Pro Spring 5

An In-Depth Guide to the Spring Framework and Its Tools

Fifth Edition

Iuliana Cosmina
Rob Harrop
Chris Schaefer
Clarence Ho

Apress®

Pro Spring 5

An In-Depth Guide to the Spring Framework and Its Tools

Fifth Edition

Iuliana Cosmina
Rob Harrop
Chris Schaefer
Clarence Ho

Apress®

Pro Spring 5: An In-Depth Guide to the Spring Framework and Its Tools

Iuliana Cosmina
Sibiu, Sibiu, Romania

Rob Harrop
Reddish, United Kingdom

Chris Schaefer
Venice, Florida, USA

Clarence Ho
Hong Kong, China

ISBN-13 (pbk): 978-1-4842-2807-4
ISBN-13 (electronic): 978-1-4842-2808-1
DOI 10.1007/978-1-4842-2808-1

Library of Congress Control Number: 2017955423

Copyright © 2017 by Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Cover image by Freepik (www.freepik.com)

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Technical Reviewer: Massimo Nardone
Coordinating Editor: Mark Powers
Copy Editor: Kim Wimpsett

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484228074. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

I dedicate this book to my friends, to my godson ștefan, and to all the musicians who have made working on this book easy with their music.

—Iuliana Cosmina

Contents at a Glance

About the Authors xxiii
About the Technical Reviewer xxv
Acknowledgments xxvii
Introduction xxix

■Chapter 1: Introducing Spring 1
■Chapter 2: Getting Started 19
■Chapter 3: Introducing IoC and DI in Spring 37
■Chapter 4: Spring Configuration in Detail and Spring Boot 125
■Chapter 5: Introducing Spring AOP 211
■Chapter 6: Spring JDBC Support 297
■Chapter 7: Using Hibernate in Spring 355
■Chapter 8: Data Access in Spring with JPA2 393
■Chapter 9: Transaction Management 467
■Chapter 10: Validation with Type Conversion and Formatting 509
■Chapter 11: Task Scheduling 537
■Chapter 12: Using Spring Remoting 557
■Chapter 13: Spring Testing 615
■Chapter 14: Scripting Support in Spring 639
■Chapter 15: Application Monitoring 655

Chapter 16: Web Applications 665

Chapter 17: WebSocket 751

Chapter 18: Spring Projects: Batch, Integration, XD, and More 773

Appendix A: Setting Up Your Development Environment 829

Index 841

Contents

About the Authors xxiii

About the Technical Reviewer xxv

Acknowledgments xxvii

Introduction xxix

■ Chapter 1: Introducing Spring 1

What Is Spring? 1

Evolution of the Spring Framework 2

Inverting Control or Injecting Dependencies? 8

Evolution of Dependency Injection 8

Beyond Dependency Injection 10

The Spring Project 15

Origins of Spring 15

The Spring Community 15

The Spring Tool Suite 16

The Spring Security Project 16

Spring Boot 16

Spring Batch and Integration 17

Many Other Projects 17

Alternatives to Spring 17

JBoss Seam Framework 17

Google Guice 17

PicoContainer 17

JEE 7 Container 18

Summary 18

Chapter 2: Getting Started 19
Obtaining the Spring Framework 20
Getting Started Quickly 20
Checking Spring Out of GitHub 20
Using the Right JDK 21
Understanding Spring Packaging 21
Choosing Modules for Your Application 24
Accessing Spring Modules on the Maven Repository 24
Accessing Spring Modules Using Gradle 26
Using Spring Documentation 26
Putting a Spring into Hello World 27
Building the Sample Hello World Application 27
Refactoring with Spring 31
Summary 35
Chapter 3: Introducing IoC and DI in Spring 37
Inversion of Control and Dependency Injection 37
Types of Inversion of Control 38
Dependency Pull 38
Contextualized Dependency Lookup 39
Constructor Dependency Injection 40
Setter Dependency Injection 41
Injection vs. Lookup 41
Setter Injection vs. Constructor Injection 42
Inversion of Control in Spring 45
Dependency Injection in Spring 46
Beans and BeanFactory 46
BeanFactory Implementations 46
ApplicationContext 48
Configuring ApplicationContext 48
Setting Spring Configuration Options 49
Basic Configuration Overview 49

Declaring Spring Components 50
Using Method Injection 84
Understanding Bean Naming 95
Understanding Bean Instantiation Mode 105
Resolving Dependencies 109
Autowiring Your Bean 112
When to Use Autowiring 121
Setting Bean Inheritance 122
Summary 124
Chapter 4: Spring Configuration in Detail and Spring Boot 125
Spring's Impact on Application Portability 126
Bean Life-Cycle Management 127
Hooking into Bean Creation 128
Executing a Method When a Bean Is Created 128
Implementing the InitializingBean Interface 132
Using the JSR-250 @PostConstruct Annotation 134
Declaring an Initialization Method Using @Bean 137
Understanding Order of Resolution 138
Hooking into Bean Destruction 139
Executing a Method When a Bean Is Destroyed 139
Implementing the DisposableBean Interface 141
Using the JSR-250 @PreDestroy Annotation 143
Declaring a Destroy Method Using @Bean 144
Understanding Order of Resolution 146
Using a Shutdown Hook 146
Making Your Beans "Spring Aware" 146
Using the BeanNameAware Interface 147
Using the ApplicationContextAware Interface 148
Use of FactoryBeans 151
FactoryBean Example: The MessageDigestFactoryBean 151

Accessing a FactoryBean Directly 156
Using the factory-bean and factory-method Attributes 157
JavaBeans PropertyEditors 158
Using the Built-in PropertyEditors 159
Creating a Custom PropertyEditor 164
More Spring ApplicationContext Configuration 167
Internationalization with the MessageSource 168
Using MessageSource in Stand-Alone Applications 171
Application Events 171
Accessing Resources 174
Configuration Using Java Classes 175
ApplicationContext Configuration in Java 175
Spring Mixed Configuration 185
Java or XML Configuration? 187
Profiles 187
An Example of Using the Spring Profiles Feature 187
Spring Profiles Using Java Configuration 190
Considerations for Using Profiles 193
Environment and PropertySource Abstraction 193
Configuration Using JSR-330 Annotations 198
Configuration Using Groovy 201
Spring Boot 204
Summary 210
■ **Chapter 5: Introducing Spring AOP 211**
AOP Concepts 212
Types of AOP 213
Using Static AOP 213
Using Dynamic AOP 213
Choosing an AOP Type 213

AOP in Spring 214
The AOP Alliance 214
Hello World in AOP 214
Spring AOP Architecture 216
Joinpoints in Spring 216
Aspects in Spring 217
About the ProxyFactory Class 217
Creating Advice in Spring 217
Interfaces for Advice 219
Creating Before Advice 219
Securing Method Access by Using Before Advice 220
Creating After-Returning Advice 224
Creating Around Advice 227
Creating Throws Advice 230
Choosing an Advice Type 232
Advisors and Pointcuts in Spring 233
The Pointcut Interface 233
Available Pointcut Implementations 235
Using DefaultPointcutAdvisor 236
Creating a Static Pointcut by Using StaticMethodMatcherPointcut 236
Creating a Dynamic Pointcut by Using DyanmicMethodMatcherPointcut 239
Using Simple Name Matching 242
Creating Pointcuts with Regular Expressions 244
Creating Pointcuts with AspectJ Pointcut Expression 246
Creating Annotation Matching Pointcuts 247
Convenience Advisor Implementations 248
Understanding Proxies 249
Using JDK Dynamic Proxies 250
Using CGLIB Proxies 250
Comparing Proxy Performance 251
Choosing a Proxy to Use 256

Advanced Use of Pointcuts 256
Using Control Flow Pointcuts 256
Using a Composable Pointcut 259
Composition and the Pointcut Interface 262
Pointcut Summary 262
Getting Started with Introductions 263
Introduction Basics 263
Object Modification Detection with Introductions 265
Introduction Summary 270
Framework Services for AOP 271
Configuring AOP Declaratively 271
Using ProxyFactoryBean 271
Using the aop Namespace 277
Using @AspectJ-Style Annotations 284
Considerations for Declarative Spring AOP Configuration 291
AspectJ Integration 291
About AspectJ 291
Using Singleton Aspects 292
Summary 295
Chapter 6: Spring JDBC Support 297
Introducing Lambda Expressions 298
Sample Data Model for Example Code 298
Exploring the JDBC Infrastructure 304
Spring JDBC Infrastructure 309
Overview and Used Packages 309
Database Connections and DataSources 310
Embedded Database Support 315
Using DataSources in DAO Classes 317
Exception Handling 319

The JdbcTemplate Class 321
Initializing JdbcTemplate in a DAO Class 321
Using Named Parameters with NamedParameterJdbcTemplate 323
Retrieving Domain Objects with RowMapper<T> 325
Retrieving Nested Domain Objects with ResultSetExtractor 327
Spring Classes That Model JDBC Operations 330
Querying Data by Using MappingSqlQuery<T> 333
Updating Data by Using SqlUpdate 337
Inserting Data and Retrieving the Generated Key 340
Batching Operations with BatchSqlUpdate 342
Calling Stored Functions by Using SqlFunction 347
Spring Data Project: JDBC Extensions 349
Considerations for Using JDBC 349
Spring Boot JDBC 350
Summary 353
■ **Chapter 7: Using Hibernate in Spring 355**
Sample Data Model for Example Code 356
Configuring Hibernate's SessionFactory 358
ORM Mapping Using Hibernate Annotations 362
Simple Mappings 363
One-to-Many Mappings 367
Many-to-Many Mappings 369
The Hibernate Session Interface 371
Querying Data by Using the Hibernate Query Language 372
Simple Querying with Lazy Fetching 372
Querying with Associations Fetching 375
Inserting Data 378
Updating Data 382
Deleting Data 384

Configuring Hibernate to Generate Tables from Entities 386
Annotating Methods or Fields? 389
Considerations When Using Hibernate 391
Summary.. 392

Chapter 8: Data Access in Spring with JPA2.. 393

Introducing JPA 2.1 394
Sample Data Model for Example Code 394
Configuring JPA's EntityManagerFactory.. 395
Using JPA Annotations for ORM Mapping 398
Performing Database Operations with JPA 400
Using the Java Persistence Query Language to Query Data.. 400
Querying with Untyped Results 410
Querying for a Custom Result Type with a Constructor Expression 412
Inserting Data 415
Updating Data... 417
Deleting data 419
Using a Native Query... 420
Using a Simple Native Query.. 421
Native Querying with SQL ResultSet Mapping 421
Using the JPA 2 Criteria API for a Criteria Query 422
Introducing Spring Data JPA 429
Adding Spring Data JPA Library Dependencies.. 429
Using Spring Data JPA Repository Abstraction for Database Operations 430
Using JpaRepository 436
Spring Data JPA with Custom Queries 437
Keeping Track of Changes on the Entity Class.. 440
Keeping Entity Versions by Using Hibernate Envers... 451
Adding Tables for Entity Versioning 452
Configuring EntityManagerFactory for Entity Versioning... 453

Enabling Entity Versioning and History Retrieval 456
Testing Entity Versioning 457
Spring Boot JPA 459
Considerations When Using JPA 465
Summary 466
Chapter 9: Transaction Management 467
Exploring the Spring Transaction Abstraction Layer 468
Transaction Types 468
Implementations of the PlatformTransactionManager 469
Analyzing Transaction Properties 470
The TransactionDefinition Interface 471
The TransactionStatus Interface 472
Sample Data Model and Infrastructure for Example Code 473
Creating a Simple Spring JPA Project with Dependencies 473
Sample Data Model and Common Classes 475
Using AOP Configuration for Transaction Management 486
Using Programmatic Transactions 488
Considerations on Transaction Management 490
Global Transactions with Spring 490
Infrastructure for Implementing the JTA Sample 491
Implementing Global Transactions with JTA 491
Spring Boot JTA 501
Considerations on Using JTA Transaction Manager 507
Summary 507
Chapter 10: Validation with Type Conversion and Formatting 509
Dependencies 510
Spring Type Conversion System 510
Conversion from a String Using PropertyEditors 511

Introducing Spring Type Conversion 514
Implementing a Custom Converter 514
Configuring ConversionService 515
Converting Between Arbitrary Types 517
Field Formatting in Spring 521
Implementing a Custom Formatter 521
Configuring ConversionServiceFactoryBean 523
Validation in Spring 524
Using the Spring Validator Interface 525
Using JSR-349 Bean Validation 527
Configuring Bean Validation Support in Spring 528
Creating a Custom Validator 531
Using AssertTrue for Custom Validation 534
Considerations for Custom Validation 535
Deciding Which Validation API to Use 535
Summary 535
Chapter 11: Task Scheduling 537
Dependencies for the Task Scheduling Samples 537
Task Scheduling in Spring 538
Introducing the Spring TaskScheduler Abstraction 539
Exploring a Sample Task 540
Using Annotations for Task Scheduling 547
Asynchronous Task Execution in Spring 551
Task Execution in Spring 554
Summary 556
Chapter 12: Using Spring Remoting 557
Using a Data Model for Samples 558
Adding Required Dependencies for the JPA Back End 560

Implementing and Configuring SingerService 562
Implementing SingerService 562
Configuring SingerService 564
Exposing the Service 567
Invoking the Service 568
Using JMS in Spring 570
Implementing a JMS Listener in Spring 573
Sending JMS Messages in Spring 574
Spring Boot Artemis Starter 576
Using RESTful-WS in Spring 579
Introducing RESTful Web Services 579
Adding Required Dependencies for Samples 580
Designing the Singer RESTful Web Service 580
Using Spring MVC to Expose RESTful Web Services 581
Configuring Castor XML 582
Implementing SingerController 584
Configuring a Spring Web Application 586
Using curl to Test RESTful-WS 590
Using RestTemplate to Access RESTful-WS 592
Securing RESTful-WS with Spring Security 597
RESTful-WS with Spring with Spring Boot 602
Using AMQP in Spring 605
Using AMQP with Spring Boot 611
Summary 613
Chapter 13: Spring Testing 615
Introducing Testing Categories 616
Using Spring Test Annotations 617
Implementing Logic Unit Tests 618
Adding Required Dependencies 619
Unit Testing Spring MVC Controllers 620

Implementing an Integration Test 623
Adding Required Dependencies 623
Configuring the Profile for Service-Layer Testing 623
Java Configuration Version 625
Implementing the Infrastructure Classes 627
Unit Testing the Service Layer 630
Dropping DbUnit 634
Implementing a Front-End Unit Test 637
Introducing Selenium 638
Summary 638
■ **Chapter 14: Scripting Support in Spring 639**
Working with Scripting Support in Java 640
Introducing Groovy 641
Dynamic Typing 642
Simplified Syntax 643
Closure 643
Using Groovy with Spring 644
Developing the Singer Domain 645
Implementing the Rule Engine 646
Implementing the Rule Factory as a Spring Refreshable Bean 648
Testing the Age Category Rule 650
Inlining Dynamic Language Code 652
Summary 654
■ **Chapter 15: Application Monitoring 655**
JMX Support in Spring 655
Exporting a Spring Bean to JMX 656
Using Java VisualVM for JMX Monitoring 657
Monitoring Hibernate Statistics 659
JMX with Spring Boot 661
Summary 664

■ Chapter 16: Web Applications 665

Implementing the Service Layer for Samples 666

Using a Data Model for the Samples 666

Implementing the DAO Layer 670

Implementing the Service Layer 670

Configuring SingerService.. 672

Introducing MVC and Spring MVC 673

Introducing MVC 674

Introducing Spring MVC.. 675

Spring MVC WebApplicationContext Hierarchy 675

Spring MVC Request Life Cycle 676

Spring MVC Configuration... 678

Creating the First View in Spring MVC.. 681

Configuring DispatcherServlet... 683

Implementing SingerController.. 684

Implementing the Singer List View.. 685

Testing the Singer List View 686

Understanding the Spring MVC Project Structure 686

Enabling Internationalization (i18n)... 687

Configuring i18n in the DispatcherServlet Configuration 688

Modifying the Singer List View for i18n Support 690

Using Theming and Templating 691

Theming Support... 691

View Templating with Apache Tiles 693

Designing the Template Layout 693

Implementing Page Layout Components... 694

Configuring Tiles in Spring MVC 698

Implementing the Views for Singer Information.. 699

Mapping URLs to the Views 699

Implementing the Show Singer View.. 700

Implementing the Edit Singer View 703
Implementing the Add Singer View 708
Enabling JSR-349 (Bean Validation) 709
Using jQuery and jQuery UI 711
Introducing jQuery and jQuery UI 711
Enabling jQuery and jQuery UI in a View 712
Rich-Text Editing with CKEditor 714
Using jqGrid for a Data Grid with Pagination 715
Enabling jqGrid in the Singer List View 715
Enabling Pagination on the Server Side 717
Handling File Upload 721
Configuring File Upload Support 721
Modifying Views for File Upload Support 723
Modifying Controllers for File Upload Support 724
Securing a Web Application with Spring Security 726
Configuring Spring Security 726
Adding Login Functions to the Application 729
Using Annotations to Secure Controller Methods 731
Creating Spring Web Applications with Spring Boot 732
Setting Up the DAO Layer 733
Setting Up the Service Layer 735
Setting Up the Web Layer 735
Setting Up Spring Security 737
Creating Thymeleaf Views 738
Using Thymeleaf Extensions 743
Using Webjars 747
Summary 749
Chapter 17: WebSocket 751
Introducing WebSocket 751
Using WebSocket with Spring 752

Using the WebSocket API 752
Using SockJS 760
Sending Messages with STOMP 765
Summary 772
■ **Chapter 18: Spring Projects: Batch, Integration, XD, and More 773**
Spring Batch 774
JSR-352 783
Spring Boot Batch 786
Spring Integration 790
Spring XD 796
Spring Framework's Five Most Notable Features 798
The Functional Web Framework 799
Java 9 Interoperabiliy 811
JDK Modularity 811
Reactive Programming with Java 9 and Spring WebFlux 814
Spring Support for JUnit 5 Jupiter 817
Summary 827
■ **Appendix A: Setting Up Your Development Environment 829**
Introducing Project pro-spring-15 829
Understanding the Gradle Configuration 831
Building and Troubleshooting 834
Deploy on Apache Tomcat 837
Index 841

About the Authors

Iuliana Cosmina is a Spring-certified Web Application Developer and is also a Spring-certified Spring Professional, as defined by Pivotal, the makers of Spring Framework, Boot, and other tools. She has authored books with Apress on core Spring certification and Spring-certified web development. She is a software architect at Bearing Point Software and is an active coder and software contributor on GitHub, Stack Overflow, and more.

Rob Harrop is a software consultant specializing in delivering high-performance, highly scalable enterprise applications. He is an experienced architect with a particular flair for understanding and solving complex design issues. With a thorough knowledge of both Java and .NET, Harrop has successfully deployed projects across both platforms. He also has extensive experience across a variety of sectors, retail and government in particular. Harrop is the author of five books, including *the book you are currently reading, not at its fifth edition*, a widely acclaimed, comprehensive resource on the Spring Framework.

Chris Schaefer is a principle software developer for Spring projects at Pivotal, the makers of Spring Framework, Boot, and other Spring tools.

Clarence Ho is the senior Java architect of a Hong Kong–based software consultancy firm, SkywideSoft Technology Limited. Having been worked in the IT field for more than 20 years, Clarence has been the team leader of many in-house application development projects, as well as providing consultancy services on enterprise solutions to clients.

About the Technical Reviewer

Massimo Nardone has more than 23 years of experience in security, web/mobile development, cloud computing, and IT architecture. His true IT passions are security and Android.

He currently works as the chief information security officer (CISO) for Cargotec Oyj and is a member of the ISACA Finland Chapter board. Over his long career, he has held these positions: project manager, software engineer, research engineer, chief security architect, information security manager, PCI/SCADA auditor, and senior lead IT security/cloud/SCADA architect. In addition, he has been a visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University).

Massimo has a master of science degree in computing science from the University of Salerno in Italy, and he holds four international patents (PKI, SIP, SAML, and proxy areas). Besides working on this book, Massimo has reviewed more than 40 IT books for different publishing companies and is the coauthor of *Pro Android Games* (Apress, 2015).

Acknowledgments

It is a huge honor for me to be the main author of the fifth edition of this book. Would you believe I got this assignment by mistake? I thought I was getting an assignment as a technical reviewer for this book. Only when I received the files did I realize that I was going to be one of the authors of the fifth edition of one of the best Spring books on the market.

Apress has published many of the books I have read and used to improve myself professionally during my studies and even after that. This is my third book with Apress, and it is great to contribute to the education of the next generation of developers.

I am grateful to all my friends who had the patience to listen to me complain about losing sleep, having too much work to do, and encountering writer's block. Thank you all for being supportive and making sure I still had some fun while writing this book.

Also, I would like to give a big thanks to all my favorite singers who made my work easier with their wonderful music, especially John Mayer; I was so determined to finish this book on time just so I could go to the United States to one of his concerts. That is why I changed the topic of the examples in this book to be about singers and their music; it's a tribute to their art and talent.

—Iuliana Cosmina

Introduction

Covering version 5 of the Spring Framework, this book is the most comprehensive Spring reference and practical guide available for harnessing the power of this leading enterprise Java application development framework.

This edition covers core Spring and its integration with other leading Java technologies, such as Hibernate, JPA 2, Tiles, Thymeleaf, and WebSocket. The focus of the book is on using Java configuration classes, lambda expressions, Spring Boot, and reactive programming. We share our insights and real-world experiences with enterprise application development, including remoting, transactions, the web and presentation tiers, and much more.

With Pro Spring 5, you'll learn how to do the following:

- Use inversion of control (IoC) and dependency injection (DI)
- Discover what's new in Spring Framework 5
- Build Spring-based web applications using Spring MVC and WebSocket
- Build Spring web reactive applications with Spring WebFlux
- Test Spring applications using Junit 5
- Utilize the new Java 8 lambda syntax
- Use Spring Boot to an advanced level to get any kind of Spring application up and running in no time
- Use Java 9 features in Spring applications

Because the Java 9 release date kept being postponed, Spring 5 was released based on Java 8. Thus, interoperability with Java 9 is covered in this book based on an early-access build.

There is a multimodule project associated with this book, configured using Gradle 4. The project is available on the Apress official repository: `https://github.com/Apress/pro-spring-5`. The project can be built right after cloning according to the instructions in its `README.adoc` file as long as Gradle is installed locally. If you do not have Gradle installed, you can rely on InteliJ IDEA to download it and use it to build your project by using the Gradle Wrapper. (`https://docs.gradle.org/current/userguide/gradle_wrapper.html`). There is a small appendix at the end of the book describing the project structure, configuration and additional details related to development tools that can be used to develop and run the code samples of the book, which are available on GitHub.

As the book was being written, new release candidate versions of Spring 5 were released, a new version of Intellij IDEA was released, and new versions of Gradle and other technologies used in the book were updated. We upgraded to the new versions to provide the most recent information and keep this book synchronized with the official documentation. Several reviewers have checked the book for technical accuracy, but if you notice any inconsistencies, please send an email to `editorial@apress.com` and errata will be created.

You can access the example source code for this book via the Download Source Code button at `www.apress.com/9781484228074`. It will be maintained, synchronized with new versions of the technologies, and enriched based on the recommendations of the developers using it to learn Spring.

We truly hope you will enjoy using this book to learn Spring as much as we enjoyed writing it.

CHAPTER 1

Introducing Spring

When we think of the community of Java developers, we are reminded of the hordes of gold rush prospectors of the late 1840s, frantically panning the rivers of North America, looking for fragments of gold. As Java developers, our rivers run rife with open source projects, but, like the prospectors, finding a useful project can be time-consuming and arduous.

A common gripe with many open source Java projects is that they are conceived merely out of the need to fill the gap in the implementation of the latest buzzword-heavy technology or pattern. Having said that, many high-quality, usable projects meet and address a real need for real applications, and in the course of this book, you will meet a subset of these projects. You will get to know one in particular rather well—Spring. The first version of Spring was released in October 2002 and consisted of a small core with an inversion of control (IoC) container that was easy to configure and use. Over the years Spring has become the main replacement of Java Enterprise Edition (JEE) servers and has grown into a full-blown technology made up of many distinct projects, each with its own purpose, so whether you want to build microservices, applications, or classical ERPs, Spring has a project for that.

Throughout this book, you will see many applications of different open source technologies, all of which are unified under the Spring Framework. When working with Spring, an application developer can use a large variety of open source tools, without needing to write reams of code and without coupling an application too closely to any particular tool.

In this chapter, as its title indicates, we introduce you to the Spring Framework, rather than presenting any solid examples or explanations. If you are already familiar with Spring, you might want to skip this chapter and proceed straight to Chapter 2.

What Is Spring?

Perhaps one the hardest parts of explaining Spring is classifying exactly what it is. Typically, Spring is described as a lightweight framework for building Java applications, but that statement brings up two interesting points.

First, you can use Spring to build any application in Java (for example, stand-alone, web, or JEE applications), unlike many other frameworks (such as Apache Struts, which is limited to web applications).

Second, the *lightweight* part of the description doesn't really refer to the number of classes or the size of the distribution but rather defines the principle of the Spring philosophy as a whole—that is, minimal impact. Spring is lightweight in the sense that you have to make few, if any, changes to your application code to gain the benefits of Spring Core, and should you choose to stop using Spring at any point, you will find doing so quite simple.

Notice that we qualified that last statement to refer to Spring Core only—many of the extra Spring components, such as data access, require a much closer coupling to the Spring Framework. However, the benefits of this coupling are quite clear, and throughout the book we present techniques for minimizing the impact this has on your application.

© Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho 2017
I. Cosmina et al., *Pro Spring 5*, https://doi.org/10.1007/978-1-4842-2808-1_1

Evolution of the Spring Framework

The Spring Framework originated from the book *Expert One-on-One: J2EE Design and Development* by Rod Johnson (Wrox, 2002). Over the last decade, the Spring Framework has grown dramatically in core functionality, associated projects, and community support. With the new major release of the Spring Framework, it's worthwhile to take a quick look back at important features that have come along with each milestone release of Spring, leading up to Spring Framework 5.0.

- *Spring 0.9*: This is the first public release of the framework, based on the book *Expert One-on-One: J2EE Design and Development*, that offered a bean configuration foundation, AOP support, a JDBC abstraction framework, abstract transaction support, and so on. This version does not have official reference documentation, but you can find the existing sources and documentation on SourceForge.[1]
- *Spring 1.x*: This is the first version released with official reference documentation. It is composed of the seven modules shown in Figure 1-1.

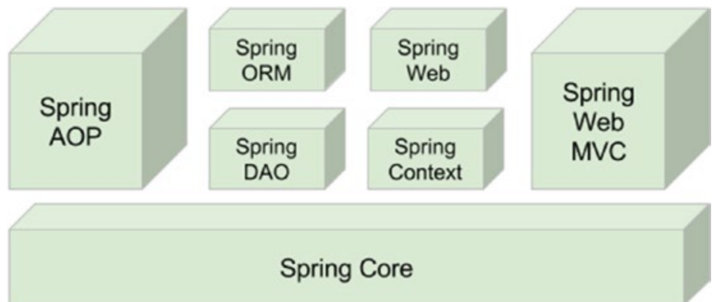

Figure 1-1. *Overview of the Spring Framework, version 1.x*

  – *Spring Core*: Bean container and supporting utilities
  – *Spring Context*: `ApplicationContext`, UI, validation, JNDI, Enterprise JavaBeans (EJB), remoting, and mail support
  – *Spring DAO*: Transaction infrastructure, Java Database Connectivity (JDBC), and data access object (DAO) support
  – *Spring ORM*: Hibernate, iBATIS, and Java Data Objects (JDO) support
  – *Spring AOP*: An AOP Alliance–compliant aspect-oriented programming (AOP) implementation
  – *Spring Web*: Basic integration features such as multipart functionality, context initialization through servlet listeners, and a web-oriented application context
  – *Spring Web MVC*: Web-based Model-View-Controller (MVC) framework

- *Spring 2.x*: This is composed of the six modules shown in Figure 1-2. The Spring Context module is now included in Spring Core, and all Spring web components have been represented here by a single item.

[1]You can download older versions of Spring including 0.9 from the SourceForge site: `https://sourceforge.net/projects/springframework/files/springframework/`.

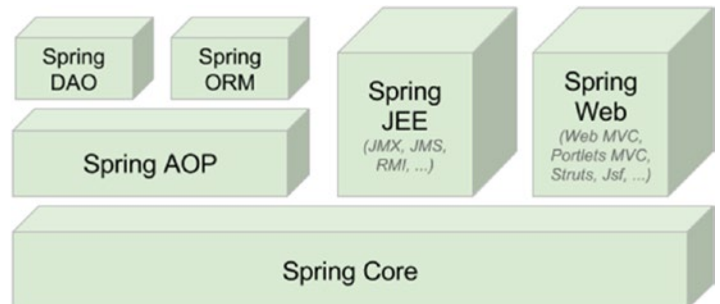

Figure 1-2. *Overview of the Spring Framework, version 2.x*

- Easier XML configuration through the use of the new XML Schema–based configuration rather than the DTD format. Notable areas of improvement include bean definitions, AOP, and declarative transactions.
- New bean scopes for web and portal usage (request, session, and global sessions).
- `@AspectJ` annotation support for AOP development.
- Java Persistence API (JPA) abstraction layer.
- Full support for asynchronous JMS message-driven POJOs (for plain old Java objects).
- JDBC simplifications including `SimpleJdbcTemplate` when using Java 5+.
- JDBC named parameter support (`NamedParameterJdbcTemplate`).
- Form tag library for Spring MVC.
- Introduction of the Portlet MVC framework.
- Dynamic language support. Beans can be written in JRuby, Groovy, and BeanShell.
- Notification support and controllable MBean registration in JMX.
- `TaskExecutor` abstraction introduced for scheduling tasks.
- Java 5 annotation support, specifically for `@Transactional`, `@Required`, and `@AspectJ`.

• *Spring 2.5.x*: This version has the following features:

- A new configuration annotation called `@Autowired` and support for JSR-250 annotations (`@Resource`, `@PostConstruct`, `@PreDestroy`)
- New stereotype annotations: `@Component`, `@Repository`, `@Service`, `@Controller`
- Automatic classpath-scanning support to detect and wire classes annotated with stereotype annotations
- AOP updates, including a new bean pointcut element and AspectJ load-time weaving
- Full WebSphere transaction management support

- In addition to the Spring MVC @Controller annotation, @RequestMapping, @RequestParam, and @ModelAttribute annotations added to support request handling through annotation configuration
- Tiles 2 support
- JSF 1.2 support
- JAX-WS 2.0/2.1 support
- Introduction of the Spring TestContext Framework, providing annotation-driven and integration testing support, agnostic of the testing framework being used
- Ability to deploy a Spring application context as a JCA adapter

- *Spring 3.0.x*: This is the first version of Spring based on Java 5 and is designed to take full advantage of Java 5 features such as generics, varargs, and other language improvements. This version introduces the Java-based @Configuration model. The framework modules have been revised to be managed separately with one source tree per module JAR. This is abstractly depicted in Figure 1-3.

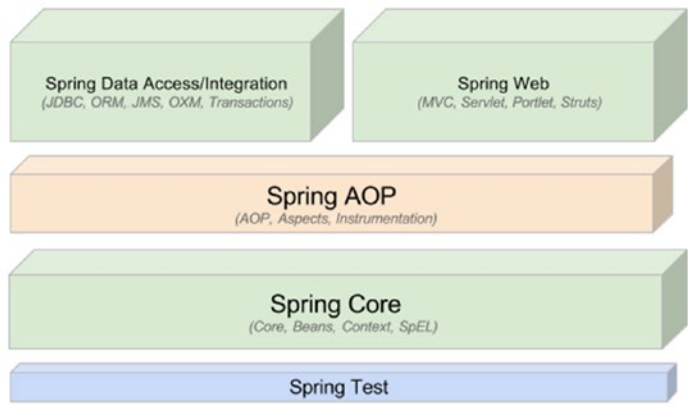

Figure 1-3. *Overview of the Spring Framework, version 3.0.x*

- Support for Java 5 features such as generics, varargs, and other improvements
- First-class support for Callables, Futures, ExecutorService adapters, and ThreadFactory integration
- Framework modules now managed separately with one source tree per module JAR
- Introduction of the Spring Expression Language (SpEL)
- Integration of core Java Config features and annotations
- General-purpose type conversion system and field-formatting system
- Comprehensive REST support
- New MVC XML namespace and additional annotations such as @CookieValue and @RequestHeaders for Spring MVC

- Validation enhancements and JSR-303 (Bean Validation) support
- Early support for Java EE 6, including `@Async`/`@Asynchronous` annotation, JSR-303, JSF 2.0, JPA 2.0, and so on
- Support for embedded databases such as HSQL, H2, and Derby

- *Spring 3.1.x*: This version has the following features:
  - New cache abstraction
  - Bean definition profiles can be defined in XML as well as support for the `@Profile` annotation
  - Environment abstraction for unified property management
  - Annotation equivalents for common Spring XML namespace elements such as `@ComponentScan`, `@EnableTransactionManagement`, `@EnableCaching`, `@EnableWebMvc`, `@EnableScheduling`, `@EnableAsync`, `@EnableAspectJAutoProxy`, `@EnableLoadTimeWeaving`, and `@EnableSpringConfigured`
  - Support for Hibernate 4
  - Spring TestContext Framework support for `@Configuration` classes and bean definition profiles
  - c: namespace for simplified constructor injection
  - Support for Servlet 3 code-based configuration of the Servlet container
  - Ability to bootstrap the JPA `EntityManagerFactory` without `persistence.xml`
  - `Flash` and `RedirectAttributes` added to Spring MVC, allowing attributes to survive a redirect by using the HTTP session
  - URI template variable enhancements
  - Ability to annotate Spring MVC `@RequestBody` controller method arguments with `@Valid`
  - Ability to annotate Spring MVC controller method arguments with the `@RequestPart` annotation
- *Spring 3.2.x*: This version has the following features:
  - Support for Servlet 3–based asynchronous request processing.
  - New Spring MVC test framework.
  - New Spring MVC annotations `@ControllerAdvice` and `@MatrixVariable`.
  - Support for generic types in `RestTemplate` and in `@RequestBody` arguments.
  - Jackson JSON 2 support.
  - Support for Tiles 3.
  - `@RequestBody` or an `@RequestPart` argument can now be followed by an `Errors` argument, making it possible to handle validation errors.
  - Ability to exclude URL patterns by using the MVC namespace and Java Config configuration options.
  - Support for `@DateTimeFormat` without Joda Time.

  - Global date and time formatting.
  - Concurrency refinements across the framework, minimizing locks and generally improving concurrent creation of scoped/prototyped beans
  - New Gradle-based build system.
  - Migration to GitHub (`https://github.com/SpringSource/spring-framework`).
  - Refined Java SE 7/OpenJDK 7 support in the framework and third-party dependencies. CGLIB and ASM are now included as part of Spring. AspectJ 1.7 is supported in addition to 1.6.
- *Spring 4.0.x*: This is a major Spring release and the first to fully support Java 8. Older versions of Java can be used, but the minimum requirement has been raised to Java SE6. Deprecated classes and methods were removed, and the module organization is pretty much the same, as depicted in Figure 1-4.

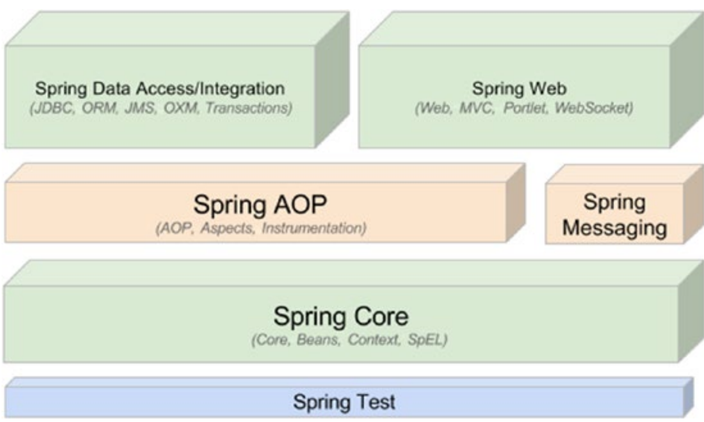

Figure 1-4. *Overview of the Spring Framework, version 4.0.x*

  - Improved getting-started experience via a series of Getting Started guides on the new `www.spring.io/guides` web site
  - Removal of deprecated packages and methods from the prior Spring 3 version
  - Java 8 support, raising the minimum Java version to 6 update 18
  - Java EE 6 and above is now considered the baseline for Spring Framework 4.0
  - Groovy bean definition DSL, allowing bean definitions to be configured via Groovy syntax
  - Core container, testing, and general web improvements
  - WebSocket, SockJS, and STOMP messaging
- *Spring 4.2.x*: This version has the following features:
  - Core improvements (eg., introduction of `@AliasFor` and modification of existing annotation to make use of it)
  - Full support for Hibernate ORM 5.0

- JMS and web improvements
- WebSocket messaging improvements
- Testing improvements, most notably the introduction of `@Commit` to replace `@Rollback(false)` and the introduction of the `AopTestUtils` utility class that allows access to the underlying object hidden behind a Spring proxy

- *Spring 4.3.x*: This version has the following features:
  - The programming model has been refined.
  - Considerable improvements in the core container (inclusions of ASM 5.1, CGLIB 3.2.4, and Objenesis 2.4 in `spring-core.jar`) and MVC.
  - Composed annotations were added.
  - Spring TestContext Framework requires JUnit 4.12 or higher.
  - Support for new libraries, including Hibernate ORM 5.2, Hibernate Validator 5.3, Tomcat 8.5 and 9.0, Jackson 2.8, and so on

- *Spring 5.0.x*: This is a major release. The entire framework codebase is based on Java 8 and is fully compatible with Java 9 as of July 2016.[2]
  - Support was dropped for Portlet, Velocity, JasperReports, XMLBeans, JDO, Guava, Tiles2, and Hibernate3.
  - XML configuration namespaces are now streamed to unversioned schemas; version-specific declarations are still supported but validated against the latest XSD schema.
  - Overall improvements were introduced by harnessing the full power of Java 8 features.
  - The `Resource` abstraction provides `isFile` indicator for defensive `getFile` access.
  - Full Servlet 3.1 signature support in Spring-provided `Filter` implementations.
  - Support for Protobuf 3.0.
  - Support for JMS 2.0+, JPA 2.1+.
  - Introduction of Spring Web Flow, a project that is an alternative to `Spring MVC` built on a reactive foundation, which means that it is fully asynchronous and non-blocking, intended for use in an event-loop execution model vs. traditional large thread pool with a thread-per-request execution model (built upon Project Reactor[3]).
  - The web and core modules were adapted to the reactive programming model.[4]
  - There are a lot of improvements in the Spring test module. JUnit 5 is now supported, and new annotations were introduced to support the Jupiter programming and extension model such as `@SpringJUnitConfig`, `@SpringJUnitWebConfig`, `@EnabledIf`, `@DisabledIf`.
  - Support for parallel test execution in the Spring TestContext Framework.

---

[2]Keep in mind that Java 9 will be released officially to the public in September 2017, according to the Oracle schedule available at `http://openjdk.java.net/projects/jdk9/`.
[3]Project Reactor implements the Reactive Streams API specification; see `https://projectreactor.io/`.
[4]Reactive programming is a style of micro-architecture involving intelligent routing and consumption of events. This should lead to nonblocking applications that are asynchronous and event-driven and require a small number of threads to scale vertically within the JVM, rather than horizontally through clustering.

Inverting Control or Injecting Dependencies?

The core of the Spring Framework is based on the principle of inversion of control. IoC is a technique that externalizes the creation and management of component dependencies. Consider an example in which class Foo depends on an instance of class Bar to perform some kind of processing. Traditionally, Foo creates an instance of Bar by using the new operator or obtains one from some kind of factory class. Using the IoC approach, an instance of Bar (or a subclass) is provided to Foo at runtime by some external process. This behavior, the injection of dependencies at runtime, led to IoC being renamed by Martin Fowler as the much more descriptive *dependency injection* (DI). Chapter 3 discusses the precise nature of the dependencies managed by DI.

As you will see in Chapter 3, using the term *dependency injection* when referring to inversion of control is always correct. In the context of Spring, you can use the terms interchangeably, without any loss of meaning.

Spring's DI implementation is based on two core Java concepts: JavaBeans and interfaces. When you use Spring as the DI provider, you gain the flexibility of defining dependency configuration within your applications in different ways (for example, XML files, Java configuration classes, annotations within your code, or the new Groovy bean definition method). JavaBeans (POJOs) provide a standard mechanism for creating Java resources that are configurable in a number of ways, such as constructors and setter methods. In Chapter 3, you will see how Spring uses the JavaBean specification to form the core of its DI configuration model; in fact, any Spring-managed resource is referred to as a *bean*. If you are unfamiliar with JavaBeans, refer to the quick primer we present at the beginning of Chapter 3.

Interfaces and DI are technologies that are mutually beneficial. Clearly designing and coding an application to interfaces makes for a flexible application, but the complexity of wiring together an application designed using interfaces is quite high and places an additional coding burden on developers. By using DI, you reduce the amount of code you need to use an interface-based design in your application to almost zero. Likewise, by using interfaces, you can get the most out of DI because your beans can utilize any interface implementation to satisfy their dependency. The use of interfaces also allows Spring to utilize JDK dynamic proxies (the Proxy pattern) to provide powerful concepts such as AOP for crosscutting concerns.

In the context of DI, Spring acts more like a container than a framework—providing instances of your application classes with all the dependencies they need—but it does so in a much less intrusive way. Using Spring for DI relies on nothing more than following the JavaBeans naming conventions within your classes—there are no special classes from which to inherit or proprietary naming schemes to follow. If anything, the only change you make in an application that uses DI is to expose more properties on your JavaBeans, thus allowing more dependencies to be injected at runtime.

Evolution of Dependency Injection

In the past few years, thanks to the popularity gained by Spring and other DI frameworks, DI has gained wide acceptance among Java developer communities. At the same time, developers were convinced that using DI was a best practice in application development, and the benefits of using DI were also well understood.

The popularity of DI was acknowledged when the Java Community Process (JCP) adopted JSR-330 (Dependency Injection for Java) in 2009. JSR-330 had become a formal Java specification request, and as you might expect, one of the specification leads was Rod Johnson—the founder of the Spring Framework. In JEE 6, JSR-330 became one of the included specifications of the entire technology stack. In the meantime, the EJB architecture (starting from version 3.0) was also revamped dramatically; it adopted the DI model in order to ease the development of various Enterprise JavaBeans apps.

Although we leave the full discussion of DI until Chapter 3, it is worth taking a look at the benefits of using DI rather than a more traditional approach.

- *Reduced glue code*: One of the biggest plus points of DI is its ability to dramatically reduce the amount of code you have to write to glue the components of your application together. Often this code is trivial, so creating a dependency involves simply creating a new instance of an object. However, the glue code can get quite complex when you need to look up dependencies in a JNDI repository or when the dependencies cannot be invoked directly, as is the case with remote resources. In these cases, DI can really simplify the glue code by providing automatic JNDI lookup and automatic proxying of remote resources.
- *Simplified application configuration*: By adopting DI, you can greatly simplify the process of configuring an application. You can use a variety of options to configure those classes that were injectable to other classes. You can use the same technique to express the dependency requirements to the "injector" for injecting the appropriate bean instance or property. In addition, DI makes it much simpler to swap one implementation of a dependency for another. Consider the case where you have a DAO component that performs data operations against a PostgreSQL database and you want to upgrade to Oracle. Using DI, you can simply reconfigure the appropriate dependency on your business objects to use the Oracle implementation rather than the PostgreSQL one.
- *Ability to manage common dependencies in a single repository*: Using a traditional approach to dependency management of common services—for example, data source connection, transaction, and remote services—you create instances (or lookup from some factory classes) of your dependencies where they are needed (within the dependent class). This will cause the dependencies to spread across the classes in your application, and changing them can prove problematic. When you use DI, all the information about those common dependencies is contained in a single repository, making the management of dependencies much simpler and less error prone.
- *Improved testability*: When you design your classes for DI, you make it possible to replace dependencies easily. This is especially handy when you are testing your application. Consider a business object that performs some complex processing; for part of this, it uses a DAO to access data stored in a relational database. For your test, you are not interested in testing the DAO; you simply want to test the business object with various sets of data. In a traditional approach, whereby the business object is responsible for obtaining an instance of the DAO itself, you have a hard time testing this, because you are unable to easily replace the DAO implementation with a mock implementation that returns your test data sets. Instead, you need to make sure your test database contains the correct data and uses the full DAO implementation for your tests. Using DI, you can create a mock implementation of the DAO object that returns the test data sets, and then you can pass this to your business object for testing. This mechanism can be extended for testing any tier of your application and is especially useful for testing web components where you can create mock implementations of `HttpServletRequest` and `HttpServletResponse`.
- *Fostering of good application design*: Designing for DI means, in general, designing against interfaces. A typical injection-oriented application is designed so that all major components are defined as interfaces, and then concrete implementations of these interfaces are created and hooked together using the DI container. This kind of design was possible in Java before the advent of DI and DI-based containers such as Spring, but by using Spring, you get a whole host of DI features for free, and you are able to concentrate on building your application logic, not a framework to support it.

As you can see from this list, DI provides a lot of benefits for your application, but it is not without its drawbacks. In particular, DI can make it difficult for someone not intimately familiar with the code to see just what implementation of a particular dependency is being hooked into which objects. Typically, this is a problem only when developers are inexperienced with DI; after becoming more experienced and following good DI coding practice (for example, putting all injectable classes within each application layer into the same package), developers will be able to discover the whole picture easily. For the most part, the massive benefits far outweigh this small drawback, but you should consider this when planning your application.

Beyond Dependency Injection

Spring Core alone, with its advanced DI capabilities, is a worthy tool, but where Spring really excels is in its myriad of additional features, all elegantly designed and built using the principles of DI. Spring provides features for all layers of an application, from helper application programming interfaces (APIs) for data access right through to advanced MVC capabilities. What is great about these features in Spring is that, although Spring often provides its own approach, you can easily integrate them with other tools in Spring, making these tools first-class members of the Spring family.

Support for Java 9

Java 8 brings many exciting features that Spring Framework 5 supports, most notably lambda expressions and method references with Spring's callback interfaces. The Spring 5 release plan was aligned with the initial release plan for JDK 9, and although the release deadline for JDK 9 has been postponed, Spring 5 was released according to plan. It is estimated that Spring 5.1 will fully embrace JDK 9. Spring 5 will make use of JDK 9 features such as compact strings, the ALPN stack, and the new HTTP Client implementation. While Spring Framework 4.0 supports Java 8, compatibility is still maintained back to JDK 6 update 18. The use of a more recent version of Java such as 7 or 8 is recommended for new development projects. Spring 5.0 requires Java 8+ because the Spring development team has applied the Java 8 language level to the entire framework codebase, but Spring 5 was built on JDK 9 too, even from the start, to provide comprehensive support for advertised features of JDK 9.

Aspect-Oriented Programming with Spring

AOP provides the ability to implement crosscutting logic—that is, logic that applies to many parts of your application—in a single place and to have that logic applied across your application automatically. Spring's approach to AOP is to create dynamic proxies to the target objects and weave the objects with the configured advice to execute the crosscutting logic. By the nature of JDK dynamic proxies, target objects must implement an interface declaring the method in which the AOP advice will be applied. Another popular AOP library is the Eclipse AspectJ project,[5] which provides more-powerful features including object construction, class loading, and stronger crosscutting capability. However, the good news for Spring and AOP developers is that starting from version 2.0, Spring offers much tighter integration with AspectJ. The following are some highlights:

- Support for AspectJ-style pointcut expressions
- Support for `@AspectJ` annotation style, while still using Spring AOP for weaving
- Support for aspects implemented in AspectJ for DI
- Support for load-time weaving within the Spring `ApplicationContext`

---

[5] www.eclipse.org/aspectj

> ⚠ Starting with Spring Framework version 3.2, `@AspectJ` annotation support can be enabled with Java configuration.

Both kinds of AOP have their place, and in most cases, Spring AOP is sufficient for addressing an application's crosscutting requirements. However, for more complicated requirements, AspectJ can be used, and both Spring AOP and AspectJ can be mixed in the same Spring-powered application.

AOP has many applications. A typical one given in many of the traditional AOP examples involves performing some kind of logging, but AOP has found uses well beyond the trivial logging applications. Indeed, within the Spring Framework itself, AOP is used for many purposes, particularly in transaction management. Spring AOP is covered in full detail in Chapter 5, where we show you typical uses of AOP within the Spring Framework and your own applications, as well as AOP performance and areas where traditional technologies are better suited than AOP.

Spring Expression Language

Expression Language (EL) is a technology to allow an application to manipulate Java objects at runtime. However, the problem with EL is that different technologies provide their own EL implementations and syntaxes. For example, Java Server Pages (JSP) and Java Server Faces (JSF) both have their own EL, and their syntaxes are different. To solve the problem, the Unified Expression Language (EL) was created.

Because the Spring Framework is evolving so quickly, there is a need for a standard expression language that can be shared among all the Spring Framework modules as well as other Spring projects. Consequently, starting in version 3.0, Spring introduced the *Spring Expression Language*. SpEL provides powerful features for evaluating expressions and for accessing Java objects and Spring beans at runtime. The result can be used in the application or injected into other JavaBeans.

Validation in Spring

Validation is another large topic in any kind of application. The ideal scenario is that the validation rules of the attributes within JavaBeans containing business data can be applied in a consistent way, regardless of whether the data manipulation request is initiated from the front end, a batch job, or remotely (for example, via web services, RESTful web services, or remote procedure calls [RPCs]).

To address these concerns, Spring provides a built-in validation API by way of the `Validator` interface. This interface provides a simple yet concise mechanism allowing you to encapsulate your validation logic into a class responsible for validating the target object. In addition to the target object, the validate method takes an `Errors` object, which is used to collect any validation errors that may occur.

Spring also provides a handy utility class, `ValidationUtils`, which provides convenience methods for invoking other validators, checking for common problems such as empty strings, and reporting errors back to the provided `Errors` object.

Driven by need, the JCP also developed JSR-303 (Bean Validation), which provides a standard way of defining bean validation rules. For example, when applying the `@NotNull` annotation to a bean's property, it mandates that the attribute shouldn't contain a `null` value before being able to persist into the database.

Starting in version 3.0, Spring provides out-of-the-box support for JSR-303. To use the API, just declare a `LocalValidatorFactoryBean` and inject the `Validator` interface into any Spring-managed beans. Spring will resolve the underlying implementation for you. By default, Spring will first look for the Hibernate Validator (`hibernate.org/subprojects/validator`), which is a popular JSR-303 implementation. Many front-end technologies (for example, JSF 2 and Google Web Toolkit), including Spring MVC, also support the application of JSR-303 validation in the user interface. The time when developers needed to program the same validation logic in both the user interface and the back-end layer is gone. Chapter 10 discusses the details.

Starting with Spring Framework version 4.0, the 1.1 version of JSR-349 (Bean Validation) is supported.

Accessing Data in Spring

Data access and persistence seem to be the most discussed topics in the Java world. Spring provides excellent integration with a choice selection of these data access tools. In addition, Spring makes plain-vanilla JDBC a viable option for many projects, with its simplified wrapper APIs around the standard API. Spring's data access module provides out-of-the-box support for JDBC, Hibernate, JDO, and the JPA.

Starting with Spring Framework version 4.0, iBATIS support has been removed. The MyBatis-Spring project provides integration with Spring, and you can find more information at `http://mybatis.github.io/spring/`.

However, in the past few years, because of the explosive growth of the Internet and cloud computing, besides relational databases, a lot of other "special-purpose" databases were developed. Examples include databases based on key-value pairs to handle extremely large volumes of data (generally referred to as NoSQL), graph databases, and document databases. To help developers support those databases and to not complicate the Spring data access module, a separate project called Spring Data[6] was created. The project was further split into different categories to support more specific database access requirements.

Spring's support of nonrelational databases is not covered in this book. If you are interested in this topic, the Spring Data project mentioned earlier is a good place to look. The project page details the nonrelational databases that it supports, with links to those databases' home pages.

The JDBC support in Spring makes building an application on top of JDBC a realistic undertaking, even for more complex applications. The support for Hibernate, JDO, and JPA makes already simple APIs even simpler, thus easing the burden on developers. When using the Spring APIs to access data via any tool, you are able to take advantage of Spring's excellent transaction support. You'll find a full discussion of this in Chapter 9.

One of the nicest features in Spring is the ability to easily mix and match data access technologies within an application. For instance, you may be running an application with Oracle, using Hibernate for much of your data access logic. However, if you want to take advantage of some Oracle-specific features, it is simple to implement that part of your data access tier by using Spring's JDBC APIs.

Object/XML Mapping in Spring

Most applications need to integrate or provide services to other applications. One common requirement is to exchange data with other systems, either on a regular basis or in real time. In terms of data format, XML is the most commonly used. As a result, you will often need to transform a JavaBean into XML format, and vice versa. Spring supports many common Java-to-XML mapping frameworks and, as usual, eliminates the need for directly coupling to any specific implementation. Spring provides common interfaces for marshalling (transforming JavaBeans into XML) and unmarshalling (transforming XML into Java objects) for DI into any Spring beans. Common libraries such as Java Architecture for XML Binding (JAXB) , Castor, XStream, JiBX, and XMLBeans are supported. In Chapter 12, when we discuss remotely accessing a Spring application for business data in XML format, you will see how to use Spring's Object/XML Mapping (OXM) support in your application.

