной строки на языке DSL и запущенного на выполнение в оболочке.

Как видите, в Spring XD предоставляется немало готовых функциональных возможностей, избавляющих разработчиков от необходимости самостоятельно реализовывать распространенные варианты использования. Для дальнейшего исследования функциональных возможностей Spring XD оставляем вам в качестве упражнения преобразование Ф.И.О. и названия песни каждого певца, как это было сделано в предыдущих примерах. Дополнительные сведения о Spring XD можно получить на странице проекта, оперативно доступной по адресу `http://projects.spring.io/spring-xd/`.

Наиболее примечательные функциональные средства каркаса Spring Framework

На момент написания данной книги была выпущена версия Spring Framework RC3, а версия Spring 5.0 считалась главным исправлением базового каркаса и сопровождалась кодовой базой, переписанной по версии Java 8, но в версии RC3 была начата ее адаптация к Java 9. В этой версии был объявлен ряд следующих основных средств.

- Модуль `spring-webflux`, построенный на основе Reactor 3.1 с поддержкой версий RxJava 1.3 и 2.1 известного каркаса Reactive Web Framework. Этот модуль служит дополнением модуля `spring-webmvc`, предоставляя модель реактивного веб-программирования, предназначенную для выполнения асинхронных прикладных интерфейсов API на сервере Tomcat, Jetty или Undertow.
- Поддержка языка Kotlin, обеспечиваемая через полноценный прикладной интерфейс API для регистрации компонентов Spring Beans и конечных точек функциональных веб-ресурсов с безопасной обработкой пустых значений.
- Интеграция с прикладными интерфейсами Java EE 8 API, включая поддержку версий Servlet 4.0, Bean Validation 2.0, JPA 2.2 и JSON Binding API (в качестве альтернативы библиотекам Jackson/Gson в модуле Spring MVC).

- Полная поддержка версии JUnit 5 Jupiter, дающая разработчикам возможность писать модульные тесты и расширения в JUnit 5, параллельно выполняя их средствами Spring TestContext Framework.
- Совместимость с версией Java 9, к которой команда разработчиков стремилась в версии Spring Framework 5. Но поскольку компания Oracle отложила выпуск версии Java 9 на несколько месяцев, то версия Spring Framework 5 была разработана и выпущена совместимой с Java 8 при поддержке реактивного программирования с помощью Project Reactor. Тем не менее обещание полной совместимости с Java 9 осталось, а его воплощение было начато в версии RC3.
- Много других доступных функциональных средств[12].

Функциональный каркас веб-приложений

Как было установлено ранее, функциональный каркас веб-приложений (модуль `spring-webflux`) служит дополнением модуля `spring-webmvc`, предоставляющим модель реактивного веб-программирования, предназначенную для применения асинхронных прикладных интерфейсов API. Он построен по принципам *реактивного программирования*, которое можно описать, как "программирование с реактивными потоками данных". Потокам данных принадлежит центральное место в модели реактивного программирования, они служат для поддержки асинхронной обработки любых данных. По существу, библиотеки реактивного программирования предоставляют возможность использовать в качестве потоков данных все, что угодно: переменные, вводимые пользователем данные, кеши, структуры данных и т.д., а следовательно, поддерживать такие характерные для потоков данных операции, как фильтрация одного потока для создания другого, слияние потоков, преобразование значений из одного потока в другой и прочее. Реактивная часть в реактивном программировании означает, что поток данных будет объектом, "наблюдаемым" компонентом, который будет соответственно реагировать на порождаемые объекты. В потоке данных могут быть порождены три типа объектов: значения, ошибки или сигналы завершения операций.

Чтобы превратить обычное приложение в реактивное, необходимо сделать первый логический шаг, который состоит в видоизменении компонентов для получения и обработки потоков данных. Имеются следующие два типа потоков данных.

- **`reactor.core.publisher.Flux`**[13]. Это поток данных, состоящий из `[0..n]` элементов. Простейший способ создать поток данных типа `Flux` демонстрируется в следующем фрагменте кода:

---

[12] Чтобы следить за последними выпусками и изменениями в составе проекта Spring, рекомендуется регулярно посещать официальный блог, который ведется по Spring на странице, доступной по адресу `https://spring.io/blog`.

[13] Более подробное разъяснение этого типа потоков данных приведено по адресу `https://projectreactor.io/docs/core/release/reference/#flux`.

```
Flux simple = Flux.just("1", "2", "3");
// или из существующего списка: List<Singer> Flux<Singer>
fromList = Flux.fromIterable(list);
```

- **reactor.core.publisher.Mono**[14]. Это поток данных, состоящий из [0..1] элементов. Простейший способ создать поток данных типа Mono демонстрируется в следующем фрагменте кода:

```
Mono simple = Mono.just("1");
// или из существующего объекта типа Singer
Mono<Singer> fromObject = Mono.justOrEmpty(singer);
```

В обоих упомянутых выше классах реализуется интерфейс org.reactivestreams.Publisher<T>, и в связи с этим у вас может возникнуть вполне резонный вопрос: а зачем вообще требуется реализация этого интерфейса в классе Mono? Она требуется из практических соображений, поскольку для обработки значений, порождаемых в потоке данных, всегда полезно знать количество составляющих его элементов. Так, если имеется класс реактивного хранилища, стоит ли возвращать поток данных типа Flux из метода findOne()? Этого краткого введения должно быть достаточно, чтобы уяснить основы реактивного программирования и составить определенное представление о функциональных возможностях модуля spring-webflux[15].

В качестве примера в этом разделе рассматривается веб-приложение Spring Boot, представленное в главе 16, но теперь вместо шаблонизатора Thymeleaf в нем будет использован модуль Spring WebFlux. Поэтому необходимо внедрить сначала модуль spring-boot-starterwebflux в качестве зависимости, а также удалить ненужные зависимости. В следующем фрагменте кода конфигурации внедряются нужные библиотеки, сконфигурированные в родительском проекте и применяемые в модуле webflux-boot:

```
// Файл конфигурации pro-spring-15/build.gradle
ext {
    // Библиотеки Spring
    bootVersion = '2.0.0.M3'
    springDataVersion = '2.0.0.M3'
    junit5Version = '5.0.0-M4'
    ...
    boot = [
       ...
       springBootPlugin : "org.springframework.boot:
                 spring-boot-gradle-plugin:$bootVersion",
```

[14] Более подробное разъяснение этого типа потоков данных приведено по адресу https://projectreactor.io/docs/core/release/reference/#mono.

[15] За дополнительными сведениями о модели реактивного программирования и практическом применении реактивных потоков данных обращайтесь к справочному руководству, примерам кода и документации на Project Reactor в формате Javadoc, доступной по адресу https://projectreactor.io/docs. А превосходное введение в реактивное программирование можно найти по адресу https://gist.github.com/staltz/868e7e9bc2a7b8c1f754.

```
        starterWeb : "org.springframework.boot:
                     spring-boot-starter-web:$bootVersion",
        starterTest : "org.springframework.boot:
                      spring-boot-starter-test:
                      $bootVersion",
        starterJpa : "org.springframework.boot:
                     spring-boot-starter-data-jpa:
                     $bootVersion",
        starterJta : "org.springframework.boot:
                      spring-boot-starter-jta-atomikos:
                      $bootVersion",
        starterWebFlux : "org.springframework.boot:
                         spring-boot-starter-webflux:
                         $bootVersion"
    ]

    testing = [
        ...
        junit5 : "org.junit.jupiter:junit-jupiter-engine:
                  $junit5Version"
    ]

    db = [
        ...
         h2 : "com.h2database:h2:$h2Version"
    ]
}

// Файл конфигурации chapter18/webflux-module/build.gradle
buildscript {
    repositories {
        ...
    }

    dependencies {
        classpath boot.springBootPlugin
    }
}

apply plugin: 'org.springframework.boot'

dependencies {
    compile boot.starterWebFlux, boot.starterWeb,
    boot.starterJpa, boot.starterJta, db.h2
    testCompile boot.starterTest, testing.junitJupiter,
    testing.junit5
}
```

Модуль `spring-boot-starter-webflux` зависит от нескольких библиотек реактивного программирования, внедряемых в веб-приложение автоматически. Зависимости веб-приложения, отображаемые в представлении Gradle, доступном в IDE IntelliJ IDEA, приведены на рис. 18.1. Обратите внимание на библиотеку `reactive-streams`, содержащую основные интерфейсы для реактивной потоковой передачи, стандартную спецификацию и следующие четыре интерфейса: `Publisher`, `Subscriber`, `Subscription` и `Processor`. Подробнее об этом см. по адресу `www.reactive-streams.org`, поскольку рассмотрение потокового прикладного интерфейса API выходит за рамки этого раздела. Реализации потоковой передачи данных, применяемые в рассматриваемом здесь примере веб-приложения, предоставляются библиотекой `reactor-core`, узнать о которой более подробно можно по адресу `https://projectreactor.io/`.

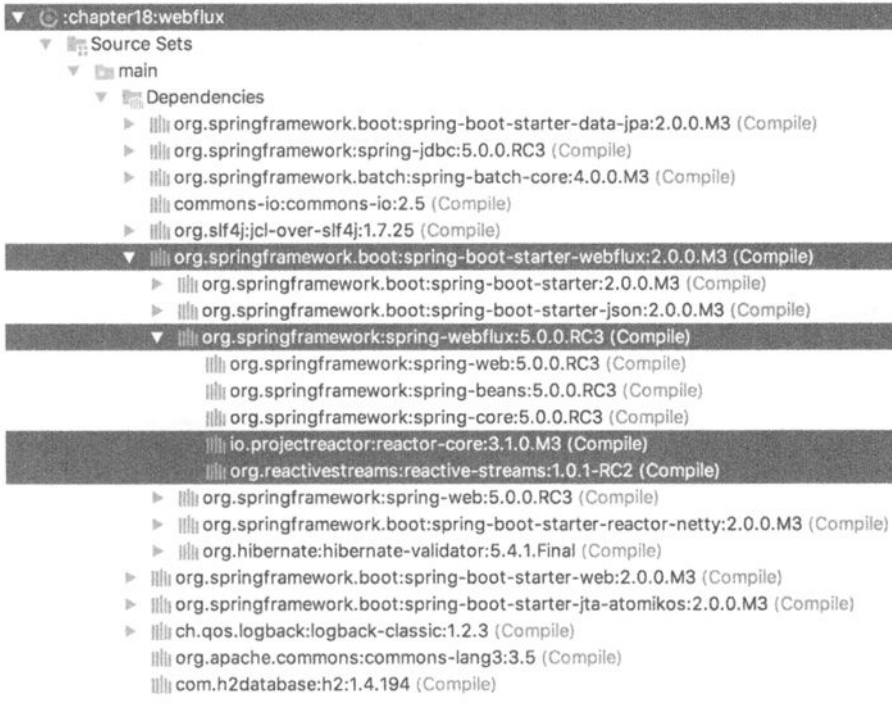


Рис. 18.1. Зависимости текущего проекта, отображаемые в представлении Gradle, доступном в IDE IntelliJ IDEA

Ради простоты из рассматриваемого здесь примера веб-приложения исключены уровень защиты и пользовательский интерфейс. В то же время в данном приложении будет применена технология REST, а для его проверки созданы тестовые классы. Кроме того, в данном примере будет использоваться приведенный ниже упрощенный вариант класса `Singer`, из которого исключены все аннотации проверки достоверности, поскольку не о них пойдет речь в этом разделе.

```
package com.apress.prospring5.ch18.entities;

import javax.persistence.*;
import java.io.Serializable;
import java.text.SimpleDateFormat;
import java.util.Date;
import static javax.persistence.GenerationType.IDENTITY;

@Entity
@Table(name = "singer")
public class Singer implements Serializable {

  @Id
  @GeneratedValue(strategy = IDENTITY)
  @Column(name = "ID")
  private Long id;
  @Version
  @Column(name = "VERSION")
  private int version;
  @Column(name = "FIRST_NAME")
  private String firstName;
  @Column(name = "LAST_NAME")
  private String lastName;
  @Temporal(TemporalType.DATE)
  @Column(name = "BIRTH_DATE")
  private Date birthDate;

  public Singer() {
  }

  public Singer(String firstName,
                String lastName, Date birthDate) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.birthDate = birthDate;
  }
  // Методы установки и получения
  ...
}
```

Для реализации информационного хранилища в данном случае послужит пустое расширение интерфейса `CrudRepository<Singer, Long>`, которое здесь не показывается снова. А новшество состоит в том, что подобным образом будет реализовано реактивное хранилище. Ниже приведено определение интерфейса `ReactiveSingerRepo`, где для обработки объектов типа `Singer` применяются потоки данных типа `Mono` и `Flux`.

```
package com.apress.prospring5.ch18.repos;

import com.apress.prospring5.ch18.entities.Singer;
```

```
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

public interface ReactiveSingerRepo {

  Mono<Singer> findById(Long id);

  Flux<Singer> findAll();

  Mono<Void> save(Mono<Singer> singer);
}
```

Этот интерфейс реализуется следующим образом:

```
package com.apress.prospring5.ch18.repos;

import com.apress.prospring5.ch18.entities.Singer;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@Service
public class ReactiveSingerRepoImpl
       implements ReactiveSingerRepo {

  @Autowired
  SingerRepository singerRepository;

  @Override public Mono<Singer> findById(Long id) {
    return Mono.justOrEmpty(singerRepository.findById(id));
  }

  @Override public Flux<Singer> findAll() {
    return Flux.fromIterable(singerRepository.findAll());
  }

  @Override public Mono<Void> save(Mono<Singer> singerMono) {
    return singerMono.doOnNext(singer ->
           singerRepository.save(singer))
    .thenEmpty((Mono.empty()));
  }
}
```

Опишем вкратце каждый метод из приведенного выше класса в отдельности.

- Метод `findById()` возвращает простой поток данных, порождающий объект типа `Singer`, если таковой обнаруживается в результате вызова метода `singerRepository.findById(id)`, а иначе выдается сигнал `onComplete`. В конечном итоге возвращается пустой объект типа `Mono<Singer>`.