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

Managing Transactions

Spring provides an excellent abstraction layer for transaction management, allowing for programmatic and declarative transaction control. By using the Spring abstraction layer for transactions, you can make it simple to change the underlying transaction protocol and resource managers. You can start with simple, local, resource-specific transactions and move to global, multiresource transactions without having to change your code. Transactions are covered in full detail in Chapter 9.

Simplifying and Integrating with JEE

With the growing acceptance of DI frameworks such as Spring, a lot of developers have chosen to construct applications by using DI frameworks in favor of JEE's EJB approach. As a result, the JCP communities also realize the complexity of EJB. Starting in version 3.0 of the EJB specification, the API was simplified, so it now embraces many of the concepts from DI.

However, for those applications that were built on EJB or need to deploy the Spring-based applications in a JEE container and utilize the application server's enterprise services (for example, Java Transaction API's Transaction Manager, data source connection pooling, and JMS connection factories), Spring also provides simplified support for those technologies. For EJB, Spring provides a simple declaration to perform the JNDI lookup and inject into Spring beans. On the reverse side, Spring also provides simple annotation for injecting Spring beans into EJBs.

For any resources stored in a JNDI-accessible location, Spring allows you to do away with the complex lookup code and have JNDI-managed resources injected as dependencies into other objects at runtime. As a side effect of this, your application becomes decoupled from JNDI, allowing you more scope for code reuse in the future.

MVC in the Web Tier

Although Spring can be used in almost any setting, from the desktop to the Web, it provides a rich array of classes to support the creation of web-based applications. Using Spring, you have maximum flexibility when you are choosing how to implement your web front end. For developing web applications, the MVC pattern is the most popular practice. In recent versions, Spring has gradually evolved from a simple web framework into a full-blown MVC implementation. First, view support in Spring MVC is extensive. In addition to standard support for JSP and Java Standard Tag Library (JSTL), which is greatly bolstered by the Spring tag libraries, you can take advantage of fully integrated support for Apache Velocity, FreeMarker, Apache Tiles, Thymeleaf, and XSLT. In addition, you will find a set of base view classes that make it simple to add Microsoft Excel, PDF, and JasperReports output to your applications.

In many cases, you will find Spring MVC sufficient for your web application development needs. However, Spring can also integrate with other popular web frameworks such as Struts, JSF, Atmosphere, Google Web Toolkit (GWT), and so on.

In the past few years, the technology of web frameworks has evolved quickly. Users have required more responsive and interactive experiences, and that has resulted in the rise of Ajax as a widely adopted technology in developing rich Internet applications (RIAs). On the other hand, users also want to be able to access their applications from any device, including smartphones and tablets. This creates a need for web frameworks that support HTML5, JavaScript, and CSS3. In Chapter 16, we discuss developing web applications by using Spring MVC.

WebSocket Support

Starting with Spring Framework 4.0, support for JSR-356 (Java API for WebSocket) is available. WebSocket defines an API for creating a persistent connection between a client and server, typically implemented in web browsers and servers. WebSocket-style development opens the door for efficient, full-duplex communication enabling real-time message exchanges for highly responsive applications. Use of WebSocket support is detailed further in Chapter 17.

Remoting Support

Accessing or exposing remote components in Java has never been the simplest of jobs. Using Spring, you can take advantage of extensive support for a wide range of remoting techniques to quickly expose and access remote services. Spring provides support for a variety of remote access mechanisms, including Java Remote Method Invocation (RMI), JAX-WS, Caucho Hessian and Burlap, JMS, Advanced Message Queuing Protocol (AMQP), and REST. In addition to these remoting protocols, Spring provides its own HTTP-based invoker that is based on standard Java serialization. By applying Spring's dynamic proxying capabilities, you can have a proxy to a remote resource injected as a dependency into one of your classes, thus removing the need to couple your application to a specific remoting implementation and also reducing the amount of code you need to write for your application. We discuss remote support in Spring in Chapter 12.

Mail Support

Sending e-mail is a typical requirement for many kinds of applications and is given first-class treatment within the Spring Framework. Spring provides a simplified API for sending e-mail messages that fits nicely with the Spring DI capabilities. Spring supports the standard JavaMail API. Spring provides the ability to create a prototype message in the DI container and uses this as the base for all messages sent from your application. This allows for easy customization of mail parameters such as the subject and sender address. In addition, for customizing the message body, Spring integrates with template engines, such as Apache Velocity; this allows the mail content to be externalized from the Java code.

Job Scheduling Support

Most nontrivial applications require some kind of scheduling capability. Whether this is for sending updates to customers or performing housekeeping tasks, the ability to schedule code to run at a predefined time is an invaluable tool for developers. Spring provides scheduling support that can fulfill most common scenarios. A task can be scheduled either for a fixed interval or by using a Unix cron expression. On the other hand, for task execution and scheduling, Spring integrates with other scheduling libraries as well. For example, in the application server environment, Spring can delegate execution to the CommonJ library that is used by many application servers. For job scheduling, Spring also supports libraries including the JDK Timer API and Quartz, a commonly used open source scheduling library. The scheduling support in Spring is covered in full in Chapter 11.

Dynamic Scripting Support

Starting with JDK 6, Java introduced dynamic language support, in which you can execute scripts written in other languages in a JVM environment. Examples include Groovy, JRuby, and JavaScript. Spring also supports the execution of dynamic scripts in a Spring-powered application, or you can define a Spring bean that was written in a dynamic scripting language and injected into other JavaBeans. Spring-supported dynamic scripting languages include Groovy, JRuby, and BeanShell. In Chapter 14, we discuss the support of dynamic scripting in Spring in detail.

Simplified Exception Handling

One area where Spring really helps reduce the amount of repetitive, boilerplate code you need to write is in exception handling. The core of the Spring philosophy in this respect is that checked exceptions are overused in Java and that a framework should not force you to catch any exception from which you are unlikely to be able to recover—a point of view that we agree with wholeheartedly. In reality, many frameworks are designed to reduce the impact of having to write code to handle checked exceptions. However, many of these frameworks take the approach of sticking with checked exceptions but artificially reducing the granularity of the exception class hierarchy. One thing you will notice with Spring is that because of the convenience afforded to the developer from using unchecked exceptions, the exception hierarchy is remarkably granular. Throughout the book, you will see examples in which the Spring exception-handling mechanisms can reduce the amount of code you have to write and, at the same time, improve your ability to identify, classify, and diagnose errors within your application.

The Spring Project

One of the most endearing things about the Spring project is the level of activity present in the community and the amount of cross-pollination between Spring and other projects such as CGLIB, Apache Geronimo, and AspectJ. One of the most touted benefits of open source is that if the project folded tomorrow, you would be left with the code; but let's face it—you do not want to be left with a codebase the size of Spring to support and improve. For this reason, it is comforting to know how well established and active the Spring community is.

Origins of Spring

As noted earlier in this chapter, the origins of Spring can be traced back to *Expert One-to-One: J2EE Design and Development*. In this book, Rod Johnson presented his own framework, called the Interface 21 Framework, which he developed to use in his own applications. Released into the open source world, this framework formed the foundation of the Spring Framework as we know it today. Spring proceeded quickly through the early beta and release candidate stages, and the first official 1.0 release was made available in March 2004. Since then, Spring has undergone dramatic growth, and at the time of this writing, the latest major version of Spring Framework is 5.0.

The Spring Community

The Spring community is one of the best in any open source project we have encountered. The mailing lists and forums are always active, and progress on new features is usually rapid. The development team is truly dedicated to making Spring the most successful of all the Java application frameworks, and this shows in the quality of the code that is reproduced. As we mentioned already, Spring also benefits from excellent relationships with other open source projects, a fact that is extremely beneficial when you consider the large amount of dependency the full Spring distribution has. From a user's perspective, perhaps one of the best features of Spring is the excellent documentation and test suite that accompany the distribution. Documentation is provided for almost all the features of Spring, making it easy for new users to pick up the framework. The test suite Spring provides is impressively comprehensive—the development team writes tests for everything. If they discover a bug, they fix that bug by first writing a test that highlights the bug and then getting the test to pass. Fixing bugs and creating new features is not limited just to the development team! You can contribute code through pull requests against any portfolio of Spring projects through the official GitHub repositories (`http://github.com/spring-projects`). Additionally, issues can be created and

tracked by way of the official Spring JIRA (https://jira.spring.io/secure/Dashboard.jspa). What does all this mean to you? Well, put simply, it means you can be confident in the quality of the Spring Framework and confident that, for the foreseeable future, the Spring development team will continue to improve what is already an excellent framework.

The Spring Tool Suite

To ease the development of Spring-based applications in Eclipse, Spring created the Spring IDE project. Soon after that, SpringSource, the company behind Spring founded by Rod Johnson, created an integrated tool called the Spring Tool Suite (STS) , which can be downloaded from https://spring.io/tools. Although it used to be a paid-for product, the tool is now freely available. The tool integrates the Eclipse IDE, Spring IDE, Mylyn (a task-based development environment in Eclipse), Maven for Eclipse, AspectJ Development Tools, and many other useful Eclipse plug-ins into a single package. In each new version, more features are being added, such as Groovy scripting language support, a graphical Spring configuration editor, visual development tools for projects such as Spring Batch and Spring Integration, and support for the Pivotal tc Server application server.

SpringSource was bought by VMware and incorporated into Pivotal Software.

In addition to the Java-based suite, a Groovy/Grails Tool Suite is available with similar capabilities but targeted at Groovy and Grails development (http://spring.io/tools).

The Spring Security Project

The Spring Security project (http://projects.spring.io/spring-security), formerly known as the Acegi Security System for Spring, is another important project within the Spring portfolio. Spring Security provides comprehensive support for both web application and method-level security. It tightly integrates with the Spring Framework and other commonly used authentication mechanisms, such as HTTP basic authentication, form-based login, X.509 certificate, and single sign-on (SSO) products (for example, CA SiteMinder). It provides role-based access control to application resources, and in applications with more-complicated security requirements (for example, data segregations), use of an access control list (ACL) is supported. However, Spring Security is mostly used in securing web applications, which we discuss in detail in Chapter 16.

Spring Boot

Setting up the basis of an application is a cumbersome job. Configuration files for the project must be created, and additional tools (like an application server) must be installed and configured. Spring Boot (http://projects.spring.io/spring-boot/) is a Spring project that makes it easy to create stand-alone, production-grade Spring-based applications that you can just run. Spring Boot comes with out-of-the-box configurations for different types of Spring applications that are packed in *starter* packages. The `web-starter` package, for example, contains a preconfigured and easily customizable web application context and supports Tomcat 7+, Jetty 8+, and Undertow 1.3 embedded servlet containers out of the box.

Spring Boot also wraps up all dependencies a Spring application needs, taking into account compatibility between versions. At the time of writing, the current version of Spring Boot is 2.0.0.RELEASE.

Spring Boot is covered in Chapter 4, as an alternative Spring project configuration, and most of the projects assigned to later chapters will be run using Spring Boot because it makes development and testing more practical and faster.

Spring Batch and Integration

Needless to say, batch job execution and integration are common use cases in applications. To cope with this need and to make it easy for developers in these areas, Spring created the Spring Batch and Spring Integration projects. Spring Batch provides a common framework and various policies for batch job implementation, reducing a lot of boilerplate code. By implementing the Enterprise Integration Patterns (EIP), Spring Integration can make integrating Spring applications with external systems easy. We discuss the details in Chapter 20.

Many Other Projects

We've covered the core modules of Spring and some of the major projects within the Spring portfolio, but there are many other projects that have been driven by the need of the community for different requirements. Some examples include Spring Boot, Spring XD, Spring for Android, Spring Mobile, Spring Social, and Spring AMQP. Some of these projects are discussed further in Chapter 20. For additional details, you can refer to the Spring by Pivotal web site (`www.spring.io/projects`).

Alternatives to Spring

Going back to our previous comments on the number of open source projects, you should not be surprised to learn that Spring is not the only framework offering dependency injection features or full end-to-end solutions for building applications. In fact, there are almost too many projects to mention. In the spirit of being open, we include a brief discussion of several of these frameworks here, but it is our belief that none of these platforms offers quite as comprehensive a solution as that available in Spring.

JBoss Seam Framework

Founded by Gavin King (the creator of the Hibernate ORM library), the Seam Framework (`http://seamframework.org`) is another full-blown DI-based framework. It supports web application front-end development (JSF), business logic layer (EJB 3), and JPA for persistence. As you can see, the main difference between Seam and Spring is that the Seam Framework is built entirely on JEE standards. JBoss also contributes the ideas in the Seam Framework back to the JCP and has become JSR-299 (Contexts and Dependency Injection for the Java EE Platform).

Google Guice

Another popular DI framework is Google Guice (`http://code.google.com/p/google-guice`). Led by the search engine giant Google, Guice is a lightweight framework that focuses on providing DI for application configuration management. It was also the reference implementation of JSR-330 (Dependency Injection for Java).

PicoContainer

PicoContainer (`http://picocontainer.com`) is an exceptionally small DI container that allows you to use DI for your application without introducing any dependencies other than PicoContainer. Because PicoContainer is nothing more than a DI container, you may find that as your application grows, you need to introduce another framework, such as Spring, in which case you would have been better off using Spring from the start. However, if all you need is a tiny DI container, then PicoContainer is a good choice, but since Spring packages the DI container separately from the rest of the framework, you can just as easily use that and keep the flexibility for the future.

JEE 7 Container[7]

As discussed previously, the concept of DI was widely adopted and also realized by JCP. When you are developing an application for application servers compliant with JEE 7 (JSR-342), you can use standard DI techniques across all layers.

Summary

In this chapter, we gave you a high-level view of the Spring Framework, complete with discussions of all the major features, and we guided you to the relevant sections of the book where these features are discussed in detail. After reading this chapter, you should understand what Spring can do for you; all that remains is to see *how* it can do it. In the next chapter, we discuss all the information you need to know to get up and running with a basic Spring application. We show you how to obtain the Spring Framework and discuss the packaging options, the test suite, and the documentation. Also, Chapter 2 introduces some basic Spring code, including a time-honored Hello World example in all its DI-based glory.

[7]The JEE8 release date has been postponed to the end of 2017; see https://jcp.org/en/jsr/detail?id=366.

CHAPTER 2

Getting Started

Often the hardest part of learning to use any new development tool is figuring out where to begin. Typically, this problem is worse when the tool offers as many choices as Spring. Fortunately, getting started with Spring isn't that hard if you know where to look first. In this chapter, we present you with all the basic knowledge you need to get off to a flying start. Specifically, you will look at the following:

- *Obtaining Spring*: The first logical step is to obtain or build the Spring JAR files. If you want to get up and running quickly, simply use the dependency management snippets in your build system with the examples provided at `http://projects.spring.io/spring-framework`. However, if you want to be on the cutting edge of Spring development, check out the latest version of the source code from Spring's GitHub repository.[1]
- *Spring packaging options*: Spring packaging is modular; it allows you to pick and choose which components you want to use in your application and to include only those components when you are distributing your application. Spring has many modules, but you need only a subset of these modules depending on your application's needs. Each module has its compiled binary code in a JAR file along with corresponding Javadoc and source JARs.
- *Spring guides*: The new Spring web site includes a Guides section located at `http://spring.io/guides`. The guides are meant to be quick, hands-on instructions for building the Hello World version of any development task with Spring. These guides also reflect the latest Spring project releases and techniques, providing you with the most up-to-date samples available.
- *Test suite and documentation*: One of the things members of the Spring community are most proud of is their comprehensive test suite and documentation set. Testing is a big part of what the team does. The documentation set provided with the standard distribution is also excellent.
- *Putting some Spring into Hello World*: All bad punning aside, we think the best way to get started with any new programming tool is to dive right in and write some code. We present a simple example, which is a full DI-based implementation of everyone's favorite Hello World application. Don't be alarmed if you don't understand all the code right away; full discussions follow later in the book.

[1]Find Spring's GitHub repository at `http://github.com/spring-projects/spring-framework`.

© Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho 2017
I. Cosmina et al., *Pro Spring 5*, https://doi.org/10.1007/978-1-4842-2808-1_2

If you are already familiar with the basics of the Spring Framework, feel free to proceed straight to Chapter 3 to dive into IoC and DI in Spring. However, even if you are familiar with the basics of Spring, you may find some of the discussions in this chapter interesting, especially those on packaging and dependencies.

Obtaining the Spring Framework

Before you can get started with any Spring development, you need to obtain the Spring libraries. You have a couple of options for retrieving the libraries: you can use your build system to bring in the modules you want to use, or you can check out and build the code from the Spring GitHub repository. Using a dependency management tool such as Maven or Gradle is often the most straightforward approach; all you need to do is declare the dependency in the configuration file and let the tool obtain the required libraries for you.

If you have an Internet connection and use a build tool such as Maven or Gradle in combination with a smart IDE like Eclipse or IntelliJ IDEA, you can download the Javadoc and libraries automatically so you can access them during development. When you upgrade the versions in the build configuration files when building the project, the libraries and Javadoc will be updated too.

Getting Started Quickly

Visit the Spring Framework project page[2] to obtain a dependency management snippet for your build system to include the latest-release RELEASE version of Spring in your project. You can also use milestones/nightly snapshots for upcoming releases or previous versions.

When using Spring Boot, there is no need to specify the Spring version you want to use, as Spring Boot provides opinionated “starter” project object model (POM) files to simplify your Maven configuration and default Gradle starter configuration. Just keep in mind that Spring Boot versions that precede version 2.0.0.RELEASE use Spring 4.*x* versions.

Checking Spring Out of GitHub

If you want to learn about new features before they make their way even into the snapshots, you can check out the source code directly from Pivotal’s GitHub repository. To check out the latest version of the Spring code, first install Git, which you can download from http://git-scm.com. Then open a terminal shell and run the following command:

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

See the `README.md` file in the project root for full details and requirements on how to build from source.

---

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

Using the Right JDK

The Spring Framework is built in Java, which means you need to be able to execute Java applications on your computer to use it. For this you need to install Java. There are three widely used Java acronyms when people talk about Java applications development.

- A Java virtual machine (JVM) is an abstract machine. It is a specification that provides a runtime environment in which Java bytecode can be executed.
- The Java Runtime Environment (JRE) is used to provide a runtime environment. It is the implementation of the JVM that physically exists. It contains a set of libraries and other files that the JVM uses at runtime. Oracle bought Sun Microsystems in 2010; since then, new versions and patches have been actively provided. Other companies, such as IBM, provide their own implementations of the JVM.
- The Java Development Kit (JDK) contains the JRE, documentation, and Java tools. This is what Java developers install on their machines. A smart editor like IntelliJ IDEA or Eclipse will require you to provide the location of the JDK so classes and documentation can be loaded and used during development.

If you are using a build tool like Maven or Gradle (the source code accompanying the book is organized in a Gradle multimodule project), it will require a JVM as well; Maven and Gradle are both Java-based projects themselves.

The latest stable Java version is Java 8, and Java 9 is scheduled to be released on 21 September 2017. You can download the JDK from `https://www.oracle.com/`. By default it will be installed in some default location on your computer, depending on your operating system. If you want to use Maven or Gradle from the command line, you need to define environment variables for the JDK and Maven/Gradle and to add the path to their executables to the system path. You can find instructions on how to do this on the official site for each product and in the appendix of this book.

Chapter 1 presented a list with Spring versions and the required JDK version. The Spring version covered in the book is 5.0.*x*. The source code presented in the book is written using Java 8 syntax, so you need at least JDK version 8 to be able to compile and run the examples.

Understanding Spring Packaging

Spring modules are simply JAR files that package the required code for that module. After you understand the purpose of each module, you can select the modules required in your project and include them in your code. As of Spring version 5.0.0.RELEASE, Spring comes with 21 modules, packaged into 21 JAR files. Table 2-1 describes these JAR files and their corresponding modules. The actual JAR file name is, for example, `spring-aop-5.0.0.RELEASE.jar`, though we have included only the specific module portion for simplicity (as in `aop`, for example).

Table 2-1. *Spring modules*

Module	Description
aop	This module contains all the classes you need to use Spring's AOP features within your application. You also need to include this JAR in your application if you plan to use other features in Spring that use AOP, such as declarative transaction management. Moreover, classes that support integration with AspectJ are packed in this module.
aspects	This module contains all the classes for advanced integration with the AspectJ AOP library. For example, if you are using Java classes for your Spring configuration and need AspectJ-style annotation-driven transaction management, you will need this module.
beans	This module contains all the classes for supporting Spring's manipulation of Spring beans. Most of the classes here support Spring's bean factory implementation. For example, the classes required for processing the Spring XML configuration file and Java annotations are packed into this module.
beans-groovy	This module contains Groovy classes for supporting Spring's manipulation of Spring beans.
context	This module contains classes that provide many extensions to Spring Core. You will find that all classes need to use Spring's ApplicationContext feature (covered in Chapter 5), along with classes for Enterprise JavaBeans (EJB), Java Naming and Directory Interface (JNDI), and Java Management Extensions (JMX) integration. Also contained in this module are the Spring remoting classes, classes for integration with dynamic scripting languages (for example, JRuby, Groovy, and BeanShell), JSR-303 (Bean Validation), scheduling and task execution, and so on.
context-indexer	This module contains an indexer implementation that provides access to the candidates that are defined in META-INF/spring.components. The core class CandidateComponentsIndex is not meant to be used externally.
context-support	This module contains further extensions to the spring-context module. On the user-interface side, there are classes for mail support and integration with templating engines such as Velocity, FreeMarker, and JasperReports. Also, integration with various task execution and scheduling libraries including CommonJ and Quartz are packaged here.
core	This is the main module that you will need for every Spring application. In this JAR file, you will find all the classes that are shared among all other Spring modules (for example, classes for accessing configuration files). Also, in this JAR, you will find selections of extremely useful utility classes that are used throughout the Spring codebase and that you can use in your own application.
expression	This module contains all support classes for Spring Expression Language (SpEL).
instrument	This module includes Spring's instrumentation agent for JVM bootstrapping. This JAR file is required for using load-time weaving with AspectJ in a Spring application.

(*continued*)

Table 2-1. (*continued*)

Module	Description
`dbc`	This module includes all classes for JDBC support. You will need this module for all applications that require database access. Classes for supporting data sources, JDBC data types, JDBC templates, native JDBC connections, and so on, are packed in this module.
`jms`	This module includes all classes for JMS support.
`messaging`	This module contains key abstractions taken from the Spring Integration project to serve as a foundation for message-based applications and adds support for STOMP messages.
`orm`	This module extends Spring's standard JDBC feature set with support for popular ORM tools including Hibernate, JDO, JPA, and the data mapper iBATIS. Many of the classes in this JAR depend on classes contained in the `spring-jdbc` JAR file, so you definitely need to include that in your application as well.
`oxm`	This module provides support for Object/XML Mapping (OXM). Classes for the abstraction of XML marshalling and unmarshalling and support for popular tools such as Castor, JAXB, XMLBeans, and XStream are packed into this module.
`test`	Spring provides a set of mock classes to aid in testing your applications, and many of these mock classes are used within the Spring test suite, so they are well tested and make testing your applications much simpler. Certainly we have found great use for the mock `HttpServletRequest` and `HttpServletResponse` classes in unit tests for our web applications. On the other hand, Spring provides a tight integration with the JUnit unit-testing framework, and many classes that support the development of JUnit test cases are provided in this module; for example, `SpringJUnit4ClassRunner` provides a simple way to bootstrap the Spring `ApplicationContext` in a unit test environment.
`tx`	This module provides all classes for supporting Spring's transaction infrastructure. You will find classes from the transaction abstraction layer to support the Java Transaction API (JTA) and integration with application servers from major vendors.
`web`	This module contains the core classes for using Spring in your web applications, including classes for loading an `ApplicationContext` feature automatically, file upload support classes, and a bunch of useful classes for performing repetitive tasks such as parsing integer values from the query string.
`web-reactive`	This module contains core interfaces and classes for Spring Web Reactive model.
`web-mvc`	This module contains all the classes for Spring's own MVC framework. If you are using a separate MVC framework for your application, you won't need any of the classes from this JAR file. Spring MVC is covered in more detail in Chapter 16.
`websocket`	This module provides support for JSR-356 (Java API for WebSocket).

Choosing Modules for Your Application

Without a dependency management tool such as Maven or Gradle, choosing which modules to use in your application may be a bit tricky. For example, if you require Spring's bean factory and DI support only, you still need several modules including `spring-core`, `spring-beans`, `spring-context`, and `spring-aop`. If you need Spring's web application support, you then need to further add `spring-web` and so on. Thanks to build tool features such as Maven's transitive dependencies support, all required third-party libraries would be included automatically.

Accessing Spring Modules on the Maven Repository

Founded by Apache Software Foundation, Maven[3] has become one of the most popular tools in managing the dependencies for Java applications, from open source to enterprise environments. Maven is a powerful application building, packaging, and dependency management tool. It manages the entire build cycle of an application, from resource processing and compiling to testing and packaging. There also exists a large number of Maven plug-ins for various tasks, such as updating databases and deploying a packaged application to a specific server (for example, Tomcat, JBoss, or WebSphere). As of this writing, the current Maven version is 3.3.9.

Almost all open source projects support distribution of libraries via the Maven repository. The most popular one is the Maven Central repository hosted on Apache, and you can access and search for the existence and related information of an artifact on the Maven Central web site.[4] If you download and install Maven into your development machine, you automatically gain access to the Maven Central repository. Some other open source communities (for example, JBoss and Spring by Pivotal) also provide their own Maven repository for their users. However, to be able to access those repositories, you need to add the repository into your Maven's setting file or in your project's POM file.

A detailed discussion of Maven is not in the scope of this book, and you can always refer to the online documentation or books that give you a detailed reference to Maven. However, since Maven is widely adopted, it's worth mentioning the typical structure of the project packaging on the Maven repository.

A group ID, artifact ID, packaging type, and version identify each Maven artifact. For example, for log4j, the group ID is `log4j`, the artifact ID is `log4j`, and the packaging type is `jar`. Under that, different versions are defined. For example, for version 1.2.12, the artifact's file name becomes `log4j-1.2.17.jar` under the group ID, artifact ID, and version folder. Maven configuration files are written in XML and must respect the Maven standard syntax defined by the `http://maven.apache.org/maven-v4_0_0.xsd` schema. The default name of a Maven configuration file for a project is `om.xml`, and a sample file is shown here:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.apress.prospring5.ch02</groupId>
  <artifactId>hello-world</artifactId>
  <packaging>jar</packaging>
  <version>5.0-SNAPSHOT</version>
  <name>hello-world</name>
```

[3] `http://maven.apache.org`
[4] `http://search.maven.org`

```
  <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>
```

Maven also defines a typical standard project structure, as depicted in Figure 2-1.

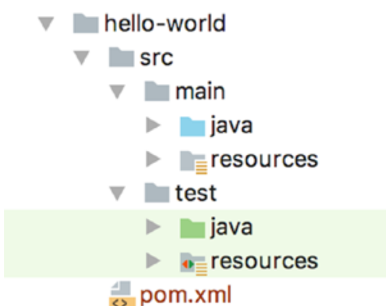

Figure 2-1. *Typical Maven project structure*

The `main` directory contains the classes (the `java` directory) and configuration files (the `resources` directory) for the application.

The `test` directory contains the classes (the `java` directory) and configuration files (the `resources` directory) that are used to test the application from the `main` directory.

Accessing Spring Modules Using Gradle

The Maven project standard structure and artifact categorization and organization is important because Gradle respects the same rules and even uses the Maven central repository to retrieve artifacts. Gradle is a powerful build tool that has given up the bloated XML for configuration and switched to the simplicity and flexibility of Groovy. At the time of writing, the current version of Gradle is 4.0.[5] Starting with version 4.*x*, the Spring team has switched to using Gradle for the configuration of every Spring product. That is why the source code for this book can be built and executed using Gradle too. The default name of a Gradle configuration file for a project is `build.gradle`. The equivalent of the `pom.xml` file depicted earlier (well, one version of it) is shown here:

```
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'
   ...
}
```

That's way more readable, right? As you can observe, the artifacts are identified using the group, artifact, and version as previously introduced with Maven, but the property names differ. And since Gradle is not in the scope of this book either, the coverage for it must end here.

Using Spring Documentation

One of the aspects of Spring that makes it such a useful framework for developers who are building real applications is its wealth of well-written, accurate documentation. In every release, the Spring Framework's documentation team works hard to ensure that all the documentation is finished and polished by the development team. This means that every feature of Spring is not only fully documented in the Javadoc but is also covered in the Spring reference manual included in every distribution. If you haven't yet familiarized yourself with the Spring Javadoc and the reference manual, do so now. This book is not a replacement for either of these resources; rather, it is a complementary reference, demonstrating how to build a Spring-based application from the ground up.

[5]On the official project site you can find detailed instructions on how to download, install, and configure Gradle for development: https://gradle.org/install.

Putting a Spring into Hello World

We hope by this point in the book you appreciate that Spring is a solid, well-supported project that has all the makings of a great tool for application development. However, one thing is missing—we haven't shown you any code yet. We are sure you are dying to see Spring in action, and because we cannot go any longer without getting into the code, let's do just that. Do not worry if you do not fully understand all the code in this section; we go into much more detail on all the topics as we proceed through the book.

Building the Sample Hello World Application

Now, we are sure you are familiar with the traditional Hello World example, but just in case you have been living on the moon for the past 30 years, the following code snippet shows the Java version in all its glory:

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

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

As examples go, this one is pretty simple—it does the job, but it is not very extensible. What if we want to change the message? What if we want to output the message differently, maybe to standard error instead of standard output or enclosed in HTML tags rather than as plain text? We are going to redefine the requirements for the sample application and say that it must support a simple, flexible mechanism for changing the message, and it must be easy to change the rendering behavior. In the basic Hello World example, you can make both of these changes quickly and easily by just changing the code as appropriate. However, in a bigger application, recompiling takes time, and it requires the application to be fully tested again. A better solution is to externalize the message content and read it in at runtime, perhaps from the command-line arguments shown in the following code snippet:

```
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!");
        }
    }
}
```

This example accomplishes what we wanted—we can now change the message without changing the code. However, there is still a problem with this application: the component responsible for rendering the message is also responsible for obtaining the message. Changing how the message is obtained means changing the code in the renderer. Add to this the fact that we still cannot change the renderer easily; doing so means changing the class that launches the application.

If we take this application a step further (away from the basics of Hello World), a better solution is to refactor the rendering and message retrieval logic into separate components. Plus, if we really want to make your application flexible, we should have these components implement interfaces and define the interdependencies between the components and the launcher using these interfaces. By refactoring the message retrieval logic, we can define a simple `MessageProvider` interface with a single method, `getMessage()`, as shown in the following code snippet:

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

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

The `MessageRenderer` interface is implemented by all components that can render messages, and such a component is depicted in the following code snippet:

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

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

As you can see, the `MessageRenderer` interface declares a method, `render()`, and also a JavaBean-style method, `setMessageProvider()`. Any `MessageRenderer` implementations are decoupled from message retrieval and delegate that responsibility to the `MessageProvider` instance with which they are supplied. Here, `MessageProvider` is a dependency of `MessageRenderer`. Creating simple implementations of these interfaces is easy, as shown in the following code snippet:

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

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

You can see that we have created a simple `MessageProvider` that always returns "Hello World!" as the message. The `StandardOutMessageRenderer` class shown next is just as simple:

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

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

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

Now all that remains is to rewrite the `main()` method of the entry class.

```
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();
    }
}
```

Figure 2-2 depicts the abstract schema of the application built so far.

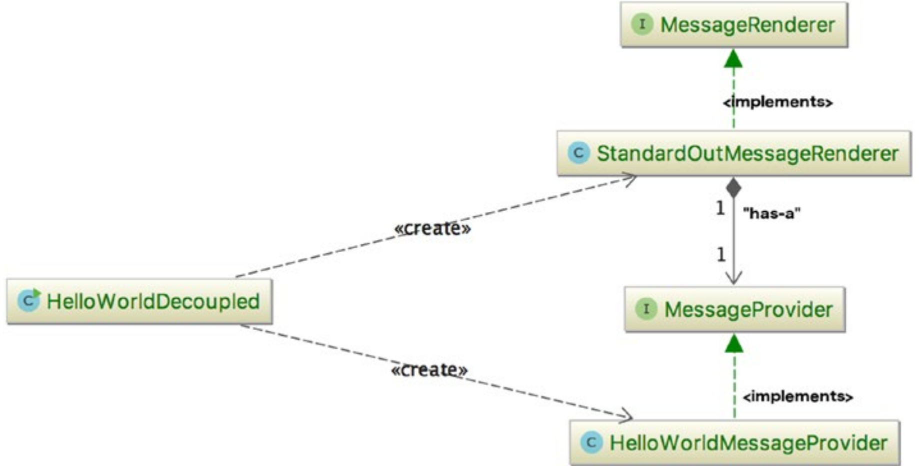

***Figure 2-2.** A little more decoupled Hello World application*

The code here is fairly simple. We instantiate instances of HelloWorldMessageProvider and StandardOutMessageRenderer, although the declared types are MessageProvider and MessageRenderer, respectively. This is because we need to interact only with the methods provided by the interface in the programming logic, and HelloWorldMessageProvider and StandardOutMessageRenderer already implemented those interfaces, respectively. Then, we pass MessageProvider to MessageRenderer and invoke MessageRenderer.render(). If we compile and run this program, we get the expected "Hello World!" output. Now, this example is more like what we are looking for, but there is one small problem. Changing the implementation of either the MessageRenderer or MessageProvider interface means a change to the code. To get around this, we can create a simple factory class that reads the implementation class names from a properties file and instantiates them on behalf of the application, as shown here:

```
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;
    }
}
```

The implementation here is trivial and naive, the error handling is simplistic, and the name of the configuration file is hard-coded, but we already have a substantial amount of code. The configuration file for this class is quite simple.

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

To make use of the previous implementation, you must modify the main method again.

```
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();
    }
}
```

Before we move on to see how we can introduce Spring into this application, let's quickly recap what we have done. Starting with the simple Hello World application, we defined two additional requirements that the application must fulfill. The first was that changing the message should be simple, and the second was that changing the rendering mechanism should also be simple. To meet these requirements, we used two interfaces: `MessageProvider` and `MessageRenderer`. The `MessageRenderer` interface depends on an implementation of the `MessageProvider` interface to be able to retrieve a message to render. Finally, we added a simple factory class to retrieve the names of the implementation classes and instantiate them as applicable.

Refactoring with Spring

The final example shown earlier met the goals laid out for the sample application, but there are still problems with it. The first problem is that we had to write a lot of glue code to piece the application together, while at the same time keeping the components loosely coupled. The second problem is that we still had to provide the implementation of `MessageRenderer` with an instance of `MessageProvider` manually. We can solve both of these problems by using Spring. To solve the problem of too much glue code, we can completely remove the `MessageSupportFactory` class from the application and replace it with a Spring interface, `ApplicationContext`. Don't worry too much about this interface; for now, it is enough to know that this interface is used by Spring for storing all the environmental information with regard to an application being managed by Spring. This interface extends another interface, `ListableBeanFactory`, which acts as the provider for any Spring-managed bean instance.

```
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();
    }
}
```

In the previous code snippet, you can see that the `main()` method obtains an instance of `ClassPathXmlApplicationContext` (the application configuration information is loaded from the file `spring/app-context.xml` in the project's classpath), typed as `ApplicationContext`, and from this, it obtains the `MessageRenderer` instances by using the `ApplicationContext.getBean()` method. Don't worry too much about the `getBean()` method for now; just know that this method reads the application configuration (in this case, an XML file), initializes Spring's `ApplicationContext` environment, and then returns the configured bean[6] instance. This XML file (`app-context.xml`) serves the same purpose as the one used for `MessageSupportFactory`.

```
<?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>
```

The previous file shows a typical Spring `ApplicationContext` configuration. First, Spring's namespaces are declared, and the default namespace is `beans`. The `beans` namespace is used to declare the beans that need to be managed by Spring and to declare their dependency requirements (for the preceding example, the renderer bean's `messageProvider` property is referencing the provider bean). Spring will resolve and inject those dependencies.

Afterward, we declare the bean with the ID `provider` and the corresponding implementation class. When Spring sees this bean definition during the `ApplicationContext` initialization, it will instantiate the class and store it with the specified ID.

Then the `renderer` bean is declared, with the corresponding implementation class. Remember that this bean depends on the `MessageProvider` interface for getting the message to render. To inform Spring about the DI requirement, we use the `p` namespace attribute. The tag attribute `p:messageProvider-ref="provider"` tells Spring that the bean's property, `messageProvider`, should be injected with another bean. The bean to be injected into the property should reference a bean with the ID `provider`. When Spring sees this definition, it will instantiate the class, look up the bean's property named `messageProvider`, and inject it with the bean instance with the ID `provider`.

[6]A *bean* is what an instance of a class is called in Spring.

As you can see, upon the initialization of Spring's ApplicationContext, the main() method now just obtains the MessageRenderer bean by using its type-safe getBean() method (passing in the ID and the expected return type, which is the MessageRenderer interface) and calls render(); Spring has created the MessageProvider implementation and injected it into the MessageRenderer implementation. Notice that we didn't have to make any changes to the classes that are being wired together using Spring. In fact, these classes have no reference to Spring and are completely oblivious to its existence. However, this isn't always the case. Your classes can implement Spring-specified interfaces to interact in a variety of ways with the DI container.

With your new Spring configuration and modified main() method, let's see it in action. Using Gradle, enter the following commands into your terminal to build the project and the root of your source code:

```
gradle clean build copyDependencies
```

The only required Spring module to be declared in your configuration file is spring-context. Gradle will automatically bring in any transitive dependencies required for this module. In Figure 2-3 you can see the transitive dependencies of spring-context.jar.

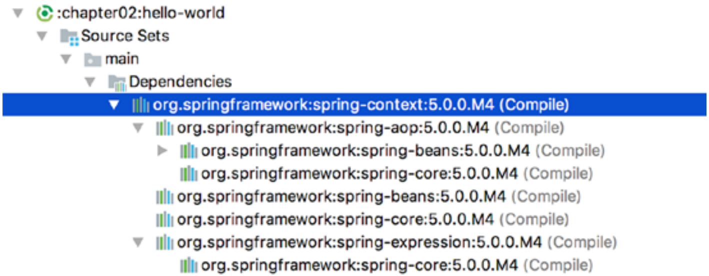

Figure 2-3. *spring-context and its transitive dependencies depicted in IntelliJ IDEA*

The previous command will build the project from scratch, deleting previously generated files, and copy all required dependencies in the same location where the resulting artifact is placed, under build/libs. This path value will also be used as an appending prefix to the library files added to MANIFEST.MF when building the JAR. See the Chapter 2 source code (available on the Apress web site), specifically the Gradle hellor-world/build.properties file, for more information if you are unfamiliar with the Gradle JAR building configuration and process. Finally, to run the Spring DI sample, enter the following commands:

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

At this point, you should see some log statements generated by the Spring container's startup process followed by the expected Hello World output.

Spring Configuration Using Annotations

Starting with Spring 3.0, XML configuration files are no longer necessary when developing a Spring application. They can be replaced with annotations and configuration classes. Configuration classes are Java classes annotated with @Configuration that contain bean definitions (methods annotated with @Bean) or are configured themselves to identify bean definitions in the application by annotating them with @ComponentScanning. The equivalent of the app-context.xml file presented earlier is shown here:

```
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 {

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

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

The main() method has to be modified to replace ClassPathXmlApplicationContext with another ApplicationContext implementation that knows how to read bean definitions from configuration classes. That class is 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();
        }
}
```

This is just one version of configuration using annotations and configuration classes. Without XML, things get pretty flexible when it comes to Spring configuration. You'll learn more about that later in this book, but the focus when it comes to configuration is on Java configuration and annotations.

Some of the interfaces and classes defined in the Hello World sample may be used in later chapters. Although we showed the full source code in this sample, future chapters may show condensed versions of code to be less verbose, especially in the case of incremental code modifications. The code has been organized a little, and all classes that can be used in Spring future examples were placed under the `com.apress.prospring5.ch2.decoupled` and `com.apress.prospring5.ch2.annotated` packages, but keep in mind in a real application you would want to layer your code appropriately.

Summary

In this chapter, we presented you with all the background information you need to get up and running with Spring. We showed you how to get started with Spring through dependency management systems and the current development version directly from GitHub. We described how Spring is packaged and the dependencies you need for each of Spring's features. Using this information, you can make informed decisions about which of the Spring JAR files your application needs and which dependencies you need to distribute with your application. Spring's documentation, guides, and test suite provide Spring users with an ideal base from which to start their Spring development, so we took some time to investigate what is made available by Spring. Finally, we presented an example of how, using Spring DI, it is possible to make the traditional Hello World a loosely coupled, extendable message-rendering application. The important thing to realize is that we only scratched the surface of Spring DI in this chapter, and we barely made a dent in Spring as a whole. In the next chapter, we take look at IoC and DI in Spring.

CHAPTER 3

Introducing IoC and DI in Spring

In Chapter 2, we covered the basic principles of inversion of control. Practically, dependency injection is a specialized form of IoC, although you will often find that the two terms are used interchangeably. In this chapter, we give you a much more detailed look at IoC and DI, formalizing the relationship between the two concepts and looking in great detail at how Spring fits into the picture.

After defining both and looking at Spring's relationship with them, we explore the concepts that are essential to Spring's implementation of DI. This chapter covers only the basics of Spring's DI implementation; we discuss more advanced DI features in Chapter 4. More specifically, this chapter covers the following topics:

- *Inversion of control concepts*: In this section, we discuss the various kinds of IoC, including dependency injection and dependency lookup. This section presents the differences between the various IoC approaches as well as the pros and cons of each.
- *Inversion of control in Spring*: This section looks at IoC capabilities available in Spring and how they are implemented. In particular, you'll see the dependency injection services that Spring offers, including setter, constructor, and Method Injection.
- *Dependency injection in Spring*: This section covers Spring's implementation of the IoC container. For bean definition and DI requirements, `BeanFactory` is the main interface an application interacts with. However, other than the first few, the remainder of the sample code provided in this chapter focuses on using Spring's `ApplicationContext` interface, which is an extension of `BeanFactory` and provides much more powerful features. We cover the difference between `BeanFactory` and `ApplicationContext` in later sections.
- *Configuring the Spring application context*: The final part of this chapter focuses on using the XML and annotation approaches for `ApplicationContext` configuration. Groovy and Java configuration are further discussed in Chapter 4. This section starts with a discussion of DI configuration and moves on to present additional services provided by `BeanFactory` such as bean inheritance, life-cycle management, and autowiring.

Inversion of Control and Dependency Injection

At its core, IoC, and therefore DI, aims to offer a simpler mechanism for provisioning component dependencies (often referred to as an object's *collaborators*) and managing these dependencies throughout their life cycles. A component that requires certain dependencies is often referred to as the *dependent object* or, in the case of IoC, the *target*. In general, IoC can be decomposed into two subtypes: dependency injection

© Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho 2017

I. Cosmina et al., *Pro Spring 5*, https://doi.org/10.1007/978-1-4842-2808-1_3

and dependency lookup. These subtypes are further decomposed into concrete implementations of the IoC services. From this definition, you can clearly see that when we are talking about DI, we are always talking about IoC, but when we are talking about IoC, we are not always talking about DI (for example, dependency lookup is also a form of IoC).

Types of Inversion of Control

You may be wondering why there are two types of IoC and why these types are split further into different implementations. There seems to be no clear answer to this question; certainly the different types provide a level of flexibility, but to us, it seems that IoC is more of a mixture of old and new ideas. The two types of IoC represent this. Dependency lookup is a much more traditional approach, and at first glance, it seems more familiar to Java programmers. Dependency injection, although it appears counterintuitive at first, is actually much more flexible and usable than dependency lookup. With dependency lookup–style IoC, a component must acquire a reference to a dependency, whereas with dependency injection, the dependencies are injected into the component by the IoC container. Dependency lookup comes in two types: dependency pull and contextualized dependency lookup (CDL). Dependency injection also has two common flavors: constructor and setter dependency injection.

⚠ For the discussions in this section, we are not concerned with how the fictional IoC container comes to know about all the different dependencies, just that at some point, it performs the actions described for each mechanism.

Dependency Pull

To a Java developer, dependency pull is the most familiar types of IoC. In dependency pull, dependencies are pulled from a registry as required. Anyone who has ever written code to access an EJB (2.1 or prior versions) has used dependency pull (that is, via the JNDI API to look up an EJB component). Figure 3-1 shows the scenario of dependency pull via the lookup mechanism.

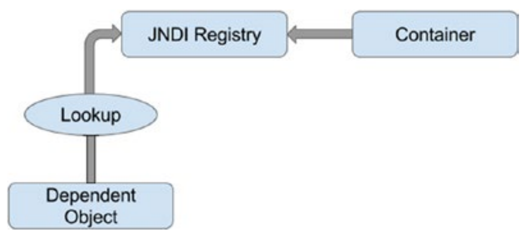

Figure 3-1. *Dependency pull via JNDI lookup*

Spring also offers dependency pull as a mechanism for retrieving the components that the framework manages; you saw this in action in Chapter 2. The following code sample shows a typical dependency pull lookup in a Spring-based application:

```
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();
    }
}
```

This kind of IoC is not only prevalent in JEE-based applications (using EJB 2.1 or prior versions), which make extensive use of JNDI lookups to obtain dependencies from a registry, but also pivotal to working with Spring in many environments.

Contextualized Dependency Lookup

Contextualized dependency lookup (CDL) is similar, in some respects, to dependency pull, but in CDL, lookup is performed against the container that is managing the resource, not from some central registry, and it is usually performed at some set point. Figure 3-2 shows the CDL mechanism.

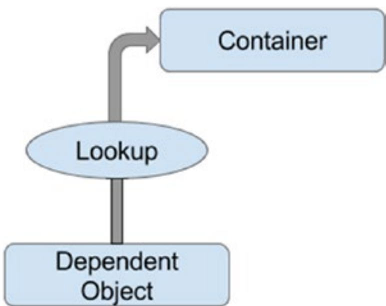

Figure 3-2. *Contextualized dependency lookup*

CDL works by having the component implement an interface similar to that in the following code snippet:

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

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

By implementing this interface, a component is signaling to the container that it wants to obtain a dependency. The container is usually provided by the underlying application server or framework (for example, Tomcat or JBoss) or framework (for example, Spring). The following code snippet shows a simple `Container` interface that provides a dependency lookup service:

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

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

When the container is ready to pass dependencies to a component, it calls `performLookup()` on each component in turn. The component can then look up its dependencies by using the `Container` interface, as shown in the following code snippet:

```
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();
    }
}
```

Constructor Dependency Injection

Constructor dependency injection occurs when a component's dependencies are provided to it in its constructor (or constructors). The component declares a constructor or a set of constructors, taking as arguments its dependencies, and the IoC container passes the dependencies to the component when instantiation occurs, as shown in the following code snippet:

```
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();
        }
}
```

An obvious consequence of using constructor injection is that an object cannot be created without its dependencies; thus, they are mandatory.

Setter Dependency Injection

In *setter dependency injection*, the IoC container injects a component's dependencies via JavaBean-style setter methods. A component's setters expose the dependencies the IoC container can manage. The following code sample shows a typical setter dependency injection–based component:

```
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();
        }
}
```

An obvious consequence of using setter injection is that an object can be created without its dependencies, and they can be provided later by calling the setter.

Within the container, the dependency requirement exposed by the `setDependency()` method is referred to by the JavaBeans-style name, dependency. In practice, setter injection is the most widely used injection mechanism, and it is one of the simplest IoC mechanisms to implement.

⚠ There is a another type of injection supported in Spring called *field injection*, but this will be covered later in the chapter, when you learn about autowiring using the `@Autowire` annotation.

Injection vs. Lookup

Choosing which style of IoC to use—injection or lookup—is not usually a difficult decision. In many cases, the type of IoC you use is mandated by the container you are using. For instance, if you are using EJB 2.1 or prior versions, you must use lookup-style IoC (via JNDI) to obtain an EJB from the JEE container. In Spring, aside from initial bean lookups, your components and their dependencies are always wired together using injection-style IoC.

⚠ When you are using Spring, you can access EJB resources without needing to perform an explicit lookup. Spring can act as an adapter between lookup and injection-style IoC systems, thus allowing you to manage all resources by using injection.

The real question is this: given the choice, which method should you use, injection or lookup? The answer is most definitely injection. If you look at the code in the previous code samples, you can clearly see that using injection has zero impact on your components' code. The dependency pull code, on the other hand, must actively obtain a reference to the registry and interact with it to obtain the dependencies, and using CDL requires your classes to implement a specific interface and look up all dependencies manually. When you are using injection, the most your classes have to do is allow dependencies to be injected by using either constructors or setters.

Using injection, you are free to use your classes completely decoupled from the IoC container that is supplying dependent objects with their collaborators manually, whereas with lookup, your classes are always dependent on the classes and interfaces defined by the container. Another drawback with lookup is that it becomes difficult to test your classes in isolation from the container. Using injection, testing your components is trivial because you can simply provide the dependencies yourself by using the appropriate constructor or setter.

For a more complete discussion of testing by using dependency injection and Spring, refer to Chapter 13.

Lookup-based solutions are, by necessity, more complex than injection-based ones. Although complexity is nothing to be afraid of, we question the validity of adding unneeded complexity to a process as central to your application as dependency management.

All of these reasons aside, the biggest reason to choose injection over lookup is that it makes your life easier. You write substantially less code when you are using injection, and the code that you do write is simple and can, in general, be automated by a good IDE. You will notice that all the code in the injection samples is passive, in that it doesn't actively try to accomplish a task. The most exciting thing you see in injection code is that objects get stored in a field only; no other code is involved in pulling the dependency from any registry or container. Therefore, the code is much simpler and less error prone. Passive code is much simpler to maintain than active code because there is very little that can go wrong. Consider the following code taken from the CDL example:

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

In this code, plenty could go wrong: the dependency key could change, the container instance could be `null`, or the returned dependency might be the incorrect type. We refer to this code as having a lot of moving parts, because plenty of things can break. Using dependency lookup might decouple the components of your application, but it adds complexity in the additional code required to couple these components back together in order to perform any useful tasks.

Setter Injection vs. Constructor Injection

Now that we have established which method of IoC is preferable, you still need to choose whether to use setter injection or constructor injection. *Constructor injection* is particularly useful when you absolutely must have an instance of the dependency class before your component is used. Many containers, Spring included, provide a mechanism for ensuring that all dependencies are defined when you use setter injection, but by using constructor injection, you assert the requirement for the dependency in a container-agnostic manner. Constructor injection also helps achieve the use of immutable objects.

Setter injection is useful in a variety of cases. If the component is exposing its dependencies to the container but is happy to provide its own defaults, setter injection is usually the best way to accomplish this. Another benefit of setter injection is that it allows dependencies to be declared on an interface,

although this is not as useful as you might first think. Consider a typical business interface with one business method, `defineMeaningOfLife()`. If, in addition to this method, you define a setter for injection such as `setEncylopedia()`, you are mandating that all implementations must use or at least be aware of the encyclopedia dependency. However, you don't need to define `setEncylopedia()` in the business interface. Instead, you can define the method in the classes implementing the business interface. While programming in this way, all recent IoC containers, Spring included, can work with the component in terms of the business interface but still provide the dependencies of the implementing class. An example of this may clarify this matter slightly. Consider the business interface in the following code snippet:

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

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

Notice that the business interface does not define any setters for dependency injection. This interface could be implemented as shown in the following code snippet:

```
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";
    }
}
```

As you can see, the `BookwormOracle` class not only implements the Oracle interface but also defines the setter for dependency injection. Spring is more than comfortable dealing with a structure like this. There is absolutely no need to define the dependencies on the business interface. The ability to use interfaces to define dependencies is an often-touted benefit of setter injection, but in actuality, you should strive to keep setters used solely for injection out of your interfaces. Unless you are absolutely sure that all implementations of a particular business interface require a particular dependency, let each implementation class define its own dependencies and keep the business interface for business methods.

Although you shouldn't always place setters for dependencies in a business interface, placing setters and getters for configuration parameters in the business interface is a good idea and makes setter injection a valuable tool. We consider configuration parameters to be a special case for dependencies. Certainly your components depend on the configuration data, but configuration data is significantly different from the types of dependency you have seen so far. We will discuss the differences shortly, but for now, consider the business interface shown in the following code snippet:

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

public interface NewsletterSender {
    void setSmtpServer(String smtpServer);
    String getSmtpServer();
```

```
    void setFromAddress(String fromAddress);
    String getFromAddress();

    void send();
}
```

Classes that send a set of newsletters via e-mail implement the `NewsletterSender` interface. The `send()` method is the only business method, but notice that we have defined two JavaBean properties on the interface. Why are we doing this when we just said that you shouldn't define dependencies in the business interface? The reason is that these values, the SMTP server address and the address the e-mails are sent from, are not dependencies in the practical sense; rather, they are configuration details that affect how all implementations of the `NewsletterSender` interface function. The question here then is this: what is the difference between a configuration parameter and any other kind of dependency? In most cases, you can clearly see whether a dependency should be classified as a configuration parameter, but if you are not sure, look for the following three characteristics that point to a configuration parameter:

- Configuration parameters are passive. In the `NewsletterSender` example depicted in the previous code snippet, the SMTP server parameter is an example of a passive dependency. Passive dependencies are not used directly to perform an action; instead, they are used internally or by another dependency to perform their actions. In the `MessageRenderer` example from Chapter 2, the `MessageProvider` dependency was not passive; it performed a function that was necessary for the `MessageRenderer` to complete its task.
- Configuration parameters are usually information, not other components. By this we mean that a configuration parameter is usually some piece of information that a component needs to complete its work. Clearly, the SMTP server is a piece of information required by the `NewsletterSender`, but the `MessageProvider` is really another component that the `MessageRenderer` needs to function correctly.
- Configuration parameters are usually simple values or collections of simple values. This is really a by-product of the previous two points, but configuration parameters are usually simple values. In Java this means they are a primitive (or the corresponding wrapper class) or a `String` or collections of these values. Simple values are generally passive. This means you can't do much with a `String` other than manipulate the data it represents; and you almost always use these values for information purposes, for example, an `int` value that represents the port number that a network socket should listen on or a `String` that represents the SMTP server through which an e-mail program should send messages.

When considering whether to define configuration options in the business interface, also consider whether the configuration parameter is applicable to all implementations of the business interface or just one. For instance, in the case of implementations of `NewsletterSender`, it is obvious that all implementations need to know which SMTP server to use when sending e-mails. However, we would probably choose to leave the configuration option that flags whether to send secure e-mail off the business interface because not all e-mail APIs are capable of this, and it is correct to assume that many implementations will not take security into consideration at all.

⚠ Recall that in Chapter 2 it was chosen to define the dependencies in the business purposes. This was for illustration purposes and should not be treated in any way as a best practice.

Setter injection also allows you to swap dependencies for a different implementation on the fly without creating a new instance of the parent component. Spring's JMX support makes this possible. Perhaps the biggest benefit of setter injection is that it is the least intrusive of the injection mechanisms.

In general, you should choose an injection type based on your use case. Setter-based injection allows dependencies to be swapped out without creating new objects and also lets your class choose appropriate defaults without the need to explicitly inject an object. Constructor injection is a good choice when you want to ensure that dependencies are being passed to a component and when designing for immutable objects. Do keep in mind that while constructor injection ensures that all dependencies are provided to a component, most containers provide a mechanism to ensure this as well but may incur a cost of coupling your code to the framework.

Inversion of Control in Spring

As mentioned earlier, inversion of control is a big part of what Spring does. The core of Spring's implementation is based on dependency injection, although dependency lookup features are provided as well. When Spring provides collaborators to a dependent object automatically, it does so using dependency injection. In a Spring-based application, it is always preferable to use dependency injection to pass collaborators to dependent objects rather than have the dependent objects obtain the collaborators via lookup. Figure 3-3 shows Spring's dependency injection mechanism. Although dependency injection is the preferred mechanism for wiring together collaborators and dependent objects, you need dependency lookup to access the dependent objects. In many environments, Spring cannot automatically wire up all of your application components by using dependency injection, and you must use dependency lookup to access the initial set of components. For example, in stand-alone Java applications, you need to bootstrap Spring's container in the `main()` method and obtain the dependencies (via the `ApplicationContext` interface) for processing programmatically. However, when you are building web applications by using Spring's MVC support, Spring can avoid this by gluing your entire application together automatically. Wherever it is possible to use dependency injection with Spring, you should do so; otherwise, you can fall back on the dependency lookup capabilities. You will see examples of both in action during the course of this chapter, and we will point them out when they first arise.

Figure 3-3. *Spring's dependency injection mechanism*

An interesting feature of Spring's IoC container is that it has the ability to act as an adapter between its own dependency injection container and external dependency lookup containers. We discuss this feature later in this chapter.

Spring supports both constructor and setter injection and bolsters the standard IoC feature set with a whole host of useful additions to make your life easier.

The rest of this chapter introduces the basics of Spring's DI container, complete with plenty of examples.

Dependency Injection in Spring

Spring's support for dependency injection is comprehensive and, as you will see in Chapter 4, goes beyond the standard IoC feature set we have discussed so far. The rest of this chapter addresses the basics of Spring's dependency injection container, looking at setter, constructor, and Method Injection, along with a detailed look at how dependency injection is configured in Spring.

Beans and BeanFactory

The core of Spring's dependency injection container is the `BeanFactory` interface. `BeanFactory` is responsible for managing components, including their dependencies as well as their life cycles. In Spring, the term *bean* is used to refer to any component managed by the container. Typically, your beans adhere, at some level, to the JavaBeans specification, but this is not required, especially if you plan to use constructor injection to wire your beans together.

If your application needs only DI support, you can interact with the Spring DI container via the `BeanFactory` interface. In this case, your application must create an instance of a class that implements the `BeanFactory` interface and configures it with bean and dependency information. After this is complete, your application can access the beans via `BeanFactory` and get on with its processing.

In some cases, all of this setup is handled automatically (for example, in a web application, Spring's `ApplicationContext` will be bootstrapped by the web container during application startup via a Spring-provided `ContextLoaderListener` class declared in the `web.xml` descriptor file). But in many cases, you need to code the setup yourself. All of the examples in this chapter require manual setup of the `BeanFactory` implementation.

Although the `BeanFactory` can be configured programmatically, it is more common to see it configured externally using some kind of configuration file. Internally, bean configuration is represented by instances of classes that implement the `BeanDefinition` interface. The bean configuration stores information not only about a bean itself but also about the beans that it depends on. For any `BeanFactory` implementation classes that also implement the `BeanDefinitionReader` interface, you can read the `BeanDefinition` data from a configuration file, using either `PropertiesBeanDefinitionReader` or `XmlBeanDefinitionReader`. `PropertiesBeanDefinitionReader` reads the bean definition from properties files, while `XmlBeanDefinitionReader` reads from XML files.

So, you can identify your beans within `BeanFactory`; each bean can be assigned an ID, a name, or both. A bean can also be instantiated without any ID or name (known as an *anonymous bean*) or as an inner bean within another bean. Each bean has at least one name but can have any number of names (additional names are separated by commas). Any names after the first are considered aliases for the same bean. You use bean IDs or names to retrieve a bean from `BeanFactory` and also to establish dependency relationships (that is, bean X depends on bean Y).

BeanFactory Implementations

The description of the `BeanFactory` interface may appear overly complex, but in practice, this is not the case. Take a look at a simple example. Let's say you have an implementation that mimics an oracle that can tell you the meaning of life.

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

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

```
//implementation
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";
    }
}
```

Now let's see, in a stand-alone Java program, how Spring's BeanFactory can be initialized and obtain the oracle bean for processing. Here's the code:

```
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());
  }
}
```

In the previous code sample, you can see that we are using DefaultListableBeanFactory, which is one of the two main BeanFactory implementations supplied with Spring, and that we are reading in the BeanDefinition information from an XML file by using XmlBeanDefinitionReader. Once the BeanFactory implementation is created and configured, we retrieve the oracle bean by using its name, oracle, which is configured in the XML configuration file.

```
<?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>
```

⚠ When declaring a Spring XSD location, it's a best practice to not include the version number. This resolution is already handled for you by Spring as the versioned XSD file is configured through a pointer in the `spring.schemas` file. This file resides in the `spring-beans` module defined as a dependency in your project. This also prevents you from having to modify all of your bean files when upgrading to a new version of Spring.

The previous file declares a Spring bean, gives it an ID of `oracle` and a name of `wiseworm`, and tells Spring that the underlying implementation class is `com.apress.prospring4.ch3.BookwormOracle`. Don't worry too much about the configuration at the moment; we discuss the details in later sections.

Having the configuration defined, run the program shown in the previous code sample; you will see the phrase returned by the `defineMeaningOfLife()` method in the console output.

In addition to `XmlBeanDefinitionReader`, Spring also provides `PropertiesBeanDefinitionReader`, which allows you to manage your bean configuration by using properties rather than XML. Although properties are ideal for small, simple applications, they can quickly become cumbersome when you are dealing with a large number of beans. For this reason, it is preferable to use the XML configuration format for all but the most trivial of applications.

Of course, you are free to define your own `BeanFactory` implementations, although be aware that doing so is quite involved; you need to implement a lot more interfaces than just `BeanFactory` to get the same level of functionality you have with the supplied `BeanFactory` implementations. If all you want to do is define a new configuration mechanism, create your definition reader by developing a class that extends the `DefaultListableBeanFactory` class, which has the `BeanFactory` interface implemented.

ApplicationContext

In Spring, the `ApplicationContext` interface is an extension to `BeanFactory`. In addition to DI services, `ApplicationContext` provides other services, such as transaction and AOP service, message source for internationalization (i18n), and application event handling, to name a few. In developing Spring-based applications, it's recommended that you interact with Spring via the `ApplicationContext` interface. Spring supports the bootstrapping of `ApplicationContext` by manual coding (instantiate it manually and load the appropriate configuration) or in a web container environment via `ContextLoaderListener`. From this point onward, all the sample code in this book uses `ApplicationContext` and its implementations.

Configuring ApplicationContext

Having discussed the basic concepts of IoC and DI and gone through a simple example of using Spring's `BeanFactory` interface, let's dive into the details of how to configure a Spring application. In the following sections, we go through various aspects of configuring Spring applications. Specifically, we should focus our attention on the `ApplicationContext` interface, which provides many more configuration options than the traditional `BeanFactory` interface.

Setting Spring Configuration Options

Before we dive into the details of configuring Spring's ApplicationContext, let's take a look at the options that are available for defining an application's configuration within Spring. Originally, Spring supported defining beans through either properties or an XML file. Since the release of JDK 5 and Spring's support of Java annotations, Spring (starting from Spring 2.5) also supports using Java annotations when configuring ApplicationContext. So, which one is better, XML or annotations? There have been lots of debates on this topic, and you can find numerous discussions on the Internet.[1] There is no definite answer, and each approach has its pros and cons. Using an XML file can externalize all configuration from Java code, while annotations allow the developer to define and view the DI setup from within the code. Spring also supports a mix of the two approaches in a single ApplicationContext. One common approach is to define the application infrastructure (for example, data source, transaction manager, JMS connection factory, or JMX) in an XML file, while defining the DI configuration (injectable beans and beans' dependencies) in annotations. However, no matter which option you choose, stick to it and deliver the message clearly across the entire development team. Agreeing on the style to use and keeping it consistent across the application will make ongoing development and maintenance activities much easier.

To facilitate your understanding of both the XML and annotation configuration, we provide sample code for XML and annotations side by side whenever appropriate, but the focus of this book will be on annotations and Java configuration, as XML was already covered in the previous editions of this book.

Basic Configuration Overview

For XML configuration, you need to declare the required namespace base provided by Spring that your application requires. The following configuration sample shows the most basic sample, which declares only the bean's namespace for you to define the Spring beans. We refer to this configuration file as app-context-xml.xml for XML-style configuration throughout the samples.

```
<?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>
```

Besides beans, Spring provides a large number of other namespaces for different purposes. Some examples include context for ApplicationContext configuration, aop for AOP support, and tx for transactional support. Namespaces are covered in the appropriate chapters.

To use Spring's annotation support in your application, you need to declare the tags shown in the next configuration sample in your XML configuration. We refer to this configuration file as app-context-annotation.xml for XML configuration with annotation support throughout the samples.

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

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

[1]For example, try the Spring community forums at http://forum.spring.io.

```
       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>
```

The `<context:component-scan>` tag tells Spring to scan the code for injectable beans annotated with `@Component`, `@Controller`, `@Repository`, and `@Service` as well as supporting the `@Autowired`, `@Inject`, and `@Resource` annotations under the package (and all its subpackages) specified. In the `<context:component-scan>` tag, multiple packages can be defined by using either a comma, a semicolon, or a space as the delimiter. Moreover, the tag supports inclusion and exclusion of a component scan for more fine-grained control. For example, consider the following configuration sample:

```
<?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>
```

The previous tag tells Spring to scan the package as specified but omit the classes that were assignable to the type as specified in the expression (can be either a class or an interface). Besides the exclude filter, you can also use an include filter. And for the type, you can use annotation, regex, assignable, AspectJ, or custom (with your own filter class that implements `org.springframework.core.type.filter.TypeFilter`) as the filter criteria. The expression format depends on the type you specified.

Declaring Spring Components

After you develop some kind of service class and want to use it in a Spring-based application, you need to tell Spring that those beans are eligible for injection to other beans and have Spring manage them for you. Consider the sample in Chapter 2, where `MessageRender` outputs the message and depends on `MessageProvider` to provide the message to render. The following code sample depicts the interfaces and the implementation of the two services:

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

//renderer interface
public interface MessageRenderer {
    void render();
    void setMessageProvider(MessageProvider provider);
    MessageProvider getMessageProvider();
}
// rendered implementation
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;
        }
}

//provider interface
public interface MessageProvider {
    String getMessage();
}

//provider implementation
public class HelloWorldMessageProvider implements MessageProvider {

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

> The classes shown previously are part of the `com.apress.prospring5.ch2.decoupled` package. They are used in the project specific to this chapter as well, because in a real production application, developers try to reuse code instead of duplicating it. That is why, as you will see when you will get the sources, the project for Chapter 2 is defined as a dependency for some of the projects for Chapter 3.

To declare bean definitions in an XML file, the `<bean ../>` tag is used, and the resulting `app-context-xml.xml` file now looks like this:

```
<?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>
```

The previous tags declare two beans, one with an ID of `provider` with the `HelloWorldMessageProvider` implementation, and the other with an ID of `renderer` with the `StandardOutMessageRenderer` implementation.

Starting with this example, namespaces will no longer be added to configuration samples, unless new namespaces will be introduced, as this will make the bean definitions more visible.

To create bean definitions using annotations, the bean classes must be annotated with the appropriate stereotype annotation,[2] and the methods or constructors must be annotated with `@Autowired` to tell the Spring IoC container to look for a bean of that type and use it as an argument when calling that method. In the following code snippet, the annotations used to create the bean definition are underlined. The stereotype annotations can have as a parameter the name of the resulting bean.

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

import com.apress.prospring5.ch2.decoupled.MessageProvider;
import org.springframework.stereotype.Component;
//simple bean
@Component("provider")
public class HelloWorldMessageProvider implements MessageProvider {
```

[2]These annotations are called *stereotype* because they are part of a package named `org.springframework.stereotype`. This package groups together all annotations used to define beans. These annotations are also relevant to the role of a bean. For example, `@Service` is used to define a service bean, which is a more complex functional bean that provides services that other beans may require, and `@Repository` is used to define a bean that is used to retrieve/save data from/to a database, etc.

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

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

//complex, service 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;
    }
}
```

When bootstrapping Spring's ApplicationContext with the XML configuration depicted here, in file app-context-annotation.xml, Spring will seek out those components and instantiate the beans with the specified names:

```
<?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>
```

Using either approach doesn't affect the way you obtain the beans from ApplicationContext.

```
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();
    }
}
```

Instead of DefaultListableBeanFactory, an instance of GenericXmlApplicationContext is instantiated. The GenericXmlApplicationContext class implements the ApplicationContext interface and is able to bootstrap Spring's ApplicationContext from the configurations defined in XML files.

You can swap the app-context-xml.xml file with app-context-annotation.xml in the provided source code for this chapter, and you will find that both cases produce the same result: "Hello World!" is printed. The only difference is that after the swap the beans providing the functionality are the ones defined with annotations in the com.apress.prospring5.ch3.annotation package.

Using Java Configuration

In Chapter 1 we mentioned that app-context-xml.xml can be replaced with a configuration class, without modifying the classes representing the bean types being created. This is useful when the bean types that the application needs are part of third-party libraries that cannot be modified. Such a configuration class is annotated with @Configuration and contains methods annotated with @Bean that are called directly by the Spring IoC container to instantiate the beans. The bean name will be the same as the name of the method used to create it. The class is shown in the following code sample, and the method names are underlined to make obvious how the resulting beans will be named:

```
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;
        }
}
```

To read the configuration from this class, a different implementation of ApplicationContext is needed.

```
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();
        }
}
```

Instead of DefaultListableBeanFactory, an instance of AnnotationConfigApplicationContext is instantiated. The AnnotationConfigApplicationContext class implements the ApplicationContext interface and is able to bootstrap Spring's ApplicationContext from the configurations defined by the HelloWorldConfiguration class.

A configuration class can be used to read the annotated beans definitions as well. In this case, because the bean's definition configuration is part of the bean class, the class will no longer need any @Bean annotated methods. But, to be able to look for bean definitions inside Java classes, component scanning has to be enabled. This is done by annotating the configuration class with an annotation that is the equivalent of the <context:component-scanning ../> element. This annotation is @ComponentScanning and has the same parameters as the XML analogous element.

```
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 {
}
```

The code to bootstrap a Spring environment using AnnotationConfigApplicationContext will work with this class too, with no additional changes.

In real-life production applications, there might be legacy code, developed with older versions of Spring, or requirements might be of such a nature that require XML and configuration classes. Fortunately, XML and Java configuration can be mixed in more than one way. For example, a configuration class can import bean definitions from an XML file (or more) using @ImportResource, and the same bootstrapping using AnnotationConfigApplicationContext will work in this case as well.

```
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 {
}
```

So, Spring allows you to be really creative when defining your beans; you'll learn more about this in Chapter 4, which is focused solely on Spring application configuration.

Using Setter Injection

To configure setter injection by using XML configuration, you need to specify <property> tags under the <bean> tag for each <property> into which you want to inject a dependency. For example, to assign the message provider bean to the messageProvider property of the messageRenderer bean, you simply change the <bean> tag for the renderer bean, as shown in the following code snippet:

```
<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>
```

From this code, we can see that the provider bean is assigned to the messageProvider property. You can use the ref attribute to assign a bean reference to a property (discussed in more detail shortly).

If you are using Spring 2.5 or newer and have the p namespace declared in your XML configuration file, you can declare the injection as shown in the following code snippet:

```
<?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>
```

The p namespace is not defined in an XSD file and exists only in Spring core; therefore, no XSD is declared in the schemaLocation attribute.

With annotations, it's even simpler. You just need to add an @Autowired annotation to the setter method, as shown in the following code snippet:

```
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;
    }
}
```

Since we declared the <context:component-scan> tag in the XML configuration file, during the initialization of Spring's ApplicationContext, Spring will discover those @Autowired annotations and inject the dependency as required.

Instead of `@Autowired`, you can use `@Resource(name="messageProvider")` to achieve the same result. `@Resource` is one of the annotations in the JSR-250 standard that defines a common set of Java annotations for use on both JSE and JEE platforms. Different from `@Autowired`, the `@Resource` annotation supports the `name` parameter for more fine-grained DI requirements. Additionally, Spring supports use of the `@Inject` annotation introduced as part of JSR-299 (Contexts and Dependency Injection for the Java EE Platform). `@Inject` is equivalent in behavior to Spring's `@Autowired` annotation.

To verify the result, you can use `DeclareSpringComponents` that was presented earlier. As in the previous section, you can swap the `app-context-xml.xml` file with `app-context-annotation.xml` in the provided source code for this chapter, and you will find that both cases produce the same result: "Hello World!" is printed.

Using Constructor Injection

In the previous example, the `MessageProvider` implementation, `HelloWorldMessageProvider`, returned the same hard-coded message for each call of the `getMessage()` method. In the Spring configuration file, you can easily create a configurable `MessageProvider` that allows the message to be defined externally, as shown in the following code snippet:

```
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;
        }
}
```

As you can see, it is impossible to create an instance of `ConfigurableMessageProvider` without providing a value for the message (unless you supply `null`). This is exactly what we want, and this class is ideally suited for use with *constructor injection*. The following code snippet shows how you can redefine the `provider` bean definition to create an instance of `ConfigurableMessageProvider`, injecting the message by using constructor injection:

```
<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>
```

In this code, instead of using a `<property>` tag, we used a `<constructor-arg>` tag. Because we are not passing in another bean this time, just a `String` literal, we use the `value` attribute instead of `ref` to specify the value for the constructor argument. When you have more than one constructor argument or your class has more than one constructor, you need to give each `<constructor-arg>` tag an index attribute to specify the index of the argument, starting at 0, in the constructor signature. It is always best to use the index attribute whenever you are dealing with constructors that have multiple arguments, to avoid confusion between the parameters and ensure that Spring picks the correct constructor.

In addition to the p namespace, as of Spring 3.1, you can also use the c namespace, as shown here:

```
<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>
```

The `c` namespace is not defined in an XSD file either and exists only in Spring Core; therefore, no XSD is declared in the `schemaLocation` attribute.

To use an annotation for constructor injection, we also use the `@Autowired` annotation in the target bean's constructor method, which is an alternative option to the one using setter injection, as shown in the following code snippet:

```
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;
        }
}
```

From the previous code, we can see that we use another annotation, @Value, to define the value to be injected into the constructor. This is the way in Spring we inject values into a bean. Besides simple strings, we can use the powerful SpEL for dynamic value injection (more on this later in this chapter).

However, hard-coding the value in the code is not a good idea; to change it, we would need to recompile the program. Even if you choose annotation-style DI, a good practice is to externalize those values for injection. To externalize the message, let's define the message as a Spring bean in the annotation configuration file, as shown in the following code snippet:

```
<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"/>
</beans>
```

Here we define a bean with an ID of message and type of java.lang.String. Notice that we also use the c namespace for constructor injection to set the string value, and _0 indicates the index for the constructor argument. Having the bean declared, we can take away the @Value annotation from the target bean, as shown in the following code snippet:

```
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;
        }
}
```

Since we declare that the message bean and its ID are the same as the name of the argument specified in the constructor, Spring will detect the annotation and inject the value into the constructor method. Now run the test by using the following code against both the XML (`app-context.xml.xml`) and annotation configurations (`app-context-annotation.xml`), and the configured message will be displayed in both cases:

```
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());
    }
}
```

In some cases, Spring finds it impossible to tell which constructor we want it to use for constructor injection. This usually arises when we have two constructors with the same number of arguments and the types used in the arguments are represented in the same way. Consider the following code:

```
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
    }
}
```

This simply retrieves a bean of type ConstructorConfusion from ApplicationContext and writes the value to console output. Now look at the following configuration code:

```
<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>
```

Which of the constructors is called in this case? Running the example yields the following output:

```
ConstructorConfusion(String) called
```

This shows that the constructor with the String argument is called. This is not the desired effect, since we want to prefix any integer values passed in by using constructor injection with Number:, as shown in the int constructor. To get around this, we need to make a small modification to the configuration, as shown in the following code snippet:

```
<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>
```

Notice now that the <constructor-arg> tag has an additional attribute, type, that specifies the type of argument Spring should look for. Running the example again with the corrected configuration yields the correct output.

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

For annotation-style construction injection, the confusion can be avoided by applying the annotation directly to the target constructor method, as shown in the following code snippet:

```
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();
	}
}
```

By applying the @Autowired annotation to the desired constructor method, Spring will use that method to instantiate the bean and inject the value as specified. As before, we should externalize the value from the configuration.

⚠ The @Autowired annotation can be applied to only one of the constructor methods. If we apply the annotation to more than one constructor method, Spring will complain while bootstrapping ApplicationContext.

Using Field Injection

There is a third type of dependency injection supported in Spring called *field injection*. As the name says, the dependency is injected directly in the field, with no constructor or setter needed. This is done by annotating the class member with the Autowired annotation. This might seem practical, because when the dependency is not needed outside of the object it is part of, it relieves the developer of writing some code that is no longer used after the initial creation of the bean. In the following code snippet, the bean of type Singer has a field of type 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());
        }
}
```

The field is private, but the Spring IoC container does not really care about that; it uses reflection to populate the required dependency. The Inspiration class code is shown here; it is a simple bean with a String member:

```
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;
        }
}
```

The following configuration uses component scanning to discover the bean definitions that will be created by the Spring IoC container:

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

Finding one bean of type `Inspiration`, the Spring IoC container will inject that bean in the `inspirationBean` member of the `singer` bean. That is why when running the example depicted in the next code snippet, “For all my running, I can understand” will be printed in the console.

```
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();
        }
}
```

But there are drawbacks, and this is why using field injection is usually avoided.

- Although it is easy to add dependencies this way, we must be careful not to violate the single responsibility principle. Having more dependencies means more responsibilities for a class, which might lead to a difficulty of separating concerns at refactoring time. The situation when a class becomes bloated is easier to see when dependencies are set using constructors or setters but is quite well hidden when using field injection.
- The responsibility of injecting dependencies is passed to the container in Spring, but the class should clearly communicate the type of dependencies needed using a public interface, through methods or constructors. Using field injections, it can become unclear what type of dependency is really needed and if the dependency is mandatory or not.
- Field injection introduces a dependency of the Spring container, as the `@Autowired` annotation is a Spring component; thus, the bean is no longer a POJO and cannot be instantiated independently.
- Field injection cannot be used for final fields. This type of fields can only be initialized using constructor injection.
- Field injection introduces difficulties when writing tests as the dependencies have to be injected manually.

Using Injection Parameters

In the three previous examples, you saw how to inject other components and values into a bean by using both setter injection and constructor injection. Spring supports a myriad of options for injection parameters, allowing you to inject not only other components and simple values but also Java collections, externally defined properties, and even beans in another factory. You can use all of these injection parameter types for both setter injection and constructor injection by using the corresponding tag under the `<property>` and `<constructor-args>` tags, respectively.

Injecting Simple Values

Injecting simple values into your beans is easy. To do so, simply specify the value in the configuration tag, wrapped inside a `<value>` tag. By default, not only can the `<value>` tag read `String` values, but it can also convert these values to any primitive or primitive wrapper class. The following code snippet shows a simple bean that has a variety of properties exposed for injection:

```
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;
        }
}
```

In addition to the properties, the `InjectSimple` class also defines the `main()` method that creates an `Application Context` and then retrieves an `InjectSimple` bean from Spring. The property values of this bean are then written to the console output. The configuration contained in `app-context-xml.xml` for this bean is depicted in the following snippet:

```
<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>
```

You can see from the two previous code snippets that it is possible to define properties on your bean that accept `String` values, primitive values, or primitive wrapper values and then inject values for these properties by using the `<value>` tag. Here is the output created by running this example as expected:

```
Name: John Mayer
Age: 39
Age in Seconds: 1241401112
Height: 1.92
Is Programmer?: false
```

For annotation-style simple value injection, we can apply the @Value annotation to the bean properties. This time, instead of the setter method, we apply the annotation to the property declaration statement, as we can see in the following code snippet (Spring supports the annotation either at the setter method or in the properties):

```
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;
        }
}
```

This achieves the same result as the XML configuration.

Injecting Values by Using SpEL

One powerful feature introduced in Spring 3 is the Spring Expression Language (SpEL). SpEL enables you to evaluate an expression dynamically and then use it in Spring's `ApplicationContext`. You can use the result for injection into Spring beans. In this section, we take a look at how to use SpEL to inject properties from other beans, by using the example in the preceding section.

Suppose now we want to externalize the values to be injected into a Spring bean in a configuration class, as shown in the following code snippet:

```
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;
        }
}
```

We can then define the bean in the XML configuration and use SpEL to inject the bean's properties into the dependent bean, as shown in the following configuration snippet:

```
<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>
```

Notice that we use the SpEL `#{injectSimpleConfig.name}` in referencing the property of the other bean. For the age, we add 1 to the value of the bean to indicate that we can use SpEL to manipulate the property as we see fit and inject it into the dependent bean. Now we can test the configuration with the program shown in the following code snippet:

```
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();
    }
}
```

The following is the output of the program:

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

When using annotation-style value injection, we just need to substitute the value annotations with the SpEL expressions (see the following code snippet):

```
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();
    }
}
```

The version of `InjectSimpleConfig` is shown here:

```
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;

        // getters here ...
}
```

In the previous snippet, instead of the @Service annotation, @Component was used. Basically, using @Component has the same effect as @Service. Both annotations are instructing Spring that the annotated class is a candidate for autodetection using annotation-based configuration and classpath scanning. However, since the InjectSimpleConfig class is storing the application configuration, rather than providing a business service, using @Component makes more sense. Practically, @Service is a specialization of @Component, which indicates that the annotated class is providing a business service to other layers within the application.

Testing the program will produce the same result. Using SpEL, you can access any Spring-managed beans and properties and manipulate them for application use by Spring's support of sophisticated language features and syntax.

Injecting Beans in the Same XML Unit

As you have already seen, it is possible to inject one bean into another by using the ref tag. The next code snippet shows a class that exposes a setter to allow a bean to be injected:

```
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();
    }
}
```

To configure Spring to inject one bean into another, you first need to configure two beans: one to be injected and one to be the target of the injection. Once this is done, you simply configure the injection by using the <ref> tag on the target bean. The following code snippet shows an example of this configuration (file app-context-xml.xml):

```
<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>
```

Running the InjectRef class produces the following output:

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

An important point to note is that the type being injected does not have to be the exact type defined on the target; the types just need to be compatible. Compatible means that if the declared type on the target is an interface, the injected type must implement this interface. If the declared type is a class, the injected type must be either the same type or a subtype. In this example, the InjectRef class defines the setOracle() method to receive an instance of Oracle, which is an interface, and the injected type is BookwormOracle, a class that implements Oracle. This is a point that causes confusion for some developers, but it is really quite simple. Injection is subject to the same typing rules as any Java code, so as long as you are familiar with how Java typing works, understanding typing in injection is easy.

In the previous example, the ID of the bean to inject is specified by using the local attribute of the <ref> tag. As you will see later, in the section “Understanding Bean Naming,” you can give a bean more than one name so that you can refer to it using a variety of aliases. When you use the local attribute, it means that the <ref> tag only looks at the bean’s ID and never at any of its aliases. Moreover, the bean definition should exist in the same XML configuration file. To inject a bean by any name or import one from other XML configuration files, use the bean attribute of the <ref> tag instead of the local attribute. The following code snippet shows an alternative configuration for the previous example, using an alternative name for the injected 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>
```

In this example, the oracle bean is given an alias by using the name attribute, and then it is injected into the injectRef bean by using this alias in conjunction with the bean attribute of the <ref> tag. Don’t worry too much about the naming semantics at this point. We discuss this in much more detail later in the chapter. Running the InjectRef class again produces the same result as the previous example.

Injection and ApplicationContext Nesting

So far, the beans we have been injecting have been located in the same ApplicationContext (and hence the same BeanFactory) as the beans they are injected into. However, Spring supports a hierarchical structure for ApplicationContext so that one context (and hence the associating BeanFactory) is considered the parent of another. By allowing ApplicationContexts to be nested, Spring allows you to split your configuration into different files, which is a godsend on larger projects with lots of beans.

When nesting ApplicationContext instances, Spring allows beans in what is considered the child context to reference beans in the parent context. ApplicationContext nesting using GenericXmlApplicationContext is simple to understand. To nest one GenericXmlApplicationContext inside another, simply call the setParent() method in the child ApplicationContext, as shown in the following code sample:

```
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();
    }
}
```

The Song class is quite simple, and it is shown here:

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

public class Song {
    private String title;

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

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

Inside the configuration file for the child ApplicationContext, referencing a bean in the parent ApplicationContext works exactly like referencing a bean in the child ApplicationContext, unless you have a bean in the child ApplicationContext that shares the same name. In that case, you simply replace the bean attribute of the ref element with parent, and you are on your way. The following configuration snippet depicts the contents of the configuration file for the parent BeanFactory, named parent-context.xml:

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

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

As you can see, this configuration simply defines two beans: childTitle and parentTitle. Both are String objects with the values Daughters and Gravity. The following configuration snippet depicts the configuration for the child ApplicationContext that is contained in 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>
```

Notice that we have defined four beans here. childTitle in this code is similar to childTitle in the parent except that the String it represents has a different value, indicating that it is located in the child ApplicationContext.

The song1 bean is using the bean ref attribute to reference the bean named parentTitle. Because this bean exists only in the parent BeanFactory, song1 receives a reference to that bean. There are two points of interest here. First, you can use the bean attribute to reference beans in both the child and the parent ApplicationContexts. This makes it easy to reference the beans transparently, allowing you to move beans between configuration files as your application grows. The second point of interest is that you can't use the local attribute to refer to beans in the parent ApplicationContext. The XML parser checks to see that the value of the local attribute exists as a valid element in the same file, preventing it from being used to reference beans in the parent context.

The song2 bean is using the bean ref attribute to reference childTitle. Because that bean is defined in both ApplicationContexts, the song2 bean receives a reference to childTitle in its own ApplicationContext.

The song3 bean is using the <ref> tag to reference childTitle directly in the parent ApplicationContext. Because song3 is using the parent attribute of the <ref> tag, the childTitle instance declared in the child ApplicationContext is ignored completely.

⚠ You may have noticed that, unlike song1 and song2, the song3 bean is not using the p namespace. While the p namespace provides handy shortcuts, it does not provide all the capabilities as when using property tags, such as referencing a parent bean. While we show it as an example, it's best to pick either the p namespace or property tags to define your beans, rather than mixing styles (unless absolutely necessary).

Here is the output from running the HierarchicalAppContextUsage class:

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

As expected, the song1 and song3 beans both get a reference to beans in the parent ApplicationContext, whereas the song2 bean gets a reference to a bean in the child ApplicationContext.

Injecting Collections

Often your beans need access to collections of objects rather than just individual beans or values. Therefore, it should come as no surprise that Spring allows you to inject a collection of objects into one of your beans. Using the collection is simple: you choose either <list>, <map>, <set>, or <props> to represent a List, Map, Set, or Properties instance, and then you pass in the individual items just as you would with any other injection. The <props> tag allows for only Strings to be passed in as the value, because the Properties class allows only for property values to be Strings. When using <list>, <map>, or <set>, you can use any tag you want when injecting into a property, even another collection tag. This allows you to pass in a List of Maps, a Map of Sets, or even a List of Maps of Sets of Lists! The following code snippet shows a class that can have all four of the collection types injected into it:

```
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;
    }
}
```

That is quite a lot of code, but it actually does very little. The `main()` method retrieves a `CollectionInjection` bean from Spring and then calls the `displayInfo()` method. This method just outputs the contents of the `Map`, `Properties`, `Set`, and `List` instances that will be injected from Spring. The configuration required to inject values for each of the properties in the `CollectionInjection` class is depicted next, and the configuration file is named `app-context-xml.xml`.

Also, notice the declaration of the Map<String,Object> property. For JDK versions newer than 5, Spring also supports the strongly typed Collection declaration and will perform the conversion from the XML configuration to the corresponding type specified accordingly.

```
<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>
```

In this code, you can see that we have injected values into all four setters exposed on the CollectionInjection class. For the map property, we have injected a Map instance by using the <map> tag. Notice that each entry is specified using an <entry> tag, and each has a String key and then an entry value. This entry value can be any value you can inject into a property separately; this example shows the use of the <value> and <ref> tags to add a String value and a bean reference to the Map. The LyricHolder class, which is the type of the lyricHolder bean injected in the map in the previous configuration, is depicted here:

```
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 + "}";
        }
}
```

For the props property, we use the <props> tag to create an instance of java.util.Properties and populate it using <prop> tags. Notice that although the <prop> tag is keyed in a similar manner to the <entry> tag, we can specify only String values for each property that goes in the Properties instance.

Also, for the <map> element there is an alternative, more compact configuration using the value and value-ref attributes, instead of the <value> and <ref> elements. The map declared here is equivalent with the one in the previous configuration:

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

Both the <list> and <set> tags work in the same way: you specify each element by using any of the individual value tags such as <value> and <ref> that are used to inject a single value into a property. In the previous configuration, you can see that we have added a String value and a bean reference to both the List and Set instances.

Here is the output generated by the main() method in the class CollectionInjection. As expected, it simply lists the elements added to the collections in the configuration file.

```
Map contents:

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

Properties contents:

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

Set contents:

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

List contents:

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

Remember, with the <list>, <map>, and <set> elements, you can employ any of the tags used to set the value of noncollection properties to specify the value of one of the entries in the collection. This is quite a powerful concept because you are not limited just to injecting collections of primitive values; you can also inject collections of beans or other collections.

Using this functionality, it is much easier to modularize your application and provide different, user-selectable implementations of key pieces of application logic. Consider a system that allows corporate staff to create, proofread, and order their personalized business stationery online. In this system, the finished artwork for each order is sent to the appropriate printer when it is ready for production. The only complication is that some printers want to receive the artwork via e-mail, some via FTP, and others using Secure Copy Protocol (SCP). Using Spring's collection injection, you can create a standard interface for this functionality, as shown in the following code snippet:

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

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

In the previous example, the Recipient class is an empty class. From this interface, you can create multiple implementations, each of which is capable of describing itself to a human, such as the one shown here:

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

public class FtpArtworkSender
        implements ArtworkSender {

    @Override
    public void sendArtwork(String artworkPath, Recipient recipient) {
        // ftp logic here...
    }

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

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

Imagine that you then develop an ArtworkManager class that supports all available implementations of the Artwork-Sender interface. With the implementations in place, you simply pass a List to your ArtworkManager class, and you are on your way. Using the getFriendlyName() method, you can display a list of delivery options for the system administrator to choose from when you are configuring each stationery template. In addition, your application can remain fully decoupled from the individual implementations if you just code to the ArtworkSender interface. We will leave the implementation of the ArtworkManager class as an exercise for you.

Besides the XML configuration, you can use annotations for collection injection. However, you would also like to externalize the values of the collections into the configuration file for easy maintenance. The following snippet is the configuration of four different Spring beans that mimic the same collection properties of the previous sample (configuration file 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>
```

Let's also develop an annotation version of the LyricHolder class. The class content is depicted here:

```
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 + "}";
        }
}
```

In the configuration depicted previously, we make use of the `util` namespace provided by Spring to declare your beans for storing collection properties: the `util` namespace. It greatly simplifies the configuration, as compared to previous versions of Spring. In the class we use to test your configuration, we inject the previous beans and use the JSR-250 `@Resource` annotation with the name specified as an argument to properly identify the beans. The `displayInfo()` method is the same as before so is no longer shown here.

```
@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();
    }
    ...
}
```

Run the test program, and you will get the same result as the sample using XML configuration.

You may wonder why the annotation `@Resource` is used instead of `@Autowired`. It's because the `@Autowired` annotation is semantically defined in a way that it always treats arrays, collections, and maps as sets of corresponding beans, with the target bean type derived from the declared collection value type. So, for example, if a class has an attribute of type `List<ContentHolder>` and has the `@Autowired` annotation defined, Spring will try to inject all beans of type `ContentHolder` within the current `ApplicationContext` into that attribute (instead of the `<util:list>` declared in the configuration file), which will result in either the unexpected dependencies being injected or Spring throwing an exception if no bean of type `ContentHolder` was defined. So, for collection type injection, we have to explicitly instruct Spring to perform injection by specifying the bean name, which the `@Resource` annotation supports.

A combination of `@Autowired` and `@Qualifier` can be used to fulfill the same purpose, but it is always preferable to use one annotation and not two. In the following code snippet, you can see the equivalent configuration to inject a collection using its bean name by using `@Autowired` and `@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;
...
}
```

Using Method Injection

Besides constructor and setter injection, another less frequently used DI feature that Spring provides is *Method Injection*. Spring's Method Injection capabilities come in two loosely related forms, lookup Method Injection and Method Replacement. Lookup Method Injection provides another mechanism by which a bean can obtain one of its dependencies. Method replacement allows you to replace the implementation of any method on a bean arbitrarily, without having to change the original source code. To provide these two features, Spring uses the dynamic bytecode enhancement capabilities of CGLIB.[3]

[3]`cglib` is a powerful, high-performance, and high-quality code generation library. It can extend Java classes and implement interfaces at runtime. It is open source, and you can find the official repository at https://github.com/cglib.

Lookup Method Injection

Lookup Method Injection was added to Spring in version 1.1 to overcome the problems encountered when a bean depends on another bean with a different life cycle, specifically, when a singleton depends on a nonsingleton. In this situation, both setter and constructor injection result in the singleton maintaining a single instance of what should be a nonsingleton bean. In some cases, you will want to have the singleton bean obtain a new instance of the nonsingleton every time it requires the bean in question.

Consider a scenario in which a LockOpener class provides the service of opening any locker. The LockOpener class relies on a KeyHelper class for opening the locker, which was injected into LockOpener. However, the design of the KeyHelper class involves some internal states that make it not suitable for reuse. Every time the openLock() method is called, a new KeyHelper instance is required. In this case, LockOpener will be a singleton. However, if we inject the KeyHelper class by using the normal mechanism, the same instance of the KeyHelper class (which was instantiated when Spring performed the injection the first time) will be reused. To make sure that a new instance of the KeyHelper instance is passed into the openLock() method every time it is invoked, we need to use Lookup Method Injection.

Typically, you can achieve this by having the singleton bean implement the ApplicationContextAware interface (we discuss this interface in the next chapter). Then, using the ApplicationContext instance, the singleton bean can look up a new instance of the nonsingleton dependency every time it needs it. Lookup Method Injection allows the singleton bean to declare that it requires a nonsingleton dependency and that it will receive a new instance of the nonsingleton bean each time it needs to interact with it, without needing to implement any Spring-specific interfaces.

Lookup Method Injection works by having your singleton declare a method, the lookup method, which returns an instance of the nonsingleton bean. When you obtain a reference to the singleton in your application, you are actually receiving a reference to a dynamically created subclass on which Spring has implemented the lookup method. A typical implementation involves defining the lookup method, and thus the bean class, as abstract. This prevents any strange errors from creeping in when you forget to configure the Method Injection and you are working directly against the bean class with the empty method implementation instead of the Spring-enhanced subclass. This topic is quite complex and is best shown by example.

In this example, we create one nonsingleton bean and two singleton beans that both implement the same interface. One of the singletons obtains an instance of the nonsingleton bean by using “traditional” setter injection; the other uses Method Injection. The following code sample depicts the Singer class, which in this example is the type of the nonsingleton 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() {
            //commented because it pollutes the output
        //System.out.println(lyric);
    }
}
```

This class is decidedly unexciting, but it serves the purposes of this example perfectly. Next, you can see the DemoBean interface, which is implemented by both of the singleton bean classes.

```
package com.apress.prospring5.ch3;
public interface DemoBean {
    Singer getMySinger();
    void doSomething();
}
```

This bean has two methods: getMySinger() and doSomething(). The sample application uses the getMySinger() method to get a reference to the Singer instance and, in the case of the method lookup bean, to perform the actual method lookup. The doSomething()method is a simple method that depends on the Singer class to do its processing. The following code snippet shows the StandardLookupDemoBean class, which uses setter injection to obtain an instance of the Singer class:

```
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();
        }
}
```

This code should all look familiar, but notice that the doSomething() method uses the stored instance of Singer to complete its processing. In the following code snippet, you can see the AbstractLookupDemoBean class, which uses Method Injection to obtain an instance of the Singer class.

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

public abstract class AbstractLookupDemoBean
     implements DemoBean {
    public abstract Singer getMySinger();

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

Notice that the getMySinger() method is declared as abstract and that this method is called by the doSomething() method to obtain a Singer instance. The Spring XML configuration for this example is contained in a file named app-context-xml.xml and is shown here:

```
<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>
```

The configuration for the singer and standardLookupBean beans should look familiar to you by now. For abstract-LookupBean, you need to configure the lookup method by using the <lookup-method> tag. The name attribute of the <lookup-method> tag tells Spring the name of the method on the bean that it should override. This method must not accept any arguments, and the return type should be that of the bean you want to return from the method. In this case, the method should return a class of type Singer, or its subclasses. The bean attribute tells Spring which bean the lookup method should return. The following code snippet shows the final piece of code for this example, which is the class containing the main() method used to run the example:

```
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");
    }
}
```

In this code, you can see that the `abstractLookupBean` and the `standardLookupBean` from the `GenericXmlApplicationContext` are retrieved and each reference is passed to the `displayInfo()` method. The instantiation of the abstract class is supported only when using Lookup Method Injection, in which Spring will use CGLIB to generate a subclass of the `AbstractLookupDemoBean` class that overrides the method dynamically. The first part of the `displayInfo()` method creates two local variables of `Singer` type and assigns them each a value by calling `getMySinger()` on the bean passed to it. Using these two variables, it writes a message to the console indicating whether the two references point to the same object.

For the `abstractLookupBean` bean, a new instance of `Singer` should be retrieved for each call to `getMySinger()`, so the references should not be the same.

For `standardLookupBean`, a single instance of `Singer` is passed to the bean by setter injection, and this instance is stored and returned for every call to `getMySinger()`, so the two references should be the same.

⚠ The `StopWatch` class used in the previous example is a utility class available with Spring. You'll find `StopWatch` very useful when you need to perform simple performance tests and when you are testing your applications.

The final part of the `displayInfo()` method runs a simple performance test to see which bean is faster. Clearly, `standardLookupBean` should be faster because it returns the same instance each time, but it is interesting to see the difference. We can now run the `LookupDemo` class for testing. Here is the output we received from this example:

```
[abstractLookupBean]: Singer Instances the Same? false
100000 gets took 431 ms

[standardLookupBean]: Singer Instances the Same? true
100000 gets took 1 ms
```

As you can see, the `Singer` instances are, as expected, the same when we use `standardLookupBean` and different when we use `abstractLookupBean`. There is a noticeable performance difference when you use `standardLookupBean`, but that is to be expected.

Of course, there is an equivalent way to configure the beans presented earlier using annotations. The `singer` bean must have an extra annotation to specify the `prototype` scope.

```
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() {
        // commented to avoid console pollution
        //System.out.println(lyric);
    }
}
```

The AbstractLookupDemoBean class is no longer an abstract class, and the method getMySinger() has an empty body and is annotated with @Lookup that receives as an argument the name of the Singer bean. The method body will be overridden, in the dynamically generated subclass.

```
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; // overriden dynamically
    }

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

The StandardLookupDemoBean class only must be annotated with @Component, and setMySinger must be annotated with @Autowired and @Qualifier to inject the singer bean.

```
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();
        }
}
```

The configuration file, named app-context-annotated.xml, only must enable component scanning for the package containing the annotated classes.

```
<beans ...>

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

The class used to execute the code is identical to class LookupDemo; the only difference is the XML file used as an argument to create the GenericXmlApplicationContext object.

If we want to get rid of XML files totally, this can be done using a configuration class to enable component scanning on the com.apress.prospring5.ch3.annotated package. And this class can be declared right where you need it, meaning in this case inside the class being run to test the beans, as shown here:

```
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) {
                // same implementation as before
                ...
        }
}
```

Alternative configurations using annotations and Java configuration are covered in more detail in Chapter 4.

Considerations for Lookup Method Injection

Lookup Method Injection is intended for use when you want to work with two beans of different life cycles. Avoid the temptation to use Lookup Method Injection when the beans share the same life cycle, especially if they are singletons. The output of running the previous example shows a noticeable difference in performance between using Method Injection to obtain new instances of a dependency and using standard DI to obtain a single instance of a dependency. Also, make sure you don't use Lookup Method Injection needlessly, even when you have beans of different life cycles.

Consider a situation in which you have three singletons that share a dependency in common. You want each singleton to have its own instance of the dependency, so you create the dependency as a nonsingleton, but you are happy with each singleton using the same instance of the collaborator throughout its life. In this case, setter injection is the ideal solution; Lookup Method Injection just adds unnecessary overhead.

When you are using Lookup Method Injection, there are a few design guidelines that you should keep in mind when building your classes. In the earlier examples, we declared the lookup method in an interface. The only reason we did this was we did not have to duplicate the `displayInfo()` method twice for two different bean types. As mentioned earlier, generally you do not need to pollute a business interface with unnecessary definitions that are used solely for IoC purposes. Another point is that although you don't have to make your lookup method abstract, doing so prevents you from forgetting to configure the lookup method and then using a blank implementation by accident. Of course, this works only with XML configuration. Annotation-based configuration forces an empty implementation of the method; otherwise, your bean won't be created.

Method Replacement

Although the Spring documentation classifies method replacement as a form of injection, it is different from what you have seen so far. So far, we have used injection purely to supply beans with their collaborators. Using method replacement, you can replace the implementation of any method on any beans arbitrarily without having to change the source of the bean you are modifying. For example, you have a third-party library that you use in your Spring application, and you need to change the logic of a certain method. However, you are not able to change the source code because it was provided by a third party, so one solution is to use method replacement to just replace the logic for that method with your own implementation.

Internally, you achieve this by creating a subclass of the bean class dynamically. You use CGLIB and redirect calls to the method you want to replace to another bean that implements the `MethodReplacer` interface. In the following code sample, you can see a simple bean that declares two overloads of the `formatMessage()` method:

```
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>";
    }
}
```

You can replace any of the methods on the `ReplacementTarget` class by using Spring's method replacement functionality. In this example, we show you how to replace the `formatMessage(String)` method, and we also compare the performance of the replaced method with that of the original.

To replace a method, you first need to create an implementation of the `MethodReplacer` interface; this is shown in the following code sample:

```
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;
        }
}
```

The MethodReplacer interface has a single method, reimplement(), that you must implement. Three arguments are passed to reimplement(): the bean on which the original method was invoked, a Method instance that represents the method that is being overridden, and the array of arguments passed to the method. The reimplement() method should return the result of your reimplemented logic, and, obviously, the type of the return value should be compatible with the return type of the method you are replacing. In the previous code sample, FormatMessageReplacer first checks to see whether the method that is being overridden is the formatMessage(String) method; if so, it executes the replacement logic (in this case, surrounding the message with <h2> and </h2>) and returns the formatted message to the caller. It is not necessary to check whether the message is correct, but this can be useful if you are using a few MethodReplacers with similar arguments. Using a check helps prevent a situation in which a different MethodReplacer with compatible arguments and return types is used accidentally.

In the configuration sample listed next, you can see an ApplicationContext instance that defines two beans of type ReplacementTarget; one has the formatMessage(String) method replaced, and the other does not (the file is named app-context-xml.xml):

```
<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>
```

As you can see, the MethodReplacer implementation is declared as a bean in ApplicationContext. You then use the <replaced-method> tag to replace the formatMessage(String) method on replacementTargetBean. The name attribute of the <replaced-method> tag specifies the name of the method to replace, and the replacer attribute is used to specify the name of the MethodReplacer bean that we want to replace the method implementation. In cases where there are overloaded methods such as in the ReplacementTarget class, you can use the <arg-type> tag to specify the method signature to match. The <arg-type> tag supports pattern matching, so String is matched to java.lang.String and also to java.lang.StringBuffer.

The following code snippet shows a simple demo application that retrieves both the standardTarget and replacement-Target beans from ApplicationContext, executes their formatMessage(String) methods, and then runs a simple performance test to see which is faster.

```
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");
            //commented to not pollute the console
            //System.out.println(out);
        }

        stopWatch.stop();

        System.out.println("1000000 invocations took: "
                + stopWatch.getTotalTimeMillis() + " ms");
    }
}
```

You should be familiar with this code by now, so we won't go into detail. On our machine, running this example yields the following output:

```
<h2>Thanks for playing, try again!</h2>
1000000 invocations took: 188 ms

<h1>Thanks for playing, try again!</h1>
1000000 invocations took: 24 ms
```

As expected, the output from the `replacementTarget` bean reflects the overridden implementation that the `Method-Replacer` provides. Interestingly, though, the dynamically replaced method is many times slower than the statically defined method. Removing the check for a valid method in `MethodReplacer` made a negligible difference across a number of executions, so we can conclude that most of the overhead is in the CGLIB subclass.

When to Use Method Replacement

Method replacement can prove quite useful in a variety of circumstances, especially when you want to override only a particular method for a single bean rather than all beans of the same type. With that said, we still prefer using standard Java mechanisms for overriding methods rather than depending on runtime bytecode enhancement.

If you are going to use method replacement as part of your application, we recommend you use one `Method-Replacer` per method or group of overloaded methods. Avoid the temptation to use a single `MethodReplacer` for lots of unrelated methods; this results in extra unnecessary String comparisons while your code works out which method it should reimplement. We have found that performing simple checks to ensure that `MethodReplacer` is working with the correct method is useful and doesn't add too much overhead to your code. If you are really concerned about performance, you can simply add a Boolean property to your `MethodReplacer`, which allows you to turn the check on and off using dependency injection.

Understanding Bean Naming

Spring supports quite a complex bean-naming structure that allows you the flexibility to handle many situations. Every bean must have at least one name that is unique within the containing `ApplicationContext`. Spring follows a simple resolution process to determine what name is used for the bean. If you give the `<bean>` tag an `id` attribute, the value of that attribute is used as the name. If no `id` attribute is specified, Spring looks for a `name` attribute, and if one is defined, it uses the first name defined in the `name` attribute. (We say the first name because it is possible to define multiple names within the `name` attribute; this is covered in more detail shortly.) If neither the `id` nor the `name` attribute is specified, Spring uses the bean's class name as the name, provided, of course, that no other bean is using the same class name. If multiple beans of the same type without an ID or name are declared, Spring will throw an exception (of type `org.springframework.beans.factory.NoSuchBeanDefinitionException`) on injection during `ApplicationContext` initialization. The following configuration sample configuration depicts all three naming schemes (`app-context-01.xml`):

```
<beans ...>
    <bean id="string1" class="java.lang.String"/>
    <bean name="string2" class="java.lang.String"/>
    <bean class="java.lang.String"/>
</beans>
```

Each of these approaches is equally valid from a technical point of view, but which is the best choice for your application? To start with, avoid using the automatic name by class behavior. This doesn't allow you much flexibility to define multiple beans of the same type, and it is much better to define your own names. That way, if Spring changes the default behavior in the future, your application continues to work. If you want to see how Spring is naming the beans, using the previous configuration, run the following example:

```
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();
    }
}
```

`ctx.getBeansOfType(String.class)` is used to obtain a map with all beans of type `String` and their IDs that exist within `ApplicationContext`. The keys of the map are the bean IDs that are printed using the lambda expression in the previous code. With the mentioned configuration, this is the output:

```
string1
string2
java.lang.String#0
```

The last line in the previous output sample is the ID that Spring gave to the bean of type `String` that was not named explicitly in the configuration. If the configuration were modified to add another `String` unnamed bean, it would look like this:

```
<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>
```

The output would change to the following:

```
string1
string2
java.lang.String#0
java.lang.String#1
```

Prior to Spring 3.1, the id attribute is the same as the XML identity (that is, xsd:ID), which places a restriction in the characters you can use. As of Spring 3.1, Spring uses xsd:String for the id attribute, so the previous restriction on the characters that you can use is gone. However, Spring will continue to ensure that the id is unique across the entire ApplicationContext. As a general practice, you should give your bean a name by using the id attribute and then associate the bean with other names by using name aliasing, as discussed in the next section.

Bean Name Aliasing

Spring allows a bean to have more than one name. You can achieve this by specifying a space-, comma-, or semicolon-separated list of names in the name attribute of the bean's <bean> tag. You can do this in place of, or in conjunction with, the id attribute. Besides using the name attribute, you can use the <alias> tag for defining aliases for Spring bean names. The following configuration sample shows a simple <bean> configuration that defines multiple names for a single bean (app-context-02.xml):

```
<beans ...>
    <bean id="john" name="john johnny,jonathan;jim" class="java.lang.String"/>
    <alias name="john" alias="ion"/>
</beans>
```

As you can see, we have defined six names: one using the id attribute and the other four as a list using all allowed bean name delimiters in the name attribute (this is just for demonstration purposes and is not recommended for real-life development). In real-life development, it's recommended you standardize on the delimiter to use for separating bean names' declarations within your application. One more alias was defined using the <alias> tag. The following code sample depicts a Java routine that grabs the same bean from the ApplicationContext instance six times using different names and verifies that they are the same bean. Also, it makes use of the previously introduced ctx.getBeansOfType(..) method to make sure there is only one String bean in the context.