- Метод `findAll()` возвращает поток данных, содержащий все объекты типа `Singer`, возвращаемые методом `textitsingerRepository.findAll()`.
- Метод `save()` получает в качестве параметра экземпляр типа `Mono<Singer>`, т.е. поток данных, состоящий из единственного объекта типа `Singer`. Экземпляр типа `java.util.function.Consumer<Singer>` объявляется таким образом, чтобы удачно порожденный объект типа `Singer` был сохранен в информационном хранилище `singerRepository`. Этот метод возвращает пустой экземпляр типа `Mono`, так как возвращать, собственно, нечего.

Теперь, когда имеется реактивное хранилище, для него потребуется реактивный обработчик запросов. В типичной для Spring аннотации `@Controller` просто употребляются потоки данных, а их содержимое возвращается в представлении, и поэтому в данном случае придется выбрать нечто другое. Ниже приведен исходный код класса `SingerHandler`, реализующего реактивный обработчик запросов.

```
package com.apress.prospring5.ch18.web;

import com.apress.prospring5.ch18.entities.Singer;
import com.apress.prospring5.ch18.repos.ReactiveSingerRepo;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.server
          .ServerRequest;
import org.springframework.web.reactive.function.server
          .ServerResponse;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import static org.springframework.http.MediaType
                 .APPLICATION_JSON;
import static org.springframework.web.reactive.function
                 .BodyInserters.fromObject;

@Component
public class SingerHandler {

  @Autowired ReactiveSingerRepo reactiveSingerRepo;
  public Mono<ServerResponse> list(ServerRequest request) {
    Flux<Singer> singers = reactiveSingerRepo.findAll();
    return ServerResponse.ok().contentType(APPLICATION_JSON)
                         .body(singers, Singer.class);
  }

  public Mono<ServerResponse> show(ServerRequest request) {
    Mono<Singer> singerMono = reactiveSingerRepo
       .findById(Long.valueOf(request.pathVariable("id")));
    Mono<ServerResponse> notFound =
         ServerResponse.notFound().build();
```

```
    return singerMono.flatMap(singer -> ServerResponse.ok()
                    .contentType(APPLICATION_JSON)
                    .body(fromObject(singer)))
                    .switchIfEmpty(notFound);
  }

  public Mono<ServerResponse> save(ServerRequest request) {
    Mono<Singer> data = request.bodyToMono(Singer.class);
    reactiveSingerRepo.save(data);
    return ServerResponse.ok().build(
                        reactiveSingerRepo.save(data));
  }
}
```

Обработчик запросов принимает в качестве параметра объект запроса типа `ServerRequest` и возвращает ответный поток данных типа `Mono<ServerResponse>`. Оба интерфейса, `ServerRequest` и `ServerResponse`, являются неизменяемыми и полностью реактивными и предоставляют доступ к базовым сообщениям по сетевому протоколу HTTP. В частности, интерфейс `ServerRequest` обеспечивает доступ к телу запроса в виде потока данных типа `Flux` или `Mono`, а интерфейс `ServerResponse` принимает любой реактивный поток данных в качестве тела ответа.

Интерфейс `ServerRequest` предоставляет доступ и к другим данным, связанным с HTTP-запросом, включая обрабатываемый URI, заголовки и переменные путей. Доступ к телу запроса осуществляется с помощью метода `bodyToMono()` или равнозначного метода `bodyToFlux()`. А интерфейс `ServerResponse` предоставляет доступ к HTTP-ответу. Но поскольку это неизменяемый интерфейс, то HTTP-ответ составляется с помощью класса построителя, исходя из кода его состояния, а для установки заголовков и тела ответа вызываются различные методы. Так, в приведенном примере кода составляется ответ с кодом состояния `200` (Нормально), типом содержимого JSON и соответствующим телом.

Но откуда происходит название *функциональный каркас веб-приложений*? А происходит оно от лямбда-выражений. Проанализировав приведенный выше фрагмент кода, можно прийти к выводу, что методы `list()` и `save()` можно было бы легко написать и в виде лямбда-функций, как показано ниже.

```
package com.apress.prospring5.ch18.web;

import com.apress.prospring5.ch18.entities.Singer;
import com.apress.prospring5.ch18.repos.ReactiveSingerRepo;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.server
          .HandlerFunction;
import org.springframework.web.reactive.function.server
          .ServerRequest;
```

```
import org.springframework.web.reactive.function.server
          .ServerResponse;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import static org.springframework.http.MediaType
                .APPLICATION_JSON;
import static org.springframework.web.reactive.function
                .BodyInserters.fromObject;

@Component
public class SingerHandler {

  @Autowired ReactiveSingerRepo reactiveSingerRepo;

  public HandlerFunction<ServerResponse> list =
         serverRequest -> ServerResponse.ok()
                          .contentType(APPLICATION_JSON)
                          .body(reactiveSingerRepo
                          .findAll(), Singer.class);

  public Mono<ServerResponse> show(ServerRequest request) {
    Mono<Singer> singerMono = reactiveSingerRepo
       .findById(Long.valueOf(request.pathVariable("id")));
    Mono<ServerResponse> notFound =
         ServerResponse.notFound().build();
    return singerMono.flatMap(singer -> ServerResponse.ok()
                     .contentType(APPLICATION_JSON)
                     .body(fromObject(singer)))
                     .switchIfEmpty(notFound);
  }

  public HandlerFunction<ServerResponse> save =
         serverRequest -> ServerResponse.ok()
                          .build(reactiveSingerRepo
                          .save(serverRequest
                          .bodyToMono(Singer.class)));
}
```

Интерфейс `HandlerFunction<ServerResponse>`, который, по существу, является функциональным интерфейсом `Function<Request, Response<T>>`, лишен побочных эффектов, поскольку он возвращает ответ непосредственно, а не принимает его в качестве параметра. В итоге лямбда-функции становятся очень удобными для практического применения, поскольку их легко тестировать, составлять и оптимизировать. Не следует, однако, забывать, что злоупотребление лямбда-выражениями делает исходный код неудобочитаемым и затрудняет его сопровождение.

И здесь у вас может возникнуть вполне обоснованный вопрос: реактивные потоки данных и лямбда-выражения — это, конечно, замечательно, но где же функциональ-

ные преобразования? Откуда контейнеру известно, какая именно функция преобразуется в HTTP-запрос? Что ж, для этой цели предусмотрен ряд новых классов.

Ниже приведен сначала исходный код класса, с которого следует запускать на выполнение рассматриваемое здесь приложение Spring, а затем его анализ.

```
package com.apress.prospring5.ch18;

import com.apress.prospring5.ch18.web.SingerHandler;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation
        .Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure
        .SpringBootApplication;
import org.springframework.boot.web.servlet
        .ServletRegistrationBean;
import org.springframework.context
        .ConfigurableApplicationContext;
import org.springframework.context.annotation.Bean;

import org.springframework.http.server.reactive
        .HttpHandler;
import org.springframework.http.server.reactive
        .ServletHttpHandlerAdapter;
import org.springframework.web.reactive.function.server
        .RouterFunction;
import org.springframework.web.reactive.function.server
        .ServerResponse;
import org.springframework.web.server.WebHandler;
import org.springframework.web.server.adapter
        .WebHttpHandlerBuilder;

import static org.springframework.web.reactive.function
                .BodyInserters.fromObject;
import static org.springframework.web.reactive.function
                .server.RequestPredicates.GET;
import static org.springframework.web.reactive.function
                .server.RequestPredicates.POST;
import static org.springframework.web.reactive.function
                .server.RouterFunctions.route;
import static org.springframework.web.reactive.function
                .server.RouterFunctions.toHttpHandler;
import static org.springframework.web.reactive.function
                .server.ServerResponse.ok;

@SpringBootApplication
public class SingerApplication {
```

```
  private static Logger logger =
          LoggerFactory.getLogger(SingerApplication.class);

  @Autowired
  SingerHandler singerHandler;

  private RouterFunction<ServerResponse>
                          routingFunction() {
    return route(GET("/test"), serverRequest -> ok()
          .body(fromObject("works!")))
          .andRoute(GET("/singers"), singerHandler.list)
          .andRoute(GET("/singers/{id}"),
                    singerHandler::show)
          .andRoute(POST("/singers"), singerHandler.save)
          .filter((request, next) -> {
             logger.info("Before handler invocation: "
                      + request.path());
             return next.handle(request);
          });
  }

  @Bean
  public ServletRegistrationBean servletRegistrationBean()
         throws Exception {
    HttpHandler httpHandler =
        RouterFunctions.toHttpHandler(routingFunction());
    ServletRegistrationBean registrationBean =
        new ServletRegistrationBean<>
         (new ServletHttpHandlerAdapter(httpHandler), "/");
    registrationBean.setLoadOnStartup(1);
    registrationBean.setAsyncSupported(true);
    return registrationBean;
  }

  public static void main(String... args)
         throws Exception {
    ConfigurableApplicationContext ctx =
      SpringApplication.run(SingerApplication.class, args);
    assert (ctx != null);
    logger.info("Application started...");

    System.in.read();
    ctx.close();
  }
}
```

Функции обработки запросов доступны из нового функционального каркаса веб-приложений через функциональный интерфейс `RouterFunction`, предназначенный для создания обработчика HTTP-запросов типа `HttpHandler` с помощью служебно-

го метода `RouterFunctions.toHttpHandler()`. Этот обработчик, в свою очередь, служит для создания компонента Spring Bean типа `ServletRegistrationBean`, применяемого в интерфейсе `ServletContextInitializer` для регистрации сервлетов в их контейнере, начиная с версии Servlet 3.0.

В интерфейсе `RouterFunction` анализируется URI запроса и проверяется, имеется ли в нем соответствующая функция обработки. Если таковая отсутствует, то возвращается пустой результат. По своему поведению интерфейс `RouterFunction` похож на аннотацию `@RequestMapping`, но выгодно отличается от нее тем, что определение маршрута не ограничивается только теми значениями, которые могут быть заданы в аннотациях, рассеянных по всему классу. Напротив, весь код находится в одном месте и может быть легко переопределен или заменен. Синтаксис для написания и составления функций маршрутизации с помощью интерфейса `RouterFunction` очень удобный. Так, в приведенном выше коде был выбран наиболее практичный синтаксис[16].

Чтобы проверить рассматриваемое здесь веб-приложение, проще всего выполнить исходный код класса `SingerApplication` и получить из браузера доступ к веб-ресурсам по URI, определенным в функциях маршрутизации с помощью интерфейса `RouterFunction`. Так, если ввести в окне браузера ссылку `http://localhost:8080/test`, то на экране появится текстовое сообщение `"works!"` (работает).

Но мы еще не продемонстрировали пример применения потоков данных, где каждое значение обрабатывается по мере его порождения. С этой целью создадим контроллер REST, непосредственно возвращающий поток данных о певцах в виде объекта типа `Flux`, но порождающий этот объект через каждые две секунды.

```
package com.apress.prospring5.ch18;

import com.apress.prospring5.ch18.entities.Singer;
import com.apress.prospring5.ch18.repos.ReactiveSingerRepo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure
          .SpringBootApplication;
import org.springframework.context
          .ConfigurableApplicationContext;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation
          .GetMapping;
import org.springframework.web.bind.annotation
          .RestController;
```

[16] За более подробными сведениями обращайтесь на официальный блог, который ведется по Spring на странице, доступной по адресу `https://spring.io/blog/2016/09/22/new-in-spring-5-functional-web-framework`.

```
import reactor.core.publisher.Flux;
import reactor.util.function.Tuple2;
import java.time.Duration;

@SpringBootApplication
@RestController
public class ReactiveApplication {

  private static Logger logger =
       LoggerFactory.getLogger(ReactiveApplication.class);

  @Autowired ReactiveSingerRepo reactiveSingerRepo;
  @GetMapping(value = "/all",
              produces = MediaType.TEXT_EVENT_STREAM_VALUE)
  public Flux<Singer> oneByOne() {
    Flux<Singer> singers = reactiveSingerRepo.findAll();
    Flux<Long> periodFlux =
                Flux.interval(Duration.ofSeconds(2));
    return Flux.zip(singers, periodFlux).map(Tuple2::getT1);
  }

  @GetMapping(value = "/one/{id}")
  public Mono<Singer> one(@PathVariable Long id) {
    return reactiveSingerRepo.findById(id);
  }

  public static void main(String... args)
         throws Exception {
    ConfigurableApplicationContext ctx =
                SpringApplication.run(
                      ReactiveApplication.class, args);
    assert (ctx != null);
    logger.info("Application started...");
    System.in.read();
    ctx.close();
  }
}
```

Константа `MediaType.TEXT_EVENT_STREAM_VALUE` обозначает специальный тип ответа простым текстом. Этим гарантируется, что на сервере будет составлен ответ в формате отравляемых событий под заголовком `Content-Type: text/event-stream`. Ответ должен содержать строку `"data:"` и последующее сообщение в этом формате. В данном случае после текстового представления объекта типа `Singer` следуют подряд две последовательности символов `"\n"`, чтобы завершить поток данных. Это относится лишь к одному сообщению и означает, что порождается один объект. Если же порождается несколько объектов, то несколько строк `"data:"` будет сформировано как единый фрагмент данных, а следовательно, будет инициировано только одно событие от сообщения. Каждая из этих строк должна завершаться един-

ственной последовательностью символов "\n", кроме последней строки, которая должна завершаться двумя подобными последовательностями.

В методе oneByOne() реализуются два объединяемых потока данных. Один из них относится к типу Flux, возвращается из реактивного хранилища и содержит экземпляры объектов типа Singer, а второй содержит промежуток в секундах, формируемый с помощью класса java.time.Duration. Метод zip() служит для объединения двух потоков данных. Как только оба потока породят по одному элементу, эти элементы объединятся в одно выводимое значение с помощью специально предоставляемого объединителя. И эта операция будет продолжаться вплоть до исчерпания любого источника данных. В данном случае в качестве объединителя служит оператор map(Tuple2::getT1) со ссылкой на метод, хотя он может быть написан как map(t -> t.getT1()), что легче понять и, по существу, означает, что один элемент потока преобразуется в другой.