```
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();
    }
}
```

Executing the previous code will print `true` five times and the “There is only one String bean” text, verifying that the beans accessed using different names are, in fact, the same bean.

You can retrieve a list of the bean aliases by calling `ApplicationContext.getAliases(String)` and passing in any of the beans’ names or IDs. The list of aliases, other than the one you specified, will then be returned as a `String` array.

It was mentioned before that prior to Spring 3.1 the `id` attribute is the same as the XML identity (that is, `xsd:ID`), which means bean IDs could not contain special characters like space-, comma-, or semicolon. Starting with Spring 3.1, `xsd:String` is used for the `id` attribute, so the previous restriction on the characters that you can use is gone. However, this does not mean you can use the following:

```
<bean name="jon johnny,jonathan;jim" class="java.lang.String"/>
```

instead of this:

```
<bean id="jon johnny,jonathan;jim" class="java.lang.String"/>
```

The `name` and `id` attribute values are treated differently by the Spring IoC. You can retrieve a list of the bean aliases by calling `ApplicationContext.getAliases(String)` and passing in any one of the beans’ names or IDs. The list of aliases, other than the one you specified, will then be returned as a `String` array. This means, in the first case, `jon` will become the id, and the rest of values will become aliases.

In the second case, when the same string is used as a value for the `id` attribute, the full string becomes a unique identifier for the bean. This can be easily tested with a configuration like the one shown here (found in file `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>
```

and a main class like the one shown in the following code sample:

```
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();
    }
}
```

When run, this will produce the following output:

```
id: jon
 aliases: jonathan, jim, johnny

id: jon johnny,jonathan;jim
 aliases:
```

As you can see, the map with `String` beans contains two beans, one with the `jon` unique identifier and three aliases and one with the `jon johnny,jonathan;jim` unique identifier and no aliases.

Bean name aliasing is a strange beast because it is not something you tend to use when you are building a new application. If you are going to have many other beans inject another bean, they may as well use the same name to access that bean. However, as your application goes into production and maintenance work gets carried out, modifications are made, and so on, bean name aliasing becomes more useful.

Consider the following scenario: you have an application in which 50 beans, configured using Spring, all require an implementation of the `Foo` interface. Twenty-five of the beans use the `StandardFoo` implementation with the bean name `standardFoo`, and the other 25 use the `SuperFoo` implementation with the `superFoo` bean name. Six months after you put the application into production, you decide to move the first 25 beans to the `SuperFoo` implementation. To do this, you have three options.

- The first is to change the implementation class of the `standardFoo` bean to `SuperFoo`. The drawback of this approach is that you have two instances of the `SuperFoo` class lying around when you really need only one. In addition, you now have two beans to make changes to when the configuration changes.
- The second option is to update the injection configuration for the 25 beans that are changing, which changes the beans' names from `standardFoo` to `superFoo`. This approach is not the most elegant way to proceed. You could perform a find and replace, but then rolling back your changes when management isn't happy means retrieving an old version of your configuration from your version control system.
- The third, and most ideal, approach is to remove (or comment out) the definition for the `standardFoo` bean and make `standardFoo` an alias to `superFoo`. This change requires minimal effort, and restoring the system to its previous configuration is just as simple.

Bean Naming with Annotation Configurations

When bean definitions are declared using annotations, bean naming is a little different than XML, and there are more interesting things you can do. Let's start with the basics, though: declaring bean definitions using stereotype annotations (`@Component` and all its specializations such as `Service`, `Repository`, and `Controller`).

Consider the following `Singer` class:

```
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);
        }
}
```

This class contains the declaration of a singleton bean of type `Singer` written using the `@Component` annotation. The `@Component` annotation does not have any arguments, so the Spring IoC container decides a unique identifier for the bean. The convention followed in this case is to name the bean, as the class itself, but downcasing the first letter. This means that the bean will be named `singer`. This convention is respected by other stereotype annotations as well. To test this, the following class can be used:

```
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();
        }
}
```

The app-context-annotated.xml configuration file contains only a component scanning declaration for com.apress.prospring5.ch3.annotated so it won't be shown again. When the previous class is run, the following output is printed in the console:

```
id: singer
```

Thus, using @Component("singer") is equivalent to annotating the Singer class with @Component. If you want to name the bean differently, the @Component annotation must receive the bean name as an argument.

```
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);
        }
}
```

As expected, if AnnotatedBeanNaming is run, the following output is produced:

```
id: johnMayer
```

But, what about aliases? As the argument for the @Component annotation becomes the unique identifier for the bean, bean aliasing is not possible when declaring the bean in this way. This is where Java configuration comes to the rescue. Let's consider the following class, which contains a static configuration class defined within it (yes, Spring allows this, and we are being practical here, keeping all the logic in the same file):

```
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();
    }
}
```

This class contains a bean definition for a bean of type Singer declared by annotating the singer() method with the @Bean annotation. When no argument is provided for this annotation, the bean unique identifier, its id, becomes the method name. Thus, when the previous class is run, we get the following output:

```
id: singer
 aliases:
```

To declare aliases, we make use of the name attribute of the @Bean annotation. This attribute is the default one for this annotation, which means in this case declaring the bean by annotating the singer() method with @Bean, @Bean("singer"), or @Bean(name="singer") will lead to the same result. The Spring IoC container will create a bean of type Singer and with the singer ID.

If the value for this attribute is a string containing an alias-specific separator (space, comma, semicolon), the string will become the ID of the bean. But, if the value for it is an array of strings, the first one becomes the id and the others become aliases. Modify the bean configuration as shown here:

```
@Configuration
    public static class AliasBeanConfig {
            @Bean(name={"johnMayer","john","jonathan","johnny"})
            public Singer singer(){
                    return new Singer();
            }
    }
```

When running the AliasConfigDemo class, the output will change to the following:

```
id: johnMayer
 aliases: jonathan, johnny, john
```

When it comes to aliases, in Spring 4.2 the `@AliasFor` annotation was introduced. This annotation is used to declare aliases for annotation attributes, and most Spring annotations make use of it. For example, the `@Bean` annotation has two attributes, `name` and `value`, which are declared as aliases for each other. Using this annotation, they are explicit aliases. The following code snippet is a snapshot of the `@Bean` annotation code and is taken from the official Spring GitHub repository. The code and documentation that are not relevant at the moment were skipped:[4]

```
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 {};

        ...
}
```

Here's an example. Declare an annotation called `@Award` that can be used on `Singer` instances, of course.

```
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 {};
}
```

[4]You can look at the full implementation here: https://github.com/spring-projects/spring-framework/blob/master/spring-core/src/main/java/org/springframework/core/annotation/AliasFor.java.

Using this annotation, you can modify the Singer class like this:

```
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);
        }
}
```

The previous annotation is equivalent to `@Award(value={"grammy", "platinum disk"})` and to `@Award({"grammy", "platinum disk"})`.

But something more interesting can be done with the `@AliasFor` annotation: aliases for meta-annotation attributes can be declared. In the following code snippet, we declare a specialization for the `@Award` annotation that declares an attribute named `name`, which is an alias for the `value` attribute of the `@Award` annotation. And we do this because we want to make it obvious that the argument is a unique bean identifier.

```
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 {};
}
```

Thus, instead of writing the Singer class like this:

```
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);
        }
}
```

we can write it like this:

```
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);
        }
}
```

Creating aliases for attributes of annotations using yet another annotation `@AliasFor` does have limitations. `@AliasFor` cannot be used on any stereotype annotations (`@Component` and its specializations). The reason is that the special handling of these `value` attributes was in place years before `@AliasFor` was invented. Consequently, because of backward compatibility issues, it is simply not possible to use `@AliasFor` with such value attributes. When writing code to do just so (aliasing `value` attributes in stereotype annotations), no compile errors will be shown to you, and the code might even run, but any argument provided for the alias will be ignored. The same goes for the
`@Qualifier` annotation as well.

Understanding Bean Instantiation Mode

By default, all beans in Spring are singletons. This means Spring maintains a single instance of the bean, all dependent objects use the same instance, and all calls to `ApplicationContext.getBean()` return the same instance. We demonstrated this in the previous section, where we were able to use identity comparison (`==`) rather than the `equals()` comparison to check whether the beans were the same.

The term *singleton* is used interchangeably in Java to refer to two distinct concepts: an object that has a single instance within the application and the Singleton design pattern. We refer to the first concept as a *singleton* and to the Singleton pattern as Singleton. The Singleton design pattern was popularized in the seminal *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma et al. (Addison-Wesley, 1994). The problem arises when people confuse the need for singleton instances with the need to apply the Singleton pattern. The following code snippet shows a typical implementation of the Singleton pattern in Java:

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

public class Singleton {
    private static Singleton instance;

    static {
        instance = new Singleton();
    }

    public static Singleton getInstance() {
        return instance;
    }
}
```

This pattern achieves its goal of allowing you to maintain and access a single instance of a class throughout your application, but it does so at the expense of increased coupling. Your application code must always have explicit knowledge of the Singleton class in order to obtain the instance—completely removing the ability to code to interfaces.

In reality, the Singleton pattern is actually two patterns in one. The first, and desired, pattern involves maintenance of a single instance of an object. The second, and less desirable, is a pattern for object lookup that completely removes the possibility of using interfaces. Using the Singleton pattern also makes it difficult to swap out implementations arbitrarily because most objects that require the Singleton instance access the Singleton object directly. This can cause all kinds of headaches when you are trying to unit test your application because you are unable to replace the Singleton with a mock for testing purposes.

Fortunately, with Spring you can take advantage of the singleton instantiation model without having to work around the Singleton design pattern. All beans in Spring are, by default, created as Singleton instances, and Spring uses the same instances to fulfill all requests for that bean. Of course, Spring is not just limited to the use of the Singleton instance; it can still create a new instance of the bean to satisfy every dependency and every call to `getBean()`. It does all of this without any impact on your application code, and for this reason, we like to refer to Spring as being instantiation mode agnostic. This is a powerful concept. If you start off with an object that is a singleton but then discover it is not really suited to multithread access, you can change it to a nonsingleton (prototype) without affecting any of your application code.

⚠ Although changing the instantiation mode of your bean won't affect your application code, it does cause some problems if you rely on Spring's life-cycle interfaces. We cover this in more detail in Chapter 4.

Changing the instantiation mode from singleton to nonsingleton is simple. The following configuration snippets present how this is done in XML and using annotations:

```
<!-- app-context-xml.xml -->
<beans ...>
    <bean id="nonSingleton" class="com.apress.prospring5.ch3.annotated.Singer"
             scope="prototype" c:_0="John Mayer"/>
    </beans>

\\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;
      }
}
```

In the XML configuration, the `Singer` class can be used as a type for a bean declared in XML. If component scanning is not enabled, the annotations in the class will simply be ignored.

As you can see, the only difference between this bean declaration and any of the declarations you have seen so far is that we add the `scope` attribute and set the value to `prototype`. Spring defaults the scope to the value `singleton`. The prototype scope instructs Spring to instantiate a new instance of the bean every time a bean instance is requested by the application. The following code snippet shows the effect this setting has on your application:

```
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();
    }
}
```

Running this example gives you the following output:

```
Identity Equal?: false
Value Equal:? false
John Mayer
John Mayer
```

You can see from this that although the values of the two `String` objects are clearly equal, the identities are not, even though both instances were retrieved using the same bean name.

Choosing an Instantiation Mode

In most scenarios, it is quite easy to see which instantiation mode is suitable. Typically, you will find that singleton is the default mode for your beans. In general, singletons should be used in the following scenarios:

- *Shared object with no state*: You have an object that maintains no state and has many dependent objects. Because you do not need synchronization if there is no state, you do not need to create a new instance of the bean each time a dependent object needs to use it for some processing.

- *Shared object with read-only state*: This is similar to the previous point, but you have some read-only state. In this case, you still do not need synchronization, so creating an instance to satisfy each request for the bean is just adding overhead.
- *Shared object with shared state*: If you have a bean that has state that must be shared, singleton is the ideal choice. In this case, ensure that your synchronization for state writes is as granular as possible.
- *High-throughput objects with writable state*: If you have a bean that is used a great deal in your application, you may find that keeping a singleton and synchronizing all write access to the bean state allows for better performance than constantly creating hundreds of instances of the bean. When using this approach, try to keep the synchronization as granular as possible without sacrificing consistency. You will find that this approach is particularly useful when your application creates a large number of instances over a long period of time, when your shared object has only a small amount of writable state, or when the instantiation of a new instance is expensive.

You should consider using nonsingletons in the following scenarios:

- *Objects with writable state*: If you have a bean that has a lot of writable state, you may find that the cost of synchronization is greater than the cost of creating a new instance to handle each request from a dependent object.
- *Objects with private state*: Some dependent objects need a bean that has private state so that they can conduct their processing separately from other objects that depend on that bean. In this case, singleton is clearly not suitable, and you should use nonsingleton.

The main positive you gain from Spring's instantiation management is that your applications can immediately benefit from the lower memory usage associated with singletons, with very little effort on your part. Then, if you find that singleton mode does not meet the needs of your application, it is a trivial task to modify your configuration to use non-singleton mode.

Implementing Bean Scopes

In addition to the singleton and prototype scopes, other scopes exist when defining a Spring bean for more specific purposes. You can also implement your own custom scope and register it in Spring's `ApplicationContext`. The following bean scopes are supported as of version 4:

- *Singleton*: The default singleton scope. Only one object will be created per Spring IoC container.
- *Prototype*: A new instance will be created by Spring when requested by the application.
- *Request*: For web application use. When using Spring MVC for web applications, beans with request scope will be instantiated for every HTTP request and then destroyed when the request is completed.
- *Session*: For web application use. When using Spring MVC for web applications, beans with session scope will be instantiated for every HTTP session and then destroyed when the session is over.
- *Global session*: For portlet-based web applications. The global session scope beans can be shared among all portlets within the same Spring MVC–powered portal application.

- *Thread*: A new bean instance will be created by Spring when requested by a new thread, while for the same thread, the same bean instance will be returned. Note that this scope is not registered by default.
- *Custom*: Custom bean scope that can be created by implementing the interface `org.springframework.beans.factory.config.Scope` and registering the custom scope in Spring's configuration (for XML, use the class `org.springframework.beans.factory.config.CustomScopeConfigurer`).

Resolving Dependencies

During normal operation, Spring is able to resolve dependencies by simply looking at your configuration file or annotations in your classes. In this way, Spring can ensure that each bean is configured in the correct order so that each bean has its dependencies correctly configured. If Spring did not perform this and just created the beans and configured them in any order, a bean could be created and configured before its dependencies. This is obviously not what you want and would cause all sorts of problems within your application.

Unfortunately, Spring is not aware of any dependencies that exist between beans in your code that are not specified in the configuration. For instance, take one bean, called `johnMayer`, of type `Singer`, which obtains an instance of another bean, called `gopher`, of type `Guitar` using `ctx.getBean()` and uses it when the `johnMayer.sing()` method is called. In this method, you get an instance of type `Guitar` by calling `ctx.getBean("gopher")`, without asking Spring to inject the dependency for you. In this case, Spring is unaware that `johnMayer` depends on `gopher`, and, as a result, it may instantiate `johnMayer` before `gopher`. You can provide Spring with additional information about your bean dependencies using the `depends-on` attribute of the `<bean>` tag. The following configuration snippet (contained in a file named `app-context-01.xml`) shows how the scenario for `johnMayer` and `gopher` would be configured:

```
<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>
```

In this configuration, we are asserting that bean `johnMayer` depends on bean `gopher`. Spring should take this into consideration when instantiating the beans and ensure that `gopher` is created before `johnMayer`. To do this, though, `johnMayer` needs to access `ApplicationContext`. Thus, we also have to tell Spring to inject this reference, so when the `johnMayer.sing()` method will be called, it can be used to procure the `gopher` bean. This is done by making the `Singer` bean implement the `ApplicationContextAware` interface. This is a Spring-specific interface that forces an implementation of a setter for an `ApplicationContext` object. It is automatically detected by the Spring IoC container, and the `ApplicationContext` that the bean is created in is injected into it. This is done after the constructor of the bean is called, so obviously using `ApplicationContext` in the constructor will lead to a `NullPointerException`. You can see the code of the `Singer` class here:

```
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();
        }
}
```

The `Guitar` class is quite simple; it contains only the `sing` method and is shown here:

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

public class Guitar {

        public void sing(){
                System.out.println("Cm Eb Fm Ab Bb");
        }
}
```

To test this example, you can use the following class:

```
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();
        }
}
```

Of course, there is an annotation configuration equivalent to the previous XML configuration. Singer and Guitar must be declared as beans using one of the stereotype annotations (in this case @Component will be used). The novelty here is the @DependsOn annotation, which is placed on the Singer class. This is the equivalent of the depends-on attribute from the XML configuration.

```
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();
        }
}
```

All you have to do now is enable component scanning and then use, in the DependsOnDemo class, application- context-02.xml, to create ApplicationContext.

```
<!-- application-context-02.xml -->
<beans...>
    <context:component-scan
            base-package="com.apress.prospring5.ch3.annotated"/>
</beans>
```

The example will run, and the output will be “Cm Eb Fm Ab Bb.”

When developing your applications, avoid designing them to use this feature; instead, define your dependencies by means of setter and constructor injection contracts. However, if you are integrating Spring with legacy code, you may find that the dependencies defined in the code require you to provide extra information to the Spring Framework.

Autowiring Your Bean

Spring supports five modes for autowiring.

- byName: When using byName autowiring, Spring attempts to wire each property to a bean of the same name. So, if the target bean has a property named foo and a foo bean is defined in ApplicationContext, the foo bean is assigned to the foo property of the target.
- byType: When using byType autowiring, Spring attempts to wire each of the properties on the target bean by automatically using a bean of the same type in ApplicationContext.
- constructor: This functions just like byType wiring, except that it uses constructors rather than setters to perform the injection. Spring attempts to match the greatest numbers of arguments it can in the constructor. So, if your bean has two constructors, one that accepts a String and one that accepts String and an Integer, and you have both a String and an Integer bean in your ApplicationContext, Spring uses the two-argument constructor.
- default: Spring will choose between the constructor and byType modes automatically. If your bean has a default (no-arguments) constructor, Spring uses byType; otherwise, it uses constructor.
- no: This is the default.

So, if you have a property of type String on the target bean and a bean of type String in ApplicationContext, then Spring wires the String bean to the target bean's String property. If you have more than one bean of the same type, in this case String, in the same ApplicationContext instance, then Spring is unable to decide which one to use for the autowiring and throws an exception (of type org.springframework.beans.factory.NoSuchBeanDefinitionException).

The following configuration snippet shows a simple configuration that autowires three beans of the same type by using each of the modes (app-context-03.xml):

```
<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>
```

This configuration should look familiar to you now. Foo and Bar are empty classes. Notice that each of the Target beans has a different value for the autowire attribute. Moreover, the lazy-init attribute is set to true to inform Spring to instantiate the bean only when it is first requested, rather than at startup, so that we can output the result in the correct place in the testing program. The following code sample shows a simple Java application that retrieves each of the Target beans from 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();
    }
}
```

In this code, you can see that the Target class has three constructors: a no-argument constructor, a constructor that accepts a Foo instance, and a constructor that accepts a Foo and a Bar instance. In addition to these constructors, the Target bean has three properties: two of type Foo and one of type Bar. Each of these properties and constructors writes a message to console output when it is called. The main() method simply retrieves each of the Target beans declared in ApplicationContext, triggering the autowire process. Here is the output from running this example:

```
Using byName:

Property fooOne set

Using byType:

Property bar set
Property fooOne set
Property fooTwo set

Using constructor:

Target(Foo, Bar) called
```

From the output, you can see that when Spring uses byName, the only property that is set is foo because this is the only property with a corresponding bean entry in the configuration file. When using byType, Spring sets the value of all three properties. The fooOne and fooTwo properties are set by the fooOne bean, and the bar property is set by the barOne bean. When using a constructor, Spring uses the two-argument constructor, because Spring can provide beans for both arguments and does not need to fall back to another constructor.

When autowiring by type, things gets complicated when bean types are related, and exceptions are thrown when you have more classes that implement the same interface and the property requiring to be autowired specifies the interface as the type, because Spring does not know which bean to inject. To create such a scenario, we'll transform Foo into an interface and declare two bean type implementing it, each with its bean declaration. Let's keep the default configuration as well, no extra naming.

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

public interface Foo {
        // empty interface, used as a marker interface
}

public class FooImplOne implements Foo {
}

public class FooImplOne implements Foo {
}
```

If we were to add a new configuration file, named app-context-04.xml, it would contain the following configuration:

```
<beans ...>

    <bean id="fooOne"
        class="com.apress.prospring5.ch3.xml.complicated.FooImplOne"/>
```

```
    <bean id="fooTwo"
        class="com.apress.prospring5.ch3.xml.complicated.FooImplOne"/>

    <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>
```

For this more simple example, we also introduce the CTarget class. This is identical with the recently introduced Target class; only the main() method differs. The code snippet is depicted here:

```
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();
        }
```

Running the previous class produces the following output:

```
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
       ...
```

The console output is way bigger, but the first lines in the previous output reveal the problem in quite a readable manner. When Spring does not know what bean to autowire, it throws an `UnsatisfiedDependencyException` with an explicit message. It tells you what beans were found but that it cannot choose which to use where. There are two ways to fix this problem. The first way is to use the `primary` attribute in the bean definition that you want Spring to consider first for autowiring and set `true` as its value.

```
<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>
```

So, if the configuration is modified as presented before, when running the example, the following output will be printed:

```
Using byType:

Property bar set
Property fooOne set
Property fooTwo set
```

So, it's all back to normal. But still, the `primary` attribute is a solution only when there are just two bean-related types. If there are more, using it will not get rid of the `UnsatisfiedDependencyException`. What will do the job is the second way, which will give you full control over which bean gets injected where, and this is to name your beans and configure them where to be injected via XML. The previous example is quite a complex and dirty implementation and was conceived just to prove how each of the autowiring types can be configured in XML. When switching to annotation, things change a bit. There is an annotation equivalent to the `lazy-init` attribute; the `@Lazy` annotation is used at the class level to declare beans that will be instantiated the first time they are accessed. Using stereotype annotations, we can create only one configuration for a bean, so it seems pretty logical that the name of the beans does not really matter, as there will be only one bean of each type. Thus, the default autowiring when using configuration via annotation is byType. When there are bean-related types, it is useful to be able to specify that autowiring should be done by name. This is done using the `@Qualifier` annotation, together with the `@Autowired` annotation, and providing the name of the bean being injected as an argument for it.

Consider the following code:

```
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();
        }
}
```

If Foo is a class as depicted here:

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

@Component
public class Foo {

}
```

then when the TrickyTarget class is run, the following output is produced:

```
Property fooOne set
Property fooTwo set
Property bar set
```

The Bar class is just as simple.

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

import org.springframework.stereotype.Component;

@Component
public class Bar {

}
```

If we were to modify the TrickyTarget class and give a name to the bean, as shown here:

```
@Component("gigi")
@Lazy
public class TrickyTarget {
...
}
```

then when running the class, the same output will be produced, as there is only one bean of type Target, and when requested from the context using ctx.getBean(TrickyTarget.class), the context returns the only bean of this type, regardless of its name. Also, if we were to provide a name for the bean of type Bar:

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

import org.springframework.stereotype.Component;

@Component("kitchen")
public class Bar {
}
```

then when running the example again, we would see the same output. This means that the default autowiring type is byType.

As mentioned, things get complicated when bean types are related. Let's transform Foo into an interface and declare two bean types implementing it, each one with its bean declaration. Let's keep the default configuration as well, with no extra naming.

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

//Foo.java
public interface Foo {
        // empty interface, used as a marker interface
}

//FooImplOne.java
@Component
public class FooImplOne implements Foo {
}

//FooImplTwo.java
@Component
public class FooImplTwo implements Foo{
}
```

The TrickyTarget class remains unchanged, and when it's run, we'll see that the output changes to something that looks probably a lot like this:

```
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
...
```

There is a lot more output, but those are the first lines, and as you can see, Spring is really explicit. It tells you that it does not know which bean to autowire through the method setFoo, and it also tells you which selection of beans it has. The names of the beans are decided by Spring based on the class name, by downcasing the first letter of the class name. Using this information, TrickyTarget can be fixed. There are two ways to do this. The first way is to use the @Primary annotation (which is the equivalent of the primary attribute introduced before) on the class defining a bean, which will tell Spring to prioritize this bean when autowiring by type. We will annotate FooImplOne.

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

import org.springframework.context.annotation.Primary;
import org.springframework.stereotype.Component;

@Component
@Primary
public class FooImplOne implements Foo {
}
```

The @Primary annotation is a marker interface; it has no attributes. Its presence on a bean configuration marks the bean as having priority when a bean of this type is needed to be autowired using byType. If you run the TrickyTarget class, the expected output will be printed again.

```
Property fooOne set
Property fooTwo set
Property bar set
```

As in the case of the primary attribute, the @Primary annotation is useful only when you have exactly two related bean types. For handling more related bean types, the Qualifier annotation is more suitable. This is placed next to the @Autowired on the setters that are ambiguous: setFooOne() and setFooTwo(). (The code left unchanged is not shown anymore.)

```
@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");
        }
        ...
}
```

Now, if you run the example, the expected output will be printed again.

```
Property fooOne set
Property fooTwo set
Property bar set
```

When using Java configuration, the only thing that changes is the way the beans are defined. Because instead of @Component on bean classes, @Bean annotations will be used on bean declaration methods in the configuration class. Such an example is shown here:

```
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();
        }
}
```

The existing classes from the `com.apress.prospring5.ch3.sandbox` package were reused here too, to avoid code duplication, because as component scanning is not enabled, any bean declaration using stereotype annotation will be ignored. If you run the previous class, you will notice that the same output from the previous example is printed. If you remember, as mentioned earlier, the convention when using bean declarations with `@Bean` is that the name of the method becomes the name of the bean, so `TrickyTarget` configured with the `@Qualifier` annotation will still work as expected.

When to Use Autowiring

In most cases, the answer to the question of whether you should use autowiring is definitely no! Autowiring can save you time in small applications, but in many cases, it leads to bad practices and is inflexible in large applications. Using `byName` seems like a good idea, but it may lead you to give your classes artificial property names so that you can take advantage of the autowiring functionality. The whole idea behind Spring is that you can create your classes as you like and have Spring work for you, not the other way around. You may be tempted to use `byType` until you realize that you can have only one bean for each type in your `ApplicationContext`—a restriction that is problematic when you need to maintain beans with different configurations of the same type. The same argument applies to the use of constructor autowiring.

In some cases, autowiring can save you time, but it does not really take that much extra effort to define your wiring explicitly, and you benefit from explicit semantics and full flexibility on property naming and on how many instances of the same type you manage. For any nontrivial application, steer clear of autowiring at all costs.

Setting Bean Inheritance

In some cases, you many need multiple definitions of beans that are the same type or implement a shared interface. This can become problematic if you want these beans to share some configuration settings but not others. The process of keeping the shared configuration settings in sync is quite error-prone, and on large projects, doing so can be quite time-consuming. To get around this, Spring allows you to provide a `<bean>` definition that inherits its property settings from another bean in the same `ApplicationContext` instance. You can override the values of any properties on the child bean as required, which allows you to have full control, but the parent bean can provide each of your beans with a base configuration. The following code sample shows a simple configuration with two beans, one of which is the child of the other (`app-context-xml.xml`):

```
<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>
```

In this code, you can see that the `<bean>` tag for the `child` bean has an extra attribute, `parent`, which indicates that Spring should consider the `parent` bean the parent of the bean and inherit its configuration. In case you don't want a parent bean definition to become available for lookup from `ApplicationContext`, you can add the attribute `abstract="true"` in the `<bean>` tag when declaring the parent bean. Because the `child` bean has its own value for the `age` property, Spring passes this value to the bean. However, `child` has no value for the `name` property, so Spring uses the value given to the `inheritParent` bean.

The `Singer` bean is quite simple.

```
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;
    }
}
```

To test it, you could write a simple class like this:

```
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);
        }
}
```

As you can see, the `main()` method of the `Singer` class grabs both the `child` and `parent` beans from `ApplicationContext` and writes the contents of their properties to `stdout`. Here is the output from this example:

```
Parent:
        Name: John Mayer
        Age: 39
Child:
        Name: John Mayer
        Age: 0
```

As expected, the `inheritChild` bean inherited the value for its `name` property from the `inheritParent` bean but was able to provide its own value for the age property.

Child beans inherit both constructor arguments and property values from the parent beans, so you can use both styles of injection with bean inheritance. This level of flexibility makes bean inheritance a powerful tool for building applications with more than a handful of bean definitions. If you are declaring a lot of beans of the same value with shared property values, avoid the temptation to use copy and paste to share the values; instead, set up an inheritance hierarchy in your configuration.

When you are using inheritance, remember that bean inheritance does not have to match a Java inheritance hierarchy. It is perfectly acceptable to use bean inheritance on five beans of the same type. Think of bean inheritance as more like a templating feature than an inheritance feature. Be aware, however, that if you are changing the type of the child bean, that type must extend the parent bean type.

Summary

In this chapter, we covered a lot of ground with both Spring Core and IoC in general. We showed you examples of the types of IoC and presented the pros and cons of using each mechanism in your applications. We looked at which IoC mechanisms Spring provides and when (and when not) to use each within your applications. While exploring IoC, we introduced the Spring `BeanFactory`, which is the core component for Spring's IoC capabilities, and then `ApplicationContext`, which extends `BeanFactory` and provides additional functionalities. For `ApplicationContext`, we focused on `GenericXmlApplicationContext`, which allows external configuration of Spring by using XML. Another method to declare DI requirements for `ApplicationContext`, that is, using Java annotations, was also discussed. Some examples with `AnnotationConfigApplicationContext` and Java configuration were also included, just to slowly introduce this way of configuring beans.

This chapter also introduced you to the basics of Spring's IoC feature set including setter injection, constructor injection, Method Injection, autowiring, and bean inheritance. In the discussion of configuration, we demonstrated how you can configure your bean properties with a wide variety of values, including other beans, using both XML and annotation type configurations and `GenericXmlApplicationContext`.

This chapter only scratched the surface of Spring and Spring's IoC container. In the next chapter, you'll look at some IoC-related features specific to Spring, and you'll take a more detailed look at other functionality available in Spring Core.

CHAPTER 4

Spring Configuration in Detail and Spring Boot

In the previous chapter, we presented a detailed look at the concept of inversion of control (IoC) and how it fits into the Spring Framework. However, we have really only scratched the surface of what Spring Core can do. Spring provides a wide array of services that supplement and extend its basic IoC capabilities. In this chapter, you are going to explore these in detail. Specifically, you will be looking at the following:

- *Managing the bean life cycle*: So far, all the beans you have seen have been fairly simple and completely decoupled from the Spring container. In this chapter, we present some strategies you can employ to enable your beans to receive notifications from the Spring container at various points throughout their life cycles. You can do this either by implementing specific interfaces laid out by Spring, by specifying methods that Spring can call via reflection, or by using JSR-250 JavaBeans life-cycle annotations.
- *Making your beans "Spring aware"*: In some cases, you want a bean to be able to interact with the ApplicationContext instance that configured it. For this reason, Spring offers two interfaces, BeanNameAware and ApplicationContextAware (introduced at the end of Chapter 3), that allow your bean to obtain its assigned name and reference its ApplicationContext, respectively. This section of the chapter covers implementing these interfaces and gives some practical considerations for using them in your application.
- *Using FactoryBeans*: As its name implies, the FactoryBean interface is meant to be implemented by any bean that acts as a factory for other beans. The FactoryBean interface provides a mechanism by which you can easily integrate your own factories with the Spring BeanFactory interface.
- *Working with JavaBeans PropertyEditors*: The PropertyEditor interface is a standard interface provided in the java.beans package. PropertyEditors are used to convert property values to and from String representations. Spring uses PropertyEditors extensively, mainly to read values specified in the BeanFactory configuration and convert them into the correct types. In this chapter, we discuss the set of PropertyEditors supplied with Spring and how you can use them within your application. We also take a look at implementing custom PropertyEditors.

© Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho 2017
I. Cosmina et al., *Pro Spring 5*, https://doi.org/10.1007/978-1-4842-2808-1_4

- *Learning more about the Spring ApplicationContext*: As we know, `ApplicationContext` is an extension of `BeanFactory` intended for use in full applications. The `ApplicationContext` interface provides a useful set of additional functionality, including internationalized message support, resource loading, and event publishing. In this chapter, we present a detailed look at the features in addition to IoC that `ApplicationContext` offers. We also jump ahead of ourselves a little to show you how `ApplicationContext` simplifies the use of Spring when you are building web applications.
- *Using Java classes for configuration*: Prior to 3.0, Spring supported only the XML base configuration with annotations for beans and dependency configuration. Starting with 3.0, Spring offers another option for developers to configure the Spring `ApplicationContext` interface using Java classes. We take a look at this new option in Spring application configuration.
- *Using Spring Boot*: Spring application configuration is made even more practical by using Spring Boot. This Spring project makes it easy to create stand-alone, production-grade, Spring-based applications that you can "just run."
- *Using configuration enhancements*: We present features that make application configuration easier, such as profile management, environment and property source abstraction, and so on. In this section, we cover those features and show how to use them to address specific configuration needs.
- *Using Groovy for configuration*: New to Spring 4.0 is the ability to configure bean definitions in the Groovy language, which can be used as an alternative or supplement to the existing XML and Java configuration methods.

Spring's Impact on Application Portability

Most of the features discussed in this chapter are specific to Spring and, in many cases, are not available in other IoC containers. Although many IoC containers offer life-cycle management functionality, they probably do so through a different set of interfaces than Spring. If the portability of your application between different IoC containers is truly important, you might want to avoid using some of the features that couple your application to Spring.

Remember, however, that by setting a constraint—meaning that your application is portable between IoC containers—you are losing out on the wealth of functionality Spring offers. Because you are likely to be making a strategic choice to use Spring, it makes sense that you use it to the best of its ability.

Be careful not to create a requirement for portability out of thin air. In many cases, the end users of your application do not care whether the application can run on three different IoC containers; they just want it to run. In our experience, it is often a mistake to try to build an application on the lowest common denominator of features available in your chosen technology. Doing so often sets your application at a disadvantage right from the get-go. However, if your application requires IoC container portability, do not see this as a drawback—it is a true requirement and, therefore, one your application should fulfill. In *Expert One-on-One: J2EE Development without EJB* (Wrox, 2004), Rod Johnson and Jürgen Höller describe these types of requirements as phantom requirements and provide a much more detailed discussion of them and how they can affect your project.

Although using these features may couple your application to the Spring Framework, in reality you are increasing the portability of your application in the wider scope. Consider that you are using a freely available, open source framework that has no particular vendor affiliation. An application built using Spring's IoC container runs anywhere Java runs. For Java enterprise applications, Spring opens up new possibilities for portability. Spring provides many of the same capabilities as JEE and also provides classes to

abstract and simplify many other aspects of JEE. In many cases, it is possible to build a web application using Spring that runs in a simple servlet container but with the same level of sophistication as an application targeted at a full-blown JEE application server. By coupling to Spring, you can increase your application's portability by replacing many features that either are vendor-specific or rely on vendor-specific configuration with equivalent features in Spring.

Bean Life-Cycle Management

An important part of any IoC container, Spring included, is that beans can be constructed in such a way that they receive notifications at certain points in their life cycle. This enables your beans to perform relevant processing at certain points throughout their life. In general, two life-cycle events are particularly relevant to a bean: post-initialization and pre-destruction.

In the context of Spring, the post-initialization event is raised as soon as Spring finishes setting all the property values on the bean and finishes any dependency checks that you configured it to perform. The pre-destruction event is fired just before Spring destroys the bean instance. However, for beans with prototype scope, the pre-destruction event will not be fired by Spring. The design of Spring is that the initialization life-cycle callback methods will be called on objects regardless of bean scope, while for beans with prototype scope, the destruction life-cycle callback methods will not be called. Spring provides three mechanisms a bean can use to hook into each of these events and perform some additional processing: interface-based, method-based, and annotation-based mechanisms.

Using the interface-based mechanism, your bean implements an interface specific to the type of notification it wants to receive, and Spring notifies the bean via a callback method defined in the interface. For the method-based mechanism, Spring allows you to specify, in your `ApplicationContext` configuration, the name of a method to call when the bean is initialized and the name of a method to call when the bean is destroyed. For the annotation mechanism, you can use JSR-250 annotations to specify the method that Spring should call after construction or before destruction.

In the case of both events, the mechanisms achieve exactly the same goal. The interface mechanism is used extensively throughout Spring so that you don't have to remember to specify the initialization or destruction each time you use one of Spring's components. However, in your own beans, you may be better served using the method-based or annotation mechanism because your beans do not need to implement any Spring-specific interfaces. Although we stated that portability often isn't as important a requirement as many books lead you to believe, this does not mean you should sacrifice portability when a perfectly good alternative exists. That said, if you are coupling your application to Spring in other ways, using the interface method allows you to specify the callback once and then forget about it. If you are defining a lot of beans of the same type that need to take advantage of the life-cycle notifications, then using the interface mechanism can avoid the need for specifying the life-cycle callback methods for every bean in the XML configuration file. Using JSR-250 annotations is also another viable option, since it's a standard defined by the JCP and you are also not coupled to Spring's specific annotations. Just make sure that the IoC container you are running your application on supports the JSR-250 standard.

Overall, the choice of which mechanism you use for receiving life-cycle notifications depends on your application requirements. If you are concerned about portability or you are just defining one or two beans of a particular type that need the callbacks, use the method-based mechanism. If you use annotation-type configuration and are certain that you are using an IoC container that supports JSR-250, use the annotation mechanism. If you are not too concerned about portability or you are defining many beans of the same type that need the life-cycle notifications, using the interface-based mechanism is the best way to ensure that your beans always receive the notifications they are expecting. If you plan to use a bean across many different Spring projects, you almost certainly want the functionality of that bean to be as self-contained as possible, so you should definitely use the interface-based mechanism.

Figure 4-1 shows a high-level overview of how Spring manages the life cycle of the beans within its container.

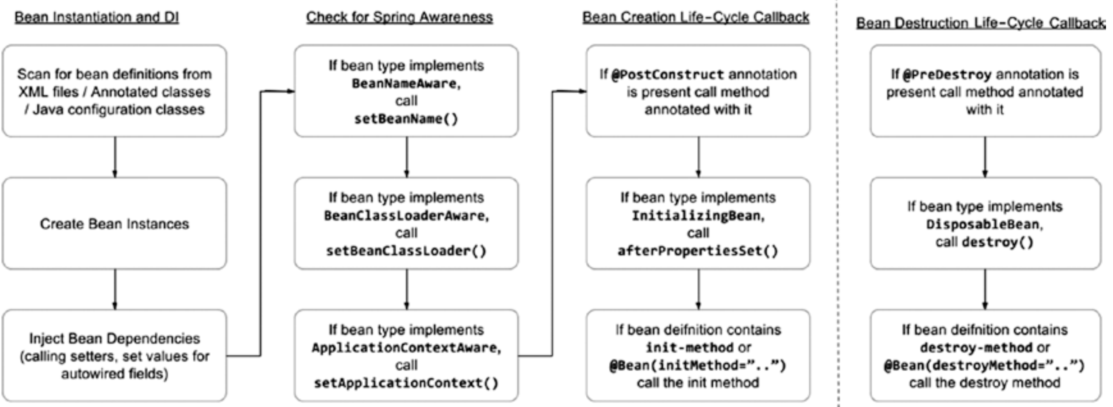

Figure 4-1. *Spring beans life cycle*

Hooking into Bean Creation

By being aware of when it is initialized, a bean can check whether all its required dependencies are satisfied. Although Spring can check dependencies for you, it is pretty much an all-or-nothing approach, and it doesn't offer any opportunities for applying additional logic to the dependency resolution procedure. Consider a bean that has four dependencies declared as setters, two of which are required and one of which has a suitable default in the event that no dependency is provided. Using an initialization callback, your bean can check for the dependencies it requires, throwing an exception or providing a default as needed.

A bean cannot perform these checks in its constructor because at this point, Spring has not had an opportunity to provide values for the dependencies it can satisfy. The initialization callback in Spring is called after Spring finishes providing the dependencies that it can and performs any dependency checks that you ask of it.

You are not limited to using the initialization callback just to check dependencies; you can do anything you want in the callback, but it is most useful for the purpose we have described. In many cases, the initialization callback is also the place to trigger any actions that your bean must take automatically in response to its configuration. For instance, if you build a bean to run scheduled tasks, the initialization callback provides the ideal place to start the scheduler—after all, the configuration data is set on the bean.

⚠ You will not have to write a bean to run scheduled tasks because this is something Spring can do automatically through its built-in scheduling feature or via integration with the Quartz scheduler. We cover this in more detail in Chapter 11.

Executing a Method When a Bean Is Created

As we mentioned previously, one way to receive the initialization callback is to designate a method on your bean as an initialization method and tell Spring to use this method as an initialization method. As discussed, this callback mechanism is useful when you have only a few beans of the same type or when you want to keep your application decoupled from Spring. Another reason for using this mechanism is to enable your

Spring application to work with beans that were built previously or were provided by third-party vendors. Specifying a callback method is simply a case of specifying the name in the `init-method` attribute of a bean's `<bean>` tag. The following code sample shows a basic bean with two dependencies:

```
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;
        }
    }
}
```

Notice that we have defined a method, `init()`, to act as the initialization callback. The `init()` method checks whether the name property has been set, and if it has not, it uses the default value stored in the `DEFAULT_NAME` constant. The `init()` method also checks whether the age property is set and throws `IllegalArgumentException` if it is not.

The `main()` method of the `SimpleBean` class attempts to obtain three beans from `GenericXmlApplicationContext`, all of type `Singer`, using its own `getBean()` method. Notice that in the `getBean()` method, if the bean is obtained successfully, its details are written to console output. If an exception is thrown in the `init()` method, as will occur in this case if the age property is not set, then Spring wraps that exception in `BeanCreationException`. The `getBean()` method catches these exceptions and writes a message to the console output informing us of the error, as well as returns a `null` value.

The following configuration snippet shows an `ApplicationContext` configuration that defines the beans used in the previous code snippet (`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"
    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>
```

As you can see, the `<bean>` tag for each of the three beans has an `init-method` attribute that tells Spring that it should invoke the `init()` method as soon as it finishes configuring the bean. The `singerOne` bean has values for both the `name` and `age` properties, so it passes through the `init()` method with absolutely no changes. The `singerTwo` bean has no value for the `name` property, meaning that in the `init()` method, the `name` property is given the default value. Finally, the `singerThree` bean has no value for the age property. The logic defined in the `init()` method treats this as an error, so `IllegalArgumentException` is thrown. Also note that in the `<beans>` tag, we added the attribute `default-lazy-init="true"` to instruct Spring to instantiate the beans defined in the configuration file only when the bean was requested from the application. If we do not specify it, Spring will try to initialize all the beans during the bootstrapping of `ApplicationContext`, and it will fail during the initialization of `singerThree`.

When all the beans in a configuration file have the same `init-method` configuration, the file can be simplified, by setting the `default-init-method` attribute on the `<beans>` element. The beans can be of different types; the only condition for them is to have a method named as the `default-init-method` attribute value. So, the previous configuration can be written like this as well:

```
<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>
```

Running the previous example yields the following output:

```
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
```

From this output, you can see that `singerOne` was configured correctly with the values that we specified in the configuration file. For `singerTwo`, the default value for the `name` property was used because no value was specified in the configuration. Finally, for `singerThree`, no bean instance was created because the `init()` method raised an error because of the lack of a value for the age property.

As you can see, using the initialization method is an ideal way to ensure that your beans are configured correctly. By using this mechanism, you can take full advantage of the benefits of IoC without losing any of the control you get from manually defining dependencies. The only constraint on your initialization method is that it cannot accept any arguments. You can define any return type, although it is ignored by Spring, and you can even use a static method, but the method must accept no arguments.

The benefits of this mechanism are negated when using a static initialization method, because you cannot access any of the bean's state to validate it. If your bean is using static state as a mechanism for saving memory and you are using a static initialization method to validate this state, then you should consider moving the static state to instance state and using a nonstatic initialization method. If you use Spring's singleton management capabilities, the end effect is the same, but you have a bean that is much simpler to test, and you also have the increased effect of being able to create multiple instances of the bean with their own state when necessary. Of course, in some instances, you need to use static state shared across multiple instances of a bean, in which case you can always use a static initialization method.

Implementing the InitializingBean Interface

The `InitializingBean` interface defined in Spring allows you to define inside your bean code that you want the bean to receive notification that Spring has finished configuring it. In the same way as when you are using an initialization method, this gives you the opportunity to check the bean configuration to ensure that it is valid, providing any default values along the way. The `InitializingBean` interface defines a single method, `afterPropertiesSet()`, that serves the same purpose as the `init()` method introduced in the previous section. The following code snippet shows a reimplementation of the previous example using the `InitializingBean` interface in place of the initialization method:

```
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;
        }
    }
}
```

As you can see, not much in this example has changed. Aside from the obvious class name change, the only differences are that this class implements `InitializingBean` and that the initialization logic has moved into the `afterPropertiesSet()` method. In the following snippet, you can see the configuration for this example (`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>
```

Again, there's not much difference between the configuration code introduced here and the configuration code in the previous section. The noticeable difference is the omission of the `init-method` attribute. Because the `SimpleBeanWithInterface` class implements the `InitializingBean` interface, Spring knows which method to call as the initialization callback, thus removing the need for any additional configuration. The output from this example is shown here:

```
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
```

Using the JSR-250 @PostConstruct Annotation

Another method that can achieve the same purpose is to use the JSR-250 life-cycle annotation, `@PostConstruct`. Starting from Spring 2.5, JSR-250 annotations are also supported to specify the method that Spring should call if the corresponding annotation relating to the bean's life cycle exists in the class. The following code sample shows the program with the `@PostConstruct` annotation applied:

```
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;
        }
    }
}
```

The program is the same as using the `init-method` approach; just apply the `@PostConstruct` annotation before the `init()` method. Note that you can assign any name to the method. In terms of configuration, since we are using annotations, we need to add the `<context:annotation-driven>` tag from the context namespace into the configuration file.

```
<?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>
```

Run the program and you will see the same output as other mechanisms.

```
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
```

All three approaches have their benefits and drawbacks. Using an initialization method, you have the benefit of keeping your application decoupled from Spring, but you have to remember to configure the initialization method for every bean that needs it. Using the `InitializingBean` interface, you have the benefit of being able to specify the initialization callback once for all instances of your bean class, but you have to couple your application to do so. Using annotations, you need to apply the annotation to the method and make sure that the IoC container supports JSR-250. In the end, you should let the requirements of your application drive the decision about which approach to use. If portability is an issue, use the initialization or annotation method; otherwise, use the `InitializingBean` interface to reduce the amount of configuration your application needs and the chance of errors creeping into your application because of misconfiguration.

> **Warning** When configuring initialization with `init-method` or `@PostConstruct`, there is the advantage of declaring the initialization method with a different access right. Initialization methods should be called only once by the Spring IoC, at bean creation time. Subsequent calls will lead to unexpected results or even failures. External additional calls can be prohibited by making the initialization method `private`. The Spring IoC will be able to call it via reflection, but any additional calls in the code won't be permitted.

Declaring an Initialization Method Using @Bean

Another way to declare the initialization method for a bean is to specify the `initMethod` attribute for the `@Bean` annotation and set the initialization method name as its value. This annotation is used to declare beans in Java configuration classes. Although Java configuration is covered a bit later in this chapter, the bean initialization part belongs here. For this example, the initial `Singer` class is used because the configuration is external, just like using the `init-method` attribute. We'll just write a configuration class and a new `main()` method to test it. Also, `default-lazy-init="true"` will be replaced by the `@Lazy` annotation on each bean declaration.

```
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();
        }

}
```

Running this code will produce the same result observed so far, shown here:

```
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
```

Understanding Order of Resolution

All initialization mechanisms can be used on the same bean instance. In this case, Spring invokes the method annotated with `@PostConstruct` first and then `afterPropertiesSet()`, followed by the initialization method specified in the configuration file. There is a technical reason for this order, and by following the path in Figure 4-1, we can notice the following steps in the bean creation process:

1. The constructor is called first to create the bean.
2. The dependencies are injected (setters are called).
3. Now that the beans exist and the dependencies were provided, the pre-initialization `BeanPostProcessor` infrastructure beans are consulted to see whether they want to call anything from this bean. These are Spring-specific infrastructure beans that perform bean modifications after they are created. The `@PostConstruct` annotation is registered by

CommonAnnotationBeanPostProcessor, so this bean will call the method found annotated with @PostConstruct. This method is executed right after the bean has been constructed and before the class is put into service,[1] before the actual initialization of the bean (before afterPropertiesSet and init-method).

4. The InitializingBean's afterPropertiesSet is executed right after the dependencies are injected. The afterPropertiesSet() method is invoked by a BeanFactory after it has set all the bean properties supplied and has satisfied BeanFactoryAware and ApplicationContextAware.

5. The init-method attribute is executed last because this is the actual initialization method of the bean.

Understanding the order of different types of bean initialization can be useful if you have an existing bean that performs some initialization in a specific method but you need to add some more initialization code when you use Spring.

Hooking into Bean Destruction

When using an ApplicationContext implementation that wraps the DefaultListableBeanFactory interface (such as GenericXmlApplicationContext, via the getDefaultListableBeanFactory() method), you can signal to BeanFactory that you want to destroy all singleton instances with a call to ConfigurableBeanFactory.destroySingletons(). Typically, you do this when your application shuts down, and it allows you to clean up any resources that your beans might be holding open, thus allowing your application to shut down gracefully. This callback also provides the perfect place to flush any data you are storing in memory to persistent storage and to allow your beans to end any long-running processes they may have started.

To allow your beans to receive notification that destroySingletons() has been called, you have three options, all similar to the mechanisms available for receiving an initialization callback. The destruction callback is often used in conjunction with the initialization callback. In many cases, you create and configure a resource in the initialization callback and then release the resource in the destruction callback.

Executing a Method When a Bean Is Destroyed

To designate a method to be called when a bean is destroyed, you simply specify the name of the method in the destroy-method attribute of the bean's <bean> tag. Spring calls it just before it destroys the singleton instance of the bean (Spring will not call this method for those beans with prototype scope). The following code snippet provides an example of using a destroy-method callback:

```
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");
```

[1]Check out this snippet from JEE official Javadoc: http://docs.oracle.com/javaee/7/api/javax/annotation/PostConstruct.html.

```
        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()");
    }
}
```

This code defines a `destroy()` method, in which the file that was created gets deleted. The `main()` method retrieves a bean of type `DestructiveBean` from `GenericXmlApplicationContext` and then invokes its `destroy()` method (which will, in turn, invoke the `ConfigurableBeanFactory.destroySingletons()` that was wrapped by the `ApplicationContext`), instructing Spring to destroy all the singletons managed by it. Both the initialization and destruction callbacks write a message to console output informing us that they have been called. In the following snippet, you can see the configuration for the `destructiveBean` bean (`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>
```

Notice that we have specified the `destroy()` method as the destruction callback by using the `destroy-method` attribute. The filePath attribute value is built by using an SpEL expression, concatenating the system properties `java.io.tmpdir` and `file.separator` before the file name `test.txt` to ensure cross-platform compatibility. Running this example yields the following output:

```
Initializing Bean
File exists: true
Calling destroy()
Destroying Bean
File exists: false
Called destroy()
```

As you can see, Spring first invokes the initialization callback, and the `DestructiveBean` instance creates the `File` instance and stores it. Next, during the call to `destroy()`, Spring iterates over the set of singletons it is managing, in this case just one, and invokes any destruction callbacks that are specified. This is where the `DestructiveBean` instance deletes the created file and logs messages to the screen indicating it no longer exists.

Implementing the DisposableBean Interface

As with initialization callbacks, Spring provides an interface, in this case `DisposableBean`, that can be implemented by your beans as a mechanism for receiving destruction callbacks. The `DisposableBean` interface defines a single method, `destroy()`, which is called just before the bean is destroyed. Using this mechanism is orthogonal to using the `InitializingBean` interface to receive initialization callbacks. The following code snippet shows a modified implementation of the `DestructiveBean` class that implements the `DisposableBean` interface:

```
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()");
    }
}
```

There is not much difference between the code that uses the callback method mechanism and the code that uses the callback interface mechanism. In this case, we even used the same method names. The configuration for this example is depicted here (`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>
```

Aside from the different class name, the only difference is the omission of the `destroy-method` attribute. Running this example yields the following output:

```
Initializing Bean
File exists: true
Calling destroy()
Destroying Bean
File exists: false
Called destroy()
```

Using the JSR-250 @PreDestroy Annotation

The third way to define a method to be called before a bean is destroyed is to use the JSR-250 life-cycle `@PreDestroy` annotation, which is the inverse of the `@PostConstruct` annotation. The following code snippet is a version of `DestructiveBean` that uses both `@PostConstruct` and `@PreDestroy` in the same class to perform program initialization and destroy actions:

```
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()");
    }
}
```

In the following snippet, you can see the configuration file for this bean, which makes use of the `<context:annotation-config>` tag (`app-context-annotation.xml`).

```
<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>
```

Declaring a Destroy Method Using @Bean

Another way to declare the destroy method for a bean is to specify the `destroyMethod` attribute for the `@Bean` annotation and set the destroy method name as its value. This annotation is used to declare beans in Java configuration classes. Although Java configuration is covered a bit later in this chapter, the bean destruction part belongs here. For this example, the initial `DestructiveBeanWithJSR250` class is used, as the configuration is external, just like using the `destroy-method` attribute. We'll just write a configuration class and a new `main()` method to test it. Also, `default-lazy-init="true"` will be replaced by the `@Lazy` annotation on each bean declaration.

```
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;

/**
 * Created by  iuliana.cosmina on  2/27/17.
 */
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()");
        }
}
```

The @PostConstruct annotation is used as well in the bean configuration; thus, running this code will produce the same result observed so far.

```
Initializing Bean
File exists: true
Calling destroy()
Destroying Bean
File exists: false
Called destroy()
```

The destruction callback is an ideal mechanism for ensuring that your applications shut down gracefully and do not leave resources open or in an inconsistent state. However, you still have to decide whether to use the destruction method callback, the DisposableBean interface, the @PreDestroy annotation, the XML destroy-attribute attribute, or the destroyMethod. Again, let the requirements of your application drive your decision in this respect; use the method callback where portability is an issue, and use the DisposableBean interface or a JSR-250 annotation to reduce the amount of configuration required.

Understanding Order of Resolution

As with the case of bean creation, you can use all mechanisms on the same bean instance for bean destruction. In this case, Spring invokes the method annotated with `@PreDestroy` first and then `DisposableBean.destroy()`, followed by your destroy method configured in your XML definition.

Using a Shutdown Hook

The only drawback of the destruction callbacks in Spring is that they are not fired automatically; you need to remember to call `AbstractApplicationContext.destroy()` before your application is closed. When your application runs as a servlet, you can simply call `destroy()` in the servlet's `destroy()` method. However, in a stand-alone application, things are not quite so simple, especially if you have multiple exit points out of your application. Fortunately, there is a solution. Java allows you to create a *shutdown hook*, which is a thread that is executed just before the application shuts down. This is the perfect way to invoke the `destroy()` method of your `AbstractApplicationContext` (which was being extended by all concrete `ApplicationContext` implementations). The easiest way to take advantage of this mechanism is to use `AbstractApplicationContext`'s `registerShutdownHook()` method. The method automatically instructs Spring to register a shutdown hook of the underlying JVM runtime. The bean declaration and configuration stay the same as before; the only thing that changes is the main method: the call of `ctx.registerShutdownHook` is added, and calls to `ctx.destroy()` or `close()` will be removed.

```
...
public class DestructiveBeanWithHook {

   public static void main(String... args) {
        GenericApplicationContext ctx =
           new AnnotationConfigApplicationContext(
               DestructiveBeanConfig.class);

           ctx.getBean(DestructiveBeanWithJSR250.class);
           ctx.registerShutdownHook();
   }
}
```

Running this code will produce the same result observed so far.

```
Initializing Bean
File exists: true
Destroying Bean
File exists: false
```

As you can see, the `destroy()` method is invoked, even though we didn't write any code to invoke it explicitly as the application was shutting down.

Making Your Beans "Spring Aware"

One of the biggest selling points of dependency injection over dependency lookup as a mechanism for achieving inversion of control is that your beans do not need to be aware of the implementation of the container that is managing them. To a bean that uses constructor or setter injection, the Spring container is the same as the container provided by Google Guice or PicoContainer. However, in certain circumstances,

you may need a bean that is using dependency injection to obtain its dependencies so it can interact with the container for some other reason. An example of this may be a bean that automatically configures a shutdown hook for you, and thus it needs access to ApplicationContext. In other cases, a bean may want to know what its name is (that is, the bean name that was assigned within the current ApplicationContext) so it can perform some additional processing based on this name.

That said, this feature is really intended for internal Spring use. Giving the bean name some kind of business meaning is generally a bad idea and can lead to configuration problems as bean names have to be artificially manipulated to support their business meaning. However, we have found that being able to have a bean find out its name at runtime is really useful for logging. Say you have many beans of the same type running under different configurations. The bean name can be included in log messages to help you differentiate between the one that is generating errors and the ones that are working fine when something goes wrong.

Using the BeanNameAware Interface

The BeanNameAware interface, which can be implemented by a bean that wants to obtain its own name, has a single method: setBeanName(String). Spring calls the setBeanName() method after it has finished configuring your bean but before any life-cycle callbacks (initialization or destroy) are called (refer to Figure 4-1). In most cases, the implementation of the setBeanName() interface is just a single line that stores the value passed in by the container in a field for use later. The following code snippet shows a simple bean that obtains its name by using BeanNameAware and then later uses this bean name to print to the console:

```
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()");
    }
}
```

This implementation is fairly trivial. Remember that BeanNameAware.setBeanName() is called before the first instance of the bean is returned to your application via a call to ApplicationContext.getBean(), so there is no need to check whether the bean name is available in the sing() method. Here you can see the configuration contained in the app-context-xml.xml file used in this example:

```
<beans ...>
    <bean id="johnMayer"
          class="com.apress.prospring5.ch4.NamedSinger"/>
</beans>
```

As you can see, no special configuration is required to take advantage of the BeanNameAware interface. In the following code snippet, you can see a simple example application that retrieves the Singer instance from ApplicationContext and then calls the sing() method:

```
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();
    }
}
```

This example generates the following log output; notice the inclusion of the bean name in the log message for the call to sing():

```
Singer johnMayer - sing()
```

Using the BeanNameAware interface is really quite simple, and it is put to good use when you are improving the quality of your log messages. Avoid being tempted to give your bean names business meaning just because you can access them; by doing so, you are coupling your classes to Spring for a feature that brings negligible benefit. If your beans need some kind of name internally, have them implement an interface such as Nameable (which is specific to your application) with a method setName() and then give each bean a name by using dependency injection. This way, you can keep the names you use for configuration concise, and you won't need to manipulate your configuration unnecessarily to give your beans names with business meaning.

Using the ApplicationContextAware Interface

ApplicationContextAware was introduced at the end of Chapter 3 to show how Spring can be used to deal with beans that require other beans to function that are not injected using constructors or setters in the configuration (the depends-on example).

Using the ApplicationContextAware interface, it is possible for your beans to get a reference to the ApplicationContext instance that configured them. The main reason this interface was created is to allow a bean to access Spring's ApplicationContext in your application, for example, to acquire other Spring beans programmatically, using getBean(). You should, however, avoid this practice and use dependency injection to provide your beans with their collaborators. If you use the lookup-based getBean() approach to obtain dependencies when you can use dependency injection, you are adding unnecessary complexity to your beans and coupling them to the Spring Framework without good reason.

Of course, ApplicationContext isn't used just to look up beans; it performs a great many other tasks. As you saw previously, one of these tasks is to destroy all singletons, notifying each of them in turn before doing so. In the previous section, you saw how to create a shutdown hook to ensure that ApplicationContext is instructed to destroy all singletons before the application shuts down. By using the ApplicationContextAware interface, you can build a bean that can be configured in ApplicationContext to create and configure a shutdown hook bean automatically. The following configuration shows the code for this bean:

```
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();
        }
    }
}
```

Most of this code should seem familiar to you by now. The ApplicationContextAware interface defines a single method, setApplicationContext(ApplicationContext), that Spring calls to pass your bean a reference to its ApplicationContext. In the previous code snippet, the ShutdownHookBean class checks whether ApplicationContext is of type GenericApplicationContext, meaning it supports the registerShutdownHook() method; if it does, it will register a shutdown hook to ApplicationContext. The following configuration snippet shows how to configure this bean to work with the DestructiveBeanWithInterface bean (app-context-annotation.xml):

```
<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>
```

Notice that no special configuration is required. The following code snippet shows a simple example application that uses ShutdownHookBean to manage the destruction of singleton 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);
    }
}
```

This code should seem quite familiar to you. When Spring bootstraps ApplicationContext and destructiveBean is defined in the configuration, Spring passes the reference of ApplicationContext to the shutdownHook bean for registering the shutdown hook. Running this example yields the following output, as expected:

```
Initializing Bean
File exists: true
Destroying Bean
File exists: false
```

As you can see, even though no calls to destroy() are in the main application, ShutdownHookBean is registered as a shutdown hook, and it calls destroy() just before the application shuts down.

Use of FactoryBeans

One of the problems that you will face when using Spring is how to create and then inject dependencies that cannot be created simply by using the new operator. To overcome this problem, Spring provides the FactoryBean interface that acts as an adapter for objects that cannot be created and managed using the standard Spring semantics. Typically, you use FactoryBeans to create beans that you cannot create by using the new operator, such as those you access through static factory methods, although this is not always the case. Simply put, a FactoryBean is a bean that acts as a factory for other beans. FactoryBeans are configured within your ApplicationContext like any normal bean, but when Spring uses the FactoryBean interface to satisfy a dependency or lookup request, it does not return FactoryBean; instead, it invokes the FactoryBean.getObject() method and returns the result of that invocation.

FactoryBeans are used to great effect in Spring; the most noticeable uses are the creation of transactional proxies, which we cover in Chapter 9, and the automatic retrieval of resources from a JNDI context. However, FactoryBeans are useful not just for building the internals of Spring; you'll find them really useful when you build your own applications because they allow you to manage many more resources by using IoC than would otherwise be available.

FactoryBean Example: The MessageDigestFactoryBean

Often the projects that we work on require some kind of cryptographic processing; typically, this involves generating a message digest or hash of a user's password to be stored in a database. In Java, the MessageDigest class provides functionality for creating a digest of any arbitrary data. MessageDigest itself is abstract, and you obtain concrete implementations by calling MessageDigest.getInstance() and passing in the name of the digest algorithm you want to use. For instance, if we want to use the MD5 algorithm to create a digest, we use the following code to create the MessageDigest instance:

```
MessageDigest md5 = MessageDigest.getInstance("MD5");
```

If we want to use Spring to manage the creation of the MessageDigest object, the best we can do without a FactoryBean is have a property, algorithmName, on your bean and then use an initialization callback to call MessageDigest.getInstance(). Using a FactoryBean, we can encapsulate this logic inside a bean. Then any beans that require a MessageDigest instance can simply declare a property, messageDigest, and use the FactoryBean to obtain the instance. The following code snippet shows an implementation of FactoryBean that does just this:

```
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;
    }
}
```

Spring calls the getObject() method to retrieve the object created by the FactoryBean. This is the actual object that is passed to other beans that use the FactoryBean as a collaborator. In the code snippet, you can see that MessageDigestFactoryBean passes a clone of the stored MessageDigest instance that is created in the InitializingBean.afterPropertiesSet() callback.

The getObjectType() method allows you to tell Spring what type of object your FactoryBean will return. This can be null if the return type is unknown in advance (for example, the FactoryBean creates different types of objects depending on the configuration, which will be determined only after the FactoryBean is initialized), but if you specify a type, Spring can use it for autowiring purposes. We return MessageDigest as our type (in this case, a class, but try to return an interface type and have the FactoryBean instantiate the concrete implementation class, unless necessary). The reason is that we do not know what concrete type will be returned (not that it matters, because all beans will define their dependencies by using MessageDigest anyway).

The isSingleton() property allows you to inform Spring whether the FactoryBean is managing a singleton instance. Remember that by setting the singleton attribute of the FactoryBean's <bean> tag, you tell Spring about the singleton status of the FactoryBean itself, not the objects it is returning. Now let's see how the FactoryBean is employed in an application. In the following code snippet, you can see a simple bean that maintains two MessageDigest instances and then displays the digests of a message passed to its digest() method:

```
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);
    }
}
```

The following configuration snippet shows an example configuration for two MessageDigestFactoryBean classes, one for the SHA1 algorithm and the other using the default (MD5) algorithm (app-context-xml.xml):

```
<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>
```

As you can see, not only we have configured the two MessageDigestFactoryBean classes, but we have also configured a MessageDigester, using the two MessageDigestFactoryBean classes, to provide values for the digest1 and digest2 properties. For the defaultDigest bean, since the algorithmName property was not specified, no injection will happen, and the default algorithm (MD5) that was coded in the class will be used. In the following code sample, you see a basic example class that retrieves the MessageDigester bean from the BeanFactory and creates the digest of a simple message:

```
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();
    }
}
```

Running this example gives the following output:

```
Using digest1
Using alogrithm: SHA1
[B@130f889
Using digest2
Using alogrithm: MD5
[B@1188e820
```

As you can see, the MessageDigest bean is provided with two MessageDigest implementations, SHA1 and MD5, even though no MessageDigest beans are configured in the BeanFactory. This is the FactoryBean at work.

FactoryBeans are the perfect solution when you are working with classes that cannot be created by using the new operator. If you work with objects that are created by using a factory method and you want to use these classes in a Spring application, create a FactoryBean to act as an adapter, allowing your classes to take full advantage of Spring's IoC capabilities.

Using FactoryBeans is different when configuration via Java configuration is used, because in this case there is a restriction from the compiler to set the property with the proper type; thus, the getObject() method must be called explicitly. In the following code snippet, you can see an example of configuring the same beans as in the previous example, but using Java configuration:

```
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();
    }
}
```

If you run this class, the same output as before is printed.

Accessing a FactoryBean Directly

Given that Spring automatically satisfies any references to a FactoryBean by the objects produced by that FactoryBean, you may be wondering whether you can actually access the FactoryBean directly. The answer is yes.

Accessing FactoryBean is simple: you prefix the bean name with an ampersand in the call to getBean(), as shown in the following code sample:

```
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();
        }
}
```

Running this program generates the following output:

```
[B@130f889
```

This feature is used in a few places in the Spring code, but your application should really have no reason to use it. The intention of FactoryBean is to be used as a piece of supporting infrastructure to allow you to use more of your application's classes in an IoC setting. Avoid accessing FactoryBean directly and invoking its getObject() manually, and let Spring do it for you; if you do this manually, you are making extra work for yourself and are unnecessarily coupling your application to a specific implementation detail that could quite easily change in the future.

Using the factory-bean and factory-method Attributes

Sometimes you need to instantiate JavaBeans that were provided by a non-Spring-powered third-party application. You don't know how to instantiate that class, but you know that the third-party application provides a class that can be used to get an instance of the JavaBean that your Spring application needs. In this case, Spring bean's `factory-bean` and `factory-method` attributes in the `<bean>` tag can be used.

To take a look at how it works, the following code snippet shows another version of the `MessageDigestFactory` that provides a method to return a `MessageDigest` bean:

```
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;
    }
}
```

The following configuration snippet shows how to configure the factory method for getting the corresponding `MessageDigest` bean instance (`app-context-xml.xml`):

```
<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>
```

Notice that two digest factory beans were defined, one using SHA1 and the other using the default algorithm. Then for the beans `shaDigest` and `defaultDigest`, we instructed Spring to instantiate the beans by using the corresponding message digest factory bean via the `factory-bean` attribute, and we specified the method to use to obtain the bean instance via the `factory-method` attribute. The following code snippet depicts the testing class:

```
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();
    }
}
```

Running the program generates the following output:

```
Using digest1
Using alogrithm: SHA1
[B@77a57272
Using digest2
Using alogrithm: MD5
[B@7181ae3f
```

JavaBeans PropertyEditors

If you are not entirely familiar with JavaBeans concepts, a `PropertyEditor` is an interface that converts a property's value to and from its native type representation into a `String`. Originally, this was conceived as a way to allow property values to be entered, as `String` values, into an editor and have them transformed into the correct type. However, because `PropertyEditors` are inherently lightweight classes, they have found uses in many settings, including Spring.

Because a good portion of property values in a Spring-based application start life in the `BeanFactory` configuration file, they are essentially `Strings`. However, the property that these values are set on may not be `String`-typed. So, to save you from having to create a load of `String`-typed properties artificially, Spring allows you to define `PropertyEditors` to manage the conversion of `String`-based property values into the correct types. Figure 4-2 shows the full list of `PropertyEditors` that are part of the `spring-beans` package; you can see this list with any smart Java editor.

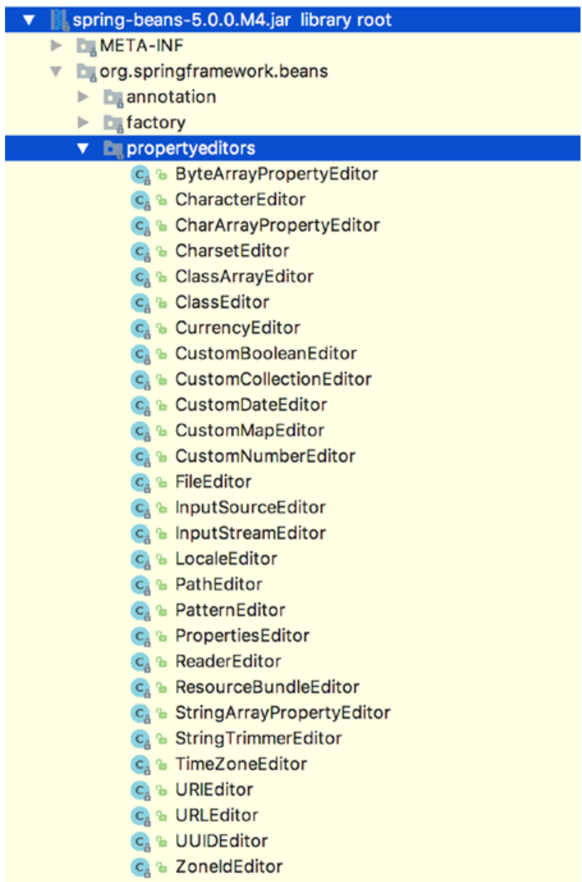

Figure 4-2. *Spring PropertyEditors*

They all extend `java.beans.PropertyEditorSupport` and can be used for implicit conversion of `String` literals into property values to be injected in beans; thus, they are preregistered with `BeanFactory`.

Using the Built-in PropertyEditors

The following code snippet shows a simple bean that declares 14 properties, one for each of the types supported by the built-in `PropertyEditor` implementations:

```
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();
    }
}
```

In the following configuration sample, you can see the configuration used to declare a bean of type PropertyEditorBean with values being specified for all the previous properties (app-config-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: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>
```

As you can see, although all the properties on the `PropertyEditorBean` are not `Strings`, the values for the properties are specified as simple `Strings`. Also note that we registered the `CustomDateEditor` and `StringTrimmerEditor`, since those two editors were not registered by default in Spring. Running this example yields the following output:

```
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/
```

As you can see, Spring has, using the built-in `PropertyEditors`, converted the `String` representations of the various properties to the correct types. Table 4-1 lists the most important built-in `PropertyEditors` available in Spring.

Table 4-1. *Spring PropertyEditors*

PropertyEditor	Description
ByteArrayPropertyEditor	Converts String values to their corresponding byte representations.
CharacterEditor	Populates a property of type Character or char from a String value.
ClassEditor	Converts from a fully qualified class name into a Class instance. When using this PropertyEditor, be careful not to include any extraneous spaces on either side of the class name when using GenericXmlApplicationContext because this results in a ClassNotFoundException.
CustomBooleanEditor	Converts a string into a Java Boolean type.
CustomCollectionEditor	Converts a source collection (e.g., represented by the util namespace in Spring) into the target Collection type.
CustomDateEditor	Converts a string representation of a date into a java.util.Date value. You need to register the CustomDateEditor implementation in Spring's ApplicationContext with the desired date format.
FileEditor	Converts a String file path into a File instance. Spring does not check whether the file exists.
InputStreamEditor	Converts a string representation of a resource (e.g., file resource using file:D:/temp/test.txt or classpath:test.txt) into an input stream property.
LocaleEditor	Converts the String representation of a locale, such as en-GB, into a java.util.Locale instance.
PatternEditor	Converts a string into the JDK Pattern object or the other way around.
PropertiesEditor	Converts a String in the format key1=value1 key2=value2 keyn=valuen into an instance of java.util.Properties with the corresponding properties configured.
StringTrimmerEditor	Performs trimming on the string values before injection. You need to explicitly register this editor.
URLEditor	Converts a String representation of a URL into an instance of java.net.URL.

This set of PropertyEditors provides a good base for working with Spring and makes configuring your application with common components such as files and URLs much simpler.

Creating a Custom PropertyEditor

Although the built-in PropertyEditors cover some of the standard cases of property type conversion, there may come a time when you need to create your own PropertyEditor to support a class or a set of classes you are using in your application. Spring has full support for registering custom PropertyEditors; the only downside is that the java.beans.PropertyEditor interface has a lot of methods, many of which are irrelevant to the task at hand, which is converting property types. Thankfully, JDK 5 or newer provides the PropertyEditorSupport class, which your own PropertyEditors can extend, leaving you to implement only a single method: setAsText(). Let's consider a simple example to see implementing a custom property editor in action. Suppose we have a FullName class with just two properties, firstName and lastName, defined as shown here:

```
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;
    }
}
```

To simplify the application configuration, let's develop a custom editor that converts a string with a space separator into the FullName class's first name and last name, respectively. The following code snippet depicts the custom property editor implementation:

```
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]));
    }
}
```

The editor is simple. It extends JDK's PropertyEditorSupport class and implements the setAsText() method. In the method, we simply split the String into a string array with a space as the delimiter. Afterward, an instance of the FullName class is instantiated, passing in the String before the space character as the first name and passing the String after the space character as the last name. Finally, the converted value is returned by calling the setValue() method with the result. To use NamePropertyEditor in your application, we need to register the editor in Spring's ApplicationContext. The following configuration sample shows an ApplicationContext configuration of a CustomEditorConfigurer and the NamePropertyEditor (app-context-02.xml):

```
<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>
```

You should notice two things in this configuration. First, custom PropertyEditors get injected into the CustomEditorConfigurer class by using the Map-typed customEditors property. Second, each entry in the Map represents a single PropertyEditor, with the key of the entry being the name of the class for which the PropertyEditor is used. As you can see, the key for NamePropertyEditor is com.apress.prospring4.ch4.FullName, which signifies that this is the class for which the editor should be used. The following code snippet shows the code for the CustomEditorExample class that is registered as a bean in the previous configuration:

```
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();
    }
}
```

The previous code is nothing special. Run the example, and you will see the following output:

```
First name: John - Last name: Mayer
```

This is the output from the `toString()` method we implemented in the `FullName` class, and you can see that the first name and last name of the `FullName` object were correctly populated by Spring by using the configured `NamePropertyEditor`. Starting from version 3, Spring introduced the Type Conversion API and the Field Formatting Service Provider Interface (SPI), which provide a simpler and well-structured API to perform type conversion and field formatting. It's especially useful for web application development. Both the Type Conversion API and the Field Formatting SPI are discussed in detail in Chapter 10.

More Spring ApplicationContext Configuration

So far, although we are discussing Spring's `ApplicationContext`, most of the features that we have covered mainly surround the `BeanFactory` interface wrapped by `ApplicationContext`. In Spring, various implementations of the `BeanFactory` interface are responsible for bean instantiation, providing dependency injection and life-cycle support for beans managed by Spring. However, as stated earlier, being an extension of the `BeanFactory` interface, `ApplicationContext` provides other useful functionalities as well. The main function of `ApplicationContext` is to provide a much richer framework on which to build your applications. `ApplicationContext` is much more aware of the beans (compared to `BeanFactory`) that you configure within it, and in the case of many of the Spring infrastructure classes and interfaces, such as `BeanFactoryPostProcessor`, it interacts with them on your behalf, reducing the amount of code you need to write in order to use Spring.

The biggest benefit of using `ApplicationContext` is that it allows you to configure and manage Spring and Spring-managed resources in a completely declarative way. This means that wherever possible, Spring provides support classes to load `ApplicationContext` into your application automatically, thus removing the need for you to write any code to access `ApplicationContext`. In practice, this feature is currently available only when you are building web applications with Spring, which allows you to initialize Spring's `ApplicationContext` in the web application deployment descriptor. When using a stand-alone application, you can also initialize Spring's `ApplicationContext` by simple coding.

In addition to providing a model that is focused more on declarative configuration, `ApplicationContext` supports the following features:

- Internationalization
- Event publication
- Resource management and access
- Additional life-cycle interfaces
- Improved automatic configuration of infrastructure components

In the following sections, we discuss some of the most important features in `ApplicationContext` besides DI.

Internationalization with the MessageSource

One area where Spring really excels is in support for internationalization (i18n). Using the MessageSource interface, your application can access String resources, called *messages*, stored in a variety of languages. For each language you want to support in your application, you maintain a list of messages that are keyed to correspond to messages in other languages. For instance, if you wanted to display "The quick brown fox jumped over the lazy dog" in English and in Czech, you would create two messages, both keyed as msg; the one for English would read "The quick brown fox jumped over the lazy dog," and the one for German would read "Der schnelle braune Fuchs sprang über den faulen Hund."

Although you don't need to use ApplicationContext to use MessageSource, the ApplicationContext interface extends MessageSource and provides special support for loading messages and for making them available in your environment. The automatic loading of messages is available in any environment, but automatic access is provided only in certain Spring-managed scenarios, such as when you are using Spring's MVC framework to build a web application. Although any class can implement ApplicationContextAware and thus access the automatically loaded messages, we suggest a better solution later in this chapter, in the section "Using MessageSource in Stand-Alone Applications."

Before continuing, if you are unfamiliar with i18n support in Java, we suggest you at least check out the Javadocs (http://download.java.net/jdk8/docs/api/index.html).

Internationalization with the MessageSource

Aside from ApplicationContext, Spring provides three MessageSource implementations.

- ResourceBundleMessageSource
- ReloadableResourceBundleMessageSource
- StaticMessageSource

The StaticMessageSource implementation should not be used in a production application because you can't configure it externally, and this is generally one of the main requirements when you are adding i18n capabilities to your application.

ResourceBundleMessageSource loads messages by using a Java ResourceBundle. ReloadableResourceBundleMessageSource is essentially the same, except it supports scheduled reloading of the underlying source files.

All three MessageSource implementations also implement another interface called HierarchicalMessageSource, which allows for many MessageSource instances to be nested. This is key to the way ApplicationContext works with MessageSource instances.

To take advantage of ApplicationContext's support for MessageSource, you must define a bean in your configuration of type MessageSource and with the name messageSource. ApplicationContext takes this MessageSource and nests it within itself, allowing you to access the messages by using ApplicationContext. This can be hard to visualize, so take a look at the following example. The following code sample shows a simple application that accesses a set of messages for both the English and German locales:

```
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();
    }
}
```

Don't worry about the calls to getMessage() just yet; we will return to those shortly. For now, just know that they retrieve a keyed message for the locale specified. In the following configuration snippet, you can see the configuration used by this application (app-context-xml.xml):

```
<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>
```

Here we define a ResourceBundleMessageSource bean with the name messageSource as required and configure it with a set of names to form the base of its file set. A Java ResourceBundle, which is used by ResourceBundleMessageSource, works on a set of properties files that are identified by base names. When looking for a message for a particular Locale, the ResourceBundle looks for a file that is named as a combination of the base name and the locale name. For instance, if the base name is foo and we are looking for a message in the en-GB (British English) locale, ResourceBundle looks for a file called foo_en_GB.properties.

For the previous example, the content of the properties files for English (labels_en.properties) and German (labels_de_DE.properties) is shown here:

```
#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}
```

Now this example just raises even more questions. What do those calls to getMessage() mean? Why did we use ApplicationContext.getMessage() rather than access the ResourceBundleMessageSource bean directly? We'll answer each of these questions in turn.

Using the getMessage() Method

The MessageSource interface defines three overloads for the getMessage() method. These are described in Table 4-2.

Table 4-2. *Overloads for MessageSource.getMessage()*

Method signature	Description
getMessage (String, Object[], Locale)	This is the standard getMessage() method. The String argument is the key of the message corresponding to the key in the properties file. In the previous code sample, the first call to getMessage() used msg as the key, and this corresponded to the following entry in the properties file for the en locale: msg=The quick brown fox jumped over the lazy dog. The Object[] array argument is used for replacements in the message. In the third call to getMessage(), we passed in an array of two Strings. The message keyed as nameMsg was My name is {0} {1}. The numbers in braces are placeholders, and each one is replaced with the corresponding entry in the argument array. The final argument, Locale, tells ResourceBundleMessageSource which properties file to look in. Even though the first and second calls to getMessage() in the example used the same key, they returned different messages that correspond to the Locale setting that was passed in to getMessage().
getMessage (String, Object[], String, Locale)	This overload works in the same way as getMessage(String, Object[], Locale), other than the second String argument, which allows we to pass in a default value in case a message for the supplied key is not available for the supplied Locale.
getMessage (MessageSourceResolvable, Locale)	This overload is a special case. We discuss it in further detail in the section "The MessageSourceResolvable Interface."

Why Use ApplicationContext As a MessageSource?

To answer this question, we need to jump a little ahead of ourselves and look at the web application support in Spring. The answer, in general, is that you shouldn't use ApplicationContext as a MessageSource because doing so couples your bean to ApplicationContext unnecessarily (this is discussed in more detail in the next section). You should use ApplicationContext when you are building a web application by using Spring's MVC framework.

The core interface in Spring MVC is Controller. Unlike frameworks such as Struts that require you to implement your controllers by inheriting from a concrete class, Spring simply requires that you implement the Controller interface (or annotate your controller class with the @Controller annotation). Having said that, Spring provides a collection of useful base classes that you will use to implement your own controllers. Each of these base classes is a subclass (directly or indirectly) of the ApplicationObjectSupport class, which is a convenient superclass for any application objects that want to be aware of ApplicationContext. Remember that in a web application setting, ApplicationContext is loaded automatically.

ApplicationObjectSupport accesses this ApplicationContext, wraps it in a MessageSourceAccessor object, and makes that available to your controller via the protected getMessageSourceAccessor() method. MessageSourceAccessor provides a wide array of convenient methods for working with MessageSource instances. Provides a wide array of convenient methods for working with MessageSource instances. This form of auto-injection is quite beneficial; it removes the need for all of your controllers to expose a MessageSource property.

However, this is not the best reason for using ApplicationContext as a MessageSource in your web application. The main reason to use ApplicationContext rather than a manually defined MessageSource bean is that Spring does, where possible, expose ApplicationContext, as a MessageSource, to the view tier. This means when you are using Spring's JSP tag library, the <spring:message> tag automatically reads messages from ApplicationContext, and when you are using JSTL, the <fmt:message> tag does the same.

All of these benefits mean that it is better to use the MessageSource support in ApplicationContext when you are building a web application, rather than manage an instance of MessageSource separately. This is especially true when you consider that all you need to do to take advantage of this feature is to configure a MessageSource bean with the name messageSource.

Using MessageSource in Stand-Alone Applications

When you are using MessageSource in stand-alone applications, where Spring offers no additional support other than to nest the MessageSource bean automatically in ApplicationContext, it is best to make the MessageSource available by using dependency injection. You can opt to make your bean ApplicationContextAware, but doing so precludes its use in a BeanFactory context. Add to this that you complicate testing without any discernible benefit, and it is clear that you should stick to using dependency injection to access MessageSource objects in a stand-alone setting.

The MessageSourceResolvable Interface

You can use an object that implements MessageSourceResolvable in place of a key and a set of arguments when you are looking up a message from a MessageSource. This interface is most widely used in the Spring validation libraries to link Error objects to their internationalized error messages.

Application Events

Another feature of ApplicationContext not present in BeanFactory is the ability to publish and receive events by using ApplicationContext as a broker. In this section, you will take a look at its usage.

Using Application Events

An event is a class derived from ApplicationEvent, which itself derives from java.util.EventObject. Any bean can listen for events by implementing the ApplicationListener<T> interface; ApplicationContext automatically registers any bean that implements this interface as a listener when it is configured. Events are published using the ApplicationEventPublisher.publishEvent() method, so the publishing class must have knowledge of ApplicationContext (which extends the ApplicationEventPublisher interface). In a web application, this is simple because many of your classes are derived from Spring Framework classes that allow access to ApplicationContext through a protected method. In a stand-alone application, you can have your publishing bean implement ApplicationContextAware to enable it to publish events.

The following code sample shows an example of a basic event class:

```
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;
    }
}
```

This code is quite basic; the only point of note is that ApplicationEvent has a single constructor that accepts a reference to the source of the event. This is reflected in the constructor for MessageEvent. Here, you can see the code for the listener:

```
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());
    }
}
```

The ApplicationListener interface defines a single method, onApplicationEvent, that is called by Spring when an event is raised. MessageEventListener shows its interest only in events of type MessageEvent (or its subclasses) by implementing the strongly typed ApplicationListener interface. If a MessageEvent was received, it writes the message to stdout. Publishing events is simple; it is just a matter of creating an instance of the event class and passing it to the ApplicationEventPublisher.publishEvent() method, as shown here:

```
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");
    }
}
```

Here you can see that the `Publisher` class retrieves an instance of itself from `ApplicationContext` and then, using the `publish()` method, publishes two `MessageEvent` instances to `ApplicationContext`. The `Publisher` bean instance accesses the `ApplicationContext` instance by implementing `ApplicationContextAware`. Here is the configuration for this example (`app-context-xml.xml`):

```
<beans ...>

    <bean id="publisher"
        class="com.apress.prospring5.ch4.Publisher"/>

    <bean id="messageEventListener"
        class="com.apress.prospring5.ch4.MessageEventListener"/>
</beans>
```

Notice that you do not need special configuration to register `MessageEventListener` with `ApplicationContext`; it is picked up automatically by Spring. Running this example results in the following output:

```
Received: I send an SOS to the world...
Received: ... I hope that someone gets my...
Received: ... Message in a bottle
```

Considerations for Event Usage

In many cases in an application, certain components need to be notified of certain events. Often you do this by writing code to notify each component explicitly or by using a messaging technology such as JMS. The drawback of writing code to notify each component in turn is that you are coupling those components to the publisher, in many cases unnecessarily.

Consider a situation whereby you cache product details in your application to avoid trips to the database. Another component allows product details to be modified and persisted to the database. To avoid making the cache invalid, the update component explicitly notifies the cache that the user details have changed. In this example, the update component is coupled to a component that, really, has nothing to do with its business responsibility. A better solution would be to have the update component publish an event every time a product's details are modified and then have interested components, such as the cache, listen for that event. This has the benefit of keeping the components decoupled, which makes it simple to remove the cache if needed or to add another listener that is interested in knowing when a product's details change.

Using JMS in this case would be overkill because the process of invalidating the product's entry in the cache is quick and is not business critical. The use of the Spring event infrastructure adds very little overhead to your application.

Typically, we use events for reactionary logic that executes quickly and is not part of the main application logic. In the previous example, the invalidation of a product in cache happens in reaction to the updating of product details, it executes quickly (or it should), and it is not part of the main function of the application. For processes that are long running and form part of the main business logic, it is recommended to use JMS or similar messaging systems such as RabbitMQ. The main benefits of using JMS are that it is more suited to long-running processes, and as the system grows, you can, if necessary, factor the JMS-driven processing of messages containing business information onto a separate machine.

Accessing Resources

Often an application needs to access a variety of resources in different forms. You might need to access some configuration data stored in a file in the file system, some image data stored in a JAR file on the classpath, or maybe some data on a server elsewhere. Spring provides a unified mechanism for accessing resources in a protocol-independent way. This means your application can access a file resource in the same way, whether it is stored in the file system, in the classpath, or on a remote server.

At the core of Spring's resource support is the `org.springframework.core.io.Resource` interface. The `Resource` interface defines ten self-explanatory methods: `contentLength()`, `exists()`, `getDescription()`, `getFile()`, `getFileName()`, `getURI()`, `getURL()`, `isOpen()`, `isReadable()`, and `lastModified()`. In addition to these ten methods, there is one that is not quite so self-explanatory: `createRelative()`. The `createRelative()` method creates a new `Resource` instance by using a path that is relative to the instance on which it is invoked. You can provide your own `Resource` implementations, although that is outside the scope of this chapter, but in most cases, you use one of the built-in implementations for accessing a file (the `FileSystemResource` class), a classpath (the `ClassPathResource` class), or URL resources (the `UrlResource` class). Internally, Spring uses another interface, `ResourceLoader`, and the default implementation, `DefaultResourceLoader`, to locate and create `Resource` instances. However, you generally won't interact with `DefaultResourceLoader`, instead using another `ResourceLoader` implementation, called `ApplicationContext`. Here is a sample application that accesses three resources by using `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("");
    }
}
```

Notice that in each call to `getResource()`, we pass in a URI for each resource. You will recognize the common `file:` and `http:` protocols that we pass in for `res1` and `res3`. The `classpath:` protocol we use for `res2` is Spring-specific and indicates that `ResourceLoader` should look in the classpath for the resource. Running this example results in the following output:

```
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
```

Notice that for both the `file:` and `http:` protocols, Spring returns a `UrlResource` instance. Spring does include a `FileSystemResource` class, but the `DefaultResourceLoader` does not use this class at all. It's because Spring's default resource-loading strategy treats the URL and file as the same type of resource with difference protocols (`file:` and `http:`). If an instance of `FileSystemResource` is required, use `FileSystemResourceLoader`. Once a `Resource` instance is obtained, you are free to access the contents as you see fit, using `getFile()`, `getInputStream()`, or `getURL()`. In some cases, such as when you are using the `http:` protocol, the call to `getFile()` results in a `FileNotFoundException`. For this reason, we recommend that you use `getInputStream()` to access resource contents because it is likely to function for all possible resource types.

Configuration Using Java Classes

Besides XML and property file configuration, you can use Java classes to configure Spring's `ApplicationContext`. Code samples were introduced here and there, until now, to make you comfortable with the annotations-style configuration. Spring JavaConfig used to be a separate project, but starting with Spring 3.0, its major features for configuration using Java classes was merged into the core Spring Framework. In this section, we show how to use Java classes to configure Spring's `ApplicationContext` and its equivalent when using XML configuration.

ApplicationContext Configuration in Java

Let's see how Spring's `ApplicationContext` can be configured by using Java classes; we'll reference the same example for message provider and renderer that we presented in Chapter 2 and Chapter 3. The following code recaps the message provider interface and a configurable message provider:[2]

```
//chapter02/hello-world/src/main/java/com/apress/prospring5/
//    ch2/decoupled/MessageProvider.java
package com.apress.prospring5.ch2.decoupled;
```

[2]The classes are not created again in the `com.apress.prospring5.ch4` package, but projects where they are defined are used as dependencies for the `java-config` project.

```
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;
        }
}
```

The following code snippet shows the MessageRenderer interface and the StandardOutMessage Renderer implementation:

```
//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;
        }
}
```

The following configuration snippet depicts the XML configuration (`app-context-xml.xml`):

```
<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>
```

The class to test this might look pretty familliar as well, as shown here:

```
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();
        }
}
```

Running this program produces the following output:

```
--> StandardOutMessageRenderer: constructor called
--> StandardOutMessageRenderer: setting the provider
This is a  configurable message
```

To get rid of the XML configuration, the `app-context-xml.xml` file must be replaced by a special class, called a *configuration class*, that will be annotated with `@Configuration`. The `@Configuration` annotation is used to inform Spring that this is a Java-based configuration file. This class will contain methods annotated with `@Bean` definitions that represent the bean declarations. The `@Bean` annotation is used to declare a Spring bean and the DI requirements. The `@Bean` annotation is equivalent to the `<bean>` tag, the method name is equivalent to the `id` attribute within the `<bean>` tag, and when instantiating the `MessageRender` bean, setter injection is achieved by calling the corresponding method to get the message provider, which is the same as using the `<ref>` attribute in the XML configuration. These annotations and this type of configuration were introduced in previous chapters to make you familiar with them, but they were not covered in detail until now. The following code snippet depicts the `AppConfig` contents that are equivalent to the XML configuration previously introduced:

```
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;
    }
}
```

The following code snippet shows how to initialize the ApplicationContext instance from the Java configuration file:

```
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();
    }
}
```

From the previous listing, we use the AnnotationConfigApplicationContext class, passing in the configuration class as the constructor argument (you can pass multiple configuration classes to it via the JDK varargs feature). Afterward, you can use ApplicationContext returned as usual. Sometimes for testing purposes, the configuration class can be declared as a static internal class, as depicted here:

```
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();
    }
}
```

The ApplicationContext instance returned can be used as usual, and the output will be the same as in the case of the XML configured application.

```
 --> StandardOutMessageRenderer: constructor called
 --> StandardOutMessageRenderer: setting the provider
Default  message
```

Having seen the basic usage of a Java configuration class, let's proceed to more configuration options. For the message provider, let's say we want to externalize the message into a properties file (message.properties) and then inject it into ConfigurableMessageProvider by using constructor injection. The content of message.properties is as follows:

```
message=Only hope can keep me together
```

Let's see the revised testing program, which loads the properties files by using the @PropertySource annotation and then injects them into the message provider implementation.

```
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;
        }
}
```

Configuration classes were introduced in Chapter 3 to show equivalents for XML elements and attributes. Bean declarations can be annotated with other annotations related to bean scopes, loading type, and dependencies. In the following code snippet, the AppConfigOne configuration class is enriched with annotations on bean declarations:

```
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;
        }
}
```

In the previous code sample, a few annotations were introduced that are explained in Table 4-3. Beans that are defined using a stereotype annotation like @Component, @Service, and others can be used in a Java configuration class, by enabling component scanning and autowiring them where needed. In the following example, we declare ConfigurableMessageProvider as a service bean.

Table 4-3. *Java Configuration Annotations Table*

Annotation	Description
@PropertySource	This annotation is used to load properties files into Spring's ApplicationContext, which accepts the location as the argument (more than one location can be provided). For XML, <context:property-placeholder> serves the same purpose.
@Lazy	This annotation instructs Spring to instantiate the bean only when requested (same as lazy-init="true" in XML). This annotation has a default value attribute that is true by default; thus, using @Lazy(value=true) is equivalent to using @Lazy.
@Scope	This is used to define the bean scope, when the desired scope is other than singleton.
@DependsOn	This annotation tells Spring that a certain bean depends on some other beans, so Spring will make sure that those beans are instantiated first.
@Autowired	This annotation is used here on the env variable, which is of Environment type. This is the Environment abstraction feature that Spring provides. We discuss it later in this chapter.

```
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;
        }
}
```

Here you can see the configuration class

```
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;
        }
}
```

`@ComponentScan` defines the packages that Spring should scan for annotations for bean definitions. It's the same as the `<context:component-scan>` tag in the XML configuration. Execute the code in the following example:

```
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();
    }
}
```

You will get the following result:

```
 --> StandardOutMessageRenderer: constructor called
 --> StandardOutMessageRenderer: setting the provider
Love on the weekend
```

An application can also have multiple configuration classes, which can be used decouple configuration and organize beans by purpose (for example, one class can be dedicated to DAO beans declaration, one for the Service beans declaration, and so forth). Let's define the `provider` bean using another configuration class named `AppConfigFour`. The bean can be accessed from another configuration class by importing the beans defined by this class. This is achieved by annotating the target configuration class `AppConfigThree` with `@Import`.

```
//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;
        }
}
```

If in the main method in class `JavaConfigExampleTwo` you replace the class `AppConfigTwo` with `AppConfigThree`, then when the example is run, the same output is printed.

Spring Mixed Configuration

But Spring can do so much more than that. Spring allows mixing XML and Java configuration classes. This is useful when applications come with legacy code that cannot be changed for some reason. To import bean declarations from an XML file, the `@ImportResource` annotation can be used. In the following configuration snippet, you can see the `provider` bean declared in an XML file named `app-context-xml-01.xml`:

```
<beans ...>

   <bean id="provider"
            class="com.apress.prospring5.ch4.ConfigurableMessageProvider"
         p:message="Love on the weekend" />

</beans>
```

The next code sample depicts class `AppConfigFive` where the beans declared in the XML file are imported. If in the main method in `classJavaConfigExampleTwo` we replace class `AppConfigTwo` with `AppConfigFive`, then when the example is run, the same output is printed.

```
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;
        }

}
```

Also, the reverse can be done as well: beans defined in Java configuration classes can be imported into XML configuration files. In the next example, the `messageRenderer` bean is defined in the XML file, and its dependency, the `provider` bean, is defined in configuration class `AppConfigSix`. The contents of the XML configuration file, `app-context-xml-02.xml`, are depicted here:

```
<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>
```

A bean of the configuration class type must be declared, and support for annotated methods must be enabled using `<context:annotation-config/>`. This enables for beans declared in the class to be configured as dependencies for beans declared in the XML file. The configuration class `AppConfigSix` is pretty simple.

```
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");
        }
}
```

Creating an `ApplicationContext` instance is done using `ClassPathXmlApplicationContext`, which has been used a lot until now.

```
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();
    }
}
```

Running the previous code yields the same output that was depicted previously.

Application infrastructure services can also be defined in Java configuration classes. For example, `@EnableTransactionManagement` defines that we will use Spring's transaction management feature, which is discussed further in Chapter 9, and `@EnableWebSecurity` and `@EnableGlobalMethodSecurity` are used to enable a Spring Security context, which will be discussed more in Chapter 16.

Java or XML Configuration?

As you already saw, using Java classes can achieve the same level of `ApplicationContext` configuration as XML. So, which one should you use? The consideration is quite like the one of whether to use XML or Java annotations for DI configuration. Each approach has its own pros and cons. However, the recommendation is the same; that is, when you and your team decide on the approach to use, stick to it and keep the configuration style persistent, instead of scattered around between Java classes and XML files. Using one approach will make the maintenance work much easier.

Profiles

Another interesting feature that Spring provides is the concept of configuration profiles. Basically, a profile instructs Spring to configure only the `ApplicationContext` instance that was defined when the specified profile was active. In this section, we demonstrate how to use profiles in a simple program.

An Example of Using the Spring Profiles Feature

Let's say there is a service called `FoodProviderService` that is responsible for providing food to schools, including kindergarten and high school. The `FoodProviderService` interface has only one method called `provideLunchSet()`, which produces the lunch set to each student for the calling school. A lunch set is a list of `Food` objects, which is a simple class that has only a `name` attribute. The following code snippet shows the `Food` class:

```
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;
    }
}
```

And here is the FoodProviderService interface:

```
package com.apress.prospring5.ch4;

import java.util.List;

public interface FoodProviderService {
    List<Food> provideLunchSet();
}
```

Now suppose that there are two providers for the lunch set, one for kindergarten and one for high school. The lunch set produced by them is different, although the service they provide is the same, that is, to provide lunch to students. So, now let's create two implementations of FoodProviderService, using the same name but putting them into different packages to identify their target school. The two classes are shown here:

```
//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;
    }
}
```

From the previous listings, you can see that the two implementations provide the same FoodProviderService interface but produce different combinations of food in the lunch set. So, now suppose a kindergarten wants the provider to deliver the lunch set for their students; let's see how we can use Spring's profile configuration to achieve this. We will run through the XML configuration first. We will create two XML configuration files, one for the kindergarten profile and the other for the high-school profile. The following configuration snippet depicts the two profile configurations:

```
<!-- 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>
```

In the previous two configurations, notice the usage of profile="kindergarten" and profile= "highschool", respectively, within the <beans> tag. It actually tells Spring that those beans in the file should be instantiated only when the specified profile is active. Now let's see how to activate the correct profile when using Spring's ApplicationContext in a stand-alone application. The following code snippet shows the testing program:

```
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();
    }
}
```

The `ctx.load()` method will load both `kindergarten-config.xml` and `highschool-config.xml`, since we pass the method the wildcard as the prefix. In this example, only the beans in the file `kindergarten-config.xml` will be instantiated by the Spring based on the `profile` attribute, which is activated by passing the JVM argument `-Dspring.profiles.active="kindergarten"`. Running the program with this JVM argument produces the following output:

```
Food: Milk
Food: Biscuits
```

This is exactly what the implementation of the kindergarten provider will produce for the lunch set. Now change the profile argument from the previous listing to high school (`-Dspring.profiles.active="highschool"`), and the output will change to the following:

```
Food: Coke
Food: Hamburger
Food: French Fries
```

You can also programmatically set the profile to use in your code by calling `ctx.getEnvironment().setActiveProfiles("kindergarten")`. Additionally, you can register classes to be enabled by profiles using Java Config by simply adding the `@Profile` annotation to your class.

Spring Profiles Using Java Configuration

Of course, there is a way to configure Spring profiles using Java configuration because developers who do not like XML configuration should get satisfaction as well. The XML files declared in the previous section have to be replaced with the equivalent Java configuration classes. This is how the configuration class for a `kindergarten` profile looks:

```
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();
        }
}
```

As you can see, a `foodProviderService` bean is defined using the `@Bean` annotation. The class is marked as specific to the `kindergarten` profile using the `@Profile` annotation. The class specific to the `highschool` profile is identical, except for the bean type, the profile name, and the class name, obviously.

```
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();
        }
}
```

These classes are used in the same way the XML files are. A context is declared to use both of them, and only one of them will actually be used to create the `ApplicationContext` instance depending on the value of the `-Dspring.profiles.active` JVM option.

```
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();
        }
}
```

By running the previous example with `kindergarten` as the value for the `-Dspring.profiles.active` JVM option, the expected output is printed.

```
Food: Milk
Food: Biscuits
```

There is also an annotation for configuring used profiles that replaces the `-Dspring.profiles.active` JVM option, but this can be used only on test classes. As testing Spring applications is covered in Chapter 13, it won't be covered in detail here. But some sample code is included.

```
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());
        }

}
```

As you have probably figured out yourself, the annotation that specifies which profile is used to run this test is `@ActiveProfiles("kindergarten")`. In complex applications, usually there is more than one profile, and more of them can be used to compose the context configuration for a test. This class can be run in any Java smart editor and is automatically run when executing `gradle clean build`.

Considerations for Using Profiles

The profiles feature in Spring creates another way for developers to manage the application's running configuration, which used to be done in build tools (for example, Maven's profile support). Build tools rely on the arguments passed into the tool to pack the correct configuration/property files into the Java archive (JAR or WAR, depending on the application type) and then deploy to the target environment. Spring's profile feature lets us as an application developer define the profiles by yourself and activate them either programmatically or by passing in the JVM argument. By using Spring's profile support, you can now use the same application archive and deploy to all environments by passing in the correct profiles as an argument during JVM startup. For example, you can have applications with different profiles such as `(dev, hibernate)`, `(prd, jdbc)`, and so on, with each combination representing the running environment (development or production) and the data access library to use (Hibernate or JDBC). It brings application profile management into the programming side.

But this approach also has its drawbacks. For example, some may argue that putting all the configurations for different environments into application configuration files or Java classes and bundling them together will be error prone if not handled carefully (for example, the administrator may forget to set the correct JVM argument in the application server environment). Packing files for all profiles together will also make the package a bit larger than usual. Again, let the application and configuration requirements drive you to select the approach that best fits your project.

Environment and PropertySource Abstraction

To set the active profile, we need to access the `Environment` interface. The `Environment` interface is an abstraction layer that serves to encapsulate the environment of the running Spring application.

Besides the profile, other key pieces of information encapsulated by the `Environment` interface are properties. Properties are used to store the application's underlying environment configuration, such as the location of the application folder, database connection information, and so on.

The `Environment` and `PropertySource` abstraction features in Spring assist developers in accessing various configuration information from the running platform. Under the abstraction, all system properties, environment variables, and application properties are served by the `Environment` interface, which Spring populates when bootstrapping `ApplicationContext`. The following code snippet shows a simple example:

```
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 EnvironmentSample {
    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();
    }
}
```

In the previous code snippet, after the `ApplicationContext` initialization, we get a reference to the `ConfigurableEnvironment` interface. Via the interface, a handle to `MutablePropertySources` (a default implementation of the `PropertySources` interface, which allows manipulation of the contained property sources) is obtained. Afterward, we construct a map, put the application properties into the map, and then construct a `MapPropertySource` class (a `PropertySource` subclass that reads keys and values from a `Map` instance) with the map. Finally, the `MapPropertySource` class is added to `MutablePropertySources` via the `addLast()` method. Run the program, and the following prints:

```
user.home: /home/jules
JAVA_HOME: /home/jules/bin/java
user.home: /home/jules
JAVA_HOME: /home/jules/bin/java
application.home:  application_home
```

For the first two lines, the JVM system property `user.home` and the environment variable `JAVA_HOME` are retrieved, as before (by using the JVM's `System` class). However, for the last three lines, you can see that all the system properties, environment variables, and application properties can be accessed via the `Environment` interface. You can see how the `Environment` abstraction can help us manage and access all the various properties within the application's running environment.