Теперь проверим обе новые функции обработки запросов. Это можно сделать двумя способами, используя браузер или команду curl. Так, если попытаться перейти в окне браузера по ссылке localhost:8080/, то появится лишь пустая страница, а браузер выдаст запрос на ее сохранение в файле. Если дать на это разрешение, то появится всплывающее окно, в котором отображается ход выполнения операции сохранения. В какой-то момент она завершится, и сохраненное содержимое файла появится в избранном редакторе. Если в этом редакторе включен режим отображения непечатных символов, то должны быть видны символы окончания строки "\n". В качестве примера на рис. 18.2 показано содержимое файла, загруженного подобным образом.

all-1 — Downloads

```
1  data:{"id":1,"version":0,"firstName":"John","lastName":"Mayer","birthDate":"1977-10-15"}
2
3  data:{"id":2,"version":0,"firstName":"Eric","lastName":"Clapton","birthDate":"1945-03-29"}
4
5  data:{"id":3,"version":0,"firstName":"John","lastName":"Butler","birthDate":"1975-03-31"}
6
7  data:{"id":4,"version":0,"firstName":"B.B.","lastName":"King","birthDate":"1925-10-15"}
8
9  data:{"id":5,"version":0,"firstName":"Jimi","lastName":"Hendrix","birthDate":"1942-12-26"}
10
11 data:{"id":6,"version":0,"firstName":"Jimmy","lastName":"Page","birthDate":"1944-02-08"}
12
13 data:{"id":7,"version":0,"firstName":"Eddie","lastName":"Van Halen","birthDate":"1955-02-25"}
14
15 data:{"id":8,"version":0,"firstName":"Saul (Slash)","lastName":"Hudson","birthDate":"1965-08-22"}
16
17 data:{"id":9,"version":0,"firstName":"Stevie","lastName":"Ray Vaughan","birthDate":"1954-11-02"}
18
19 data:{"id":10,"version":0,"firstName":"David","lastName":"Gilmour","birthDate":"1946-04-05"}
20
21 data:{"id":11,"version":0,"firstName":"Kirk","lastName":"Hammett","birthDate":"1992-12-17"}
22
23 data:{"id":12,"version":0,"firstName":"Angus","lastName":"Young","birthDate":"1955-04-30"}
24
25 data:{"id":13,"version":0,"firstName":"Dimebag","lastName":"Darrell","birthDate":"1966-09-19"}
26
27 data:{"id":14,"version":0,"firstName":"Carlos","lastName":"Santana","birthDate":"1947-08-19"}
28
29
```

Рис. 18.2. Ответ в формате, задаваемом под заголовком **Content-Type: text/event-stream**

Но, помимо медленной загрузки, в данном случае не видно, что объект типа `Singer` порождается каждые две секунды. Это можно увидеть только в том случае, если воспользоваться командой `curl`, доступной в Unix-подобных системах. А в Windows можно попробовать выполнить команду `Invoke-RestMethod` в оболочке PowerShell. Если выполнить команду `curl` с URI, сопоставленными в предыдущем примере кода, то на консоль будет выведен приведенный ниже результат. Следует, однако, иметь в виду, что для выполнения команды `curl http://localhost:8080/all` потребуется определенное время, но каждая строка будет выводиться через каждые две секунды.

```
$ curl http://localhost:8080/one/1
{"id":1,"version":0,"firstName":"John",
  "lastName":"Mayer","birthDate":"1977-10-15"}
$ curl http://localhost:8080/all
data:{"id":1,"version":0,"firstName":"John",
  "lastName":"Mayer","birthDate":"1977-10-15"}

data:{"id":2,"version":0,"firstName":"Eric",
  "lastName":"Clapton","birthDate":"1945-03-29"}

data:{"id":3,"version":0,"firstName":"John","lastName":"Butler",
  "birthDate":"1975-03-31"}

data:{"id":4,"version":0,"firstName":"B.B.","lastName":"King",
  "birthDate":"1925-10-15"}

data:{"id":5,"version":0,"firstName":"Jimi",
  "lastName":"Hendrix","birthDate":"1942-12-26"}

data:{"id":6,"version":0,"firstName":"Jimmy",
  "lastName":"Page","birthDate":"1944-02-08"}

data:{"id":7,"version":0,"firstName":"Eddie",
  "lastName":"Van Halen","birthDate":"1955-02-25"}

data:{"id":8,"version":0,"firstName":"Saul Slash",
  "lastName":"Hudson","birthDate":"1965-08-22"}

data:{"id":9,"version":0,"firstName":"Stevie",
  "lastName":"Ray Vaughan","birthDate":"1954-11-02"}

data:{"id":10,"version":0,"firstName":"David",
  "lastName":"Gilmour","birthDate":"1946-04-05"}

data:{"id":11,"version":0,"firstName":"Kirk",
  "lastName":"Hammett","birthDate":"1992-12-17"}

data:{"id":12,"version":0,"firstName":"Angus",
```

```
  "lastName":"Young","birthDate":"1955-04-30"}

data:{"id":13,"version":0,"firstName":"Dimebag",
  "lastName":"Darrell","birthDate":"1966-09-19"}

data:{"id":14,"version":0,"firstName":"Carlos",
  "lastName":"Santana","birthDate":"1947-08-19"}

$
```

По URI типа `http://localhost:8080/one/1` данные автоматически преобразуются в текст, поскольку это стандартный заголовок приема, и в конечном итоге возвращается единственная запись. А по URI типа `http://localhost:8080/all` запрос зависает, и каждое сообщение преобразуется в текст по мере его поступления. Обратите внимание на разный синтаксис. Так, в потоке из нескольких сообщений каждый отправляемый элемент снабжается префиксом **`"data:"`**.

Еще один способ получить доступ к потоку данных, возвращаемому по HTTP-запросу, отправленному по ссылке `http://localhost:8080/all`, состоит в применении реактивного веб-клиента. Для этого придется, конечно, предоставить пользовательский интерфейс, и здесь поясняется, как это делается. В связи с тем, что требуется общий контекст, объявим клиент в том же самом классе `Application` из модуля Spring Boot, как показано ниже. И если затем обратить внимание на консоль, то можно заметить, что клиент запускается на выполнение сразу же после данного веб-приложения.

```
package com.apress.prospring5.ch18;
...
@SpringBootApplication
@RestController
public class ReactiveApplication {
...
  @Bean WebClient client() {
    return WebClient.create("http://localhost:8080");
  }

  @Bean CommandLineRunner clr(WebClient client) {
    return args -> {
      client.get().uri("/all")
            .accept(MediaType.TEXT_EVENT_STREAM)
            .exchange()
            .flatMapMany(cr -> cr.bodyToFlux(Singer.class))
            .subscribe(System.out::println);
    };
  }
}
```

Компонент Spring Bean типа `WebClient` служит в качестве реактивной, неблокирующей альтернативы компоненту типа `RestTemplate`, который в последующих

версиях Spring должен прийти на смену компоненту типа `AsyncRestTemplate`. А в версии Spring 5 последний уже отмечен как не рекомендованный к употреблению.

В состав модуля Spring WebFlux входит немало новых компонентов, которыми можно воспользоваться при разработке реактивных веб-приложений, но рамки данной книги не позволяют описать все эти компоненты. Поэтому рекомендуется регулярно посещать официальный блог, который ведется по Spring на странице, доступной по адресу `https://spring.io/blog`, чтобы следить за внедрением новых компонентов и пояснениями относительно их применения. А теперь перейдем к другим функциональным средствам, внедренным в версии Spring 5.

Совместимость с версией Java 9

В связи с тем что выпуск версии Java 9 был отложен, версию Spring 5 пришлось выпустить на основе Java 8, но команда разработчиков не прекращала параллельно работать над адаптацией Spring к версии Java 9, пользуясь ранними версиям сборок JDK. На момент написания данной книги большинство новых компонентов Java 9 уже функционировали устойчиво, хотя и планировались к выпуску лишь в сентябре 2017 года. Поэтому описание совместимости с версией Java 9 в данной книге опирается лишь на ранние версии сборок JDK.

Модульность комплекта JDK

Модульность JDK относится к самым крупным усовершенствованиям в версии Java 9. Переход на модульный характер комплекта JDK дает преимущества в масштабируемости, поскольку теперь приложения Java могут быть развернуты на более мелких устройствах. Функциональные средства модульности разрабатывались в рамках проекта Jigsaw и планировались к внедрению еще в версии Java 8, но в конечном итоге они были перенесены в версию Java 9, поскольку еще не считались достаточно устойчивыми. Поэтому понятие модуля введено в версии Java 9 и обозначает программный модуль, конфигурируемый в файле `module-info.java`, находящемся в каталоге исходного кода проекта и содержащем следующие сведения.

- **Имя модуля согласно принятым соглашениям об именовании пакетов.** По принятым соглашениям имя модуля должно быть таким же, как и у корневого пакета.
- **Ряд экспортируемых пакетов.** Такие пакеты считаются открытыми прикладными интерфейсами API, предназначенными для применения в других модулях. Если класс входит в состав экспортируемого пакета, он может быть доступен и использован за пределами модуля.
- **Ряд обязательных пакетов.** Это модули, от которых зависит данный модуль. Все типы данных, открытые в пакетах, экспортируемых в такие модули, могут быть доступны и использованы в зависящем от них модуле.

Приведенные выше сведения, по существу, означают, что доступность больше не обеспечивается только четырьмя модификаторами: `public`, `private`, `default` и `protected`. А поскольку при конфигурировании модуля решается, каким из других зависимых от него модулей можно воспользоваться, то в отношении доступности необходимо принимать во внимание следующее.

- Открытость на уровне модуля везде, где этот модуль читается (оператор `exports`).
- Открытость на уровне модуля из списка модулей (это фильтруемый доступ, оператор `export to`).
- Открытость в самом модуле и для всех остальных классов в этом модуле.
- Закрытость в самом модуле (типичный закрытый доступ).
- Доступ по умолчанию в самом модуле (типичный доступ по умолчанию).
- Защищенный доступ в самом модуле (типичный защищенный доступ).

Эта модель была применена в комплекте JDK, и благодаря ей было устранено немало циклических и неочевидных зависимостей. Для этого потребовалось также внести ряд правок, поскольку в JDK имелись пакеты, выходившие за рамки JSE[17].

Рассматриваемый здесь вопрос очень важен для совместимости с Spring, поскольку некоторые компоненты JDK могут оказаться больше недоступными непосредственно. До сих пор о версии Java 9 в данной книге почти ничего не говорилось, поскольку рассматриваемый здесь проект Spring был построен на основе Java 8, а версия Java 9 была еще не устойчивой, чтобы воспользоваться ею в данном проекте. Но теперь для этого настало время, и поэтому после внесения корректив в комплект JDK установку Gradle придется обновить до последней промежуточной версии 4.2, чтобы обеспечить поддержку версии Java 9. После этого первая же сборка текущего проекта завершится неудачно со следующим сообщением об ошибке:

```
$ gradle clean build -x test
> Task :chapter03:collections:compileJava
/workspace/pro-spring-15/chapter03/collections/src/main
/java/com/apress/prospring5/ch3/annotated
/CollectionInjection.java:13:
  error: package javax.annotation is not visible[18]
  import javax.annotation.Resource;
         ^

  (package javax.annotation is declared in module
   java.xml.ws.annotation, which is not in the module graph)
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

[17] Подробнее об этом можно узнать по адресу `https://blog.codecentric.de/en/2015/11/first-steps-with-java9-jigsaw-part-1/`.

[18] ошибка: пакет javax.annotation недоступен

```
1 error

FAILURE: Build failed with an exception.[19]
```

В данном случае пакет `javax.annotation` не обнаружен среди экспортированных, поэтому любые объявленные в нем аннотации не могут быть больше использованы. Недоступность данного пакета и многих связанных с ним пакетов объясняется тем, что они содержат корпоративные компоненты, более пригодные для включения в состав платформы JEE. Эти пакеты включены в состав модуля `java.se.ee`, и чтобы все заработало как следует, необходимо, как описано выше, включить в файл `module-info.java` следующий модуль:

```
module collections.main {
    requires java.se.ee;
}
```

Но если теперь открыть файл `module-info.java` описания модуля `java.se.ee`, то в нем обнаружится следующее:

```
@java.lang.Deprecated(since = "9", forRemoval = true)
module java.se.ee {
    requires transitive java.xml.bind;
    requires transitive java.activation;
    requires transitive java.corba;
    requires transitive java.se;
    requires transitive java.transaction;
    requires transitive java.xml.ws;
    requires transitive java.xml.ws.annotation;
}
```

Как видите, в этом файле отсутствует строка с оператором `exports java.xml.ws.annotation;`, где экспортируется модуль, содержащий пакет `javax.annotation`. Какой же выход из этого затруднительного положения? Разумеется, необходимо внедрить зависимость от платформы JEE, содержащую данный пакет. Ниже показано, каким образом эта зависимость определяется в файле конфигурации Gradle.

```
misc = [
    ...
    jsr250 : "javax:javaee-endorsed-api:7.0"
    ...
]
```

---

[19] (пакет javax.annotation объявлен в модуле java.xml.ws.annotation, отсутствующем в графе модулей)
Примечание: в некоторых входных файлах применяются непроверяемые и небезопасные операции.
Примечание: для получения более подробных сведений, повторите компиляцию с параметром -Xlint:unchecked
Обнаружена 1 ошибка
ОТКАЗ: сборка завершилась неудачно с исключением.

Безусловно, зависимость `misc.jsr250` придется внедрить везде, где применяются аннотации из пакета `javax.annotation`. Если сделать еще одну попытку построить текущий проект, то результат может оказаться похожим на приведенный ниже, хотя это зависит от того, когда вы будете читать данную книгу.