For the `PropertySource` abstraction, Spring will access the properties in the following default order:

- System properties for the running JVM
- Environment variables
- Application-defined properties

So, for example, suppose we define the same application property, `user.home`, and add it to the `Environment` interface via the `MutablePropertySources` class. If you run the program, you will still see that `user.home` is retrieved from the JVM properties, not yours. However, Spring allows you to control the order in which `Environment` retrieves the properties. The following code snippet shows the revised version:

```
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 EnvironmentSample {
    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("application.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();
    }
}
```

In the previous code sample, we have defined an application property also called `user.home` and added it as the first one to search for via the `addFirst()` method of the `MutablePropertySources` class. When you run the program, you will see the following output:

```
user.home: /home/jules
JAVA_HOME: /home/jules/bin/java
user.home: application_home
JAVA_HOME: /home/jules/bin/java
```

The first two lines remain the same because we still use the `getProperty()` and `getenv()` methods of the JVM `System` class to retrieve them. However, when using the `Environment` interface, you will see that the `user.home` property we defined takes precedence since we defined it as the first one to search for property values.

In real life, you seldom need to interact directly with the Environment interface but will use a property placeholder in the form of ${} (for example, ${application.home}) and inject the resolved value into Spring beans. Let's see this in action. Suppose we had a class to store all the application properties loaded from a property file. The following shows the AppProperty class:

```
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;
    }
}
```

Here you can see the contents of the application.properties file:

```
application.home=application_home
user.home=/home/jules-new
```

Note that the property file also declares the user.home property. Let's take a look at the Spring XML configuration; see the following code (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>
```

We use the `<context:property-placeholder>` tag to load the properties into Spring's `Environment`, which is wrapped into the `ApplicationContext` interface. We also use the SpEL placeholders to inject the values into the `AppProperty` bean. The following code snippet shows the testing program:

```
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();
    }
}
```

Let's run the program, and you will see the following output:

```
application.home: application_home
user.home: /Users/jules
```

You will see the `application.home` placeholder is properly resolved, while the `user.home` property is still retrieved from the JVM properties, which is correct because it's the default behavior for the `PropertySource` abstraction. To instruct Spring to give precedence for the values in the `application.properties` file, we add the attribute `local-override="true"` to the `<context:property-placeholder>` tag.

```
<context:property-placeholder local-override="true"
    location="classpath:env/application.properties"/>
```

The `local-override` attribute instructs Spring to override the existing properties with the properties defined in this placeholder. Run the program, and you will see that now the `user.home` property from the `application.properties` file is retrieved.

```
application.home: application_home
user.home: /home/jules-new
```

Configuration Using JSR-330 Annotations

As we discussed in Chapter 1, JEE 6 provides support for JSR-330 (Dependency Injection for Java), which is a collection of annotations for expressing an application's DI configuration within a JEE container or other compatible IoC framework. Spring also supports and recognizes those annotations, so although you may not be running your application in a JEE 6 container, you can still use JSR-330 annotations within Spring. Using JSR-330 annotations can help you ease the migration to the JEE 6 container or other compatible IoC container (for example, Google Guice) away from Spring.

Again, let's take the message renderer and message provider as an example and implement it using JSR-330 annotations. To support JSR-330 annotations, you need to add the `javax.inject`[3] dependency to the project.

The following code snippet shows the `MessageProvider` and `ConfigurableMessageProvider` implementation:

```
 //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;
    }
}
```

[3]You can find dependency characteristics, like group IDs and latest versions, on the Maven public repository. For example, this is the dedicated page for `java.inject`: `https://mvnrepository.com/artifact/javax.inject/javax.inject`.

You will notice that all annotations belong to the javax.inject package, which is the JSR-330 standard. This class used @Named in two places. First, it can be used to declare an injectable bean (the same as the @Component annotation or other stereotype annotations in Spring). In the listing, the @Named("messageProvider") annotation specifies that ConfigurableMessageProvider is an injectable bean and gives it the name messageProvider, which is the same as the name attribute in Spring's <bean> tag. Second, we use constructor injection by using the @Inject annotation before the constructor that accepts a string value. Then, we use @Named to specify that we want to inject the value that had the name message assigned. Let's move on to see the MessageRenderer interface and StandardOutMessageRenderer implementation.

```
 //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;
    }
}
```

In the previous code snippet, we used @Named to define that it's an injectable bean. Notice the @Singleton annotation. It's worth noting that in the JSR-330 standard, a bean's default scope is nonsingleton, which is like Spring's prototype scope. So, in a JSR-330 environment, if you want your bean to be a singleton, you need to use the @Singleton annotation. However, using this annotation in Spring actually doesn't have any effect because Spring's default scope for bean instantiation is already singleton. We just put it here for a demonstration, and it's worth noting the difference between Spring and other JSR-330–compatible containers.

For the messageProvider property, we use @Inject for setter injection this time and specify that a bean with the name messageProvider should be used for injection. The following configuration snippet defines a simple Spring XML configuration for the application (app-context-annotation.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>
```

You don't need any special tags to use JSR-330; just configure your application like a normal Spring application. We use <context:component-scan> to instruct Spring to scan for the DI-related annotations, and Spring will recognize those JSR-330 annotations. We also declare a Spring bean called message for constructor injection into the ConfigurableMessageProvider class. The following code snippet shows the testing program:

```
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();
        }
}
```

Running the program yields the following output:

```
Gravity is working against me
```

By using JSR-330, you can ease the migration into other JSR-330–compatible IoC containers (for example, JEE 6–compatible application servers or other DI containers such as Google Guice). However, Spring's annotations are much more feature rich and flexible than JSR-330 annotations. Some main differences are highlighted here:

- When using Spring's `@Autowired` annotation, you can specify a `required` attribute to indicate that the DI must be fulfilled (you can also use Spring's `@Required` annotation to declare this requirement), while for JSR-330's `@Inject` annotation, there is no such equivalent. Moreover, Spring provides the `@Qualifier` annotation, which allows more fine-grained control for Spring to perform autowiring of dependencies based on qualifier name.
- JSR-330 supports only singleton and nonsingleton bean scopes, while Spring supports more scopes, which is useful for web applications.
- In Spring, you can use the `@Lazy` annotation to instruct Spring to instantiate the bean only when requested by the application. There's no such equivalent in JSR-330.

You can also mix and match Spring and JSR-330 annotations in the same application. However, it is recommended that you settle on either one to maintain a consistent style for your application. One possible way is to use JSR-330 annotations as much as possible and use Spring annotations when required. However, this brings you fewer benefits, because you still need to do quite a bit of work in migrating to another DI container. In conclusion, Spring's annotations approach is recommended over JSR-330 annotations, because Spring's annotations are much more powerful, unless there is a requirement that your application should be IoC container independent.

Configuration Using Groovy

New to Spring Framework 4.0 is the ability to configure your bean definitions and `ApplicationContext` by using the Groovy language. This provides developers with another choice in configuration to either replace or supplement XML and/or annotation-based bean configuration. A Spring `ApplicationContext` can be created directly in a Groovy script or loaded from Java, both by way of the `GenericGroovyApplicationContext` class. First let's dive into the details by showing how to create bean definitions from an external Groovy script and loading them from Java. In previous sections and chapters, we introduced various bean classes, and to promote some code reusability, we will use in this example the `Singer` class introduced in Chapter 3. The following code snippet shows the content of this class:

```
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;
    }
}
```

As you can see, this is just a Java class with a couple of properties describing a singer. We use this simple Java class here to show that just because you configure your beans in Groovy doesn't mean your entire code base needs to be rewritten in Groovy. Not only that, but Java classes can be imported from dependencies and used within Groovy scripts. Now, let's create the Groovy script (beans.groovy) that will be used to create the bean definition, as shown in the previous code snippet:

```
package com.apress.prospring5.ch4

import com.apress.prospring5.ch3.xml.Singer

beans {
    singer(Singer, name: 'John Mayer', age: 39)
}
```

This Groovy script starts with a top-level closure called beans, which provides bean definitions to Spring. First, we specify the bean name (singer), and then as arguments we provide the class type (Singer) followed by the property names and values that we would like to set. Next, let's create a simple test driver in Java, loading bean definitions from the Groovy script, as shown here:

```
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);
    }
}
```

As you can see, the creation of ApplicationContext is carried out in typical fashion, but it's done by using the GenericGroovyApplicationContext class and providing your Groovy script that builds the bean definitions.

Before you can run the samples in this section, you need to add a dependency to this project: the groovy-all library. The build.gradle configuration file content for the project groovy-config-java is shown here:

```
apply plugin: 'groovy'

dependencies {
        compile misc.groovy
        compile project(':chapter03:bean-inheritance')
}
```

The compile project(':chapter03:bean-inheritance') line specifies that the Chapter 3 bean-inheritance project must be compiled and used as a dependency for this project. This is the project containing the Singer class.

Gradle configuration files use Groovy syntax, and misc.groovy references the groovy property of the misc array defined in the build.gradle file of the parent project. A snippet of the content of this file (configurations related to Groovy) is shown here:

```
ext {
        springVersion =  '5.0.0.M4'
        groovyVersion =  '2.4.5'
...

        misc = [
                ...
                    groovy: "org.codehaus.groovy:groovy-all:$groovyVersion"
        ]
...
}
```

Running the GroovyBeansFromJava class yields the following output:

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

Now that you have seen how to load bean definitions from Java via an external Groovy script, how can we go about creating the ApplicationContext and bean definitions from a Groovy script alone? Let's take a look at the Groovy code (GroovyConfig.groovy) listed here:

```
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")
```

When we run this sample, we get same output as before. This time we create an instance of a typical GenericApplicationContext but use GroovyBeanDefinitionReader, which will be used to pass bean definitions to. Then, as in the previous sample, we create a bean from your simple POJO, refresh ApplicationContext, and print the string representation of the Singer bean. It doesn't get any easier than that!

As you probably can tell, we are only scratching the surface of what can be done with the Groovy support in Spring. Since you have the full power of the Groovy language, you can do all sorts of interesting things when creating bean definitions. As you have full access to ApplicationContext, not only can you configure beans, but you can also work with profile support, property files, and so on. Just keep in mind, with great power comes great responsibility.

Spring Boot

Until now you've learned more than one way to configure a Spring application. Whether it is XML, annotations, Java configuration classes, Groovy scripts, or a mix of all these, now you should have a basic idea of how it's done. But what if we tell you there is something even cooler than all that?

The Spring Boot project aims to simplify the getting-started experience of building an application by using Spring. Spring Boot takes the guesswork out of manually gathering dependencies and provides some of the most common features needed by most applications, such as metrics and health checks.

Spring Boot takes an "opinionated" approach to achieve the goal of developer simplification by way of providing starter projects for various types of applications that already contain the proper dependencies and versions, which means less time spent to get started. For those who may be looking to get away from XML completely, Spring Boot does not require any configuration to be written in XML.

In this example, we will create the traditional Hello World web application with a twist. You may be surprised to see the minimal amount of code required to do so as compared to your typical Java web application setup. Typically, we have started off samples by defining the dependencies we need to add to your project. Part of Spring Boot's simplification model is to prepare all the dependencies for you, and when using Maven, for example, you as the developer utilize a parent POM to obtain this functionality. When using Gradle, things become even simpler. There is no parent needed, except a Gradle plug-in and a starter dependency. In the following example, we will create a Spring application that lists all the beans in the context, and then the helloWorld bean is accessed. The Gradle configuration of the boot-simple project is depicted here:

```
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
}
```

The `boot.springBootPlugin` line references the `springBootPlugin` property of the `boot` array defined in the `build.gradle` file of the parent project. A snippet of the content of this file (configurations related to Spring Boot only) is depicted here:

```
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"
    ]
    ...
}
```

At the time of this writing, Spring Boot version 2.0.0 has not been released yet. That is why the version is `2.0.0.BUILD-SNAPSHOT` and we need to add the Spring Snapshot repository, `https://repo.spring.io/ libs-snapshot`, in the configuration. It is quite possible that after this book, an official version will be released.

Each release of Spring Boot provides a curated list of dependencies it supports. The versions of the necessary libraries are selected so the API matches perfectly, and this is handled by Spring Boot. Therefore, the manual configuration of dependencies versions is not necessary. Upgrading Spring Boot will ensure that those dependencies are upgraded as well. With the previous configuration, a set of dependencies will be added to the project, each with the proper versions so that their API will be compatible. In a smart editor like IntelliJ IDEA, there is a Gradle Projects view, where you can expand each module and inspect the available tasks and dependencies, as shown in Figure 4-3.

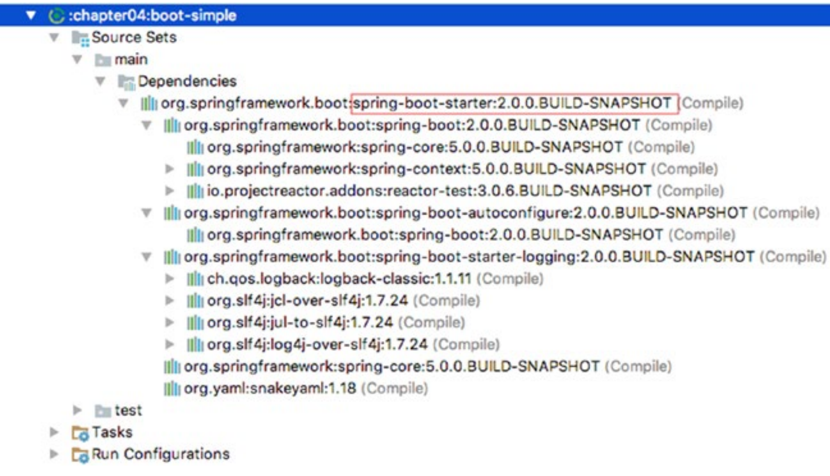

Figure 4-3. *Gradle Projects view of the boot-simple project*

Now that the setup is in place, let's create the classes. The HelloWorld class is shown here:

```
package com.apress.prospring5.ch4;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

@Component
public class HelloWorld {

        private static Logger logger =
             LoggerFactory.getLogger(HelloWorld.class);

        public void sayHi() {
                logger.info("Hello World!");
        }
}
```

There is nothing special or complicated about this; it's just a class with a method and a bean declaration annotation on it. Let's see how to build a Spring application using Spring Boot and create an ApplicationContext that contains this 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:");

                // listing all bean definition    names
                Arrays.stream(ctx.getBeanDefinitionNames()).forEach(logger::info);

                HelloWorld hw = ctx.getBean(HelloWorld.class);
                hw.sayHi();

                System.in.read();
                ctx.close();
        }
}
```

That is all. Really. And this class could have been smaller, but we wanted to show you how to do some extra things. Let's cover them all.

- *Check that we have a context*: The `assert` statement is used to test your assumption that `ctx` is not `null`.
- *Set up logging*: Spring Boot comes with a set of logging libraries, so we only have to put under the `resources` directory the configuration for the one we want to use. In our case, we chose `logback`.
- *List all bean definitions in the context*: Using Java 8 lambda expressions, you can list all bean definitions in the context in one line. So, we added that line so you can see what beans are autoconfigured for you out of the box by Spring Boot. In the list you will find the `helloWorld` bean too.
- *Confirm exit*: Without the `System.in.read();` method, the application prints the beans names, prints `HelloWorld`, and then exits. We added that call so the application will wait for the developer to press a key and then exit.

The novelty here is the `@SpringBootApplication` annotation. This annotation is a top-level annotation designed to be used only at the class level. It a convenience annotation that is equivalent to declaring the following three:

- `@Configuration`: Marks this class as a configuration class that can declare beans with `@Bean`.
- `@EnableAutoConfiguration`: This is a specific Spring Boot annotation from the package `org.springframework.boot.autoconfigure` that can enable Spring `ApplicationContext`, attempting to guess and configure beans that you are likely to need based on the specified dependencies.

  `@EnableAutoConfiguration` works well with Spring-provided starter dependencies, but it is not directly tied to them, so other dependencies outside the starters can be used. For example, if there is a specific embedded server on the classpath, it will be used, unless there is another `EmbeddedServletContainerFactory` configuration in the project.
- `@ComponentScan`: We can declare classes annotated with stereotype annotations that will become beans of some kind. The attribute used to list the packages to scan used with `@SpringBootApplication` is `basePackages`. In version 1.3.0, another attribute was added for component scanning: `basePackageClasses`. This attribute provides a type-safe alternative to `basePackages` for specifying the packages to scan for annotated components. The package of each class specified will be scanned.

If the `@SpringBootApplication` annotation has no component scanning attribute defined, it will scan only the package in which the class annotated with it is located. That is why in the example presented here, the `helloWorld` bean definition is found, and the bean is created.

The previous Spring Boot application was a simple console application with one developer-defined bean and a full-blown out-of-the-box environment. But Spring Boot provides starter dependencies for web applications as well. The dependency to use is `spring-boot-starter-web`, and in Figure 4-4 you can see the transitive dependencies of this Boot starter library. In the `boot-web` project, the `HelloWorld` class is a Spring controller, which is a special type of class that is used to create a Spring web bean. This is a typical Spring MVC controller class that you will learn about in Chapter 16.

:chapter04:boot-web
Source Sets
main
Dependencies
org.springframework.boot:spring-boot-starter-web:2.0.0.BUILD-SNAPSHOT (Compile)
org.springframework.boot:spring-boot-starter:2.0.0.BUILD-SNAPSHOT (Compile)
org.springframework.boot:spring-boot-starter-tomcat:2.0.0.BUILD-SNAPSHOT (Compile)
org.apache.tomcat.embed:tomcat-embed-core:8.5.11 (Compile)
org.apache.tomcat.embed:tomcat-embed-el:8.5.11 (Compile)
org.apache.tomcat.embed:tomcat-embed-websocket:8.5.11 (Compile)
org.hibernate:hibernate-validator:5.4.0.Final (Compile)
com.fasterxml.jackson.core:jackson-databind:2.8.7 (Compile)
org.springframework:spring-web:5.0.0.BUILD-SNAPSHOT (Compile)
org.springframework:spring-core:5.0.0.BUILD-SNAPSHOT (Compile)
org.springframework:spring-aop:5.0.0.BUILD-SNAPSHOT (Compile)
org.springframework:spring-beans:5.0.0.BUILD-SNAPSHOT (Compile)
org.springframework:spring-context:5.0.0.BUILD-SNAPSHOT (Compile)
org.springframework:spring-webmvc:5.0.0.BUILD-SNAPSHOT (Compile)
org.springframework:spring-web:5.0.0.BUILD-SNAPSHOT (Compile)
org.springframework:spring-core:5.0.0.BUILD-SNAPSHOT (Compile)
org.springframework:spring-aop:5.0.0.BUILD-SNAPSHOT (Compile)
org.springframework:spring-beans:5.0.0.BUILD-SNAPSHOT (Compile)
org.springframework:spring-context:5.0.0.BUILD-SNAPSHOT (Compile)
org.springframework:spring-expression:5.0.0.BUILD-SNAPSHOT (Compile)
test
Tasks

Figure 4-4. *Gradle Projects view of the boot-web project*

```
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!";
         }
}
```

The annotation used to declare Spring web beans is a specialization of `@Component`: the `@Controller` annotation. These types of classes contain methods annotated with `@RequestMapping`, which are mapped to a certain request URL. The annotation you see in the example `@RestController` is used for practical reasons. It is the `@Controller` annotation used for REST services. And exposing the `helloWorld` bean as a REST service is useful here because you do not have to create a full-blown web application with an user interface and other web components that will pollute the basic idea in this section. All the web components introduced here are covered in Chapter 16.

Next we create your bootstrap class by using a simple `main()` method, as shown here:

```
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();
        }
}
```

Because the HelloWorld controller was declared in a different package than the WebApplication class, we created an opportunity to depict how the scanBasePackageClasses attribute is used.

At this point, you may be asking yourself: where is the web.xml configuration file and all the other components I must create for a basic web application? You already defined everything needed in the previous listings! Don't believe it? Compile the project and run the Application class until you see the log message indicating that the application has been started. If you look at the generated log files, you will see a lot going on with such little code. Most notably, it looks like Tomcat is running, and various endpoints such as health checks, environment output information, and metrics are already defined for you. First navigate to http://localhost:8080 and you will see the Hello World web page displayed as expected. Next take a look at some of the preconfigured endpoints (for example, http://localhost:8080/health, which returns a JSON representation of the application status). Going further, load http://localhost:8080/metrics to get a better understanding of various metrics that are being collected such as heap size, garbage collection, and so on.

As you may be able to tell already from this one sample alone, Spring Boot radically simplifies the way you go about creating any type of application. Gone are the days of having to configure numerous files to get a simple web application going, and with an embedded servlet container ready to serve your web application, everything "just works."

While we have shown you a simple example, keep in mind that Spring Boot does not lock you into using what it chooses; it simply takes the "opinionated" approach and chooses defaults for you. If you don't want to use embedded Tomcat but Jetty instead, simply modify the configuration file to exclude the Tomcat starter module from the spring-boot-starter-web dependency. Utilizing the Gradle Projects view is one way to help you visualize what dependencies are being brought into your project. Spring Boot also provides many other starter dependencies for other types of applications, and you are encouraged to read through the documentation for more details.

For more information on Spring Boot, please see its project page at http://projects.spring.io/spring-boot/.

Summary

In this chapter, you saw a wide range of Spring-specific features that complement the core IoC capabilities. You saw how to hook into the life cycle of a bean and to make it aware of the Spring environment. We introduced `FactoryBeans` as a solution for IoC, enabling a wider set of classes. We also showed how you can use `PropertyEditors` to simplify application configuration and to remove the need for artificial `String`-typed properties. We showed you more than one way to define beans using XML, annotations, and Java configuration. Moreover, we finished with an in-depth look at some additional features offered by `ApplicationContext`, including i18n, event publication, and resource access.

We also covered features such as using Java classes and the new Groovy syntax instead of XML configuration, profiles support, and the environment and property source abstraction layer. Finally, we discussed using JSR-330 standard annotations in Spring.

The icing on the cake was how to use Spring Boot to configure beans and boot up your application as soon as possible and with little effort.

So far, we have covered the main concepts of the Spring Framework and its features as a DI container as well as other services that the core Spring Framework provides. In the next chapter and onward, we discuss using Spring in specific areas such as AOP, data access, transaction support, and web application support.

CHAPTER 5

Introducing Spring AOP

Besides dependency injection (DI), another core feature that the Spring Framework offers is support for aspect-oriented programming (AOP). AOP is often referred to as a tool for implementing crosscutting concerns. The term *crosscutting concerns* refers to logic in an application that cannot be decomposed from the rest of the application and may result in code duplication and tight coupling. By using AOP for modularizing individual pieces of logic, known as *concerns*, you can apply them to many parts of an application without duplicating the code or creating hard dependencies. Logging and security are typical examples of crosscutting concerns that are present in many applications. Consider an application that logs the start and end of every method for debugging purposes. You will probably refactor the logging code into a special class, but you still have to call methods on that class twice per method in your application in order to perform the logging. Using AOP, you can simply specify that you want the methods on your logging class to be invoked before and after each method call in your application.

It is important to understand that AOP complements object-oriented programming (OOP), rather than competing with it. OOP is very good at solving a wide variety of problems that we, as programmers, encounter. However, if you look at the logging example again, it is obvious to see where OOP is lacking when it comes to implementing crosscutting logic on a large scale. Using AOP on its own to develop an entire application is practically impossible, given that AOP functions on top of OOP. Likewise, although it is certainly possible to develop entire applications by using OOP, you can work smarter by employing AOP to solve certain problems that involve crosscutting logic.

This chapter covers the following topics:

- *AOP basics*: Before discussing Spring's AOP implementation, we cover the basics of AOP as a technology. Most of the concepts covered in the "AOP Concepts" section are not specific to Spring and can be found in any AOP implementation. If you are already familiar with another AOP implementation, feel free to skip the "AOP Concepts" section.
- *Types of AOP*: There are two distinct types of AOP: static and dynamic. In static AOP, like that provided by AspectJ's[1] compile-time weaving mechanisms, the crosscutting logic is applied to your code at compile time, and you cannot change it without modifying the code and recompiling. With dynamic AOP, such as Spring AOP, crosscutting logic is applied dynamically at runtime. This allows you to make changes to the AOP configuration without having to recompile the application. These types of AOP are complementary, and, when used together, they form a powerful combination that you can use in your applications.
- *Spring AOP architecture*: Spring AOP is only a subset of the full AOP feature set found in other implementations such as AspectJ. In this chapter, we take a high-level look at which features are present in Spring, how they are implemented, and why some features are excluded from the Spring implementation.

[1] `http://eclipse.org/aspectj`

© Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho 2017

I. Cosmina et al., *Pro Spring 5*, https://doi.org/10.1007/978-1-4842-2808-1_5

- *Proxies in Spring AOP*: Proxies are a huge part of how Spring AOP works, and you must understand them to get the most out of Spring AOP. In this chapter, we look at the two kinds of proxy: the JDK dynamic proxy and the CGLIB proxy. In particular, we look at the different scenarios in which Spring uses each proxy, the performance of the two proxy types, and some simple guidelines to follow in your application to get the most from Spring AOP.
- *Using Spring AOP*: In this chapter, we present some practical examples of AOP usage. We start off with a simple Hello World example to ease you into Spring's AOP code, and we continue with a detailed description of the AOP features that are available in Spring, complete with examples.
- *Advanced use of pointcuts*: We explore the `ComposablePointcut` and `ControlFlowPointcut` classes, introductions, and appropriate techniques you should employ when using pointcuts in your application.
- *AOP framework services*: The Spring Framework fully supports configuring AOP transparently and declaratively. We look at three ways (the `ProxyFactoryBean` class, the `aop` namespace, and `@AspectJ`-style annotations) to inject declaratively defined AOP proxies into your application objects as collaborators, thus making your application completely unaware that it is working with advised objects.
- *Integrating AspectJ*: AspectJ is a fully featured AOP implementation. The main difference between AspectJ and Spring AOP is that AspectJ applies advice to target objects via weaving (either compile-time or load-time weaving), while Spring AOP is based on proxies. The feature set of AspectJ is much greater than that of Spring AOP, but it is much more complicated to use than Spring. AspectJ is a good solution when you find that Spring AOP lacks a feature you need.

AOP Concepts

As with most technologies, AOP comes with its own specific set of concepts and terms, and it's important to understand what they mean. The following are the core concepts of AOP:

- *Joinpoints*: A joinpoint is a well-defined point during the execution of your application. Typical examples of joinpoints include a call to a method, the method invocation itself, class initialization, and object instantiation. Joinpoints are a core concept of AOP and define the points in your application at which you can insert additional logic using AOP.
- *Advice*: The code that is executed at a particular joinpoint is the advice, defined by a method in your class. There are many types of advice, such as before, which executes before the joinpoint, and after, which executes after it.
- *Pointcuts*: A pointcut is a collection of joinpoints that you use to define when advice should be executed. By creating pointcuts, you gain fine-grained control over how you apply advice to the components in your application. As mentioned previously, a typical joinpoint is a method invocation, or the collection of all method invocations in a particular class. Often you can compose pointcuts in complex relationships to further constrain when advice is executed.
- *Aspects*: An aspect is the combination of advice and pointcuts encapsulated in a class. This combination results in a definition of the logic that should be included in the application and where it should execute.

- *Weaving*: This is the process of inserting aspects into the application code at the appropriate point. For compile-time AOP solutions, this weaving is generally done at build time. Likewise, for runtime AOP solutions, the weaving process is executed dynamically at runtime. AspectJ supports another weaving mechanism called *load-time weaving* (LTW), in which it intercepts the underlying JVM class loader and provides weaving to the bytecode when it is being loaded by the class loader.
- *Target*: An object whose execution flow is modified by an AOP process is referred to as the target object. Often you see the target object referred to as the *advised* object.
- *Introduction*: This is the process by which you can modify the structure of an object by introducing additional methods or fields to it. You can use introduction AOP to make any object implement a specific interface without needing the object's class to implement that interface explicitly.

Don't worry if you find these concepts confusing; this will all become clear when you see some examples. Also, be aware that you are shielded from many of these concepts in Spring AOP, and some are not relevant because of Spring's choice of implementation. We will discuss each of these features in the context of Spring as we progress through the chapter.

Types of AOP

As we mentioned earlier, there are two distinct types of AOP: static and dynamic. The difference between them is really the point at which the weaving process occurs and how this process is achieved.

Using Static AOP

In static AOP, the weaving process forms another step in the build process for an application. In Java terms, you achieve the weaving process in a static AOP implementation by modifying the actual bytecode of your application, changing and extending the application code as necessary. This is a well-performing way of achieving the weaving process because the end result is just Java bytecode, and you do not perform any special tricks at runtime to determine when advice should be executed. The drawback of this mechanism is that any modifications you make to the aspects, even if you simply want to add another joinpoint, require you to recompile the entire application. AspectJ's compile-time weaving is an excellent example of a static AOP implementation.

Using Dynamic AOP

Dynamic AOP implementations, such as Spring AOP, differ from static AOP implementations in that the weaving process is performed dynamically at runtime. How this is achieved is implementation-dependent, but as you will see, Spring's approach is to create proxies for all advised objects, allowing for advice to be invoked as required. The drawback of dynamic AOP is that, typically, it does not perform as well as static AOP, but the performance is steadily increasing. The major benefit of dynamic AOP implementations is the ease with which you can modify the entire aspect set of an application without needing to recompile the main application code.

Choosing an AOP Type

Choosing whether to use static or dynamic AOP is quite a hard decision. Both have their own benefits, and you are not restricted to using only one type. In general, static AOP implementations have been around longer and tend to have more feature-rich implementations, with a greater number of available joinpoints. Typically, if performance is absolutely critical or you need an AOP feature that is not implemented in Spring,

you will want to use AspectJ. In most other cases, Spring AOP is ideal. Keep in mind that many AOP-based solutions such as transaction management are already provided for you by Spring, so check the framework capabilities before rolling your own! As always, let the requirements of your application drive your choice of AOP implementation, and don't restrict yourself to a single implementation if a combination of technologies would better suit your application. In general, Spring AOP is less complex than AspectJ, so it tends to be an ideal first choice.

AOP in Spring

Spring's AOP implementation can be viewed as two logical parts. The first part is the AOP core, which provides fully decoupled, purely programmatic AOP functionality (also known as the Spring AOP API). The second part of the AOP implementation is the set of framework services that make AOP easier to use in your applications. On top of this, other components of Spring, such as the transaction manager and EJB helper classes, provide AOP-based services to simplify the development of your application.

The AOP Alliance

The AOP Alliance (`http://aopalliance.sourceforge.net/`) is a joint effort among representatives of many open source AOP projects to define a standard set of interfaces for AOP implementations. Wherever applicable, Spring uses the AOP Alliance interfaces rather than defining its own. This allows you to reuse certain advice across multiple AOP implementations that support the AOP Alliance interfaces.

Hello World in AOP

Before we dive into discussing the Spring AOP implementation in detail, let's take a look at an example. We'll see how to change the typical Hello World example, and we'll turn to movies for inspiration. We'll write a class named `Agent`, and we'll implement it to print `Bond`. When using AOP, the instance of this class will be transformed at runtime to print `James Bond!`. The following code depicts the `Agent` class:

```
package com.apress.prospring5.ch5;

public class Agent {
    public void speak() {
        System.out.print("Bond");
    }
}
```

With the name-printing method implemented, let's *advise*—which in AOP terms means add advice to—this method so that `speak()` prints `James Bond!` instead.

To do this, we need to execute code prior to the existing body executing (to write `James`), and we need to execute code after that method body executes (to write `!`). In AOP terms, what we need is around advice, which executes around a joinpoint. In this case, the joinpoint is the invocation of the `speak()` method. The following code snippet shows the code of the `AgentDecorator` class, which acts as the around-advice implementation:

```
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;
    }
}
```

The MethodInterceptor interface is a standard AOP Alliance interface for implementing around advice for method invocation joinpoints. The MethodInvocation object represents the method invocation that is being advised, and using this object, we control when the method invocation is allowed to proceed. Because this is around advice, we are capable of performing actions before the method is invoked as well as after it is invoked but before it returns. So in the previous code snippet, we simply write James to console output, invoke the method with a call to invocation.proceed(), and then write ! to console output.

The final step in this sample is to weave the AgentDecorator advice (more specifically, the invoke() method) into the code. To do this, we create an instance of Agent, the target, and then create a proxy of this instance, instructing the proxy factory to weave in the AgentDecorator advice. This is shown here:

```
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();
    }
}
```

The important part here is that we use the ProxyFactory class to create the proxy of the target object, weaving in the advice at the same time. We pass the AgentDecorator advice to the ProxyFactory with a call to addAdvice() and specify the target for weaving with a call to setTarget(). Once the target is set and some advice is added to the ProxyFactory, we generate the proxy with a call to getProxy(). Finally, we call speak() on both the original target object and the proxy object. Running the previous code results in the following output:

```
Bond
James Bond!
```

As you can see, calling speak() on the untouched target object results in a standard method invocation, and no extra content is written to console output. However, the invocation of the proxy causes the code in AgentDecorator to execute, creating the desired James Bond! output. From this example, you can see that the advised class had no dependencies on Spring or the AOP Alliance interfaces; the beauty of Spring AOP, and indeed AOP in general, is that you can advise almost any class, even if that class was created without AOP in mind. The only restriction, in Spring AOP at least, is that you can't advise final classes because they cannot be overridden and therefore cannot be proxied.

Spring AOP Architecture

The core architecture of Spring AOP is based on proxies. When you want to create an advised instance of a class, you must use ProxyFactory to create a proxy instance of that class, first providing ProxyFactory with all the aspects that you want to be woven into the proxy. Using ProxyFactory is a purely programmatic approach to creating AOP proxies. For the most part, you don't need to use this in your application; instead, you can rely on the declarative AOP configuration mechanisms provided by Spring (the ProxyFactoryBean class, the aop namespace, and @AspectJ-style annotations) to take advantage of declarative proxy creation. However, it is important to understand how proxy creation works, so we will first show the programmatic approach to proxy creation and then dive into Spring's declarative AOP configurations.

At runtime, Spring analyzes the crosscutting concerns defined for the beans in ApplicationContext and generates proxy beans (which wrap the underlying target bean) dynamically. Instead of calling the target bean directly, callers are injected with the proxied bean. The proxy bean then analyzes the running condition (that is, joinpoint, pointcut, or advice) and weaves in the appropriate advice accordingly. Figure 5-1 shows a high-level view of a Spring AOP proxy in action. Internally, Spring has two proxy implementations: JDK dynamic proxies and CGLIB proxies. By default, when the target object to be advised implements an interface, Spring will use a JDK dynamic proxy to create proxy instances of the target. However, when the advised target object doesn't implement an interface (for example, it's a concrete class), CGLIB will be used for proxy instance creation. One major reason is that the JDK dynamic proxy supports only the proxying of interfaces. We discuss proxies in detail in the section "Understanding Proxies."

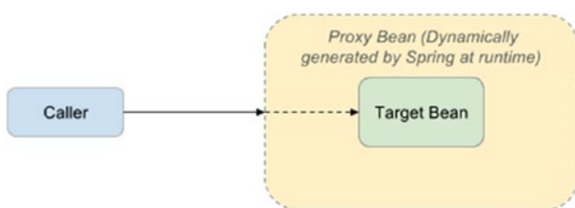

Figure 5-1. *Spring AOP proxy in action*

Joinpoints in Spring

One of the more noticeable simplifications in Spring AOP is that it supports only one joinpoint type: method invocation. At first glance, this might seem like a severe limitation if you are familiar with other AOP implementations such as AspectJ, which supports many more joinpoints, but in fact this makes Spring more accessible.

The method invocation joinpoint is by far the most useful joinpoint available, and using it, you can achieve many of the tasks that make AOP useful in day-to-day programming. Remember that if you need to advise some code at a joinpoint other than a method invocation, you can always use Spring and AspectJ together.

Aspects in Spring

In Spring AOP, an aspect is represented by an instance of a class that implements the `Advisor` interface. Spring provides convenience `Advisor` implementations that you can reuse in your applications, thus removing the need for you to create custom `Advisor` implementations. There are two subinterfaces of `Advisor`: `PointcutAdvisor` and `IntroductionAdvisor`.

The `PointcutAdvisor` interface is implemented by all `Advisor` implementations that use pointcuts to control the advice applied to joinpoints. In Spring, introductions are treated as special kinds of advice, and by using the `IntroductionAdvisor` interface, you can control those classes to which an introduction applies.

We discuss `PointcutAdvisor` implementations in detail in the upcoming section "Advisors and Pointcuts in Spring."

About the ProxyFactory Class

The `ProxyFactory` class controls the weaving and proxy creation process in Spring AOP. Before you can create a proxy, you must specify the advised or target object. You can do this, as you saw earlier, by using the `setTarget()` method. Internally, `ProxyFactory` delegates the proxy creation process to an instance of `DefaultAopProxyFactory`, which in turn delegates to either `Cglib2AopProxy` or `JdkDynamicAopProxy`, depending on the settings of your application. We discuss proxy creation in more detail later in this chapter.

The `ProxyFactory` class provides the `addAdvice()` method that you saw in the previous code sample for cases where you want advice to apply to the invocation of all methods in a class, not just a selection. Internally, `addAdvice()` wraps the advice you pass it in an instance of `DefaultPointcutAdvisor`, which is the standard implementation of `PointcutAdvisor`, and configures it with a pointcut that includes all methods by default. When you want more control over the `Advisor` that is created or when you want to add an introduction to the proxy, create the `Advisor` yourself and use the `addAdvisor()` method of the `ProxyFactory`.

You can use the same `ProxyFactory` instance to create many proxies, each with different aspects. To help with this, `ProxyFactory` has `removeAdvice()` and `removeAdvisor()` methods, which allow you to remove any advice or advisors from the `ProxyFactory` that you previously passed to it. To check whether a `ProxyFactory` has particular advice attached to it, call `adviceIncluded()`, passing in the advice object for which you want to check.

Creating Advice in Spring

Spring supports six flavors of advice, described in Table 5-1.

Table 5-1. *Advice Types in Spring*

Advice Name	Interface	Description
Before	org.springframework.aop.MethodBeforeAdvice	Using before advice, you can perform custom processing before a joinpoint executes. Because a joinpoint in Spring is always a method invocation, this essentially allows you to perform preprocessing before the method executes. Before advice has full access to the target of the method invocation as well as the arguments passed to the method, but it has no control over the execution of the method itself. If the before advice throws an exception, further execution of the interceptor chain (as well as the target method) will be aborted, and the exception will propagate back up the interceptor chain.
After-Returning	org.springframework.aop.AfterReturningAdvice	After-returning advice is executed after the method invocation at the joinpoint has finished executing and has returned a value. The after- returning advice has access to the target of the method invocation, the arguments passed to the method, and the return value. Because the method has already executed when the after-returning advice is invoked, it has no control over the method invocation at all. If the target method throws an exception, the after-returning advice will not be run, and the exception will be propagated up to the call stack as usual.
After(finally)	org.springframework.aop.AfterAdvice	After-returning advice is executed only when the advised method completes normally. However, the after (finally) advice will be executed no matter the result of the advised method. The advice is executed even when the advised method fails and an exception is thrown.
Around	org.aopalliance.intercept.MethodInterceptor	In Spring, around advice is modeled using the AOP Alliance standard of a method interceptor. Your advice is allowed to execute before and after the method invocation, and you can control the point at which the method invocation is allowed to proceed. You can choose to bypass the method altogether if you want, providing your own implementation of the logic.
Throws	org.springframework.aop.ThrowsAdvice	Throws advice is executed after a method invocation returns, but only if that invocation threw an exception. It is possible for throws advice to catch only specific exceptions, and if you choose to do so, you can access the method that threw the exception, the arguments passed into the invocation, and the target of the invocation.
Introduction	org.springframework.aop.IntroductionInterceptor	Spring models introductions as special types of interceptors. Using an introduction interceptor, you can specify the implementation for methods that are being introduced by the advice.

Interfaces for Advice

From our previous discussion of the ProxyFactory class, recall that advice is added to a proxy either directly, by using the addAdvice() method, or indirectly, by using an Advisor implementation with the addAdvisor() method. The main difference between advice and an advisor is that an advisor carries advice with the associated pointcut, which provides more fine-grained control on which joinpoints the advice will intercept. With regard to advice, Spring has created a well-defined hierarchy for Advice interfaces. This hierarchy is based on the AOP Alliance interfaces and is shown in detail in Figure 5-2.

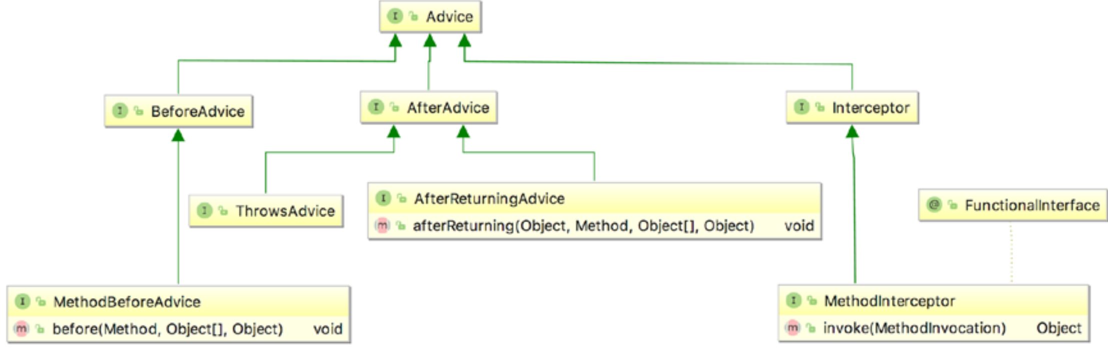

Figure 5-2. *Interfaces for Spring advice types as depicted in IntelliJ IDEA*

This kind of hierarchy has the benefit of not only being sound OO design but also enabling you to deal with advice types generically, such as by using a single addAdvice() method on the ProxyFactory, and you can add new advice types easily without having to modify the ProxyFactory class.

Creating Before Advice

Before advice is one of the most useful advice types available in Spring. This advice can modify the arguments passed to a method and can prevent the method from executing by raising an exception. In this section, we show you two simple examples of using before advice: one that writes a message to console output containing the name of the method before the method executes and another that you can use to restrict access to methods on an object. In the following code snippet, you can see the code for the SimpleBeforeAdvice class:

```
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.");
    }
}
```

Class Guitarist is simple and has just one method, sing(), that prints out a lyric in the console. It extends the Singer interface, which will be used all throughout the book.

```
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);
        }
}
```

Basically, the advice makes sure the johnMayer bean tunes his guitar before singing. In this code, you can see that we have advised an instance of the Guitarist class that we created earlier with an instance of the SimpleBeforeAdvice class. The MethodBeforeAdvice interface, which is implemented by SimpleBeforeAdvice, defines a single method, before(), which the AOP framework calls before the method at the joinpoint is invoked. Remember that, for now, we are using the default pointcut provided by the addAdvice() method, which matches all methods in a class. The before() method is passed three arguments: the method that is to be invoked, the arguments that will be passed to that method, and the Object that is the target of the invocation. The SimpleBeforeAdvice class uses the Method argument of the before() method to write a message to console output containing the name of the method to be invoked. Running this example gives we the following output:

```
Before 'sing', tune guitar.
You're gonna live forever in me
```

As you can see, the output from the call to sing() is shown, but just before it, you can see the output generated by SimpleBeforeAdvice.

Securing Method Access by Using Before Advice

In this section, we are going to implement before advice that checks user credentials before allowing the method invocation to proceed. If the user credentials are invalid, an exception is thrown by the advice, thus preventing the method from executing. The example in this section is simplistic. It allows users to authenticate with any password, and it also allows only a single, hard-coded user access to the secured methods. However, it does illustrate how easy it is to use AOP to implement a crosscutting concern such as security.

⚠ This is just an example demonstrating the use of before advice. For securing the method execution of Spring beans, the Spring Security project already provides comprehensive support; you don't need to implement the features by yourself.

The following code snippet shows the SecureBean class. This is the class that we will be securing using AOP.

```
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");
    }
}
```

The SecureBean class imparts a small pearl of wisdom from Homer Simpson, and it's wisdom that we don't want everyone to see. Because this example requires users to authenticate, we are somehow going to need to store their details. The following code snippet shows the UserInfo class we can use to store a user's credentials:

```
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;
    }
}
```

This class simply holds data about the user so that we can do something useful with it. The following code snippet shows the SecurityManager class, which is responsible for authenticating users and storing their credentials for later retrieval:

```
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();
    }
}
```

The application uses the SecurityManager class to authenticate a user and, later, to retrieve the details of the currently authenticated user. The application authenticates a user by using the login() method. This is only a mock implementation. In a real application, the login() method would probably check the supplied credentials against a database or LDAP directory, but here we can assume that all users are allowed to authenticate. The login() method creates a UserInfo object for the user and stores it on the current thread by using ThreadLocal. The logout() method sets any value that might be stored in ThreadLocal to null. Finally, the getLoggedOnUser() method returns the UserInfo object for the currently authenticated user. This method returns null if no user is authenticated.

To check whether a user is authenticated and, if so, whether the user is permitted to access the methods on Secure-Bean, we need to create advice that executes before the method and checks the UserInfo object returned by Security-Manager.getLoggedOnUser() against the set of credentials for allowed users. The code for this advice, SecurityAdvice, is shown here:

```
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());
        }
    }
}
```

The SecurityAdvice class creates an instance of SecurityManager in its constructor and then stores this instance in a field. You should note that the application and SecurityAdvice don't need to share the same SecurityManager instance because all data is stored with the current thread by using ThreadLocal. In the before() method, we perform a simple check to see whether the username of the authenticated user is John. If so, we allow the user access; otherwise, an exception is raised. Also notice that we check for a null UserInfo object, which indicates that the current user is not authenticated.

In the following code snippet, you can see a sample application that uses the SecurityAdvice class to secure the SecureBean class:

```
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;
    }
}
```

In the getSecureBean() method, we create a proxy of the SecureBean class that is advised using an instance of SecurityAdvice. This proxy is returned to the caller. When the caller invokes any method on this proxy, the call is first routed to the instance of SecurityAdvice for a security check. In the main() method, we test three scenarios, invoking the SecureBean.writeSecureMessage() method with two sets of user credentials and then no user credentials at all. Because SecurityAdvice allows method calls to proceed only if the currently authenticated user is John, we can expect that the only successful scenario in the previous code is the first. Running this example gives the following output:

```
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
```

As you can see, only the first invocation of SecureBean.writeSecureMessage() was allowed to proceed. The remaining invocations were prevented by the SecurityException exception thrown by SecurityAdvice. This example is simple, but it does highlight the usefulness of before advice. Security is a typical example of before advice, but we also find it useful when a scenario demands the modification of arguments passed to the method.

Creating After-Returning Advice

After-returning advice is executed after the method invocation at the joinpoint returns. Given that the method has already executed, you can't change the arguments that are passed to it. Although you can read these arguments, you can't change the execution path, and you can't prevent the method from executing. These restrictions are expected; what is not expected, however, is that you cannot modify the return value in the after-returning advice. When using after-returning advice, you are limited to adding processing. Although after-returning advice cannot modify the return value of a method invocation, it can throw an exception that can be sent up the stack instead of the return value.

In this section, we look at two examples of using after-returning advice in an application. The first example simply writes a message to console output after the method has been invoked. The second example shows how you can use after-returning advice to add error checking to a method. Consider a class, KeyGenerator, which generates keys for cryptographic purposes. Many cryptographic algorithms suffer from the problem that a small number of keys are considered weak. A weak key is any key whose characteristics make it significantly easier to derive the original message without knowing the key. For the DES algorithm, there are a total of 256 possible keys. From this key space, 4 keys are considered weak, and another 12 are considered semiweak. Although the chance of one of these keys being generated randomly is small (1 in 252), testing for the keys is so simple that it's worthwhile to do so. In the second example of this section, we build after-returning advice that checks for weak keys generated by KeyGenerator, raising an exception if one is found.

⚠ For more information on weak keys and cryptography at large, we recommend a visit to William Stallings' site at http://williamstallings.com/Cryptography/.

In the following code snippet, you can see the SimpleAfterReturningAdvice class, which demonstrates the use of after-returning advice by writing a message to console output after a method has returned. The previously introduced class Guitarist is reused here.

```
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.");
    }
}
```

Notice that the AfterReturningAdvice interface declares a single method, afterReturning(), which is passed the return value of method invocation, a reference to the method that was invoked, the arguments that were passed to the method, and the target of the invocation. Running this example results in the following output:

```
You're gonna live forever in me
After 'sing' put down guitar.
```

The output is similar to that of the before-advice example except that, as expected, the message written by the advice appears after the message written by the writeMessage() method. A good use of after-returning advice is to perform some additional error checking when it is possible for a method to return an invalid value.

In the scenario we described earlier, it is possible for a cryptographic key generator to generate a key that is considered weak for a particular algorithm. Ideally, the key generator would check for these weak keys, but since the chance of these keys arising is often very small, many generators do not check. By using after-returning advice, we can advise the method that generates the key and performs this additional check. The following is an extremely primitive key generator:

```
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;
    }
}
```

This key generator should not be considered secure. It's purposely simple for this example and has a one-in-three chance of producing a weak key. In the following snippet, you can see WeakKeyCheckAdvice, which checks to see whether the result of the getKey() method is a weak key:

```
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");
            }
        }
    }
}
```

In the afterReturning() method, we check first to see whether the method that was executed at the joinpoint was the getKey() method. If so, we then check the result value to see whether it was the weak key. If we find that the result of the getKey() method was a weak key, then we throw a SecurityException to inform the calling code of this. The following code snippet shows a simple application that demonstrates the use of this advice:

```
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!");
            }
        }
    }
}
```

After creating an advised proxy of the KeyGenerator target, the AfterAdviceDemo class attempts to generate ten keys. If a SecurityException is thrown during a single generation, a message is written to the console, informing the user that a weak key was generated; otherwise, the generated key is displayed. A single run of this on our machine generated the following output:

```
Key: 48658904092028502
Weak Key Generated!
Key: 48658904092028502
Weak Key Generated!
Weak Key Generated!
Weak Key Generated!
Key: 48658904092028502
Key: 48658904092028502
Key: 48658904092028502
Key: 48658904092028502
```

As you can see, the KeyGenerator class sometimes generates weak keys, as expected, and WeakKeyCheckAdvice ensures that SecurityException is raised whenever a weak key is encountered.

Creating Around Advice

Around advice functions like a combination of before and after advice, with one big difference: you can modify the return value. Not only that, but you can prevent the method from executing. This means that by using around advice, you can essentially replace the entire implementation of a method with new code. Around advice in Spring is modeled as an interceptor using the MethodInterceptor interface. There are many uses for around advice, and you will find that many features of Spring are created by using method interceptors, such as the remote proxy support and the transaction management features. Method interception is also a good mechanism for profiling the execution of your application, and it forms the basis of the example in this section.

We are not going to see how to build a simple example for method interception; instead, we refer to the first example using the Agent class, which shows how to use a basic method interceptor to write a message on either side of a method invocation. Notice from this earlier example that the invoke() method of the MethodInterceptor interface does not provide the same set of arguments as MethodBeforeAdvice and AfterReturningAdvice. The method is not passed the target of the invocation, the method that was invoked, or the arguments used. However, you can access this data by using the MethodInvocation object that is passed to invoke(). You will see a demonstration of this in the following example.

For this example, we want to achieve some way to advise a class so we get basic information about the runtime performance of its methods. Specifically, we want to know how long the method took to execute. To achieve this, we can use the StopWatch class included in Spring, and we clearly need a MethodInterceptor, because we need to start StopWatch before the method invocation and stop it right afterward.

The following code snippet shows the WorkerBean class that we are going to profile by using the StopWatch class and around advice:

```
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("");
    }
}
```

This is a simple class. The doSomeWork() method accepts a single argument, noOfTimes, and calls the work() method exactly the number of times specified by this method. The work() method simply has a dummy call to System.out.print(), which passes in an empty String. This prevents the compiler from optimizing the work() method and thus the call to work().

In the following snippet, you can see the ProfilingInterceptor class that uses the StopWatch class to profile method invocation times. We use this interceptor to profile the WorkerBean class shown previously.

```
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");
    }
}
```

In the invoke() method, which is the only method in the MethodInterceptor interface, we create an instance of StopWatch and then start it running immediately, allowing the method invocation to proceed with a call to MethodInvocation.proceed(). As soon as the method invocation has ended and the return value has been captured, we stop StopWatch and pass the total number of milliseconds taken, along with the MethodInvocation object, to the dumpInfo() method. Finally, we return the Object returned by MethodInvocation.proceed() so that the caller obtains the correct return value. In this case, we did not want to disrupt the call stack in any way; we were simply acting as an eavesdropper on the method invocation. If we had wanted, we could have changed the call stack completely, redirecting the method call to another object or a remote service, or we could simply have reimplemented the method logic inside the interceptor and returned a different return value.

The dumpInfo() method simply writes some information about the method call to console output, along with the time taken for the method to execute. In the first three lines of dumpInfo(), you can see how you can use the MethodInvocation object to determine the method that was invoked, the original target of the invocation, and the arguments used.

The following code sample shows the ProfilingDemo class that first advises an instance of WorkerBean with a ProfilingInterceptor and then profiles the doSomeWork() method.

```
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();
    }
}
```

Running this example on our machine produces the following output:

```
Executed method: doSomeWork
On object of type: com.apress.prospring5.ch5.WorkerBean
With arguments:
       > 10000000
Took: 1139 ms
```

From this output, you can see which method was executed, what the class of the target was, what arguments were passed in, and how long the invocation took.

Creating Throws Advice

Throws advice is similar to after-returning advice in that it executes after the joinpoint, which is always a method invocation, but throws advice executes only if the method throws an exception. Throws advice is also similar to after-returning advice in that it has little control over program execution. If you are using throws advice, you can't choose to ignore the exception that was raised and return a value for the method instead. The only modification you can make to the program flow is to change the type of exception that is thrown. This is quite a powerful concept and can make application development much simpler. Consider a situation where you have an API that throws an array of poorly defined exceptions. Using throws advice, you can advise all classes in that API and reclassify the exception hierarchy into something more manageable and descriptive. Of course, you can also use throws advice to provide centralized error logging across your application, thus reducing the amount of error-logging code that is spread across your application.

As you saw from the diagram in Figure 5-2, throws advice is implemented by the `ThrowsAdvice` interface. Unlike the interfaces you have seen so far, `ThrowsAdvice` does not define any methods; instead, it is simply a marker interface used by Spring. The reason for this is that Spring allows typed throws advice, which allows you to define exactly which `Exception` types your throws advice should catch. Spring achieves this by detecting methods with certain signatures using reflection. Spring looks for two distinct method signatures. This is best demonstrated with a simple example. The following code snippet shows a simple bean with two methods that both simply throw exceptions of different types:

```
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");
    }
}
```

Here you can see the SimpleThrowsAdvice class that demonstrates both of the method signatures that Spring looks for on throws advice:

```
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");
    }
}
```

The first thing Spring looks for in throws advice is one or more public methods called `afterThrowing()`. The return type of the methods is unimportant, although we find it best to stick with `void` because this method can't return any meaningful value. The first `afterThrowing()` method in the `SimpleThrowsAdvice` class has a single argument of type `Exception`. You can specify any type of `Exception` as the argument, and this method is ideal when you are not concerned about the method that threw the exception or the arguments that were passed to it. Note that this method catches `Exception` and any subtypes of `Exception` unless the type in question has its own `afterThrowing()` method.

In the second `afterThrowing()` method, we declared four arguments to catch the method that threw the exception, the arguments that were passed to the method, and the target of the method invocation. The order of the arguments in this method is important, and you must specify all four. Notice that the second `afterThrowing()` method catches exceptions of type `IllegalArgumentException` (or its subtype). Running this example produces the following output:

```
***
Generic Exception Capture
Caught: java.lang.Exception
***

***
IllegalArgumentException Capture
Caught: java.lang.IllegalArgumentException
Method: otherErrorProneMethod
***
```

As you can see, when a plain old `Exception` is thrown, the first `afterThrowing()` method is invoked, but when an `IllegalArgumentException` is thrown, the second `afterThrowing()` method is invoked. Spring invokes a `afterThrowing()` method only for each `Exception`, and as you saw from the example in class `SimpleThrowsAdvice`, Spring uses the method whose signature contains the best match for the `Exception` type. In the situation where your after-throwing advice has two `afterThrowing()` methods, both declared with the same `Exception` type but one with a single argument and the other with four arguments, Spring invokes the four-argument `afterThrowing()` method.

After-throwing advice is useful in a variety of situations; it allows you to reclassify entire `Exception` hierarchies as well as build centralized `Exception` logging for your application. We have found that after-throwing advice is particularly useful when we are debugging a live application because it allows us to add extra logging code without needing to modify the application's code.

Choosing an Advice Type

In general, choosing an advice type is driven by the requirements of your application, but you should choose the most specific advice type for your need. That is to say, don't use around advice when before advice will do. In most cases, around advice can accomplish everything that the other three advice types can, but it may be overkill for what you are trying to achieve. By using the most specific type of advice, you are making the intention of your code clearer, and you are also reducing the possibility of errors. Consider advice that counts method calls. When you are using before advice, all you need to code is the counter, but with around advice, you need to remember to invoke the method and return the value to the caller. These small things can allow spurious errors to creep into your application. By keeping the advice type as focused as possible, you reduce the scope for errors.

Advisors and Pointcuts in Spring

Thus far, all the examples you have seen have used the ProxyFactory class. This class provides a simple way of obtaining and configuring AOP proxy instances in custom user code. The ProxyFactory.addAdvice() method is to configure advice for a proxy. This method delegates to addAdvisor() behind the scenes, creating an instance of DefaultPointcutAdvisor and configuring it with a pointcut that points to all methods. In this way, the advice is deemed to apply to all methods on the target. In some cases, such as when you are using AOP for logging purposes, this may be desirable, but in other cases you may want to limit the methods to which the advice applies.

Of course, you could simply perform the checking in the advice itself that the method being advised is the correct one, but this approach has several drawbacks. First, hard-coding the list of acceptable methods into the advice reduces the advice's reusability. By using pointcuts, you can configure the methods to which an advice applies, without needing to put this code inside the advice; this clearly increases the reuse value of the advice. Other drawbacks with hard-coding the list of methods into the advice are performance related. To inspect the method being advised in the advice, you need to perform the check each time any method on the target is invoked. This clearly reduces the performance of your application. When you use pointcuts, the check is performed once for each method, and the results are cached for later use. The other performance-related drawback of not using pointcuts to restrict the list-advised methods is that Spring can make optimizations for nonadvised methods when creating a proxy, which results in faster invocations on nonadvised methods. These optimizations are covered in greater detail when we discuss proxies later in the chapter.

We strongly recommend that you avoid the temptation to hard-code method checks into your advice and instead use pointcuts wherever possible to govern the applicability of advice to methods on the target. That said, in some cases it is necessary to hard-code the checks into your advice. Consider the earlier example of the after-returning advice designed to catch weak keys generated by the KeyGenerator class. This kind of advice is closely coupled to the class it is advising, and it is wise to check inside the advice to ensure that it is applied to the correct type. We refer to this coupling between advice and target as *target affinity*. In general, you should use pointcuts when your advice has little or no target affinity. That is, it can apply to any type or a wide range of types. When your advice has strong target affinity, try to check that the advice is being used correctly in the advice itself; this helps reduce head-scratching errors when advice is misused. We also recommend you avoid advising methods needlessly. As you will see, this results in a noticeable drop in invocation speed that can have a large impact on the overall performance of your application.

The Pointcut Interface

Pointcuts in Spring are created by implementing the Pointcut interface, which is shown here:

```
package org.springframework.aop;

public interface Pointcut {
    ClassFilter getClassFilter ();
    MethodMatcher getMethodMatcher();
}
```

As you can see from this code, the Pointcut interface defines two methods, getClassFilter() and getMethodMatcher(), which return instances of ClassFilter and MethodMatcher, respectively. Obviously, if you choose to implement the Pointcut interface, you will need to implement these methods. Thankfully, as you will see in the next section, this is usually unnecessary because Spring provides a selection of Pointcut implementations that cover most, if not all, of your use cases.

When determining whether a Pointcut applies to a particular method, Spring first checks to see whether the Pointcut interface applies to the method's class by using the ClassFilter instance returned by Pointcut.getClassFilter(). Here is the ClassFilter interface:

```
org.springframework.aop;

public interface ClassFilter {
    boolean matches(Class<?> clazz);
}
```

As you can see, the ClassFilter interface defines a single method, matches(), that is passed an instance of Class that represents the class to be checked. As you have no doubt determined, the matches() method returns true if the pointcut applies to the class and false otherwise.

The MethodMatcher interface is more complex than the ClassFilter interface, as shown here:

```
package org.springframework.aop;

public interface MethodMatcher {
    boolean matches(Method m, Class<?> targetClass);
    boolean isRuntime();
    boolean matches(Method m, Class<?> targetClass, Object[] args);
}
```

Spring supports two types of MethodMatcher, static and dynamic, which are determined by the return value of isRuntime(). Before using MethodMatcher, Spring calls isRuntime() to determine whether MethodMatcher is static, indicated by a return value of false, or dynamic, indicated by a return value of true.

For a static pointcut, Spring calls the matches(Method, Class<T>) method of the MethodMatcher once for every method on the target, caching the return value for subsequent invocations of those methods. In this way, the check for method applicability is performed only once for each method, and subsequent invocations of a method do not result in an invocation of matches().

With dynamic pointcuts, Spring still performs a static check by using matches(Method, Class<T>) the first time a method is invoked to determine the overall applicability of a method. However, in addition to this and provided that the static check returned true, Spring performs a further check for each invocation of a method by using the matches(Method, Class<T>, Object[]) method. In this way, a dynamic MethodMatcher can determine whether a pointcut should apply based on a particular invocation of a method, not just on the method itself. For example, a pointcut needs to be applied only when the argument is an Integer with a value larger than 100. In this case, the matches(Method, Class<T>, Object[]) method can be coded to perform further checking on the argument for each invocation.

Clearly, static pointcuts perform much better than dynamic pointcuts because they avoid the need for an additional check per invocation. Dynamic pointcuts provide a greater level of flexibility for deciding whether to apply advice. In general, we recommend you use static pointcuts wherever you can. However, in cases where your advice adds substantial overhead, it may be wise to avoid any unnecessary invocations of your advice by using a dynamic pointcut.

In general, you rarely create your own Pointcut implementations from scratch because Spring provides abstract base classes for both static and dynamic pointcuts. We will look at these base classes, along with other Pointcut implementations, over the next few sections.

Available Pointcut Implementations

As of version 4.0, Spring provides eight implementations of the `Pointcut` interface: two abstract classes intended as convenience classes for creating static and dynamic pointcuts, and six concrete classes, one for each of the following:

- Composing multiple pointcuts together
- Handling control flow pointcuts
- Performing simple name-based matching
- Defining pointcuts using regular expressions
- Defining pointcuts using AspectJ expressions
- Defining pointcuts that look for specific annotations at the class or method level

Table 5-2 summarizes the eight `Pointcut` interface implementations.

Table 5-2. *Summary of Spring Pointcut Implementations*

Implementation Class	Description
`org.springframework.aop.support.annotation.AnnotationMatchingPointcut`	This implementation looks for a specific Java annotation on a class or method. This class requires JDK 5 or higher.
`org.springframework.aop.aspectj.AspectJExpressionPointcut`	This implementation uses an AspectJ weaver to evaluate a pointcut expression in AspectJ syntax.
`org.springframework.aop.support.ComposablePointcut`	The `ComposablePointcut` class is used to compose two or more pointcuts together with operations such as `union()` and `intersection()`.
`org.springframework.aop.support.ControlFlowPointcut`	`ControlFlowPointcut` is a special case pointcut that matches all methods within the control flow of another method, that is, any method that is invoked either directly or indirectly as the result of another method being invoked.
`org.springframework.aop.support.DynamicMethodMatcherPointcut`	This implementation is intended as a base class for building dynamic pointcuts.
`org.springframework.aop.support.JdkRegexpMethodPointcut`	This implementation allows you to define pointcuts using JDK 1.4 regular expression support. This class requires JDK 1.4 or newer.
`org.springframework.aop.support.NameMatchMethodPointcut`	Using `NameMatchMethodPointcut`, you can create a pointcut that performs simple matching against a list of method names.
`org.springframework.aop.support.StaticMethodMatcherPointcut`	The `StaticMethodMatcherPointcut` class is intended as a base for building static pointcuts.

Figure 5-3 shows the Unified Modeling Language (UML)[2] diagram for the Pointcut implementation classes.

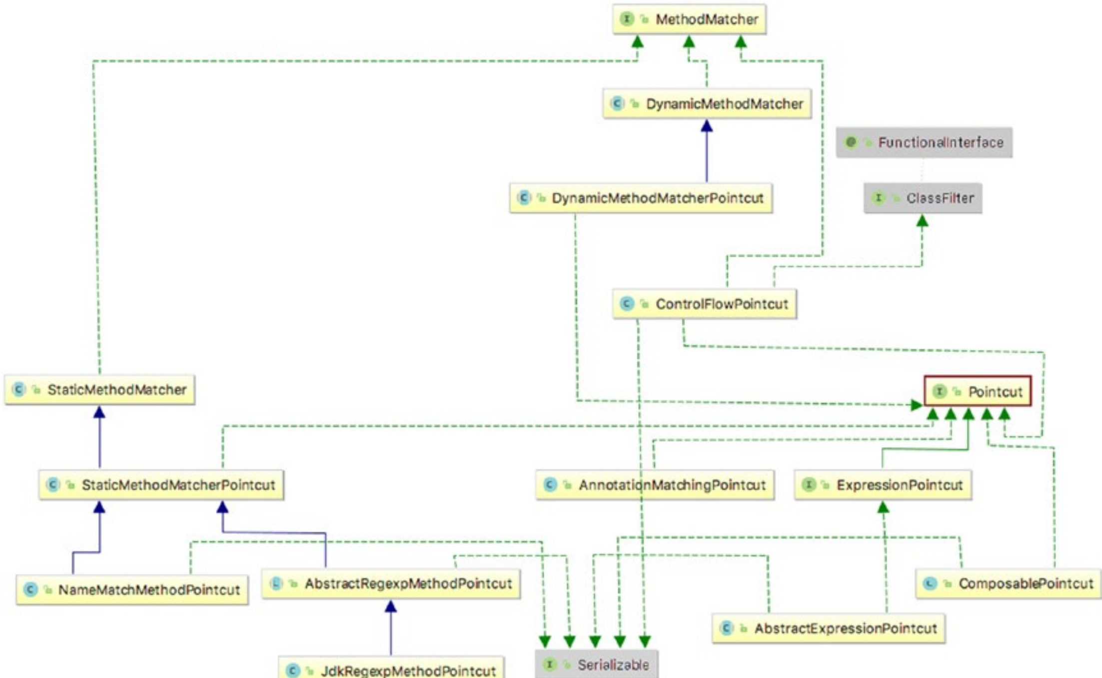

Figure 5-3. *Pointcut implementation classes represented as an UML diagram in Intellij IDEA*

Using DefaultPointcutAdvisor

Before you can use any Pointcut implementation, you must first create an instance of the Advisor interface, or more specifically a PointcutAdvisor interface. Remember from our earlier discussions that Advisor is Spring's representation of an aspect (see the previous section called "Aspects in Spring"), which is a coupling of advice and pointcuts that governs which methods should be advised and how they should be advised. Spring provides a number of implementations of PointcutAdvisor, but for now we will concern yourself with just one, DefaultPointcutAdvisor. This is a simple PointcutAdvisor for associating a single Pointcut with a single Advice.

Creating a Static Pointcut by Using StaticMethodMatcherPointcut

In this section, we will create a simple static pointcut by extending the abstract StaticMethodMatcherPointcut class. Since the StaticMethodMatcherPointcut class extends the StaticMethodMatcher class (an abstract class too), which implements the MethodMatcher interface, you are required to implement the method matches(Method, Class<?>). The rest of the Pointcut implementation is handled automatically. Although this is the only method you are required to implement (when extending the StaticMethodMatcherPointcut class), you may want to override the getClassFilter() method as shown in this example to ensure that only methods of the correct type get advised.

[2]UML is quite important in development because it is a way to simplify application logic and make it visual, thus making it easy to detect issues before the code is even written. It can also be used as documentation when introducing new members to a team, making them productive as soon as possible. You can find more information at www.uml.org/.

For this example, we have two classes, GoodGuitarist and GreatGuitarist, with identical methods defined in both, which are implementations of the method in interface 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");
        }
}
```

With this example, we want to be able to create a proxy of both classes by using the same DefaultPointcutAdvisor but have the advice apply only to the sing() method of the GoodGuitarist class. To do this, we created the SimpleStaticPointcut class as shown here:

```
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);
    }
}
```

Here you can see that we implement the matches(Method, Class<?>) method as required by the StaticMethodMatcher abstract class. The implementation simply returns true if the name of the method is sing; otherwise, it returns false. Using lambda expressions, the creation of an anonymous class implementing ClassFilter in the getClassFilter() method is hidden in the previous code sample. The expanded lambda expression looks like this:

```
public ClassFilter getClassFilter() {
        return new ClassFilter() {
            public boolean matches(Class<?> cls) {
                return (cls == GoodGuitarist.class);
            }
        };
    }
```

Notice that we have also overridden the getClassFilter() method to return a ClassFilter instance whose matches() method returns true only for the GoodGuitarist class. With this static pointcut, we are saying that only methods of the GoodGuitarist class will be matched, and furthermore, only the sing() method of that class will be matched.

The following code snippet shows the SimpleAdvice class that simply writes out a message on either side of the method invocation:

```
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;
    }
}
```

In the following code snippet, you can see a simple driver application for this example that creates an instance of DefaultPointcutAdvisor by using the SimpleAdvice and SimpleStaticPointcut classes. Also, because both classes implement the same interface, you can see that the proxies can be created based on the interface, not on the concrete classes.

```
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();
    }
}
```

Notice that the `DefaultPointcutAdvisor` instance is then used to create two proxies: one for an instance of `GoodGuitarist` and one for an instance of `EricClapton`. Finally, the `sing()` method is invoked on the two proxies. Running this example results in the following output:

```
>> 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
```

As you can see, the only method for which `SimpleAdvice` was actually invoked was the `sing()` method for the `GoodGuitarist` class, exactly as expected. Restricting the methods to which advice applies is quite simple and, as you will see when we discuss proxy options, is key to getting the best performance out of your application.

Creating a Dynamic Pointcut by Using DyanmicMethod MatcherPointcut

Creating a dynamic pointcut is not much different from creating a static one, so for this example, we will create a dynamic pointcut for the class shown here:

```
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()");
    }
}
```

For this example, we want to advise only the `foo()` method, but unlike the previous example, we want to advise this method only if the `int` argument passed to it is greater or less than 100.

As with static pointcuts, Spring provides a convenient base class for creating dynamic pointcuts: `DynamicMethodMatcherPointcut`. The `DynamicMethodMatcherPointcut` class has a single abstract method, `matches(Method, Class<?>, Object[])` (via the `MethodMatcher` interface that it implements), that you must implement, but as you will see, it is also prudent to implement the `matches(Method, Class<?>)` method to control the behavior of the static checks. The following code snippet shows the `SimpleDynamicPointcut` class:

```
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);
    }
}
```

As you can see from the previous code sample, we override the `getClassFilter()` method in a similar manner as in the previous section. This removes the need to check the class in the method-matching methods, which is something that is especially important for the dynamic check. Although you are required to implement only the dynamic check, we implement the static check as well. The reason for this is that you know the `bar()` method will never be advised. By indicating this by using the static check, Spring never has to perform a dynamic check for this method. This is because when the static check method is implemented, Spring will first check against it, and if the checking result is not a match, Spring will stop doing any further dynamic checking. Moreover, the result of the static check will be cached for better performance. But if

we neglect the static check, Spring performs a dynamic check each time the `bar()` method is invoked.
As a recommended practice, perform the class checking in the `getClassFilter()` method, the method checking in the `matches(Method, Class<?>)` method, and the argument checking in the `matches(Method, Class<?>, Object[])` method. This will make your pointcut much easier to understand and maintain, and performance will be better too.

In the `matches(Method, Class<?>, Object[])` method, you can see that we return `false` if the value of the `int` argument passed to the `foo()` method is not equal to 100; otherwise, we return `true`. Note that in the dynamic check, we know that we are dealing with the `foo()` method because no other method makes it past the static check. In the following code snippet, you can see an example of this pointcut in action:

```
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();
    }
}
```

Notice that we have used the same advice class as in the static pointcut example. However, in this example, only the first two calls to `foo()` should be advised. The dynamic check prevents the third call to `foo()` from being advised, and the static check prevents the `bar()` method from being advised. Running this example yields the following output:

```
Static check for bar
Static check for foo
Static check for toString
Static check for clone
Static check for foo
Dynamic check for foo
>> Invoking foo
Invoked foo() with: 1
>> Done
```

```
Dynamic check for foo
>> Invoking foo
Invoked foo() with: 10
>> Done

Dynamic check for foo
Invoked foo() with: 100
Static check for bar
Invoked bar()
Invoked bar()
Invoked bar()
```

As we expected, only the first two invocations of the `foo()` method were advised. Notice that none of the `bar()` invocations is subject to a dynamic check, thanks to the static check on `bar()`. An interesting point to note here is that the `foo()` method is subject to two static checks: one during the initial phase when all methods are checked and another when it is first invoked.

As you can see, dynamic pointcuts offer a greater degree of flexibility than static pointcuts, but because of the additional runtime overhead they require, you should use a dynamic pointcut only when absolutely necessary.

Using Simple Name Matching

Often when creating a pointcut, we want to match want to match based on just the name of the method, ignoring method signature and return type. In this case, you can avoid needing to create a subclass of `StaticMethodMatcherPointcut` and use `NameMatchMethodPointcut` (which is a subclass of `StaticMethodMatcherPointcut`) to match against a list of method names instead. When you are using `NameMatchMethodPointcut`, no consideration is given to the signature of the method, so if you have methods `sing()` and `sing(guitar)`, they are both matched for the name `foo`.

In the following code snippet, you can see the `GrammyGuitarist` class, which is yet another implementation of `Singer`, because this Grammy award singer sings using his voice, uses a guitar, and, being human, occasionally talks and rests.

```
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";
        }
}
```

For this example, we want to match the `sing()`, `sing(Guitar)`, and `rest()` methods by using `NameMatchMethodPointcut;`. This translates to matching the names `foo` and `bar`. This is shown in the following code snippet:

```
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();
    }
}
```

There is no need to create a class for the pointcut; you can simply create an instance of NameMatchMethodPointcut, and you are on your way. Notice that we have added two method names to the pointcut, sing and rest, using the addMethodName() method. Running this example results in the following output:

```
>> 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
```

As expected, the sing, sing(Guitar), and rest methods are advised, thanks to the pointcut, but the talk() method is left unadvised.

Creating Pointcuts with Regular Expressions

In the previous section, we discussed how to perform simple matching against a predefined list of methods. But what if you don't know all of the methods' names in advance, and instead you know the pattern that the names follow? For instance, what if you want to match all methods whose names start with get? In this case, you can use the regular expression pointcut JdkRegexpMethodPointcut to match a method name based on a regular expression. Here you can see yet another Guitarist class, which contains three methods:

```
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("Just keep me where  the light is");
        }

        public void rest() {
                System.out.println("zzz");
        }
}
```

Using a regular expression-based pointcut, we can match all methods in this class whose name starts with `string`. This is shown here:

```
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();
    }
}
```

Notice we do not need to create a class for the pointcut; instead, we just create an instance of `JdkRegexpMethodPointcut` and specify the pattern to match, and we are finished. The interesting thing to note is the pattern. When matching method names, Spring matches the fully qualified name of the method, so for `sing1()`, Spring is matching against `com.apress.prospring5.ch5.Guitarist.sing1`, which is why there's the leading `.*` in the pattern. This is a powerful concept because it allows you to match all methods within a given package, without needing to know exactly which classes are in that package and what the names of the methods are. Running this example yields the following output:

```
>> Invoking sing
Just keep me where the light is
>> Done

>> Invoking sing2
Oh gravity, stay the hell away from me
>> Done

zzz
```

As you would expect, only the `sing()` and `sing2()` methods have been advised because the `rest()` method does not match the regular expression pattern.

Creating Pointcuts with AspectJ Pointcut Expression

Besides JDK regular expressions, you can use AspectJ's pointcut expression language for pointcut declaration. Later in this chapter, you will see that when we declare the pointcut in XML configuration by using the aop namespace, Spring defaults to using AspectJ's pointcut language. Moreover, when using Spring's @AspectJ annotation-style AOP support, you need to use AspectJ's pointcut language. So when declaring pointcuts by using expression language, using an AspectJ pointcut expression is the best way to go. Spring provides the class AspectJExpressionPointcut for defining pointcuts via AspectJ's expression language. To use AspectJ pointcut expressions with Spring, you need to include two AspectJ library files, aspectjrt.jar and aspectjweaver.jar, in your project's classpath. The dependencies and their versions are configured in the main build.gradle configuration file (and configured as dependencies for all the modules of the chapter05 project).

```
ext {
    aspectjVersion = '1.9.0.BETA-5'
...
    misc = [
        ...
        Aspectjweaver     : "org.aspectj:aspectjweaver:$aspectjVersion",
        Aspectjrt         : "org.aspectj:aspectjrt:$aspectjVersion"
    ]
...
```

Considering the previous implementation of the Guitarist class, the same functionality implemented with JDK regular expressions can be implemented using an AspectJ expression. Here's the code for that:

```
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();
    }
}
```

Note that we use the AspectJExpressionPointcut's setExpression() method to set the matching criteria. The expression execution(* sing*(..)) means that the advice should apply to the execution of any methods that start with sing, have any arguments, and return any types. Running the program will get the same result as the previous example using JDK regular expressions.

Creating Annotation Matching Pointcuts

If your application is annotation-based, you may want to use your own specified annotations for defining pointcuts, that is, apply the advice logic to all methods or types with specific annotations. Spring provides the class AnnotationMatchingPointcut for defining pointcuts using annotations. Again, let's reuse the previous example and see how to do it when using an annotation as a pointcut.

First we define an annotation called AdviceRequired, which is an annotation that we will use for declaring a pointcut. The following code snippet shows the annotation class:

```
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 {
}
```

In the previous code sample, you can see that we declare the interface as an annotation by using @interface as the type, and the @Target annotation defines that the annotation can apply at either the type or method level. The following code snippet shows yet another Guitarist class implementation with your annotation on one of its methods:

```
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");
        }
}
```

The following code snippet shows the testing program:

```
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();
    }
}
```

In the previous listing, an instance of `AnnotationMatchingPointcut` is acquired by calling its static method `forMethodAnnotation()` and passing in the annotation type. This indicates that we want to apply the advice to all the methods annotated with the given annotation. It's also possible to specify annotations applied at the type level by calling the `forClassAnnotation()` method. The following shows the output when the program runs:

```
>> Invoking sing
play: G C G C Am D7
>> Done

zzz
```

As you can see, since we annotated the `sing()` method, only that method was advised.

Convenience Advisor Implementations

For many of the `Pointcut` implementations, Spring also provides a convenience `Advisor` implementation that acts as the pointcut. For instance, instead of using `NameMatchMethodPointcut` coupled with `DefaultPointcutAdvisor` in the previous example, we could simply have used `NameMatchMethodPointcutAdvisor`, as shown in the following code snippet:

```
package com.apress.prospring5.ch5;
...
import org.springframework.aop.support.NameMatchMethodPointcutAdvisor;
```

```
public class NamePointcutUsingAdvisor {
        public static void main(String... args) {
                GrammyGuitarist johnMayer = new GrammyGuitarist();

                NameMatchMethodPointcut pc = new NameMatchMethodPointcut();
                pc.addMethodName("sing");
                pc.addMethodName("rest");

                Advisor advisor =
                  new NameMatchMethodPointcutAdvisor(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();
        }
}
```

Notice that rather than creating an instance of NameMatchMethodPointcut, we configure the pointcut details on the instance of NameMatchMethodPointcutAdvisor. In this way, NameMatchMethodPointcutAdvisor is acting as both the advisor and the pointcut.

You can find full details of the different Advisor implementations by exploring the Javadoc for the org.springframework.aop.support package. There is no noticeable performance difference between the two approaches, and aside from there being slightly less code in the second example, there is very little difference in the actual coding approach. We prefer to stick with the first approach because we feel the intent is slightly clearer in the code. At the end of the day, the style you choose comes down to personal preference.

Understanding Proxies

So far, we have taken only a cursory look at the proxies generated by ProxyFactory. We mentioned that two types of proxy are available in Spring: JDK proxies created by using the JDK Proxy class and CGLIB-based proxies created by using the CGLIB Enhancer class. You may be wondering exactly what the difference between the two proxies is and why Spring needs two types of proxy. In this section, we take a detailed look at the differences between the proxies.

The core goal of a proxy is to intercept method invocations and, where necessary, execute chains of advice that apply to a particular method. The management and invocation of advice is largely proxy independent and is managed by the Spring AOP framework. However, the proxy is responsible for intercepting calls to all methods and passing them as necessary to the AOP framework for the advice to be applied.

In addition to this core functionality, the proxy must support a set of additional features. It is possible to configure the proxy to expose itself via the AopContext class (which is an abstract class) so that you can retrieve the proxy and invoke advised methods on the proxy from the target object. The proxy is responsible for ensuring that when this option is enabled via ProxyFactory.setExposeProxy(), the proxy class is appropriately exposed. In addition, all proxy classes implement the Advised interface by default, which allows for, among other things, the advice chain to be changed after the proxy has been created. A proxy must also ensure that any methods that return this (that is, return the proxied target) do in fact return the proxy and not the target.

As you can see, a typical proxy has quite a lot of work to perform, and all of this logic is implemented in both the JDK and CGLIB proxies.

Using JDK Dynamic Proxies

JDK proxies are the most basic type of proxy available in Spring. Unlike the CGLIB proxy, the JDK proxy can generate proxies only of interfaces, not classes. In this way, any object you want to proxy must implement at least one interface, and the resulting proxy will be an object that implements that interface. Figure 5-4 shows an abstract schema of such a proxy.

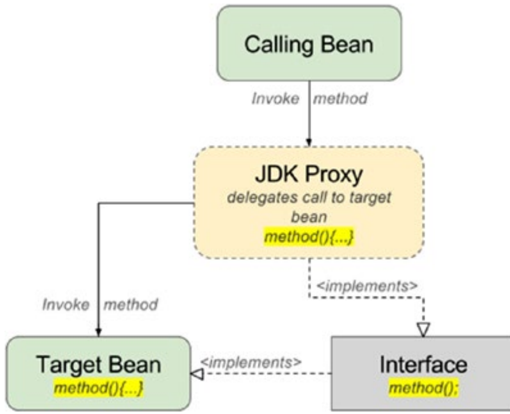

Figure 5-4. *JDK proxy abstract schema*

In general, it is good design to use interfaces for your classes, but it is not always possible, especially when you are working with third-party or legacy code. In this case, you must use the CGLIB proxy. When you are using the JDK proxy, all method calls are intercepted by the JVM and routed to the `invoke()` method of the proxy. This method then determines whether the method in question is advised (by the rules defined by the pointcut), and if so, it invokes the advice chain and then the method itself by using reflection. In addition to this, the `invoke()` method performs all the logic discussed in the previous section.

The JDK proxy makes no determination between methods that are advised and unadvised until it is in the `invoke()` method. This means that for unadvised methods on the proxy, the `invoke()` method is still called, all the checks are still performed, and the method is still invoked by using reflection. Obviously, this incurs runtime overhead each time the method is invoked, even though the proxy often performs no additional processing other than to invoke the unadvised method via reflection.

You can instruct `ProxyFactory` to use a JDK proxy by specifying the list of interfaces to proxy by using `setInterfaces()` (in the `AdvisedSupport` class that the `ProxyFactory` class extends indirectly).

Using CGLIB Proxies

With the JDK proxy, all decisions about how to handle a particular method invocation are handled at runtime each time the method is invoked. When you use CGLIB, CGLIB dynamically generates the bytecode for a new class on the fly for each proxy, reusing already generated classes wherever possible. The resulting proxy type in this case will be a subclass of the target object class. Figure 5-5 shows an abstract schema of such a proxy.

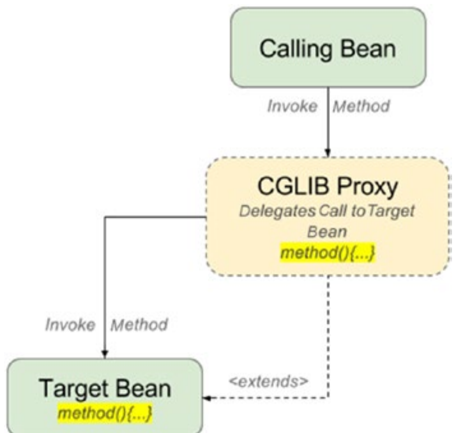

Figure 5-5. *CGLIB proxy abstract schema*

When a CGLIB proxy is first created, CGLIB asks Spring how it wants to handle each method. This means that many of the decisions that are performed in each call to `invoke()` on the JDK proxy are performed just once for the CGLIB proxy. Because CGLIB generates actual bytecode, there is also a lot more flexibility in the way you can handle methods. For instance, the CGLIB proxy generates the appropriate bytecode to invoke any unadvised methods directly, reducing the overhead introduced by the proxy. In addition, the CGLIB proxy determines whether it is possible for a method to return this; if not, it allows the method call to be invoked directly, again reducing the runtime overhead.

The CGLIB proxy also handles fixed-advice chains differently than the JDK proxy. A fixed-advice chain is one that you guarantee will not change after the proxy has been generated. By default, you are able to change the advisors and advice on a proxy even after it is created, although this is rarely a requirement. The CGLIB proxy handles fixed-advice chains in a particular way, reducing the runtime overhead for executing an advice chain.

Comparing Proxy Performance

So far, all we have done is discuss in loose terms the differences in implementation between the proxy types. In this section, we are going to run a simple test to compare the performance of the CGLIB proxy with the JDK proxy.

Let's create a class named `DefaultSimpleBean`, which we will use as the target object for proxying. Here are the `SimpleBean` interface and the `DefaultSimpleBean` class:

```
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();
    }
}
```

In the following example, the `TestPointcut` class is shown that provides static checking on the method to advise:

```
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()));
    }
}
```

The next code snippet depicts the `NoOpBeforeAdvice` class, which is just simple before advice without any operation:

```
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 {
        // no-op
    }
}
```

In the following code snippet, you can see the code used to test the different types of proxies:

```
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");
    }
}
```

In this code, you can see that you are testing three kinds of proxies:

- A standard CGLIB proxy
- A CGLIB proxy with a frozen advice chain (that is, when a proxy is frozen by calling the `setFrozen()` method in the `ProxyConfig` class that `ProxyFactory` extends indirectly, CGLIB will perform further optimization; however, further advice change will not be allowed)
- A JDK proxy

For each proxy type, you run the following five test cases:

- *Advised method (test 1)*: This is a method that is advised. The advice type used in the test is before advice that performs no processing, so it reduces the effects of the advice on the performance tests.
- *Unadvised method (test 2)*: This is a method on the proxy that is unadvised. Often your proxy has many methods that are not advised. This test looks at how well unadvised methods perform for the different proxies.
- *The equals() method (test 3)*: This test looks at the overhead of invoking the `equals()` method. This is especially important when you use proxies as keys in a `HashMap` or similar collection.
- *The hashCode() method (test 4)*: As with the `equals()` method, the `hashCode()` method is important when you are using `HashMaps` or similar collections.
- *Executing methods on the Advised interface (test 5)*: As we mentioned earlier, a proxy implements the `Advised` interface by default, allowing you to modify the proxy after creation and to query information about the proxy. This test looks at how fast methods on the `Advised` interface can be accessed using the different proxy types.

Table 5-3 shows the results of these tests.

Table 5-3. *Proxy Performance Test Results (in Milliseconds)*

	CGLIB (Standard)	CGLIB (Frozen)	JDK
Advised method	245	135	224
Unadvised method	92	42	78
`equals()`	9	6	77
`hashCode()`	29	13	23
`Advised.getProxyTargetClass()`	9	6	15

As you can see, the performance between standard CGLIB and JDK dynamic proxy for both advised and unadvised methods doesn't differ much. As always, these numbers will vary based on hardware and the JDK being used.

However, there is a noticeable difference when you are using a CGLIB proxy with a frozen advice chain. Similar figures apply to the `equals()` and `hashCode()` methods, which are noticeably faster when you are using the CGLIB proxy. For methods on the `Advised` interface, you will notice that they are also faster on the CGLIB frozen proxy. The reason for this is that `Advised` methods are handled early on in the `intercept()` method, so they avoid much of the logic that is required for other methods.

Choosing a Proxy to Use

Deciding which proxy to use is typically easy. The CGLIB proxy can proxy both classes and interfaces, whereas JDK proxies can proxy only interfaces. In terms of performance, there is no significant difference between JDK and CGLIB standard mode (at least in running both advised and unadvised methods), unless you use CGLIB in frozen mode, in which case the advice chain can't be changed and CGLIB performs further optimization when in frozen mode. When proxying a class, the CGLIB proxy is the default choice because it is the only proxy capable of generating a proxy of a class. To use the CGLIB proxy when proxying an interface, you must set the value of the `optimize` flag in `ProxyFactory` to `true` by using the `setOptimize()` method.

Advanced Use of Pointcuts

Earlier in the chapter, we looked at the six basic `Pointcut` implementations Spring provides; for the most part, we have found that these meet the needs of our applications. However, sometimes you might need more flexibility when defining pointcuts. Spring provides two additional `Pointcut` implementations, `ComposablePointcut` and `ControlFlowPointcut`, which provide exactly the flexibility you need.

Using Control Flow Pointcuts

Spring control flow pointcuts, implemented by the `ControlFlowPointcut` class, are similar to the `cflow` construct available in many other AOP implementations, although they are not quite as powerful. Essentially, a control flow pointcut in Spring applies to all method calls below a given method or below all methods in a class. This is quite hard to visualize and is better explained using an example.

The following code snippet shows a `SimpleBeforeAdvice` class that writes out a message describing the method it is advising:

```
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);
    }
}
```

This advice class allows we to see which methods the `ControlFlowPointcut` applies to. Here, you can see the simple `TestBean` class:

```
package com.apress.prospring5.ch5;

public class TestBean {
    public void foo() {
        System.out.println("foo()");
    }
}
```

You can see the simple foo() method that we want to advise. We have, however, a special requirement: we want to advise this method only when it is called from another, specific method. The following code snippet shows a simple driver program for this example:

```
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();
    }
}
```

In the previous code snippet, the advised proxy is assembled with ControlFlowPointcut, and then the foo() method is invoked twice, once directly from the main() method and once from the test() method. Here is the line of particular interest:

```
        Pointcut pc = new ControlFlowPointcut(ControlFlowDemo.class, "test");
```

In this line, we are creating a ControlFlowPointcut instance for the test() method of the ControlFlowDemo class. Essentially, this says, "Pointcut all methods that are called from the ControlFlowExample.test() method." Note that although we said "Pointcut all methods," in fact this really means "Pointcut all methods on the proxy object that is advised using the Advisor corresponding to this instance of ControlFlowPointcut." Running the previous example yields the following result in the console:

```
        Trying normal invoke
foo()

        Trying under ControlFlowDemo.test()
Before method: public void com.apress.prospring5.ch5.TestBean.foo()
foo()
```

As you can see, when the `sing()` method is first invoked outside of the control flow of the `test()` method, it is unadvised. When it executes for a second time, this time inside the control flow of the `test()` method, the `ControlFlowPointcut` indicates that its associated advice applies to the method, and thus the method is advised. Note that if we had called another method from within the `test()` method, one that was not on the advised proxy, it would not have been advised.

Control flow pointcuts can be extremely useful, allowing you to advise an object selectively only when it is executed in the context of another. However, be aware that you take a substantial performance hit for using control flow pointcut over other pointcuts.

Let's consider an example. Suppose we have a transaction processing system, which contains a `TransactionService` interface as well as an `AccountService` interface. We would like to apply after advice so that when the `AccountService.updateBalance()` method is called by `TransactionService.reverseTransaction()`, an e-mail notification is sent to the customer, after the account balance is updated. However, an e-mail will not be sent under any other circumstances. In this case, the control flow pointcut will be useful. Figure 5-6 shows the UML sequence diagram for this scenario.

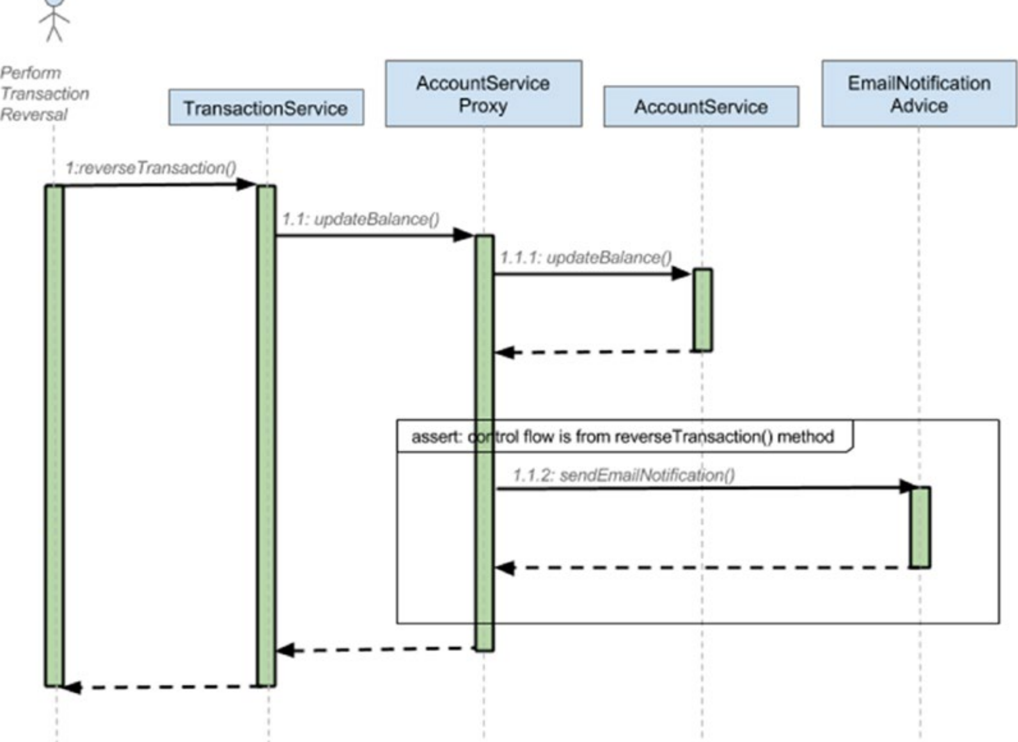

Figure 5-6. *UML sequence diagram for a control flow pointcut*

Using a Composable Pointcut

In previous pointcut examples, we used just a single pointcut for each Advisor. In most cases, this is usually enough, but in some cases, you may need to compose two or more pointcuts together to achieve the desired goal. Say you want to pointcut all getter and setter methods on a bean. You have a pointcut for getters and a pointcut for setters, but you don't have one for both. Of course, you could just create another pointcut with the new logic, but a better approach is to combine the two pointcuts into a single pointcut by using ComposablePointcut.

ComposablePointcut supports two methods: union() and intersection(). By default, ComposablePointcut is created with a ClassFilter that matches all classes and a MethodMatcher that matches all methods, although you can supply your own initial ClassFilter and MethodMatcher during construction. The union() and intersection() methods are both overloaded to accept ClassFilter and MethodMatcher arguments.

The ComposablePointcut.union() method can be called by passing in an instance of either the ClassFilter, MethodMatcher, or Pointcut interface. The result of a union operation is that ComposablePointcut will add an "or" condition into its call chain for matching with the joinpoints. It's the same for the ComposablePointcut.intersection() method, but this time an "and" condition will be added instead, which means that all ClassFilter, MethodMatcher, and Pointcut definitions within ComposablePointcut should be matched for applying an advice. You can imagine it as the WHERE clause in a SQL query, with the union() method like the "or" operator and the intersection() method like the "and" operator.

As with control flow pointcuts, this is quite difficult to visualize, and it is much easier to understand with an example. The following example shows the GrammyGuitarist class used in a previous example with its four methods:

```
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");
        }
}
```

With this example, we are going to generate three proxies by using the same ComposablePointcut instance, but each time, we are going to modify ComposablePointcut by using either the union() or intersection() method. Following this, we will invoke all three methods on the target bean proxy and look at which ones have been advised. The following code sample depicts this:

```
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"));
        }
    }
}
```

The first thing to notice in this example is the set of three private MethodMatcher implementations. SingMethodMatcher matches all methods that start with get. This is the default MethodMatcher that we use to assemble ComposablePointcut. Because of this, we expect that the first round of invocations on the GrammyGuitarist methods will result in only the sing() methods being advised.

TalkMethodMatcher matches all methods named talk, and it is combined with ComposablePointcut by using union() for the second round of invocations. At this point, we have a union of two MethodMatchers—one that matches all methods starting with si and one that matches all methods named talk. We now expect that all invocations during the second round will be advised. TalkMethodMatcher is very specific and matches only the talk() method. This MethodMatcher is combined with ComposablePointcut by using intersection() for the third round for invocations.

Because RestMethodMatcher is being composed by using intersection(), we expect none of the methods to be advised in the third round because there is no method that matches all the composed MethodMatchers.

Running this example results in the following output:

```
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
```

Although this example demonstrated the use of MethodMatchers only in the composition process, it is just as simple to use ClassFilter when you are building the pointcut. Indeed, you can use a combination of MethodMatchers and ClassFilters when building your composite pointcut.

Composition and the Pointcut Interface

In the previous section, you saw how to create a composite pointcut by using multiple MethodMatchers and ClassFilters. You can also create composite pointcuts by using other objects that implement the Pointcut interface.

Another way to construct a composite pointcut is to use the org.springframework.aop.support.Pointcuts class. The class provides three static methods. The intersection() and union() methods both take two pointcuts as arguments to construct a composite pointcut. On the other hand, a matches(Pointcut, Method, Class, Object[]) method is provided for performing a quick check on whether a pointcut matches with the provided method, class, and method arguments.

The Pointcuts class supports operations on only two pointcuts. So, if you need to combine MethodMatcher and ClassFilter with Pointcut, you need to use the ComposablePointcut class. However, when you need to combine just two pointcuts, the Pointcuts class will be more convenient.

Pointcut Summary

Spring offers a powerful set of Pointcut implementations that should meet most, if not all, of your application's requirements. Remember that if you can't find a pointcut to suit your needs, you can create your own implementation from scratch by implementing Pointcut, MethodMatcher, and ClassFilter.

You can use two patterns to combine pointcuts and advisors. The first pattern, the one we have used so far, involves having the pointcut implementation decoupled from the advisor. In the code we have seen up to this point, we created instances of Pointcut implementations and then used the DefaultPointcutAdvisor implementation to add advice along with the Pointcut to the proxy.

The second option, one that is adopted by many of the examples in the Spring documentation, is to encapsulate the Pointcut inside your own Advisor implementation. This way, you have a class that implements both Pointcut and PointcutAdvisor, with the PointcutAdvisor.getPointcut() method simply returning this. This is an approach many classes, such as StaticMethodMatcherPointcutAdvisor, use in Spring. We find that the first approach is the most flexible, allowing you to use different Pointcut

implementations with different Advisor implementations. However, the second approach is useful in situations where you are going to be using the same combination of Pointcut and Advisor in different parts of your application or, indeed, across many applications.

The second approach is useful when each Advisor must have a separate instance of a Pointcut; by making the Advisor responsible for creating the Pointcut, you can ensure that this is the case. If you recall the discussion on proxy performance from the previous chapter, you will remember that unadvised methods perform much better than methods that are advised. For this reason, you should ensure that, by using Pointcuts, you advise only the methods that are absolutely necessary. This way, you reduce the amount of unnecessary overhead added to your application by using AOP.

Getting Started with Introductions

Introductions are an important part of the AOP feature set available in Spring. By using introductions, you can introduce new functionality to an existing object dynamically. In Spring, you can introduce an implementation of any interface to an existing object. You may well be wondering exactly why this is useful. Why would you want to add functionality dynamically at runtime when you can simply add that functionality at development time? The answer to this question is easy. You add functionality dynamically when the functionality is crosscutting and is not easily implemented using traditional advice.

Introduction Basics

Spring treats *introductions* as a special type of advice, more specifically, as a special type of around advice. Because introductions apply solely at the class level, you cannot use pointcuts with introductions; semantically, the two don't match. An introduction adds new interface implementations to a class, and a pointcut defines which methods the advice applies to. You create an introduction by implementing the IntroductionInterceptor interface, which extends the MethodInterceptor and DynamicIntroductionAdvice interfaces. Figure 5-7 shows this structure along with the methods of both interfaces, as depicted by the IntelliJ IDEA UML plug-in. As you can see, the MethodInterceptor interface defines an invoke() method. Using this method, you provide the implementation for the interfaces that you are introducing and perform interception for any additional methods as required. Implementing all methods for an interface inside a single method can prove troublesome, and it is likely to result in an awful lot of code that you will have to wade through just to decide which method to invoke. Thankfully, Spring provides a default implementation of IntroductionInterceptor, called DelegatingIntroductionInterceptor, which makes creating introductions much simpler. To build an introduction by using DelegatingIntroductionInterceptor, you create a class that both inherits from DelegatingIntroductionInterceptor and implements the interfaces you want to introduce. The DelegatingIntroductionInterceptor implementation then simply delegates all calls to introduced methods to the corresponding method on itself. Don't worry if this seems a little unclear; you will see an example of it in the next section.

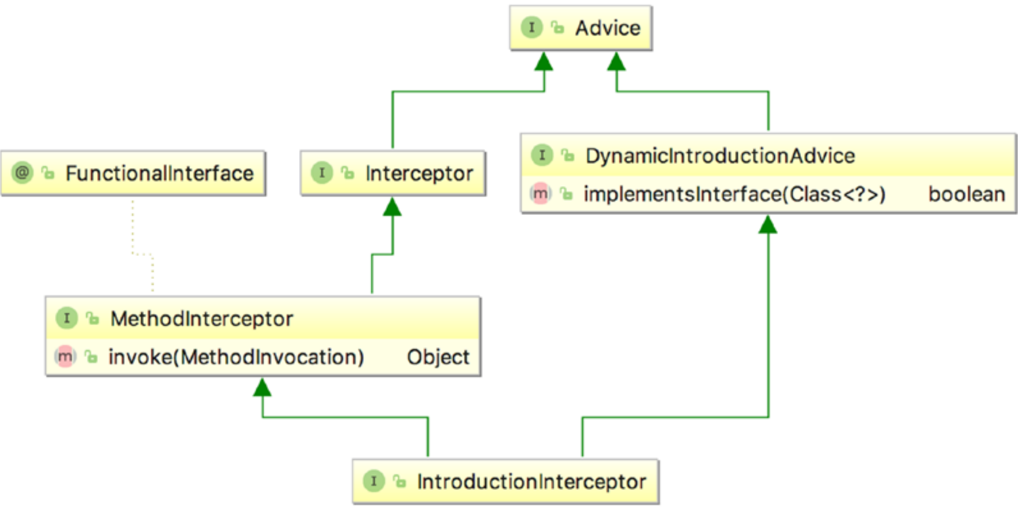

Figure 5-7. *Interface structure for introductions*

Just as you need to use PointcutAdvisor when you are working with pointcut advice, you need to use IntroductionAdvisor to add introductions to a proxy. The default implementation of IntroductionAdvisor is DefaultIntroductionAdvisor, which should suffice for most, if not all, of your introduction needs. You should be aware that adding an introduction by using ProxyFactory.addAdvice() is not permitted and results in AopConfigException being thrown. Instead, you should use the addAdvisor() method and pass an instance of the IntroductionAdvisor interface.

When using standard advice—that is, not introductions—it is possible for the same advice instance to be used for many objects. The Spring documentation refers to this as the per-class life cycle, although you can use a single advice instance for many classes. For introductions, the introduction advice forms part of the state of the advised object, and as a result, you must have a distinct advice instance for every advised object. This is called the *per-instance life cycle*. Because you must ensure that each advised object has a distinct instance of the introduction, it is often preferable to create a subclass of DefaultIntroductionAdvisor that is responsible for creating the introduction advice. This way, you need to ensure only that a new instance of your advisor class is created for each object because it will automatically create a new instance of the introduction. For example, say you want to apply before advice to the setFirstName() method on all instances of the Contact class. Figure 5-8 shows the same advice that applies to all objects of the Contact type. Now let's say you want to mix an introduction into all instances of the Contact class, and the introduction will carry information for each Contact instance (for example, an attribute isModified that indicates whether the specific instance was modified).

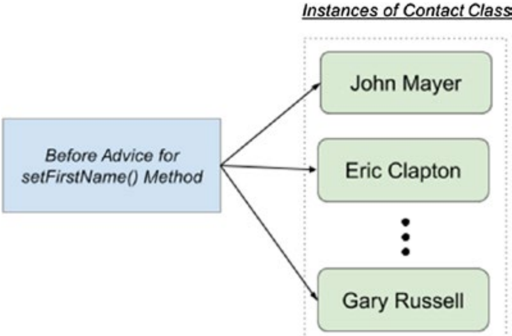

Figure 5-8. *Per-class life cycle of advice*

In this case, the introduction will be created for each instance of `Contact` and tied to that specific instance, as shown in Figure 5-9. That covers the basics of introduction creation. We will now discuss how you can use introductions to solve the problem of object modification detection.

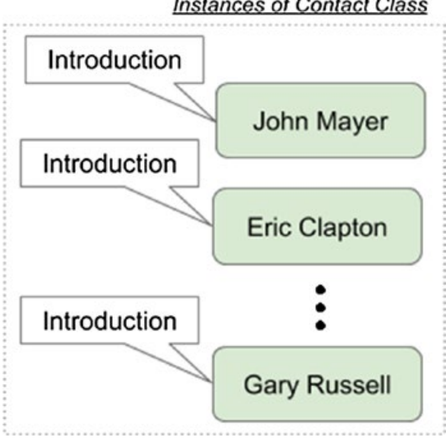

Figure 5-9. *Per-instance introduction*

Object Modification Detection with Introductions

Object modification detection is a useful technique for many reasons. Typically, you apply modification detection to prevent unnecessary database access when you are persisting object data. If an object is passed to a method for modification but it comes back unmodified, there is little point in issuing an update statement to the database. Using a modification check in this way can really increase application throughput, especially when the database is already under a substantial load or is located on a remote network, making communication an expensive operation.