```
gradle clean build -x test
# до сих пор все хорошо
...
:chapter05:aspectj-aspects:compileAspect
[ant:iajc] java.nio.file.NoSuchFileException:
  /Library/Java/JavaVirtualMachines/jdk-9.jdk/Contents
  /Home/jrt-fs.jar
[ant:iajc] at java.base/sun.nio.fs.UnixException
              .translateToIOExceptionUnixException.java:92
[ant:iajc] at java.base/sun.nio.fs.UnixException
              .rethrowAsIOExceptionUnixException.java:111
...
[ant:iajc] /workspace/pro-spring-15/chapter05
           /aspectj-aspects/src/main/java/com/apress
           /prospring5/ch5/MessageWriter.java:
  5 [error] Implicit super constructor Object is undefined.
  Must explicitly invoke another constructor
[ant:iajc] public MessageWriter {
[ant:iajc]        ^^^^^^^^^^
[ant:iajc] /workspace/pro-spring-15/chapter05
           /aspectj-aspects/src/main/java/com/apress
           /prospring5/ch5/MessageWriter.java:
  9 [error] System cannot be resolved
[ant:iajc] System.out.println"foobar!";
[ant:iajc]
[ant:iajc] /workspace/pro-spring-15/chapter05
           /aspectj-aspects/src/main/java/com/apress
           /prospring5/ch5/MessageWriter.java:
  13 [error] System cannot be resolved
[ant:iajc] System.out.println"foo";
[ant:iajc]
[ant:iajc]
[ant:iajc] 11 errors
:chapter05:aspectj-aspects:compileAspect FAILED
276 actionable tasks: 186 executed, 90 up-to-date
```

В данном случае проект `aspectj-aspects` не удалось скомпилировать, поскольку аспекты не распознаны. И это произошло потому, что соответствующий архивный JAR-файл отсутствовал в нужном месте согласно требованиям модуля `aspects`, подключаемого к Gradle. Этот подключаемый модуль явно устарел и не отвечает большей части новой внутренней структуры JDK, но это положение можно исправить, скопировав архивный файл `jrt-fs.jar` из каталога `$JAVA_HOME/libs` в то месте, где его пытается обнаружить подключаемый модуль `aspects`.

В процессе сборки появится немало предупреждений об устаревших компонентах, не рекомендованных к применению, но, по крайней мере, сборка завершится удачно. И теперь можно при желании добавить файлы `module-info.java` ко всем модулям. Возьмем ради простоты примера небольшой проект `chapter02/hello-world` и определим для него файл `module-info.java`, как показано ниже.

```
// Файл описания модуля pro-spring-15/chapter02/hello-world
//                      /src/main/java/module-info.java
module com.apress.prospring5.ch2 {
    requires spring.context;
    requires logback.classic;
    exports com.apress.prospring5.ch2;
}
```

Вот, собственно, и все. Чтобы придерживаться простых имен модулей, команда разработчиков Spring решила нарушить соглашение об именовании. Ведь в противном случае вместо оператора `requires spring.context;` пришлось бы ввести оператор `requires org.springframework.context`[20].

Модульность, внедренная в проекте Jigsaw, не только разделяет комплект JDK и строгий доступ к определенным пакетам и компонентам (при этом рефлексия не пригодна для неэкспортируемых модулей), но и позволяет обнаруживать циклические зависимости на стадии компиляции, повысить удобочитаемость исходного кода и выявлять дублирующиеся зависимости от модулей, отличающиеся только версией, устраняя тем самым конфликты или хаотичное поведение приложения. А поскольку все это приносит явные выгоды на стадии разработки, то данное нововведение в версии Java 9 заслуживает отдельного раздела в книге.

Реактивное программирование средствами JDK 9 и Spring WebFlux

Ранее в этой главе была представлена реактивная модель и упоминался стандартный для нее прикладной интерфейс API. Но, начиная с версии Java 9, придется отказаться от архивного файла `reactive-streams.jar`, поскольку теперь в JDK внедрен новый модуль `java.base`, экспортирующий пакет `java.util.concurrent`, который, среди прочего, содержит четыре функциональных интерфейса, имеющих то же самое назначение и определенных в классе `java.util.concurrent.Flow`, как поясняется ниже.

- `Flow.Processor` — равнозначен интерфейсу `org.reactivestreams.Processor<T>`.
- `Flow.Publisher` — равнозначен интерфейсу `org.reactivestreams.Publisher<T>`.

[20] О дискуссии, приведшей к такому решению, см. по адресу `https://jira.spring.io/browse/SPR-14579`.

- `Flow.Subscriber` — равнозначен интерфейсу `org.reactivestreams.Subscriber<T>`.
- `Flow.Subscription` — равнозначен интерфейсу `org.reactivestreams.Subscription<T>`.

Эти интерфейсы выполняют те же функции, что и интерфейсы, определенные в проекте Reactive Streams. Они служат для поддержки приложений, создаваемых по модели "издатель–подписчик", т.е. реактивных приложений. В версии JDK 9 предоставляется простая реализация функционального интерфейса `Publisher`, предназначенная для простых вариантов использования, хотя она может быть также расширена в зависимости от конкретных требований. В частности, класс `SubmissionPublisher<T>` предоставляет реализацию не только функционального интерфейса `Flow.Publisher<>`, но и интерфейса `AutoCloseable`, а следовательно, он может применяться и в блоке оператора `try` с ресурсами.

Библиотека RxJava[21] служит в качестве одной из реактивных реализаций для виртуальной машины JVM. Она предназначена для поддержки последовательностей данных и событий и предоставляет дополнительные операции, позволяющие составлять и фильтровать потоки данных, беря на себя такие низкоуровневые обязанности, как синхронизация и безопасность потоков исполнения. На момент написания данной книги имелись две версии библиотеки RxJava: RxJava 1.*x* и RxJava 2.*x*. Так, в версии RxJava 1.*x* реализован прикладной интерфейс ReactiveX (Reactive Extensions) API для асинхронного программирования с наблюдаемыми потоками[22]. Но версия RxJava 1.*x* отменена с 2018 года, так как полностью переписана на основе представленного ранее прикладного интерфейса Reactive Streams API. И хотя в Spring 5 можно применять обе упомянутых версии библиотеки RxJava, далее будет рассматриваться вторая версия, как более перспективная.

В представленном ранее примере реактивного программирования в Spring с помощью библиотеки Project Reactor были реализованы реактивное хранилище и реактивный клиент, а в рассматриваемом здесь примере то же самое будет сделано с помощью библиотеки RxJava 2.*x*. Сначала определим интерфейс `Rx2SingerRepo` для реактивного хранилища, как показано ниже.

```
package com.apress.prospring5.ch18.repos;

import com.apress.prospring5.ch18.entities.Singer;
import io.reactivex.Flowable;
import io.reactivex.Single;

public interface Rx2SingerRepo {
  Single<Singer> findById(Long id);
  Flowable<Singer> findAll();
  Single<Void> save(Single<Singer> singer);
}
```

[21] Подробнее см. по адресу `https://github.com/ReactiveX/RxJava`.

[22] Подробнее см. по адресу `http://reactivex.io/`.

Внимательно проанализировав приведенный выше фрагмент кода, вы, вероятно, заметили следующее сходство: в классе `Flowable` реализуется поток элементов `[0..n]`, а в классе `Single` — поток элементов `[0..1]`. Создать эти потоки также не составляет большого труда, а в прикладном интерфейсе API можно обнаружить небольшие отличия для типа `Single`, касающиеся создания пустого потока данных, как показано ниже.

```
Flowable simple = Flowable.just("1", "2", "3");
// или из существующего списка: List<Singer>
Flowable<Singer> fromList = Flowable.fromIterable(list);

Single simple = Single.just("1");
// или из существующего объекта типа Singer
Single<Singer> fromObject = Single.just(null);
```

А так выглядит реализация реактивного хранилища с помощью библиотеки RxJava 2.*x*:

```
package com.apress.prospring5.ch18.repos;

import com.apress.prospring5.ch18.entities.Singer;
import io.reactivex.Flowable;
import io.reactivex.Single;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.stereotype.Component;

@Component
public class Rx2SingerRepoImpl implements Rx2SingerRepo {

  @Autowired
  SingerRepository singerRepository;
  @Override public Single<Singer> findById(Long id) {
    return Single.just(
                  singerRepository.findById(id).get());
  }

  @Override public Flowable<Singer> findAll() {
    return Flowable.fromIterable(
                    singerRepository.findAll());
  }

  @Override public Single<Void>
            save(Single<Singer> singerSingle) {
    singerSingle.doOnSuccess(singer ->
                singerRepository.save(singer));
    return Single.just(null);
  }
}
```

Обратите внимание на сходство синтаксиса в приведенном выше коде, хотя в нем можно заметить и небольшие отличия в прикладном интерфейсе API, которые следует принять во внимание. Если переписать и методы преобразования, то они будут выглядеть так:

```
package com.apress.prospring5.ch18;
...
@SpringBootApplication
@RestController
public class Rx2ReactiveApplication {

  private static Logger logger =
     LoggerFactory.getLogger(Rx2ReactiveApplication.class);

  @Autowired Rx2SingerRepo rx2SingerRepo;

  @GetMapping(value = "/all",
              produces = MediaType.TEXT_EVENT_STREAM_VALUE)
  public Flowable<Singer> all() {
    Flowable<Singer> singers = rx2SingerRepo.findAll();
    Flowable<Long> periodFlowable =
                   Flowable.interval(2, TimeUnit.SECONDS);
    return singers.zipWith(periodFlowable,
           (singer, aLong) -> {
                    Thread.sleep(aLong);
                    return singer;
    });
  }

  @GetMapping(value = "/one/{id}")
  public Single<Singer> one(@PathVariable Long id) {
    return rx2SingerRepo.findById(id);
  }
  ...
}
```

Итак, реализовать функцию архивирования с помощью библиотеки RxJava 2.*x* немного труднее. Для проверки преобразований можно по-прежнему воспользоваться упоминавшимся ранее реактивным компонентом Spring Bean типа `WebClient`, который вполне для этого подходит. А с другой стороны, для проведения тестов можно воспользоваться новым HTTP-клиентом из комплекта JDK 9, как показано ниже. Нужно лишь внедрить зависимость в модуль `jdk.incubator.httpclient`, определив ее в файле `module-info.java`.

```
import jdk.incubator.http.HttpClient;
import jdk.incubator.http.HttpRequest;
import jdk.incubator.http.HttpResponse;
...
```

```
URI oneURI = new URI("http://localhost:8080/one/1");
HttpClient client = HttpClient.newBuilder().build();

HttpRequest httpRequest = HttpRequest.newBuilder()
                                     .GET().build();
HttpResponse httpResponse = client.send(httpRequest,
             HttpResponse.BodyHandler.asString());

System.out.println(httpResponse.statusCode());
System.out.println(httpResponse.body());
```

Вот, собственно, все, что можно сказать о совместимости с версией Java 9. Самые крупные изменения в каркасе Spring Framework относятся к модуляризации и новому реактивному прикладному интерфейсу API из библиотеки RxJava 2.*x*, который уже поддерживается в Spring. Со всеми остальными нововведениями в версии Java 9 можно ознакомиться в другой интересной книге *Java 9 Revealed*, вышедшей в издательстве Apress в 2017 году (в русском переводе книга вышла под названием *Java 9. Полный обзор нововведений* в издательстве "ДМК-Пресс", 2018 г.).

Поддержка JUnit 5 Jupiter в Spring

О платформе JUnit 5 для модульного тестирования кратко упоминалось в главе 13, теперь настало время рассмотреть ее более подробно. Если вам любопытно, в чем здесь дело, простейший ответ на этот вопрос вы найдете на странице официальной документации на JUnit 5, оперативно доступной по адресу `http://junit.org/junit5/docs/current/user-guide/#overview`.

Платформа JUnit служит основанием для запуска на выполнение тестовых сред на виртуальной машине JVM (рис. 18.3). В состав этой платформы входит модуль Console Launcher, предназначенный для ее запуска из командной строки и построения модулей, подключаемых к Gradle и Maven. Этим модулем запуска можно пользоваться для выявления, отсеивания и выполнения тестов, не прибегая к услугам подключаемого модуля Surefire или специальной настройке тестовых задач в Gradle.

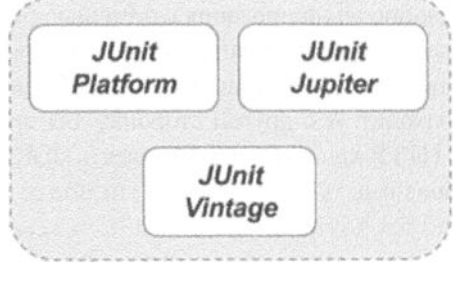


Рис. 18.3. Составные части JUnit 5

Имеется также возможность подключать сторонние библиотеки вроде Spock, Cucumber или FitNesse к инфраструктуре запуска платформы JUnit, предоставив для этого специальную реализацию интерфейса `TestEngine`.

Для написания модульных тестов и расширений в JUnit 5 служит подпроект JUnit Jupiter, сочетающий в себе новую модель программирования, основанную на ряде новых для JUnit 5 аннотаций из пакета `org.junit.jupiter.api`, с моделью расширения (в виде прикладного интерфейса Extension API с аннотацией `@ExtendWith`, предназначенной для замены аннотаций `@Rule` и `@ClassRule` из класса `Runner` в версии JUnit 4). Кроме того, в подпроекте Jupiter предоставляется реализация интерфейса `TestEngine` для выполнения на платформе JUnit тестов, составленных на основании Jupiter.

И, наконец, в модуле JUnit Vintage предоставляется реализация интерфейса `TestEngine` для выполнения на платформе JUnit тестов, написанных в версиях JUnit 3 и 4. Выясним, однако, как же пользоваться платформой JUnit, выбрав в качестве примера класс `SingerHandler` для тестирования. Начнем с методики отрицательного тестирования и, в частности, попытаемся получить сведения о несуществующем певце:

```
package com.apress.prospring5.ch18;

import com.apress.prospring5.ch18.entities.Singer;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.HttpMethod;
import org.springframework.http.HttpStatus;
import org.springframework.http.client.reactive
        .ReactorClientHttpConnector;
import org.springframework.web.reactive.function
        .BodyInserters;
import org.springframework.web.reactive.function
        .client.ClientRequest;
import org.springframework.web.reactive.function
        .client.ClientResponse;
import org.springframework.web.reactive.function
        .client.ExchangeFunction;
import org.springframework.web.reactive.function
        .client.ExchangeFunctions;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.net.URI;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.List;
import static org.junit.jupiter.api.Assertions.*;
```

```
public class SingerHandlerTest {

  private static Logger logger =
          LoggerFactory.getLogger(SingerHandlerTest.class);

  public static final String HOST = "localhost";

  public static final int PORT = 8080;

  private static ExchangeFunction exchange;

  @BeforeAll
  public static void init() {
    exchange = ExchangeFunctions.create(
                new ReactorClientHttpConnector());
  }

  @Test
  public void noSinger() {
    // получить сведения о певце
    URI uri = URI.create(String.format(
                  "http://%s:%d/singers/99", HOST, PORT));
    logger.debug("GET REQ: "+ uri.toString());
    ClientRequest request = ClientRequest.method(
                            HttpMethod.GET, uri).build();
    Mono<Singer> singerMono = exchange.exchange(request)
        .flatMap(response ->
                 response.bodyToMono(Singer.class));
    Singer singer = singerMono.block();
    assertNull(singer);
  }
  ...
}
```

Функциональный интерфейс `ExchangeFunction` служит в качестве альтернативы упоминавшемуся ранее компоненту Spring Bean типа `WebClient` для обмена запросом типа `ClientRequest` и задержанным ответом типа `ClientResponse` между клиентом и сервером. Используя такую реализацию, можно отправить запрос работающему серверу и затем проанализировать полученные от него ответы. В приведенном выше примере тестового кода необходимо отметить следующее.

- Аннотация `@BeforeAll` действует подобно аннотации `@BeforeClass` из классической платформы JUnit, а следовательно, снабженные ею методы должны выполняться прежде любых методов, снабженных аннотацией `@Test` (или производных от нее аннотаций вроде `@RepeatedTest`) в текущем классе. Отличается эта аннотация тем, что ею можно снабдить нестатические методы, если на уровне класса употребляется аннотация `@TestInstance(Lifecycle.PER_CLASS)`.

- Аннотация `@Test` из пакета `org.junit.jupiter.api` равнозначна аналогичной аннотации из классической платформы JUnit. Отличается эта аннотация лишь тем, что в ней не объявляется никаких атрибутов, поскольку расширения тестов в JUnit Jupiter действуют на основании собственных специализированных аннотаций.
- Оператор `assertNull` из класса `org.junit.jupiter.api.Assertions` действует таким же образом, как и аналогичная аннотация из классической платформы JUnit. Для всех статических методов из класса `Assertions`, в которых удобно применять лямбда-выражения, внедренные в версии Java 8, имеются аналогичные реализации и даже некоторые расширения.

Рассмотрим теперь более сложный пример. На этот раз проверим результат редактирования экземпляра типа `Singer`. Сначала необходимо проверить, действительно ли был извлечен экземпляр, представляющий нужного певца. С этой целью составляется группа утверждений, где проверяются свойства `firstName` и `lastName` извлеченного экземпляра типа `Singer`, как показано ниже.

```
package com.apress.prospring5.ch18;

import com.apress.prospring5.ch18.entities.Singer;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.HttpMethod;
import org.springframework.http.HttpStatus;
import org.springframework.http.client.reactive
          .ReactorClientHttpConnector;
import org.springframework.web.reactive.function
          .BodyInserters;
import org.springframework.web.reactive.function
          .client.ClientRequest;
import org.springframework.web.reactive.function
          .client.ClientResponse;
import org.springframework.web.reactive.function
          .client.ExchangeFunction;
import org.springframework.web.reactive.function
          .client.ExchangeFunctions;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.net.URI;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.List;
import static org.junit.jupiter.api.Assertions.*;
```

```
public class SingerHandlerTest {

  private static Logger logger =
          LoggerFactory.getLogger(SingerHandlerTest.class);

  public static final String HOST = "localhost";

  public static final int PORT = 8080;

  private static ExchangeFunction exchange;

  @BeforeAll
  public static void init() {
    exchange = ExchangeFunctions.create(
               new ReactorClientHttpConnector());
  }

  @Test
  public void editSinger() {
    // получить сведения о певце
    URI uri = URI.create(String.format(
              "http://%s:%d/singers/1", HOST, PORT));
    logger.debug("GET REQ: "+ uri.toString());
    ClientRequest request =
         ClientRequest.method(HttpMethod.GET, uri).build();
    Mono<Singer> singerMono = exchange.exchange(request)
        .flatMap(response ->
                 response.bodyToMono(Singer.class));
    Singer singer = singerMono.block();
    assertNotNull(singer);
    assertAll("singer",
        () -> assertEquals("John", singer.getFirstName()),
        () -> assertEquals("Mayer", singer.getLastName()));

    logger.info("singer:" + singer.toString());

    // отредактировать сведения о певце
    singer.setFirstName("John Clayton");
    uri = URI.create(String.format("http://%s:%d/singers", HOST, PORT));
    logger.debug("UPDATE REQ: "+ uri.toString());
    request = ClientRequest.method(HttpMethod.POST, uri)
           .body(BodyInserters.fromObject(singer)).build();
    Mono<ClientResponse> response =
                          exchange.exchange(request);
    assertEquals(HttpStatus.OK,
                 response.block().statusCode());
    logger.info("Update Response status: "
              + response.block().statusCode());
  }
}
```

В приведенном выше тестовом коде можно сделать еще больше, организовав выполнение утверждений `assertEquals` по условию в утверждении `assetNotNull`. Так, выполнение обоих операторов `assertEquals` в приведенном выше тестовом коде завершается неудачно, если возвращает экземпляр, представляющий не того певца[23]. А ниже приведен код, в котором выполнение утверждений `assertEquals` зависит от результата выполнения утверждения `assetNotNull`.

```
assertAll("singer", () -> {
  assertNotNull(singer);
  assertAll("singer",
        () -> assertEquals("John", singer.getFirstName()),
        () -> assertEquals("Mayer", singer.getLastName()));
});
```

Остановимся на этом, завершив описание тестовых компонентов, характерных для JUnit, и перейдем к другим нововведениям в Spring для тестирования. Помимо интерфейса `WebClient`, предназначенного для модели реактивного программирования, в состав модуля `spring-test` теперь входит интерфейс `WebTestClient` для интеграции поддержки тестирования в модуле Spring WebFlux. Этот новый интерфейс, аналогично классу `MockMvc`, не требует наличия работающего сервера. Но это совсем не означает, что его нельзя использовать на уже работающем сервере. Так, в следующем примере кода интерфейс `WebTestClient` используется в приложении, запускаемом на выполнение из класса `SingerApplication`:

```
package com.apress.prospring5.ch18;

import com.apress.prospring5.ch18.entities.Singer;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import org.springframework.http.MediaType;
import org.springframework.test.web.reactive
          .server.FluxExchangeResult;
import org.springframework.test.web.reactive
          .server.WebTestClient;
import reactor.core.publisher.Mono;

import java.util.Date;
import java.util.GregorianCalendar;
import static org.junit.jupiter.api.Assertions.*;

public class AnotherSingerHandlerTest {

  private static WebTestClient client;

  @BeforeAll
```

[23] Для этого достаточно изменить URI на `http://%s:%d/singers/2`.

```
public static void init() {
  client = WebTestClient
           .bindToServer()
           .baseUrl("http://localhost:8080")
           .build();
}

@Test
public void getSingerNotFound() throws Exception {
  client.get().uri("/singers/99").exchange()
        .expectStatus().isNotFound()
        .expectBody().isEmpty();
}

@Test
public void getSingerFound() throws Exception {
  client.get().uri("/singers/1").exchange()
     .expectStatus().isOk().expectBody(Singer.class)
     .consumeWith(seer -> {
        Singer singer = seer.getResponseBody();
        assertAll("singer", () ->
        {
          assertNotNull(singer);
          assertAll("singer",
            () -> assertEquals("John",
                               singer.getFirstName()),
            () -> assertEquals("Mayer",
                               singer.getLastName()));
        });
  });
}

@Test
public void getAll() throws Exception {
  client.get().uri("/singers").accept(MediaType
                              .TEXT_EVENT_STREAM)
        .exchange()
        .expectStatus().isOk()
        .expectHeader().contentType(
                        MediaType.APPLICATION_JSON)
        .expectBodyList(Singer.class).consumeWith(
                        Assertions::assertNotNull);
}

@Test
public void create() throws Exception {
  Singer singer = new Singer();
  singer.setFirstName("Ed");
  singer.setLastName("Sheeran");
```

```
    singer.setBirthDate(new Date(
            (new GregorianCalendar(1991, 2, 17))
            .getTime().getTime()));
    client.post().uri("/singers").body(Mono.just(singer),
            Singer.class).exchange().expectStatus().isOk();
  }
}
```

Интерфейс `WebTestClient` является главным компонентом для тестирования конечных точек подключения к серверу WebFlux. У него имеется такой же прикладной интерфейс API, как и у интерфейса `WebClient`, а бо́льшую часть своей работы он поручает внутреннему интерфейсу `WebClient`, сосредоточенному в основном на предоставлении тестового контекста. Чтобы провести комплексное тестирование на конкретном действующем сервере, необходимо вызвать метод `bindToServer()`. Приведенный выше пример тестирования довольно прост. Компонент типа `WebTestClient` привязывается к адресу того места, где выполняется конкретное приложение. Следовательно, ему не требуются собственные определяемые операции преобразования или функции маршрутизации. Но иногда это может быть сделано, если все же потребуется, поскольку для этого имеется немало методов конфигурирования. Так, в следующем фрагменте кода применяется функциональный интерфейс `RouterFunction` для определения функции маршрутизации, которая приводится в действие через вызов `bindToRouterFunction(function)`:

```
@Test
public void testCustomRouting() {
    RouterFunction function = RouterFunctions.route(
          RequestPredicates.GET("/test"),
          request -> ServerResponse.ok().build()
    );

  WebTestClient
        .bindToRouterFunction(function)
        .build().get().uri("/test")
        .exchange()
        .expectStatus().isOk()
        .expectBody().isEmpty();
}
```

Здесь стоит также упомянуть, что, начиная с версии 5 каркаса Spring и платформы JUnit, комплексные тесты можно выполнять параллельно. Но для того чтобы продемонстрировать, каким образом можно выполнять тесты параллельно, нам придется оставить пример приложения для певцов и создать простое приложение Spring Boot, в котором объявляется компонент Spring Bean типа `FluxGenerator`, формирующий экземпляры потоков данных типа `Flux`. Так, в приведенном ниже фрагменте кода объявляется простой компонент Spring Bean типа `FluxGenerator`, а также создается и конфигурируется точка входа в данное приложение.

```
// Исходный файл FluxGenerator.java
package com.apress.prospring5.ch18;
import org.springframework.stereotype.Component;
import reactor.core.publisher.Flux;

@Component
public class FluxGenerator {
  public Flux<String> generate(String... args) {
    return Flux.just(args);
  }
}

// Исходный файл Application.java
package com.apress.prospring5.ch18;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.autoconfigure
          .SpringBootApplication;
import org.springframework.boot.builder
          .SpringApplicationBuilder;
import org.springframework.context
          .ConfigurableApplicationContext;

@SpringBootApplication
public class Application {

  private static final Logger logger =
          LoggerFactory.getLogger(Application.class);

  public static void main(String args) throws Exception {
    ConfigurableApplicationContext ctx =
         new SpringApplicationBuilder(Application.class)
        .run(args);
    assert (ctx != null);
    logger.info("Application started...");
    System.in.read();
    ctx.close();
  }
}
```

Создадим далее два тестовых класса, `IntegrationOneTest` и `IntegrationTwoTest`, в каждом из которых объявляются два тестовых метода. В них не будет ничего проверяться, а будет только использоваться компонент Spring Bean типа `FluxGenerator` для получения экземпляра потока данных типа `Flux` и вывода его содержимого. Оба тестовых класса приведены ниже, и, как видите, устанавливаемые в них значения отличаются, чтобы в режиме командной строки можно было отслеживать выполнение тестов и убедиться, что они действительно выполняются параллельно.

```
// Исходный файл IntegrationOneTest.java
package com.apress.prospring5.ch18.test;

import com.apress.prospring5.ch18.FluxGenerator;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.boot.test.context
          .SpringBootTest;
import org.springframework.test.context.junit4
          .SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class IntegrationOneTest {

  private final Logger logger =
       LoggerFactory.getLogger(IntegrationOneTest.class);

  @Autowired FluxGenerator fluxGenerator;

  @Test
  public void test1One() {
    fluxGenerator.generate("1", "2", "3").collectList()
        .block().forEach(s -> executeSlow(2000, s) );
  }

  @Test
  public void test2One() {
    fluxGenerator.generate("11", "22", "33").collectList()
        .block().forEach(s -> executeSlow(1000, s));
  }

  private void executeSlow(int duration, String s) {
    try {
      Thread.sleep(duration);
      logger.info(s);
    } catch (InterruptedException e) {
    }
  }
}

// Исходный файл IntegrationTwoTest.java
package com.apress.prospring5.ch18.test;
... // те же самые операторы импорта, что и выше
```

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class IntegrationTwoTest {
  private final Logger logger =
       LoggerFactory.getLogger(IntegrationTwoTest.class);

  @Autowired FluxGenerator fluxGenerator;

  @Test
  public void test1One() {
    fluxGenerator.generate(2, "a", "b", "c").collectList()
                 .block().forEach(logger::info);
  }

  @Test
  public void test2One() {
  fluxGenerator.generate(3, "aa", "bb", "cc").collectList()
               .block().forEach(logger::info);
  }
}
```

Ниже приведен класс, предназначенный для выполнения обоих тестов. Он содержит два тестовых метода: один — для параллельного, другой — для последовательного выполнения тестов.

```
package com.apress.prospring5.ch18.test;

import org.junit.experimental.ParallelComputer;
import org.junit.jupiter.api.Test;
import org.junit.runner.Computer;
import org.junit.runner.JUnitCore;

public class ParallelTests {
  @Test
  void executeTwoInParallel() {
    final Class<?> classes = {
      IntegrationOneTest.class, IntegrationTwoTest.class
    };
    JUnitCore.runClasses(new ParallelComputer(true, true), classes);
  }

  @Test
  void executeTwoLinear() {
    final Class<?> classes = {
      IntegrationOneTest.class, IntegrationTwoTest.class
    };
    JUnitCore.runClasses(new Computer(), classes);
  }
}
```

Класс `JUnitCore` служит фасадом для выполнения тестов. В нем поддерживаются тексты, составленные в версиях JUnit 3.8 и 4, а также их комбинации. В качестве параметра он принимает экземпляр класса `Computer` или расширение экземпляра класса `ParallelComputer`. В частности, класс `Computer` служит для обычного, последовательного выполнения тестов, а класс `ParallelComputer` — для параллельного их выполнения, для чего конструктор этого класса принимает два аргумента логического типа. Первый аргумент служит для установки режима параллельного выполнения классов, а второй аргумент — такого же режима для методов. В приведенном выше примере тестового кода установлено логическое значение `true` обоих аргументов, чтобы уведомить исполнитель тестов в JUnit о необходимости параллельного выполнения не только классов, но и их методов. Именно поэтому классы `IntegrationOneTest` и `IntegrationTwoTest` несколько отличаются.

Если выполнить приведенный выше тестовый код, то на консоль будет выведен результат, аналогичный следующему:

```
...
17:29:30.460 [pool-2-thread-2] INFO
    c.a.p.c.t.IntegrationTwoTest - aa
17:29:30.460 [pool-2-thread-2] INFO
    c.a.p.c.t.IntegrationTwoTest - bb
17:29:30.460 [pool-2-thread-1] INFO
    c.a.p.c.t.IntegrationTwoTest - a
17:29:30.460 [pool-2-thread-2] INFO
    c.a.p.c.t.IntegrationTwoTest - cc
17:29:30.460 [pool-2-thread-1] INFO
    c.a.p.c.t.IntegrationTwoTest - b
17:29:30.460 [pool-2-thread-1] INFO
    c.a.p.c.t.IntegrationTwoTest - c
17:29:31.463 [pool-1-thread-2] INFO
    c.a.p.c.t.IntegrationOneTest - 11
17:29:32.461 [pool-1-thread-1] INFO
    c.a.p.c.t.IntegrationOneTest - 1
17:29:32.468 [pool-1-thread-2] INFO
    c.a.p.c.t.IntegrationOneTest - 22
17:29:33.472 [pool-1-thread-2] INFO
    c.a.p.c.t.IntegrationOneTest - 33
17:29:34.466 [pool-1-thread-1] INFO
    c.a.p.c.t.IntegrationOneTest - 2
17:29:36.471 [pool-1-thread-1] INFO
    c.a.p.c.t.IntegrationOneTest - 3
...
```

И последний обсуждаемый здесь вопрос касается применения прикладного интерфейса Extension API в JUnit 5. По существу, он служит всего лишь маркерным интерфейсом `Extension` для компонентов, которые могут быть зарегистрированы с помощью аннотации `@ExtendWith` или автоматически через доступный в Java механизм типа `ServiceLoader`. В версии Spring 5 в модуль `spring-test` внедрен класс

SpringExtension, реализующий немало интерфейсных производных аннотации @ExtendWith в Jupiter для интеграции каркаса Spring TestContext Framework в модель программирования Jupiter, внедренную в версии JUnit 5. Чтобы воспользоваться данным расширением, достаточно снабдить тестовый класс, основанный на модели программирования Jupiter в JUnit, аннотацией @ExtendWith(SpringExtension.class), @SpringJUnitConfig или @SpringJUnitWebConfig.

Обратимся к простому примеру, создав сначала класс namedTestConfig, в котором объявляется компонент Spring Bean типа FluxGenerator, а затем тестовый класс для проверки этого компонента. Исходный код обоих классов приведен ниже.

```
// Исходный файл TestConfig.java
package com.apress.prospring5.ch18.test.config;

import com.apress.prospring5.ch18.FluxGenerator;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
          .Configuration;

@Configuration
public class TestConfig {
  @Bean FluxGenerator generator() {
    return new FluxGenerator();
  }
}

// Исходный файл JUnit5IntegrationTest.java
package com.apress.prospring5.ch18.test;

import com.apress.prospring5.ch18.Application;
import com.apress.prospring5.ch18.FluxGenerator;
import com.apress.prospring5.ch18.test.config.TestConfig;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation
          .Autowired;
import org.springframework.test.context
          .ContextConfiguration;
import org.springframework.test.context.junit.jupiter
          .SpringExtension;

import java.util.List;
import static org.junit.jupiter.api.Assertions
                .assertEquals;

@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = TestConfig.class)
public class JUnit5IntegrationTest {
```

```
  @Autowired FluxGenerator fluxGenerator;
  @Test
  public void testGenerator() {
    List<String> list = fluxGenerator.generate("2", "3")
                                     .collectList().block();
    assertEquals(2, list.size());
  }
}
```

Для поддержки JUnit 5 в Spring придется приложить еще немало усилий, поэтому исходный код примеров, представленных в этом разделе, может претерпеть некоторые изменения в последующих выпусках Spring. Это настолько новый предмет, что он еще не отражен в документации на Spring[24], которая, вероятно, когда-нибудь будет соответственно обновлена.

Резюме

В этой главе был сделан общий обзор нескольких проектов, входящих в состав Spring. В ней были вкратце рассмотрены проекты Spring Batch, Spring Integration, Spring XD, Spring WebFlux, спецификация JSR-352 и поддержка платформы JUnit 5 в Spring. Все эти инструментальные средства предоставляют свои особые возможности с целью упростить решение конкретных задач разработки приложений в Spring. Одни проекты являются новыми, тогда как другие доказали свою надежность и прочность, послужив в качестве основания для других каркасов. Мы рекомендуем изучить эти проекты более основательно, поскольку уверены, что в целом они значительно упростят работу над проектами, выполняемыми на Java.

[24] В разделе "Testing" (Тестирование) справочного руководства по Spring уже упоминается о версии JUnit 5; см. по адресу `https://docs.spring.io/spring/docs/current/spring-framework-reference/#testing`.

ПРИЛОЖЕНИЕ

Установка среды разработки

В этом приложении приведен справочный материал, который поможет вам установить свою среду разработки для написания, компиляции и выполнения примеров приложений Spring, обсуждавшихся в данной книге.

Введение в проект `pro-spring-15`

Проект `pro-spring-15` имеет трехуровневую структуру в Gradle. Он является корневым и родительским для проектов `chapter02`–`chapter18`, а каждый из проектов `chapter**` — родительским для вложенных в него модульных проектов[1]. Данный проект структурирован именно таким образом, чтобы помочь вам сориентироваться в исходном коде примеров из данной книги и легко находить соответствие исходного кода с материалом каждой главы. Структура данного проекта приведена на рис. A.1

Описание конфигурации Gradle

В проекте `pro-spring-15` определяется ряд библиотек, доступных для применения в порожденных модулях, а также имеется конфигурация Gradle в файле, обычно называемом `build.gradle`. У всех модулей на втором уровне иерархии (иными словами, проектов `chapter**`) имеется файл конфигурации Gradle под названием [*имя_модуля*]`.gradle` (например, `chapter02.gradle`), что дает возможность

[1] На момент написания этого приложения к книге данный проект еще пребывал в сыром виде. В дальнейшем он был завершен, переименован в `pro-spring-5` и находится в официальном хранилище, доступном по адресу `https://github.com/Apress/pro-spring-5`. Тем не менее все написанное выше остается в силе, за исключением наименования проекта.

быстро найти нужный файл конфигурации с общими настройками для всех проектов из отдельной главы. Кроме того, в файле `pro-spring-15/settings.gradle` имеется приведенный ниже элемент замыкания, где во время сборки проверяется наличие файлов конфигурации для модулей всех глав.

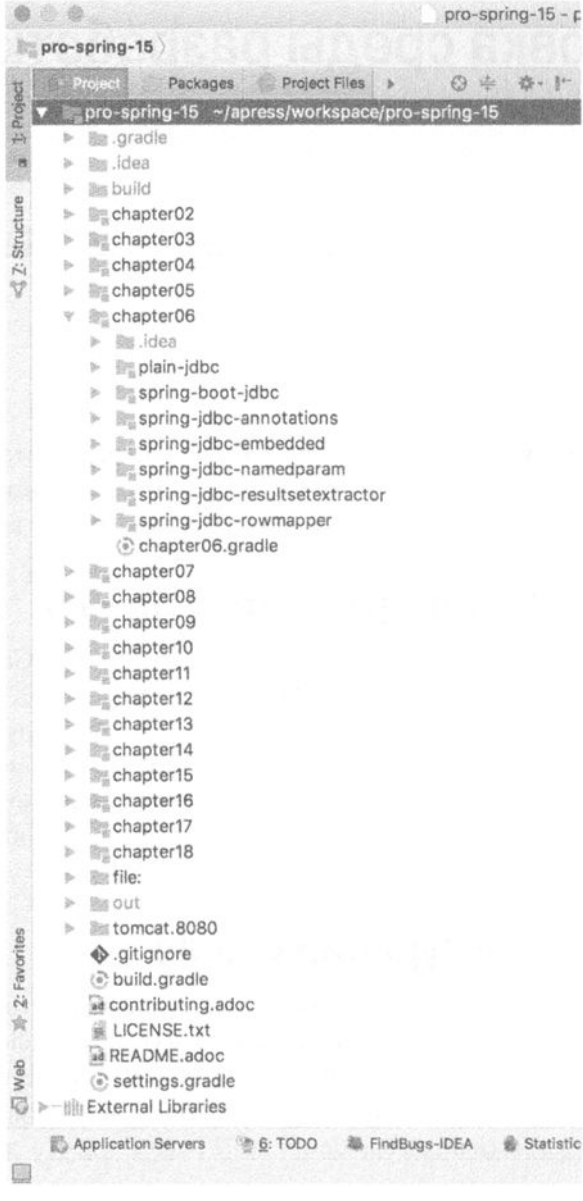


Рис. А.1. Структура проекта **`pro-spring-15`** в IDE Intellij IDEA

```
rootProject.children.each { project ->
    project.buildFileName = "${project.name}.gradle"
    assert project.projectDir.isDirectory()
    assert project.buildFile.exists()
    assert project.buildFile.isFile()
}
```

Если файл сборки Gradle назван иначе, чем модуль, то при выполнении любого задания в Gradle возникнет ошибка. Так, если переименовать файл `chapter02.gradle` на `chapter02_2.gradle`, то возникнет ошибка, аналогичная приведенной ниже.

```
$ gradle clean
FAILURE: Build failed with an exception.

* Where:
Settings file '/workspace/pro-spring-15/settings.gradle'
  line: 328

* What went wrong:
A problem occurred evaluating settings 'pro-spring-15'.[2]
> assert project.buildFile.exists
         |       |         |
         |       |         false
         |       /workspace/pro-spring-15/chapter02
         /chapter02.gradle:chapter02

* Try:
Run with --stacktrace option to get the stack trace.
Run with --info or -debug option to get more log output.

BUILD FAILED in 0s[3]
```

Такое решение было принято на стадии разработки для того, чтобы файл конфигурации было легче обнаружить в IDE. Если же потребуется изменить файл конфигу-

---

[2] ОТКАЗ: сборка завершилась неудачно с исключением.
* Где:
Файл настроек '/workspace/pro-spring-15/settings.gradle'
  строка: 328

* Что пошло не так:
Возникла ошибка при анализе параметров 'pro-spring-15'.

[3] Попробуйте:
Выполнить сборку с параметром -stacktrace, чтобы получить трассировку стека.
Выполнить сборку с параметром --info или -debug, чтобы получить более подробный вывод записей в журнал.
СБОРКА ЗАВЕРШИЛАСЬ НЕУДАЧНО через 0 с

рации для модуля отдельной главы, то его нетрудно будет обнаружить по однозначному имени в такой IDE, как, например, IntelliJ IDEA.

У каждого модуля на третьем уровне иерархии проекта `pro-spring-15` (иными словами, порожденного модуля из проекта `chapter**`) имеется свой файл `build.gradle` для конфигурации Gradle. Такое решение было также принято на стадии разработки в связи с тем, что в Gradle не допускается наличие однозначно именованных файлов конфигурации на данном уровне этой конкретной конфигурации.

В качестве другого подхода к построению многомодульного проекта можно было бы ввести элементы отдельных замыканий в главный файл `build.gradle`, чтобы специально настроить конфигурацию каждого модуля. Но, придерживаясь надлежащих норм практики разработки, мы решили сделать конфигурации модулей как можно более независимыми друг от друга и расположить их в том же месте, где и содержимое каждого модуля.

Файл конфигурации `pro-spring-15/build.gradle` содержит переменную для каждой версии применяемого программного обеспечения. Эти переменные служат для специальной настройки отдельных строк объявления зависимостей, сгруппированных в массивы, именуемые по конкретному назначению или применяемой технологии. В этом файле содержится список открытых хранилищ, из которых загружаются зависимости.

В следующем примере конфигурации демонстрируются версии и массив из библиотек сохраняемости, применяемых в проекте:

```
ext {
    ...
    // библиотеки сохраняемости
    hibernateVersion = '5.2.10.Final'
    hibernateJpaVersion = '1.0.0.Final'
    hibernateValidatorVersion = '5.4.1.Final' //6.0.0.Beta2
    atomikosVersion = '4.0.4'

    hibernate = [
       validator : "org.hibernate:hibernate-validator:
                   $hibernateValidatorVersion",
       jpaModelGen: "org.hibernate:hibernate-jpamodelgen:
                    $hibernateVersion",
       ehcache : "org.hibernate:hibernate-ehcache:
                 $hibernateVersion",
       em : "org.hibernate:hibernate-entitymanager:
            $hibernateVersion",
       envers : "org.hibernate:hibernate-envers:
                $hibernateVersion",
       jpaApi : "org.hibernate.javax.persistence:
                hibernate-jpa-2.1-api:
                $hibernateJpaVersion",
       querydslapt: "com.mysema.querydsl:
                    querydsl-apt:2.7.1",
```

```
        tx : "com.atomikos:transactions-hibernate4:
             $atomikosVersion"
    ]
}
...
subprojects {
  version '5.0-SNAPSHOT'

  repositories {
    mavenLocal()
    mavenCentral()
    maven { url "http://repo.spring.io/release" }
    maven { url "http://repo.spring.io/snapshot" }
    maven { url "https://repo.spring.io/libs-snapshot" }
    maven { url "http://repo.spring.io/milestone" }
    maven { url "https://repo.spring.io/libs-milestone" }
  }
}

tasks.withType(JavaCompile) {
  options.encoding = "UTF-8"
}
```

В проектах на втором уровне иерархии (т.е. в проектах `chapter**`) можно обнаружить файлы конфигурации `chapter**.gradle`, в которых зависимости обозначаются по именам их массивов и связанных с ними именам. Эти файлы содержат также групповое имя проекта, характерное для каждой главы, дополнительные подключаемые модули и задания Gradle. В качестве примера ниже приведено содержимое файла конфигурации `chapter08.gradle`.

```
subprojects {
    group 'com.apress.prospring5.ch08'
    apply plugin: 'java'
    /* Задание на копирование всех зависимостей
       в каталог build/libs */
    task copyDependenciestype: Copy {
      from configurations.compile
      into 'build/libs'
    }

    dependencies {
      if !project.name.contains"boot" {
        compile spring.contextSupport {
          exclude module: 'spring-context'
          exclude module: 'spring-beans'
          exclude module: 'spring-core'
      }
      compile spring.orm, spring.context, misc.slf4jJcl,
        misc.logback, db.h2, misc.lang3, hibernate.em
```

```
    }
    testCompile testing.junit
  }
}
```

Здесь условный оператор `if (!project.name.contains("boot"))` необходим потому, что в проект `chapter08` вложен один или несколько проектов Spring Boot. А поскольку у модуля Spring Boot имеются свои зависимости и их версии, то вряд ли стоит наследовать конфигурацию, определенную в данном файле, ведь это может привести к конфликтам или непредсказуемому поведению.

Для проектов на третьем уровне иерархии можно дополнительно настроить конфигурации, наследуемые из родительского проекта `chapter**`, внедрив их собственные зависимости и объявив их собственные спецификации в файле манифеста, а также подключаемые модули и задания. В качестве примера ниже приведен простой файл конфигурации `chapter12\spring-invoker\build.gradle` (более сложную конфигурацию Gradle можно при желании просмотреть в файле `chapter08\jpa-criteria\build.gradle`).

```
apply plugin: 'war'

dependencies {
    compile project(':chapter12:base-remote')
    compile spring.webmvc, web.servlet
    testCompile spring.test
}

war {
    archiveName = 'remoting.war'
    manifest {
        attributes("Created-By" : "Iuliana Cosmina",
                   "Specification-Title": "Pro Spring 5",
                   "Class-Path" : configurations.compile
                       .collect { it.getName() }.join(' '))
    }
}
```

Построение проекта и устранение неполадок

Скопировав (или загрузив) исходный код по ссылке, указанной во введении к данной книге, необходимо импортировать рассматриваемый здесь проект в избранную IDE. Ниже поясняется, как это делается в IDE IntelliJ IDEA.

1. Выберите команду меню File⇨New⇨Project from Existing Sources (Файл⇨Создать⇨Проект из существующих источников), как показано на рис. А.2.
2. После выбора надлежащей команды появится всплывающее окно, в котором запрашивается местоположение проекта, как показано на рис. А.3.

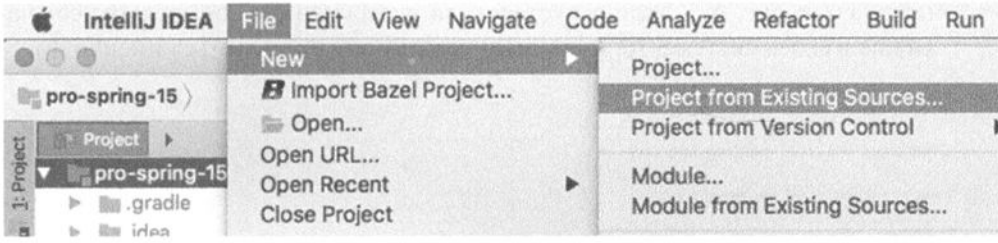


Рис. А.2. Пункты меню выбора импортируемых проектов в IDE IntelliJ IDEA

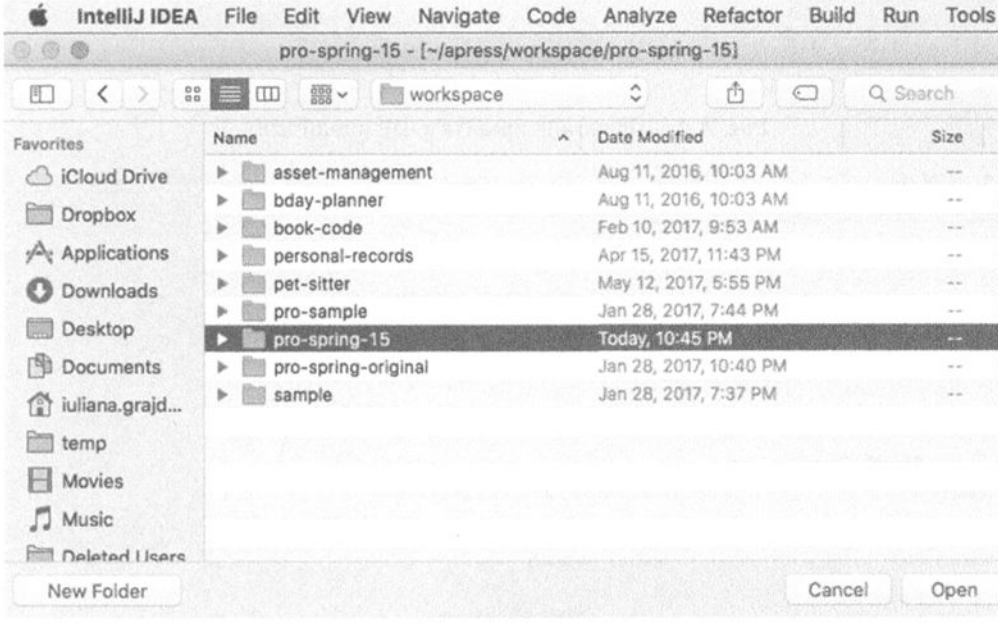


Рис. А.3. Всплывающее окно для выбора корневого каталога проекта в IDE IntelliJ IDEA

3. Выберите каталог `pro-spring-15`. Появится всплывающее окно, в котором запрашивается тип проекта. В IDE Intellij IDEA можно создать собственный тип проекта из выбранных источников и построить его с помощью встроенного в эту IDE построителя проектов на Java, но в данном случае такая возможность не используется, поскольку проект `pro-spring-15` относится к типу проектов Gradle.

4. Выберите сначала кнопку-переключатель Import project from external model (Импортировать проект из внешней модели), а затем пункт меню Gradle, как показано на рис. А.4.

5. Появится последнее всплывающее меню, в котором запрашивается местоположение файла конфигурации `build.gradle`, а также исполняемого файла Gradle. Все параметры настройки в этом окне заполняются автоматически, как

показано на рис. А.5. Если вы установили инструментальное средство сборки Gradle в своей системе, то, вероятнее всего, будете пользоваться именно им.

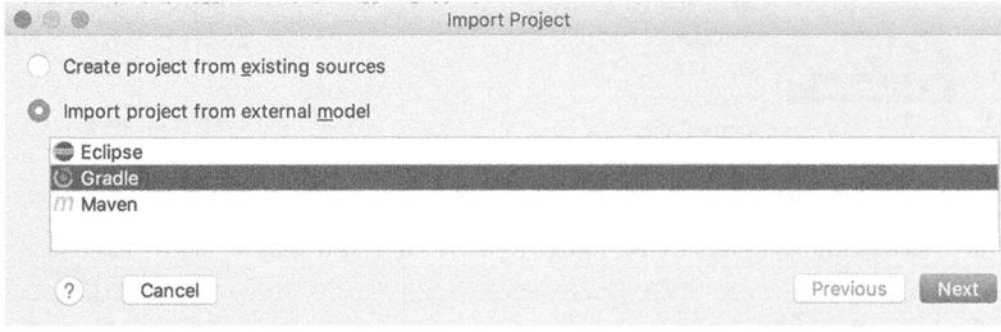


Рис. А.4. Выбор типа проекта в IDE IntelliJ IDEA

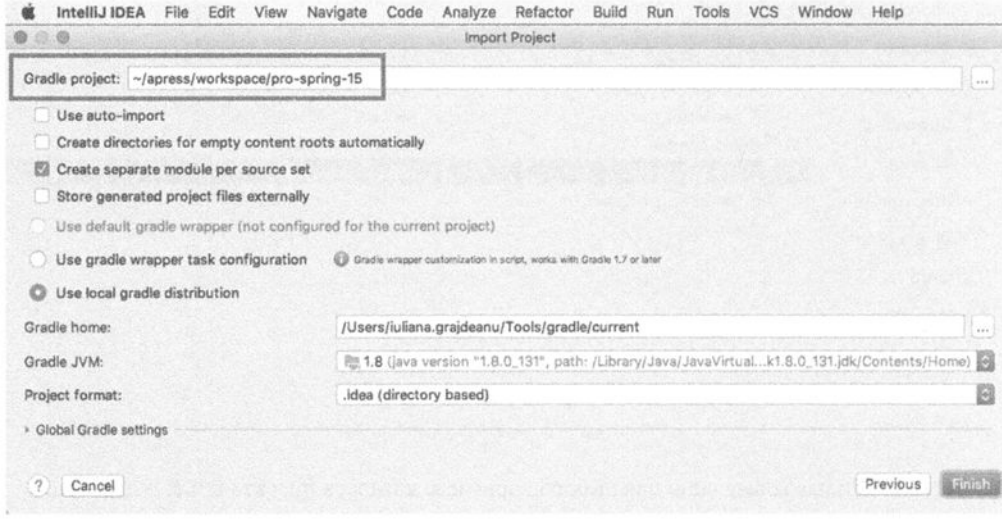


Рис. А.5. Вид последнего всплывающего окна для импорта проекта в IDE IntelliJ IDEA

Прежде чем приступить к работе над проектом, необходимо построить его. Это можно сделать непосредственно в IDE IntelliJ IDEA, щелкнув на кнопке Refresh (Обновить), как обозначено меткой `(1)` на рис. А.6. В итоге IDE IntelliJ IDEA просмотрит конфигурацию проекта и разрешит все его зависимости, включая загрузку недостающих библиотек и внутреннее выполнение упрощенной сборки проекта, которой должно быть достаточно для устранения ошибок компиляции, обусловленных отсутствующими зависимостями.

Задание `compileJava` в Gradle, обозначенное меткой `(3)` на рис. А.6, выполняет полную сборку проекта. То же самое можно сделать из командной строки, выполнив следующую команду сборки в Gradle:

```
.../workspace/pro-spring-15 $ gradle build
```

А в IDE Intellij IDEA для этого достаточно дважды щелкнуть на задании `build`, обозначенном меткой `(2)` на рис. А.6.

Рис. А.6. Построение проекта **`pro-spring-15`** в IDE IntelliJ IDEA

В итоге будут выполнены следующие задания на сборку каждого модуля:

```
:chapter02:hello-world:compileJava
:chapter02:hello-world:processResources
:chapter02:hello-world:classes
:chapter02:hello-world:jar
:chapter02:hello-world:assemble
:chapter02:hello-world:compileTestJava
```

```
:chapter02:hello-world:compileTestResources
...
```

Выше перечислены только задания на сборку модулей из проекта `chapter02`, но задание на сборку в Gradle выполнит все задания, от которых оно зависит. Как видите, здесь не выполняется задание `clean`, поэтому его следует выполнить вручную несколько раз при построении проекта, чтобы воспользоваться самыми последними версиями классов.

Данный проект содержит тесты, которые специально написаны так, чтобы они не проходили, и поэтому выполнение задания `build` завершится неудачно. Вместо этого можно выполнить лишь задания `clean` и `compileJava` или же выполнить задание `build` в Gradle, но с параметром **`-x`**, чтобы пропустить тесты, как показано ниже.

```
.../workspace/pro-spring-15 $ gradle build -x test
```

Развертывание на сервере Apache Tomcat

В состав проекта `pro-spring-15` входит ряд веб-приложений. Это в основном приложения Spring Boot, предназначенные для выполнения на встроенном сервере. Тем не менее применение внешнего сервера вроде контейнера сервлетов Apache Tomcat дает определенные преимущества. В частности, запустить такой сервер в режиме отладки и воспользоваться точками прерывания для отладки веб-приложения не составит большого труда, хотя это далеко не единственное преимущество данного подхода. Во внешнем контейнере сервлетов можно выполнить несколько веб-приложений одновременно, не останавливая сервер. Встроенные серверы, как, например, в модуле Spring Boot, удобны для тестирования, быстрой разработки и учебных целей, но в производственной среде более предпочтительными оказываются серверы приложений.

Далее поясняется, что следует сделать, чтобы воспользоваться внешним сервером. Прежде всего загрузите самую последнюю версию Apache Tomcat с официального веб-сайта[4]. Загрузить можно версию 8 или 9, поскольку обе подходят для выполнения примеров веб-приложений из данной книги. Распакуйте загруженный архивный файл в любом удобном месте своей системы, а затем настройте модуль запуска IDE IntelliJ IDEA на запуск сервера и развертывание выбранного веб-приложения. Это нетрудно сделать, выполнив следующие действия.

1. Выберите команду Edit Configurations (Править конфигурацию) из меню действующей конфигурации, как обозначено меткой `(1)` на рис. А.7. Появится всплывающее окно, в котором перечислен целый ряд модулей запуска. Щелкните на экранной кнопке со знаком + и выберите вариант Tomcat Server. В итоге меню расширится. Выберите из него пункт Local, обозначенный меткой `(2)` на рис. А.7, поскольку вам предстоит воспользоваться сервером, установленным на вашем компьютере.

[4] Официальный веб-сайт Apache Tomcat находится по адресу `http://tomcat.apache.org/`.

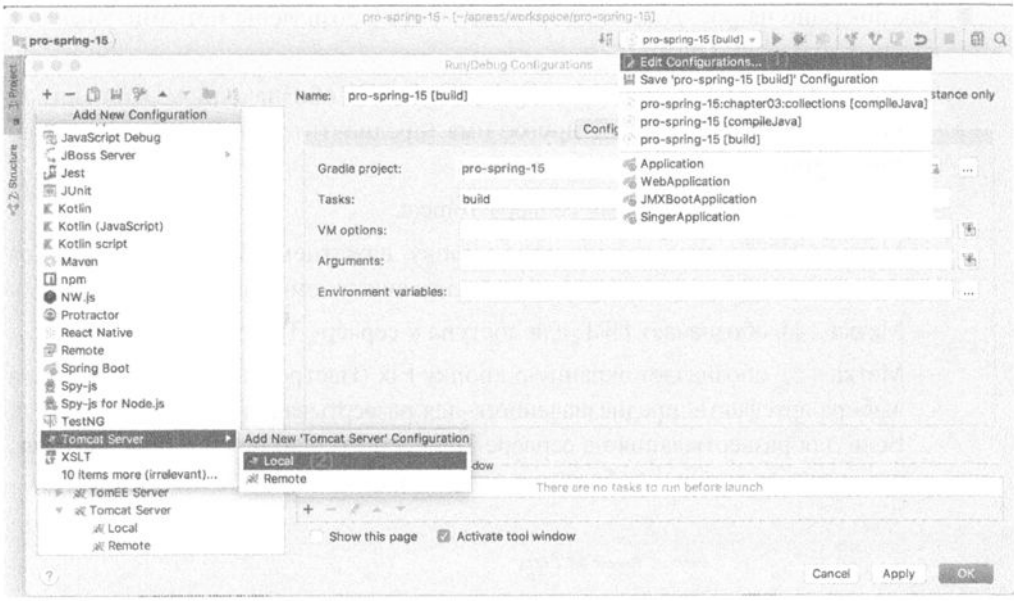


Рис. А.7. Пункты меню для выбора модуля запуска сервера Tomcat в IDE Intellij IDEA

2. Появится всплывающее меню, аналогичное приведенному на рис. А.8. В этом окне запрашивается некоторая информация.

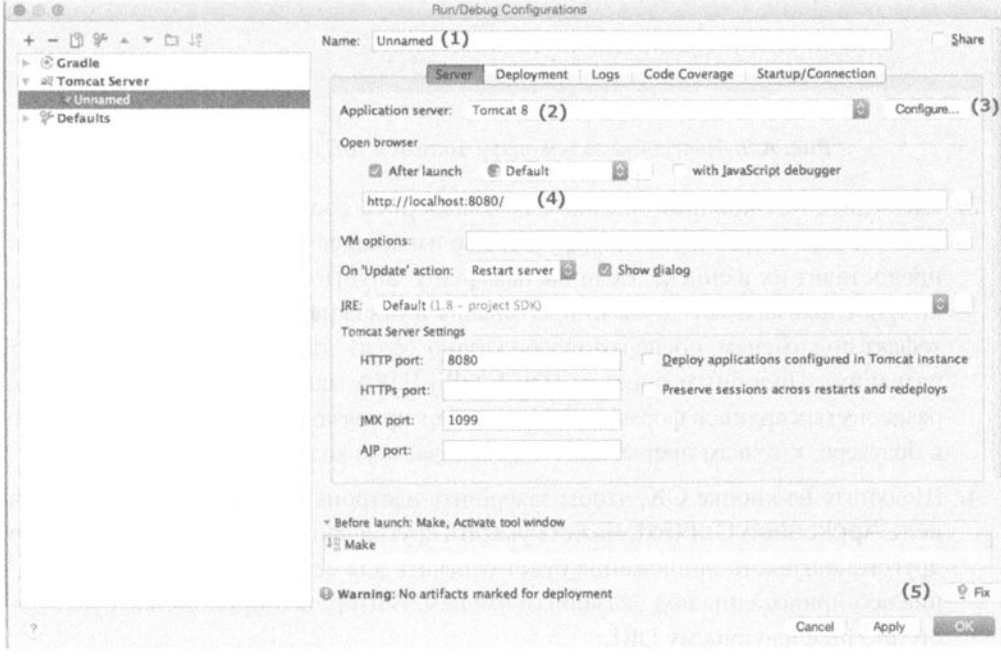


Рис. А.8. Всплывающее окно для настройки модуля запуска Tomcat в IDE Intellij IDEA

- Как показано на рис. А.8, некоторые элементы обозначены метками, значение которых приведено ниже.
  - — Метка (1) обозначает имя модуля запуска. В обозначенном этой меткой поле можно ввести любое другое имя (предпочтительно имя развертываемого приложения).
  - — Метка (2) обозначает имя сервера Tomcat.
  - — Метка (3) обозначает экранную кнопку, нажатием которой открывается всплывающее окно для ввода местоположения экземпляра Tomcat (рис. А.9).
  - — Метка (4) обозначает URL для доступа к серверу Tomcat.
  - — Метка (5) обозначает экранную кнопку Fix (Настроить), нажимаемую для выбора артефакта, предназначенного для развертывания экземпляра Tomcat. Если для развертывания на сервере Tomcat не задано ни одного веб-архива, эта экранная кнопка отобразится с пиктограммой красной лампочки.

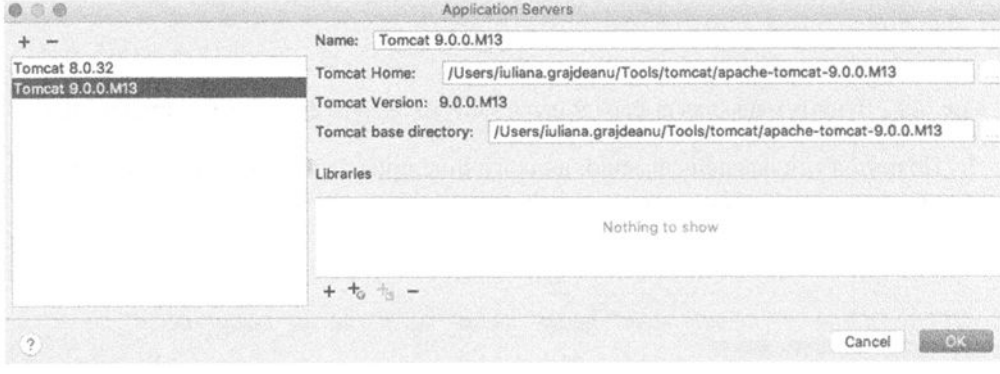


Рис. А.9. Настройка экземпляра Tomcat в IDE Intellij IDEA

3. Щелкните на экранной кнопке Fix и выберите соответствующий артефакт. В итоге IDE IntelliJ IDEA обнаружит все имеющиеся артефакты (рис. А.10) и предоставит их в списке. Если вы намерены запустить сервер в режиме отладки и воспользоваться точками прерывания в отлаживаемом коде, выберите артефакт под именем, после которого указано (exploded), т.е. данный артефакт развернут. Подобным образом IDE Intellij IDEA манипулирует содержимым развернутых архивов формата WAR и может привязать действия, выполняемые в браузере, к точкам прерывания в отлаживаемом коде.
4. Щелкните на кнопке OK, чтобы завершить настройку. Введя новое значение в поле Application Context, можете указать другой контекст приложения. Выбор другого контекста приложения будет означать для сервера Tomcat развертывание веб-приложения под указанным именем. В итоге веб-приложение будет доступно по следующему URL:

```
http://localhost:8080/[имя_контекста_приложения]/
```

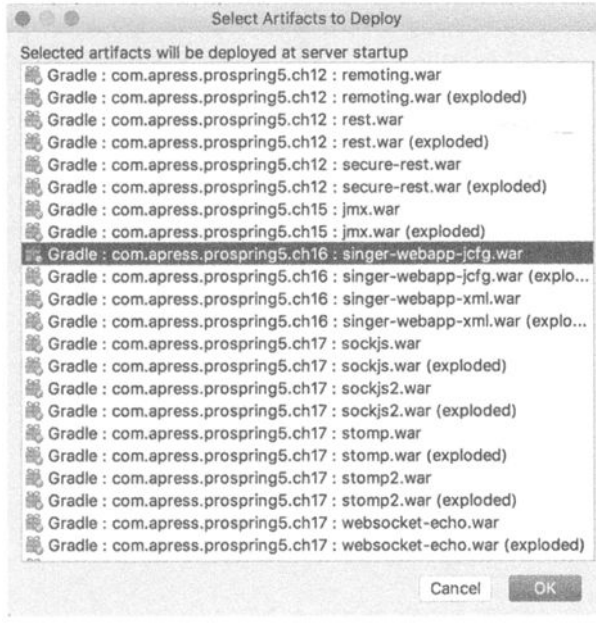
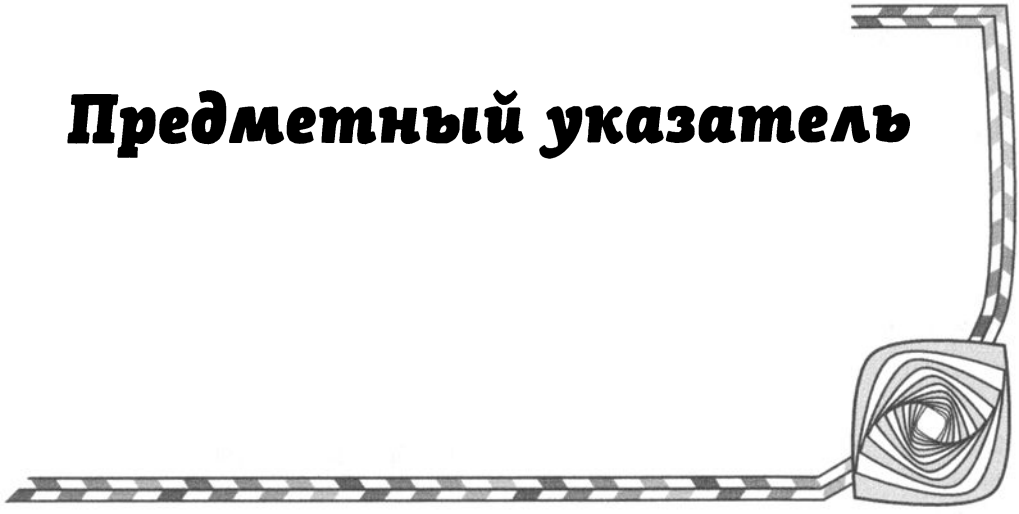


Рис. А.10. Список развертываемых артефактов в IDE Intellij IDEA

Другие серверы приложений могут быть использованы аналогичным образом, при условии, что для них в IDE IntelliJ IDEA предоставляются подключаемые модули. Настройки модулей запуска могут дублироваться, а несколько экземпляров сервера Tomcat — запускаться одновременно, если они действуют через разные порты. Благодаря этому может быть ускорено тестирование и сравнение реализаций. IntelliJ IDEA — очень гибкая и удобная интегрированная среда разработки, именно поэтому она и рекомендуется для проработки примеров и упражнений из данной книги. Проекты Gradle могут быть также импортированы в Eclipse и другие IDE и редакторы исходного кода Java, в которых поддерживается Gradle.



Предметный указатель

A

Б

В

З

И

К

Л

М

О

П

P

C

T

JAVA
ЭФФЕКТИВНОЕ ПРОГРАММИРОВАНИЕ
ТРЕТЬЕ ИЗДАНИЕ

Джошуа Блох

www.dialektika.com

Говоря о третьем издании книги *Java: эффективное программирование*, достаточно упомянуть ее автора, Джошуа Блоха, и это будет наилучшей ее рекомендацией.
Книга представляет собой овеществленный опыт ее автора как программиста на Java. Новые возможности этого языка программирования, появившиеся в версиях, вышедших со времени предыдущего издания книги, по сути, знаменуют появление совершенно новых концепций, так что для их эффективного использования недостаточно просто узнать об их существовании и программировать на современном Java с использованием старых парадигм.
К программированию в полной мере относится фраза Евклида о том, что в геометрии нет царских путей. Но пройти путь изучения и освоения языка программирования вам может помочь проводник, показывающий наиболее интересные места и предупреждающий о ямах и ухабах. Таким проводником может послужить книга Джошуа Блоха. С ней вы не заблудитесь и не забредете в дебри, из которых будете долго и трудно выбираться с помощью отладчика.

ISBN 978-5-6041394-4-8

в продаже

ПРОФЕССИОНАЛАМ ОТ ПРОФЕССИОНАЛОВ

Spring 5 для профессионалов

Эта книга воплощает знания и опыт работы авторов с каркасом Spring Framework и сопутствующими технологиями удаленного взаимодействия, Hibernate, EJB и пр. Она дает возможность читателю не только усвоить основные понятия и принципы работы с Spring Framework, но и научиться рационально пользоваться этим каркасом для построения различных уровней и частей корпоративных приложений на языке Java, включая обработку транзакций, представление веб-содержимого и прочего содержимого, развертывание и многое другое. Полноценные примеры подобных приложений, представленные в этой книге, наглядно демонстрируют особенности совместного применения различных технологий и методик разработки приложений в Spring.

Пятое издание этой книги, давно уже пользующейся успехом у читателей, обновлено по новой версии Spring Framework 5 и является самым исчерпывающим и полным руководством по применению Spring среди всех имеющихся. В нем представлен новый функциональный каркас веб-приложений, микрослужбы, совместимость с версией Java 9 и прочие функциональные возможности Spring. Прочитав эту обстоятельную книгу, вы сможете включить в арсенал своих средств весь потенциал Spring для основательного построения сложных приложений.

Гибкий, легковесный каркас Spring Framework с открытым кодом продолжает оставаться фактически ведущим в области разработки корпоративных приложений на языке Java и самым востребованным среди разработчиков и программирующих на Java. Он превосходно взаимодействует с другими гибкими, легковесными технологиями Java с открытым кодом, включая Hibernate, Groovy, MyBatis и прочие, а также с платформами Java EE и JPA 2.

Эта книга поможет вам:

- Выявить новые функциональные возможности в версии Spring Framework 5
- Научиться пользоваться Spring Framework вместе с Java 9
- Овладеть механизмом доступа к данным и обработки транзакций
- Освоить новый функциональный каркас веб-приложений
- Научиться создавать микрослужбы и другие веб-службы

Категория: программирование
Предмет рассмотрения: каркас Spring и язык Java
Уровень: промежуточный/продвинутый

ISBN 978-5-907114-07-4

9 785907 114074 18076

ДИАЛЕКТИКА

www.williamspublishing.com

Apress®