Unfortunately, this kind of functionality is difficult to implement by hand because it requires you to add to every method that can modify object state to check whether the object state is actually being modified. When you consider all the `null` checks that have to be made and the checks to see whether the value is actually changing, you are looking at around eight lines of code per method. You could refactor this into a single method, but you still have to call this method every time you need to perform the check. Spread this across a typical application with many classes that require modification checks, and you have a disaster waiting to happen.

This is clearly a place where introductions will help. We don't want to have each class that requires modification checks inherit from some base implementation, losing its only chance for inheritance as a result, nor do we really want to be adding checking code to each and every state-changing method. Using introductions, we can provide a flexible solution to the modification detection problem without having to write a bunch of repetitive, error-prone code.

In this example, we are going to build a full modification check framework using introductions. The modification check logic is encapsulated by the `IsModified` interface, an implementation of which will be introduced into the appropriate objects, along with interception logic to perform modification checks automatically. For the purposes of this example, we use JavaBeans conventions, in that we consider a modification to be any call to a setter method. Of course, we don't just treat all calls to a setter method as a modification; we check to see whether the value being passed to the setter is different from the one currently stored in the object. The only flaw with this solution is that setting an object back to its original state will still reflect a modification if any one of the values on the object has changed. For example, you have a `Contact` object with the `firstName` attribute. Let's say that during processing, the `firstName` attribute was changed from `Peter` to `John`. As a result, the object was marked as modified. However, it will still be marked as modified, even if the value is then changed back from `John` to its original value `Peter` in later processing. One way to keep track of such changes is to store the full history of changes in the object's entire life cycle. However, the implementation here is nontrivial and suffices for most requirements. Implementing the more complete solution would result in an overly complex example.

Using the IsModified Interface

Central to the modification check solution is the `IsModified` interface, which the fictional application uses to make intelligent decisions about object persistence. We do not cover how the application would use `IsModified`; instead, we will focus on the implementation of the introduction. The following code snippet shows the `IsModified` interface:

```
package com.apress.prospring5.ch5.introduction;

public interface IsModified {
    boolean isModified();
}
```

There's nothing special here—just a single method, `isModified()`, indicating whether an object has been modified.

Creating a Mixin

The next step is to create the code that implements `IsModified` and that is introduced to the objects; this is referred to as a *mixin*. As we mentioned earlier, it is much simpler to create mixins by subclassing `DelegatingIntroductionInterceptor` than to create one by directly implementing the `IntroductionInterceptor` interface. The mixin class, `IsModifiedMixin`, subclasses `DelegatingIntroductionInterceptor` and also implements the `IsModified` interface. This is shown here:

```
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;
        }
    }
}
```

The first thing to notice here is the implementation of `IsModified`, which consists of the private modified field and the `isModified()` method. This example highlights why you must have one mixin instance per advised object—the mixin introduces not only methods to the object but also state. If you share a single instance of this mixin across many objects, then you are also sharing the state, which means all objects show as modified the first time a single object becomes modified.

You do not actually have to implement the `invoke()` method for a mixin, but in this case, doing so allows us to detect automatically when a modification occurs. We start by performing the check only if the object is still unmodified; we do not need to check for modifications once we know that the object has been modified. Next, we check to see whether the method is a setter, and if it is, we retrieve the corresponding getter method. Note that we cache the getter/setter pairs for quicker future retrieval. Finally, we compare the value returned by the getter with that passed to the setter to determine whether a modification has occurred. Notice that we check for the different possible combinations of `null` and set the modifications appropriately. It is important to remember that when you are using `DelegatingIntroductionInterceptor`, you must call `super.invoke()` when overriding `invoke()` because it is the `DelegatingIntroductionInterceptor` that dispatches the invocation to the correct location, either the advised object or the mixin itself.

You can implement as many interfaces as you like in your mixin, each of which is automatically introduced into the advised object.

Creating an Advisor

The next step is to create an `Advisor` to wrap the creation of the mixin class. This step is optional, but it does help ensure that a new instance of the mixin is being used for each advised object. The following code snippet shows the `IsModifiedAdvisor` class:

```
package com.apress.prospring5.ch5.introduction;

import org.springframework.aop.support.DefaultIntroductionAdvisor;

public class IsModifiedAdvisor extends DefaultIntroductionAdvisor {
    public IsModifiedAdvisor() {
        super(new IsModifiedMixin());
    }
}
```

Notice that we have extended `DefaultIntroductionAdvisor` to create your `IsModifiedAdvisor`. The implementation of this advisor is trivial and self-explanatory.

Putting It All Together

Now that we have a mixin class and an Advisor class, we can test the modification check framework. The class that we are going to use is the Contact class that was mentioned earlier, which is part of a common package. This class is often used as a dependency for projects in this book for reasons of reusability. The contents of this class are shown here:

```
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;
        }

        // getters and setter for other fields
        ...
}
```

This bean has a set of properties, but only the name property for testing the modification check mixin. The following code snippet shows how to assemble the advised proxy and then tests the modification check code:

```
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());
    }
}
```

Notice that when we are creating the proxy, we set the optimize flag to true to force the use of the CGLIB proxy. The reason for this is that when you are using the JDK proxy to introduce a mixin, the resulting proxy will not be an instance of the object class (in this case Contact); the proxy implements only the mixin interfaces, not the original class. With the CGLIB proxy, the original class is implemented by the proxy along with the mixin interfaces.

Notice in the code that we test first to see whether the proxy is an instance of Contact and then to see whether it is an instance of IsModified. Both tests return true when you are using the CGLIB proxy, but only the IsModified test returns true for the JDK proxy. Finally, we test the modification check code by first setting the name property to its current value and then to a new value, checking the value of the isModified flag each time. Running this example results in the following output:

```
Is Contact?: true
Is IsModified?: true
Has been modified?: false
Has been modified?: false
Has been modified?: true
```

As expected, both instanceof tests return true. Notice that the first call to isModified(), before any modification occurred, returns false. The next call, after we set the value of name to the same value, also returns false. For the final call, however, after we set the value of name to a new value, the isModified() method returns true, indicating that the object has in fact been modified.

Introduction Summary

Introductions are one of the most powerful features of Spring AOP; they allow you not only to extend the functionality of existing methods but to extend the set of interfaces and object implementations dynamically. Using introductions is the perfect way to implement crosscutting logic that your application interacts with through well-defined interfaces. In general, this is the kind of logic that you want to apply declaratively rather than programmatically. By using IsModifiedMixin defined in this example and the framework services discussed in the next section, we can declaratively define which objects are capable of modification checks, without needing to modify the implementations of those objects.

Obviously, because introductions work via proxies, they add a certain amount of overhead. All methods on the proxy are considered advised since pointcuts cannot be used in conjunction with introductions. However, in the case of many of the services that you can implement by using introductions such as the object modification check, this performance overhead is a small price to pay for the reduction in code required to implement the service, as well as the increase in stability and maintainability that comes from fully centralizing the service logic.

Framework Services for AOP

Up to now, we have had to write a lot of code to advise objects and generate the proxies for them. Although this in itself is not a huge problem, it does mean that all advice configuration is hard-coded into your application, removing some of the benefits of being able to advise a method implementation transparently. Thankfully, Spring provides additional framework services that allow you to create an advised proxy in your application configuration and then inject this proxy into a target bean just like any other dependencies.

Using the declarative approach to AOP configuration is preferable to the manual, programmatic mechanism. When you use the declarative mechanism, not only do you externalize the configuration of advice but you also reduce the chance of coding errors. You can also take advantage of DI and AOP combined to enable AOP so that it can be used in a completely transparent environment.

Configuring AOP Declaratively

When using declarative configuration of Spring AOP, three options exist.

- *Using ProxyFactoryBean*: In Spring AOP, `ProxyFactoryBean` provides a declarative way to configure Spring's `ApplicationContext` (and hence the underlying `BeanFactory`) when creating AOP proxies based on defined Spring beans.
- *Using the Spring aop namespace*: Introduced in Spring 2.0, the `aop` namespace provides a simplified way (when compared to `ProxyFactoryBean`) to define aspects and their DI requirements in Spring applications. However, the `aop` namespace also uses `ProxyFactoryBean` behind the scenes.
- *Using @AspectJ-style annotations*: Besides the XML-based `aop` namespace, you can use the `@AspectJ`-style annotations within your classes for configuring Spring AOP. Although the syntax it uses is based on AspectJ and you need to include some AspectJ libraries when using this option, Spring still uses the proxy mechanism (that is, creates proxied objects for the targets) when bootstrapping `ApplicationContext`.

Using ProxyFactoryBean

The `ProxyFactoryBean` class is an implementation of `FactoryBean` that allows you to specify a bean to target, and it provides a set of advice and advisors for that bean that are eventually merged into an AOP proxy. `ProxyFactoryBean` is used to apply interceptor logic to an existing target bean in a way that when methods on that bean are invoked, the interceptors are executed before and after that method call. Because you can use both advisor and advice with `ProxyFactoryBean`, you can configure not only the advice declaratively but the pointcuts as well.

`ProxyFactoryBean` shares a common interface (the `org.springframework.aop.framework.Advised` interface) with `ProxyFactory` (both classes extend the `org.springframework.aop.framework.AdvisedSupport` class indirectly, which implements the `Advised` interface), and as a result, it exposes many of the same flags such as `frozen`, `optimize`, and `exposeProxy`. The values for these flags are passed directly to the underlying `ProxyFactory`, which allows you to configure the factory declaratively as well.

ProxyFactoryBean in Action

Using ProxyFactoryBean is simple. You define a bean that will be the target bean, and then using ProxyFactoryBean, you define the bean that your application will access, using the target bean as the proxy target. Where possible, define the target bean as an anonymous bean inside the proxy bean declaration. This prevents your application from accidentally accessing the unadvised bean. However, in some cases, such as the sample we are about to show you, you may want to create more than one proxy for the same bean, so you should use a normal top-level bean for this case.

For the following example, imagine this scenario: you have a singer working together with a documentarist to produce a documentary of a tour. In this case, Documentarist has a dependency of the Singer implementation. The Singer implementation that we will use here is the previously introduced GrammyGuitarist. The contents are shown again here:

```
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");
        }
}
```

The Documentarist class that will basically tell the singer what to do while filming the documentary is shown here:

```
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;
        }
}
```

For this example, we are going to create two proxies for a single GrammySinger instance, both with the same basic advice shown here:

```
package com.apress.prospring5.ch5;

import org.aspectj.lang.JoinPoint;

public class AuditAdvice {
    public void simpleBeforeAdvice(JoinPoint joinPoint) {
        System.out.println("Executing: " +
                joinPoint.getSignature().getDeclaringTypeName() + " "
                + joinPoint.getSignature().getName());
    }
}
```

The first proxy will just advise the target by using the advice directly; thus, all methods will be advised. For the second proxy, we will configure AspectJExpressionPointcut and DefaultPointcutAdvisor so that only the sing() method of the GrammySinger class is advised. To test the advice, we will create two bean definitions of type Documentarist, each of which will be injected with a different proxy. Then we will invoke the execute() method on each of these beans and observe what happens when the advised methods on the dependency are invoked. Figure 5-10 shows the configuration for this example (app-context-xml.xml). We used an image to depict this configuration because it might look a little confusing and we wanted to make sure it is easy to see where each bean is injected. In the example, we are simply setting the properties that we set in code using Spring's DI capabilities. The only points of interest are that we use an anonymous bean for the pointcut, and we use the ProxyFactoryBean class. We prefer to use anonymous beans for pointcuts when they are not being shared because it keeps the set of beans that are directly accessible as small and as relevant to the application as possible. The important point to realize when you are using ProxyFactoryBean is that the ProxyFactoryBean declaration is the one to expose to your application and the one to use when you are fulfilling dependencies. The underlying target bean declaration is not advised, so you should use this bean only when you want to bypass the AOP framework, although in general, your application should not be aware of the AOP framework and thus should not want to bypass it. For this reason, you should use anonymous beans wherever possible to avoid accidental access from the application.

```
<?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>
```

Figure 5-10. *Declarative AOP configuration*

The following code snippet shows a simple class that obtains the two `Documentarist` instances from `ApplicationContext` and then runs the `execute()` method for each one:

```
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();
    }
}
```

Running this example produces the following output:

```
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
```

As expected, both the `sing()` and `talk()` methods in the first proxy are advised because no pointcut was used in its configuration. For the second proxy, however, only the `sing()` method was advised because of the pointcut used in the configuration.

Using ProxyFactoryBean for Introductions

You are not limited in using the `ProxyFactoryBean` class for just advising an object but also for introducing mixins to your objects. Remember from the earlier discussion on introductions that you must use an `IntroductionAdvisor` to add an introduction; you cannot add an introduction directly. The same rule applies when you are using `ProxyFactoryBean` with introductions. When you are using `ProxyFactoryBean`, it becomes much easier to configure your proxies if you created a custom `Advisor` for your mixin. The following configuration snippet shows a sample configuration for the `IsModifiedMixin` introduction from earlier in the chapter (`app-context-xml.xml`):

```
<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>
```

As you can see from the configuration, we use the IsModifiedAdvisor class as the advisor for ProxyFactoryBean, and because we do not need to create another proxy of the same target object, we use an anonymous declaration for the target bean. The following code snippet shows a modification of the previous introduction example that obtains the proxy from 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());
    }
}
```

Running this example yields exactly the same output as the previous introduction example, but this time the proxy is obtained from ApplicationContext and no configuration is present in the application code.

And since we've already covered Java configuration, the XML configuration depicted previously can be replaced with a configuration class like the one shown here:

```
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;
        }
}
```

To test that the previous class actually works, in the main() method of class IntroductionConfigDemo, just replace the lines that initialize the context with the following:

```
GenericApplicationContext ctx =
     new AnnotationConfigApplicationContext(AppConfig.class);
```

The difference in the configuration class is that there is no need to refer to the advisor bean by name or add it in a list to provide it as an argument to ProxyFactoryBean because addAdvisor(..) can be called directly and the advisor bean can be provided as the argument. This obviously simplifies the configuration.

ProxyFactoryBean Summary

When you use ProxyFactoryBean, you can configure AOP proxies that provide all the flexibility of the programmatic method without needing to couple your application to the AOP configuration. Unless you need to perform decisions at runtime as to how your proxies should be created, it is best to use the declarative method of proxy configuration over the programmatic method. Let's move on so you can see the other two options for declarative Spring AOP, which are both preferred options for applications based on Spring 2.0 or newer with JDK 5 or newer.

Using the aop Namespace

The aop namespace provides a greatly simplified syntax for declarative Spring AOP configurations. To show you how it works, let's reuse the previous ProxyFactoryBean example with a slightly modified version in order to demonstrate its usage. The GrammyGuitarist class from the previous example is still used, but Documentarist will be extended to call the sing() method with a Guitar argument.

```
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();
        }
}
```

The advice class changes to the following:

```
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());
        }
}
```

You will see that the advice class no longer needs to implement the `MethodBeforeAdvice` interface. Also, the before advice accepts the joinpoint as an argument but not the method, object, and arguments. Actually, for the advice class, this argument is optional, so you can leave the method with no argument. However, if in the advice you need to access the information of the joinpoint being advised (in this case, we want to dump the information of the calling type and method name), then we need to define the acceptance of the argument. When the argument is defined for the method, Spring will automatically pass the joinpoint into the method for your processing. Here is the Spring XML configuration with the `aop` namespace from the `app-context-xml-01.xml` file:

```
<?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>
```

First, we need to declare the aop namespace in the `<beans>` tags. Second, all the Spring AOP configuration was put under the tag `<aop:config>`. Under `<aop:config>`, you can then define the pointcut, aspects, advisors, and so on, and reference other Spring beans as usual.

From the previous configuration, we defined a pointcut with the ID `singExecution`. The expression

```
"execution(*
   com.apress.prospring5.ch5..sing*(com.apress.prospring5.ch2.common.Guitar)
)"
```

means that we want to advise all methods with the prefix `sing`, and the classes are defined under the package `com.apress.prospring5.ch5` (including all the subpackages). Also, the `sing()` method should receive one argument with the `Guitar` type. Afterward, the aspect was declared by using the `<aop:aspect>` tag, and the advice class is referencing the Spring bean with the ID `advice`, which is the `SimpleAdvice` class. The `pointcut-ref` attribute is referencing the defined pointcut with the ID `singExecution`, and the before advice (declared by using the `<aop:before>` tag) is the method `simpleBeforeAdvice()` within the advice bean. To test the previous configuration, you can use the following class:

```
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();
    }
}
```

In this example, we simply initialize ApplicationContext as usual, retrieve the bean, and call its execute() method. Running the program will yield the following output:

```
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
```

As you can see, only the call to the sing(..) method with a Guitar argument was advised; the sing() method with no argument and the talk() method were not. This exactly works as expected, and you can see the configuration was greatly simplified when compared to the ProxyFactoryBean configuration.

Let's further revise the previous sample into a bit more complicated case. Suppose now we want to advise only those methods with Spring beans with an ID starting with john and a parameter of type Guitar that has the brand property set to Gibson.

For this, first the Guitar class must be changed to add the brand property. We'll make it nonmandatory and populate it with a default value, just to keep the previous examples working.

```
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;
        }
}
```

Then we need to make NewDocumentarist call the sing() method with a special brand of guitar.

```
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();
        }
}
```

Now we need a new and more complex type of advice. The argument guitar was added into the signature of the before advice. Second, in the advice, we check and execute the logic only when the argument's brand property is equal to 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());
        }
    }
}
```

Also, the XML configuration needs to be revised because we need to use the new type of advice and update the pointcut expression. (You can find the full configuration in app-context-xml-02.xml, and except for the lines shown next, everything else is identical to the contents of app-context-xml-01.xml, so they won't be depicted again here.)

```
<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>
```

Two more directives were added to the pointcut expression. First, args(value) instructs Spring to also pass the argument with the name value into the before advice. Second, the bean(john*) directive instructs Spring to advise only the beans with an ID that has john as the prefix. This is a powerful feature; if you have a well-defined structure of Spring beans naming, you can easily advise the objects you want. For example, you can have advice that applies to all DAO beans by using bean(*DAO*) or all service layer beans using bean(*Service*), instead of using the fully qualified class name for matching. Running the same testing program with the new configuration file app-context-xml02.xml produces the following output:

```
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
```

You can see that only the sing() method with the Guitar argument and brand equal to Gibson was advised.

Let's see one more example of using the aop namespace for the around advice. Instead of creating another class to implement the MethodInterceptor interface, we can simply add a new method to the ComplexAdvice class. The following code sample shows the new method named simpleAroundAdvice() in the revised ComplexAdvice class:

```
//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;
}
```

The newly added simpleAroundAdvice() method needs to take at least one argument of type ProceedingJoinPoint so that it can proceed with the invocation of the target object. We also add the value argument to display the value in the advice. The XML configuration for <aop:aspect> must be adapted to add the new advice. (You can find the full configuration in app-context-xml-03.xml, and except for the lines shown next, everything else is identical to the contents from app-context-xml-02.xml, so they won't be depicted again here.)

```
<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>
```

We just added the new tag <aop:around> to declare the around advice and reference the same pointcut. Let's modify the NewDocumentarist.execute() method again to include a sing() call with a default Guitar to obtain the behavior we want to analyze.

```
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();
        }
}
```

Run the testing program again, and you will have the following output:

```
sing: Gravity is working against me
And gravity wants to bring me down

Executing: com.apress.prospring5.ch5.GrammyGuitarist sing
Before execution: 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

Before execution: com.apress.prospring5.ch5.GrammyGuitarist sing argument: Martin
play: G C G C Am D7
After execution: com.apress.prospring5.ch5.GrammyGuitarist sing argument: Martin
talk
```

There are two interesting points here. First, you see that the around advice was applied to both invocations of the sing(..) method with a Guitar argument, since it doesn't check the argument. Second, for the sing() method with "Gibson" Guitar as an argument, both the before and around advice were executed, and by default the before advice takes precedence.

When using the aop namespace or the @AspectJ style, there are two types of after advice. The after-returning advice (using the <aop:after-returning> tag) applies only when the target method is completed normally. Another one is the after advice (using the <aop:after> tag), which takes place whether the method was completed normally or the method runs into an error and an exception is thrown. If you need advice that executes regardless of the execution result of the target method, you should use after advice.

Using @AspectJ-Style Annotations

When using Spring AOP with JDK 5 or newer, you can also use the @AspectJ-style annotations to declare your advice. However, as stated before, Spring still uses its own proxying mechanism for advising the target methods, not AspectJ's weaving mechanism.

In this section, we will go through how to implement the same aspects as the ones in the aop namespace by using @AspectJ-style annotations. AspectJ is a general-purpose aspect-oriented extension to Java born out of need to solve issues or concerns that are not well captured by traditional programming methodologies, in other words, crosscutting concerns. For the examples in this section, we will use annotations for other Spring beans as well, and we will use Java configuration classes.

The following example depicts the GrammyGuitarist class with the bean being declared using annotations:

```
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");
        }
}
```

The NewDocumentarist class needs adapting too.

```
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;
        }
}
```

We annotate both classes with the `@Component` annotation and assign them with the corresponding name. In the `GrammyGuitarist` class, the setter method of the property `guitarist` was annotated with `@Autowired` for automatic injection by Spring.

Now let's see the `AnnotationAdvice` class using `@AspectJ`-style annotations. We will implement the pointcuts, before advice, and around advice altogether in one shot. The following code snippet shows the `AnnotationAdvice` class:

```
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;
    }
}
```

You will notice that the code structure is quite like the one we used in the aop namespace, just in this case we used annotations instead. However, there are still a few points worth noting.

- We used both @Component and @Aspect to annotate the AnnotatedAdvice class. The @Aspect annotation is used to declare that it's an aspect class. To allow Spring to scan the component when we use the <context:component-scan> tag in the XML configuration, you also need to annotate the class with @Component.
- The pointcuts were defined as methods that return void. In the class, we defined two pointcuts; both are annotated with @Pointcut. We intentionally split the pointcut expression in the aop namespace example into two. The first one (indicated by the method singExecution(Guitar value)) defines the pointcut for execution of sing*() methods within all classes under the package com.apress.prospring4.ch5 with a guitar argument, and the argument (value) will also be passed into the advice. The other one (indicated by the method isJohn()) is to define another pointcut that defines all method executions with Spring beans' names prefixed by john. Also note that we need to use && to define the "and" condition in the pointcut expression, while for the aop namespace, we need to use the and operator.
- The before-advice method was annotated with @Before, while the around advice was annotated with @Around. For both advice types, we pass in the value that uses the two pointcuts defined in the class. The value singExecution(value) && isJohn() means the condition of both pointcuts should be matched for applying the advice, which is the same as the intersection operation in ComposablePointcut.
- The before-advice logic and the around-advice logic are the same as in the aop namespace example.

With all the annotations in place, the XML configuration becomes simple.

```
<?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>
```

Only two tags were declared. The `<aop:aspect-autoproxy>` tag is to inform Spring to scan for `@AspectJ`-style annotations, while the `<context:component-scan>` tag was still required for Spring to scan for Spring beans within the package that the advice resides. We also need to annotate the advice class with `@Component` to indicate that it's a Spring component.

The class to test this configuration is depicted in the following code snippet:

```
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();
    }
}
```

If you were to run the example as it is, you would get a little surprise as this is what you would see in the console:

```
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'
...
```

So, what is going on here? Well, GrammyGuitarist implements the Singer interface, and by default interface-based JDK dynamic proxies are created. But NewDocumentarist strictly requires the dependency to be of type Grammy-Guitarist or an extension of it. Thus, the previous exception is thrown. How do we fix it? There are two ways: one is to modify NewDocumentarist to accept a Singer dependency, but this is not suitable for our example as we want to access methods in the GrammyGuitarist class, which are not implementations of methods defined in the Singer interface. The second way is to request Spring to generate CGLIB, class-based proxies. In XML, this can be done by modifying the configuration of the <aop:aspectj-autoproxy/> tag and setting the proxy-target-class attribute value to true.

The Java configuration class is even simpler than this:

```
@Configuration
@ComponentScan(basePackages = {"com.apress.prospring5.ch5"})
@EnableAspectJAutoProxy(proxyTargetClass = true)
public class AppConfig {
}
```

Notice the @EnableAspectJAutoProxy annotation. It is the equivalent of <aop:aspectj-autoproxy/> and also has an attribute called proxyTargetClass that is analogous to the proxy-target-class attribute. This annotation enables support for handling components marked with AspectJ's @Aspect annotation and is designed to be used on classes annotated with @Configuration.

The following is the testing program. It was designed as a JUnit test case so that both XML and Java configuration examples can be placed in the same class. Smart editors like IntelliJ IDEA offer the possibility to execute every test method in isolation.

```
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();
        }
}
```

Running any of these test methods, if it passes, yields the following output:

```
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
```

Spring Boot provides a special AOP starter library that removes a little of the hassle of configuration. The library is configured as usual in the pro-spring-15/build.properties file and added as a dependency using its given name in the child project configuration file aspectj-boot/build.gradle.

```
//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"
    ]
}
//aspectj-boot/build.gradle
buildscript {
          ...
    dependencies {
        classpath boot.springBootPlugin
    }
}

apply plugin: 'org.springframework.boot'

dependencies {
    compile boot.starterAop
}
```

In Figure 5-11 you can see the set of libraries added as dependencies to the Spring Boot project. By adding this library to your application as a dependency, the `@EnableAspectJAutoProxy(proxyTargetClass = true)` annotation is no longer needed because the AOP Spring support is already enabled by default. The attribute does not have to be set anywhere either because Spring Boot automatically detects what type of proxies you need. Considering the previous example, you can remove the `AppConfig` class and replace it with a typical Spring Boot application class.

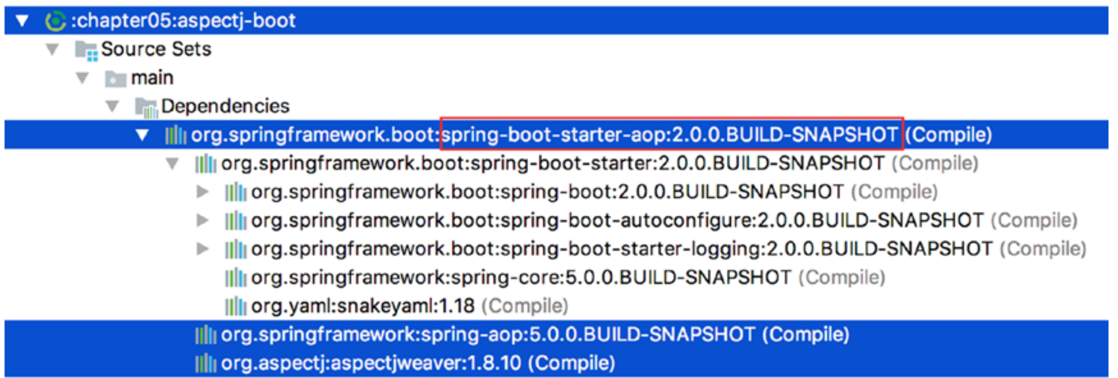

Figure 5-11. *Spring Boot AOP starter tranzitive dependencies as depicted in IntelliJ IDEA*

```
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();
        }
}
```

Considerations for Declarative Spring AOP Configuration

So far, we have discussed three ways of declaring Spring AOP configuration, including the `ProxyFactoryBean`, the `aop` namespace, and `@AspectJ`-style annotations. We believe you will agree that the `aop` namespace is much simpler than `ProxyFactoryBean`. So, the general question is, do you use the `aop` namespace or `@AspectJ`-style annotations?

If your Spring application is XML configuration–based, using the `aop` namespace approach is a natural choice because it keeps the AOP and DI configuration styles consistent. On the other hand, if your application is mainly annotation based, use the `@AspectJ` annotation. Again, let the requirements of your application drive the configuration approach and make your best effort to be consistent.

Moreover, there are some other differences between the `aop` namespace and `@AspectJ` annotation approaches.

- The pointcut expression syntax has some minor differences (for example, in the previous discussions, we need to use `and` in the `aop` namespace, but `&&` in `@AspectJ` annotation).
- The `aop` namespace approach supports only the "singleton" aspect instantiation model.
- In the `aop` namespace, you can't "combine" multiple pointcut expressions. In the example using `@AspectJ`, we can combine the two pointcut definitions (that is, `singExecution(value) && isJohn()`) in the before and around advice. When using the `aop` namespace and you need to create a new pointcut expression that combines the matching conditions, you need to use the `ComposablePointcut` class.

AspectJ Integration

AOP provides a powerful solution to many of the common problems that arise with OOP-based applications. When using Spring AOP, you can take advantage of a select subset of AOP functionality that, in most cases, enables you to solve problems you encounter in your application. However, in some cases, you may want to use some AOP features that are outside the scope of Spring AOP.

From the joinpoint perspective, Spring AOP supports only pointcuts matching on the execution of public nonstatic methods. However, in some cases, you may need to apply advice to protected/private methods, during object construction or field access, and so on.

In those cases, you need to look at an AOP implementation with a fuller feature set. Our preference, in this case, is to use AspectJ, and because you can now configure AspectJ aspects using Spring, AspectJ forms the perfect complement to Spring AOP.

About AspectJ

AspectJ is a fully featured AOP implementation that uses a weaving process (either compile-time or load-time weaving) to introduce aspects into your code. In AspectJ, aspects and pointcuts are built using a Java-like syntax, which reduces the learning curve for Java developers. We are not going to spend too much time looking at AspectJ and how it works because that is outside the scope of this book. Instead, we present some simple AspectJ examples and show you how to configure them using Spring. For more information on AspectJ, you should definitely read *AspectJ in Action: Enterprise AOP with Spring Applications* by Ramnivas Laddad (Manning, 2009).

⚠ We are not going to cover how to weave AspectJ aspects into your application. Refer to the AspectJ documentation for details or take a look at the provided Gradle build in Chapter 5's `aspectj-aspects` project.

Using Singleton Aspects

By default, AspectJ aspects are singletons, meaning you get a single instance per class loader. The problem Spring faces with any AspectJ aspect is that it cannot create the aspect instance, as this is already handled by AspectJ itself. However, each aspect exposes a method called `org.aspectj.lang.Aspects.aspectOf()` that can be used to access the aspect instance. Using the `aspectOf()` method and a special feature of Spring configuration, you can have Spring configure the aspect for you. With this support, you can take full advantage of AspectJ's powerful AOP feature set without losing out on Spring's excellent DI and configuration abilities. This also means you do not need two separate configuration methods for your application; you can use the same Spring `ApplicationContext` approach for all your Spring-managed beans and for your AspectJ aspects.

To support aspects in a Spring application, a Gradle plug-in needs to be added to the configuration. You can find the source code and instructions on how to use it in a Gradle application here: https://github.com/eveoh/gradle-aspectj. Here you can see the contents of the `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(' '))
   }
}
```

In the following code snippet, you can see a basic class, `MessageWriter`, that we will advise using AspectJ:

```
package com.apress.prospring5.ch5;

public class MessageWriter {
    public void writeMessage() {
        System.out.println("foobar!");
    }
```

```
    public void foo() {
        System.out.println("foo");
    }
}
```

For this example, we are going to use AspectJ to advise the `writeMessage()` method and write out a message before and after the method invocation. These messages will be configurable using Spring. The following code sample shows the `MessageWrapper` aspect (the file name is `MessageWrapper.aj`, which is an AspectJ file instead of a standard Java class):

```
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);
    }
}
```

Essentially, we create an aspect called `MessageWrapper`, and, just as with a normal Java class, we give the aspect two properties, `suffix` and `prefix`, which we will use when advising the `writeMessage()` method. Next, we define a named pointcut, `doWriting()`, for a single joinpoint, in this case, the execution of the `writeMessage()` method. AspectJ has a large number of joinpoints, but coverage of those is outside the scope of this example. Finally, we define two bits of advice: one that executes before the `doWriting()`

pointcut and one that executes after it. The following configuration snippet shows how this aspect is configured in Spring (app-config-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="aspect" class="com.apress.prospring5.ch5.MessageWrapper"
        factory-method="aspectOf" p:prefix="The Prefix" p:suffix="The Suffix"/>
</beans>
```

As you can see, much of the configuration of the aspect bean is similar to standard bean configuration. The only difference is the use of the factory-method attribute of the <bean> tag. The factory-method attribute is intended to allow classes that follow a traditional Factory pattern to be integrated seamlessly into Spring. For instance, if you have a class Foo with a private constructor and then a static factory method, getInstance(), using the factory-method attribute allows a bean of this class to be managed by Spring. The aspectOf() method exposed by every AspectJ aspect allows you to access the instance of the aspect and thus allows Spring to set the properties of the aspect. Here you can see a simple driver application for this example:

```
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();
    }
}
```

Notice that first we load ApplicationContext to allow Spring to configure the aspect. Next we create an instance of MessageWriter and then invoke the writeMessage() and foo() methods. The output from this example is as follows:

```
The Prefix
foobar!
The Suffix
foo
```

As you can see, the advice in the MessageWrapper aspect was applied to the writeMessage() method, and the prefix and suffix values specified in the ApplicationContext configuration were used by the advice when writing out the before and after messages.

Summary

In this chapter, we covered a large number of AOP core concepts and looked at how these concepts translate into the Spring AOP implementation. We discussed the features that are (and are not) implemented in Spring AOP, and we pointed to AspectJ as an AOP solution for those features that Spring does not implement. We spent some time explaining the details of the advice types available in Spring, and you saw examples of the four types in action. We also looked at how you limit the methods to which advice applies by using pointcuts. In particular, we looked at the six basic pointcut implementations available with Spring. We also covered the details of how the AOP proxies are constructed, the different options, and what makes them different. We compared performance among three proxy types and highlighted some major differences and restrictions for choosing between a JDK vs. CGLIB proxy. We covered the advanced options for pointcutting, as well as how to extend the set of interfaces implemented by an object using introductions. We also covered Spring Framework services to configure AOP declaratively, thus avoiding the need to hard-code AOP proxy construction logic into your code. We spent some time looking at how Spring and AspectJ are integrated to allow you to use the added power of AspectJ without losing any of the flexibility of Spring. That's certainly a lot of AOP!

In the next chapter, we move on to a completely different topic—how we can use Spring's JDBC support to radically simplify the creation of JDBC-based data access code.

CHAPTER 6

Spring JDBC Support

By now you have seen how easy it is to build a fully Spring-managed application. You have a solid understanding of bean configuration and aspect-oriented programming (AOP). However, one part of the puzzle is missing: how do you get the data that drives the application?

Besides simple throwaway command-line utilities, almost every application needs to persist data to some sort of data store. The most usual and convenient data store is a relational database.

These are the top seven relational enterprise databases for 2017:

- Oracle Database
- Microsoft SQL Server
- IBM DB2
- SAP Sybase ASE
- PostgreSQL
- MariaDB Enterprise
- MySQL

If you are not working for a big company that can afford licenses for the first four, you probably are using one of the last three in the previous list. The most used open source relational databases are perhaps MySQL (`http://mysql.com`) and PostgreSQL (`postgresql.org`). MySQL is generally more widely used for web application development, especially on the Linux platform.[1] On the other side, PostgreSQL is friendlier to Oracle developers because its procedural language, PLpgSQL, is very close to Oracle's PL/SQL language.

Even if you choose the fastest and most reliable database, you cannot afford to lose its speed and flexibility by using a poorly designed and implemented data access layer. Applications tend to use the data access layer very frequently; thus, any unnecessary bottlenecks in the data access code impact the entire application, no matter how well designed it is.

In this chapter, we show you how you can use Spring to simplify the implementation of data access code using JDBC. We start by looking at the large and repetitive amount of code you would normally need to write without Spring and then compare it to a class implemented using Spring's data access classes. The result is truly amazing because Spring allows you to use the full power of human-tuned SQL queries while minimizing the amount of support code you need to implement. Specifically, we discuss the following:

- *Comparing traditional JDBC code and Spring JDBC support*: We explore how Spring simplifies the old-style JDBC code while keeping the same functionality. You will also see how Spring accesses the low-level JDBC API and how this low-level API is mapped into convenient classes such as `JdbcTemplate`.

[1]WordPress is a widely used blogging platform that uses MySQL or MariaDB for storing data.

© Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho 2017

I. Cosmina et al., *Pro Spring 5*, https://doi.org/10.1007/978-1-4842-2808-1_6

- *Connecting to the database*: Even though we do not go into every little detail of database connection management, we do show you the fundamental differences between a simple `Connection` and a `DataSource`. Naturally, we discuss how Spring manages the data sources and which data sources you can use in your applications.
- *Retrieving and mapping the data to Java objects*: We show you how to retrieve data and then how to effectively map the selected data to Java objects. You also learn that Spring JDBC is a viable alternative to object-relational mapping (ORM) tools.
- *Inserting, updating, and deleting data*: Finally, we discuss how you can implement the insert, update, and delete operations by using Spring to execute these types of queries.

Introducing Lambda Expressions

The release of Java version 8 brings lambda expression support, among many other features. Lambda expressions make a great replacement for anonymous inner class usage and are an ideal candidate for working with Spring's JDBC support. The usage of lambda expressions requires the use of Java 8. This book was written during the prerelease versions of Java 8 and the first General Availability version, so we understand not everybody will be using Java 8 yet. Given that, the chapter code samples and source code download show both flavors where applicable. Lambda expressions are suitable in most places of the Spring API where templates or callbacks are used, not just limited to JDBC. This chapter does not cover lambda expressions themselves, as they are a Java language feature, and you should be familiar with lambda concepts and syntax. Please refer to the Lambda Expressions tutorial at `http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html` for more information.

Sample Data Model for Example Code

Before proceeding with the discussion, we introduce a simple data model that is used for the examples throughout this chapter, as well as the next few chapters when discussing other data access techniques (we will expand the model accordingly to fulfill the needs of each topic as we go).

The model is a simple music database with two tables. The first one is the `SINGER` table, which stores a singer's information, and the other table is `ALBUM`, which stores the albums released by that singer. Each singer can have zero or more albums; in other words, it's a one-to-many relationship between `SINGER` and `ALBUM`. A singer's information includes their first and last names and date of birth. Figure 6-1 shows the entity-relationship (ER) diagram of the database.

Figure 6-1. *Simple data model for the example code*

As you can see, both tables have an ID column that will be automatically assigned by the database during insertion. For the `ALBUM` table, there is a foreign-key relation to the `SINGER` table, which is linked by the column `SINGER_ID` with the primary key of the `SINGER` table (that is, the ID column).

In this chapter, we use the open source database MySQL to show interactions with a real database in some examples. This will require you to have an instance of MySQL available to use. We do not cover how to install MySQL. You can use another database of your choice, but you may need to modify the schema and function definitions. We also cover embedded database usage, which does not require a MySQL database.

Just in case you want to use MySQL, on the official site you can find very good tutorials on installing and configuring MySQL. After you have downloaded MySQL[2] and installed it, you can access it using the `root` account. Usually, when you develop an application, you need a new schema and user. For the code samples in this chapter, the schema is named `MUSICDB`, and the user to access it is named `prospring5`. The SQL code to execute to create them is depicted next, and you can find them in the `ddl.sql` file, under the `resources` directory of the `plain-jdbc` project. Included is a fix for a bug in the MySQL Community Server version 5.17.18, the current version at the time this book was written.

```
CREATE USER 'prospring5'@'localhost' IDENTIFIED BY 'prospring5';

CREATE SCHEMA MUSICDB;
GRANT ALL PRIVILEGES ON MUSICDB . * TO 'prospring5'@'localhost';
FLUSH PRIVILEGES;

/*in case of java.sql.SQLException: The server timezone value 'UTC'
       is unrecognized or represents more than one timezone. */
SET GLOBAL time_zone = '+3:00';
```

The following code snippet depicts the SQL code necessary to create the two tables mentioned previously. This code is in the `resources` directory of the `plain-jdbc` project in `schema.sql`.

```
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)
);
```

[2]Download MySQL Community Server from https://dev.mysql.com/downloads/.

If you use a smart editor like IntelliJ IDEA, you can use the Database view to inspect your schema and tables. In Figure 6-2 you can see the contents of the MUSICDB schema as depicted in IntelliJ IDEA.

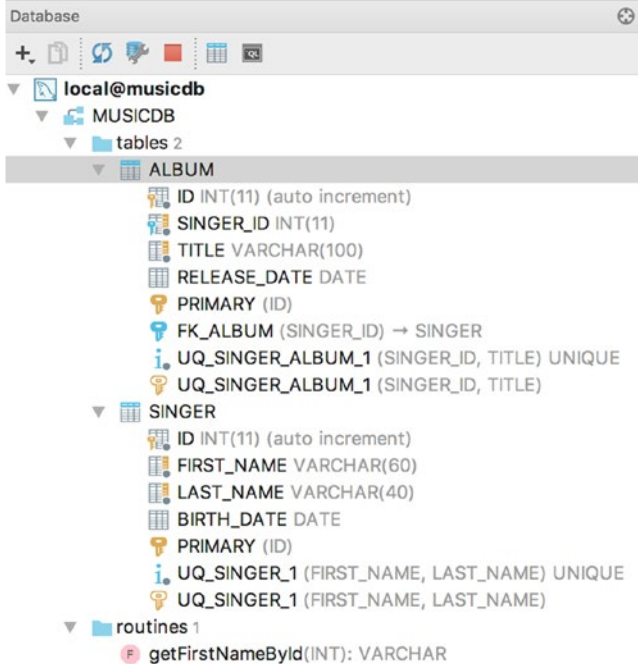

Figure 6-2. *Contents of the MUSICDB schema*

Because we need data to test the JDBC usage, a file named `test-data.sql` is also provided containing a set of `INSERT` statements that populate the two tables.

```
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');
```

In later sections of this chapter, you will see examples of retrieving the data via JDBC from the database and directly mapping the result set into Java objects (that is, POJOs). These classes that map to the records in tables are also called `entities`. For the `SINGER` table, a `Singer` class will be created that will be instantiated to create Java objects that map to singer records.

```
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;
    }
}
```

In a similar manner, an `Album` class is created as well.

```
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;
    }
}
```

Let's start with a simple interface for `SingerDao` that encapsulates all the data access services for singer information. The code is as follows:

```
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);
}
```

In the previous interface, we define two finder methods and the insert, update, and delete methods, respectively. They correspond to the CRUD terms (create, read, update, delete).

Finally, to facilitate testing, let's modify the `logback.xml` configuration file to turn the log level to `DEBUG` for all classes. At the `DEBUG` level, the Spring JDBC module will output all the underlying SQL statements being fired to the database so you know what exactly is going on; this is especially useful for troubleshooting SQL statement syntax errors. The following configuration sample depicts the contents of the `logback.xml` file (residing under `src/main/resources` with the source code files for the Chapter 6 project) with the `DEBUG` level turned on.

```
<?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>
```

Exploring the JDBC Infrastructure

JDBC provides a standard way for Java applications to access data stored in a database. The core of the JDBC infrastructure is a driver that is specific to each database; it is this driver that allows Java code to access the database.

Once a driver is loaded, it registers itself with a `java.sql.DriverManager` class. This class manages a list of drivers and provides static methods for establishing connections to the database. The `DriverManager`'s `getConnection()` method returns a driver-implemented `java.sql.Connection` interface. This interface allows you to run SQL statements against the database.

The JDBC framework is quite complex and well tested; however, with this complexity comes difficulty in development. The first level of complexity lies in making sure your code manages the connections to the database. A connection is a scarce resource and is very expensive to establish. Generally, the database creates a thread or spawns a child process for each connection. Also, the number of concurrent connections is usually limited, and an excessive number of open connections will slow down the database.

We will show you how Spring helps manage this complexity, but before we can proceed any further, we need to show you how to select, delete, and update data in pure JDBC.

All projects covered in this chapter need special databases libraries as dependencies: `mysql-connector`, `spring-jdbc`, `dbcp`, and so on. Just look in the `build.gradle` configuration files for each project and in the `pro-spring-15/build.gradle` for the version and libraries used.

Let's create a plain form of implementation of the SingerDao interface for interacting with the database via pure JDBC. Keeping in mind what we already know about database connections, we will take the cautious and expensive (in terms of performance) approach of creating a connection for each statement. This greatly degrades the performance of Java and adds extra stress to the database because a connection has to be established for each query. However, if we kept a connection open, we could bring the database server to a halt. The following code snippet shows the code required for managing a JDBC connection, using MySQL as an example:

```
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);
        }
    }
...
```

This code is far from complete, but it gives you an idea of the steps you need in order to manage a JDBC connection. This code does not even deal with connection pooling, which is a common technique for managing connections to the database more effectively. We do not discuss connection pooling at this point (connection pooling is discussed in the "Database Connections and DataSources" section later in this chapter); instead, the following code snippet shows an implementation of the findAll(), insert(), and delete() methods of the SingerDao interface using plain 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);
        }
    }
    ...
}
```

To test the PlainSingerDao class, you can use the following class:

```
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());
        }
    }
}
```

As you can notice, we will now use a logger to print the messages in the console. Running the previous program yields the following result (assuming you have a locally installed MySQL database called `MUSICDB` that has a username and password set to `prospring5` and the sample data was loaded):

```
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
```

As shown in the output, the first block of lines shows the initial data. The second block of lines shows that the new record was added. The final block of lines shows that the newly created singer (Ed Sheeran) was deleted.

As you can see in the previous code samples, a lot of code needs to be moved to a helper class or, even worse, duplicated in each DAO class. This is the main disadvantage of JDBC from the application programmer's point of view; you just do not have time to write repetitive code in every DAO class. Instead, you want to concentrate on writing code that actually does what you need the DAO class to do: select, update, and delete the data. The more helper code you need to write, the more checked exceptions you need to handle, and the more bugs you may introduce in your code.

This is where a DAO framework and Spring come in. A framework eliminates the code that does not actually perform any custom logic and allows you to forget about all the housekeeping that needs to be performed. In addition, Spring's extensive JDBC support makes your life a lot easier.

Spring JDBC Infrastructure

The code we discussed in the first part of the chapter is not very complex, but it is tedious, and because there is so much of it to write, the likelihood of coding errors is quite high. It is time to take a look at how Spring makes things easier and more elegant.

Overview and Used Packages

JDBC support in Spring is divided into the five packages detailed in Table 6-1; each handles different aspects of JDBC access.

Table 6-1. *Spring JDBC Packages*

Package	Description
`org.springframework.jdbc.core`	This contains the foundations of JDBC classes in Spring. It includes the core JDBC class, `JdbcTemplate`, which simplifies programming database operations with JDBC. Several subpackages provide support of JDBC data access with more specific purposes (e.g., a `JdbcTemplate` class that supports named parameters) and related support classes as well.
`org.springframework.jdbc.datasource`	This contains helper classes and `DataSource` implementations that you can use to run JDBC code outside a JEE container. Several subpackages provide support for embedded databases, database initialization, and various data source lookup mechanisms.
`org.springframework.jdbc.object`	This contains classes that help convert the data returned from the database into objects or lists of objects. These objects and lists are plain Java objects and therefore are disconnected from the database.
`org.springframework.jdbc.support`	The most important class in this package is `SQLException` translation support. This allows Spring to recognize error codes used by the database and map them to higher-level exceptions.
`org.springframework.jdbc.config`	This contains classes that support JDBC configuration within Spring's `ApplicationContext`. For example, it contains a handler class for the `jdbc` namespace (e.g., `<jdbc:embedded-database>` tags).

Let's start the discussion of Spring JDBC support by looking at the lowest-level functionality. The first thing that needs to be done before running SQL queries is establishing a connection to the database.

Database Connections and DataSources

You can use Spring to manage the database connection for you by providing a bean that implements javax.sql.DataSource. The difference between a DataSource and a Connection is that a DataSource provides and manages Connections.

DriverManagerDataSource (under the package org.springframework.jdbc.datasource) is the simplest implementation of a DataSource. By looking at the class name, you can guess that it simply calls DriverManager to obtain a connection. The fact that DriverManagerDataSource doesn't support database connection pooling makes this class unsuitable for anything other than testing. The configuration of DriverManagerDataSource is quite simple, as you can see in the following code snippet; you just need to supply the driver class name, a connection URL, a username, and a password (drivermanager-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>
```

You most likely recognize the properties in the listing. They represent the values you normally pass to JDBC to obtain a Connection interface. The database connection information typically is stored in a properties file for easy maintenance and substitution in different deployment environments. The following snippet shows a sample jdbc.properties from which Spring's property placeholder will load the connection information:

```
jdbc.driverClassName=com.mysql.cj.jdbc.Driver jdbc.url=jdbc:mysql://localhost:3306/
musicdb?useSSL=true
jdbc.username=prospring5
jdbc.password=prospring5
```

By adding the util namespace in the mix, the same configuration can be written like this as well (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>
```

One change is required here. The properties are loaded into a `java.util.Properties` bean named `jdbc`, so you need to change the name of the property names in the properties file to be able to access them with the `jdbc` prefix. Here's `jdbc2.properties` file content:

```
driverClassName=com.mysql.cj.jdbc.Driver
url=jdbc:mysql://localhost:3306/musicdb?useSSL=true
username=prospring5
password=prospring5
```

Since this book is focused more on Java `Configuration` classes, here is a configuration class as well:

```
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;
            }
    }
}
```

To test any of these classes, you can use the following test class:

```
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();
            }
        }
    }
}
```

Again, a test class was used because it is more practical to reuse some of the code and also teach you to work with JUnit to quickly write tests for any piece of code you write. The first method, `testOne()`, is used to test XML configurations, and the second is used to test the `DbConfig` configuration class. After obtaining the `dataSource` bean from any configuration, the mock query `SELECT 1` is used to test the connection to the MySQL database.

In real-world applications, you can use the Apache Commons BasicDataSource[3] or a DataSource implemented by a JEE application server (for example, JBoss, WebSphere, WebLogic, or GlassFish), which may further increase the performance of the application. You could use a DataSource in the plain JDBC code and get the same pooling benefits; however, in most cases, you would still miss a central place to configure the DataSource. Spring, on the other hand, allows you to declare a dataSource bean and set the connection properties in the ApplicationContext definition files. See the following configuration sample; the file name is 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>
```

This particular Spring-managed DataSource is implemented in org.apache.commons.dbcp.BasicDataSource. The most important bit is that the dataSource bean implements javax.sql.DataSource, and you can immediately start using it in your data access classes.

Another way to configure a dataSource bean is to use JNDI. If the application you are developing is going to run in a JEE container, we can take advantage of the container-managed connection pooling. To use a JNDI-based data source, you need to change the dataSource bean declaration, as shown in the following example (datasource-jndi.xml):

```
<beans ...>
    <bean id="dataSource"
      class="org.springframework.jndi.JndiObjectFactoryBean"

      p:jndiName="java:comp/env/jdbc/musicdb"/>
</beans>
```

In the previous example, we use Spring's JndiObjectFactoryBean to obtain the data source by JNDI lookup. Starting from version 2.5, Spring provides the jee namespace, which further simplifies the configuration. Here you can see the same JNDI data source configuration using the jee namespace (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>
```

[3]Here is the official site: http://commons.apache.org/dbcp.

In the previous configuration snippet, we declare the jee namespace in the <beans> tag and then the <jee:jndi-lookup> tag to declare the data source. If you take the JNDI approach, you must not forget to add a resource reference (resourceref) in the application descriptor file. See the following code snippet:

```
<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>
```

<root-node> is a placeholder value; you need to change it depending on how your module is packaged. For example, it becomes <web-app> in the web deployment descriptor (WEB-INF/web.xml) if the application is a web module. Most likely, you will need to configure resource-ref in an application server–specific descriptor file as well. However, notice that the resource-ref element configures the jdbc/musicdb reference name and that the dataSource bean's jndiName is set to java:comp/env/jdbc/musicdb.

As you can see, Spring allows you to configure the DataSource in almost any way you like, and it hides the actual implementation or location of the data source from the rest of the application's code. In other words, your DAO classes do not know and do not need to know where the DataSource points.

The connection management is also delegated to the dataSource bean, which in turn performs the management itself or uses the JEE container to do all the work.

Embedded Database Support

Starting from version 3.0, Spring also offers embedded database support, which automatically starts an embedded database and exposes it as a DataSource for the application. The following configuration snippet shows the configuration of an embedded database (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>
```

In the previous listing, we first declare the jdbc namespace in the <beans> tag. Afterward, we use <jdbc:embedded-database> to declare the embedded database and assign it an ID of dataSource. Within the tag, we also instruct Spring to execute the scripts specified to create the database schema and populate testing data accordingly. Note that the order of the scripts is important, and the file that contains Data Definition Language (DDL) should always appear first, followed by the file with Data Manipulation Language (DML). For the type attribute, we specify the type of embedded database to use. As of version 4.0, Spring supports HSQL (the default), H2, and DERBY.

Embedded databases can be configured using Java Configuration classes too. The class to use is EmbeddedDatabaseBuilder, which uses the database creation and loading data scripts as arguments to create an instance of EmbeddedDatabase that implements DataSource.

```
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;
        }
    }
...
}
```

The embedded database support is extremely useful for local development or unit testing. Throughout the rest of this chapter, we use the embedded database to run the sample code, so your machine doesn't require a database to be installed in order to run the samples.

Not only can you utilize the embedded database support via the JDBC namespace but also initialize a database instance running elsewhere, such as MySQL, Oracle, and so on. Rather than specifying type and embedded-database, simply use initialize-database, and your scripts will execute against the intended dataSource just as they would with an embedded database.

Using DataSources in DAO Classes

The Data Access Object (DAO) pattern is used to separate low-level data accessing APIs or operations from high-level business services. The Data Access Object pattern requires the following components:

- *DAO interface*: This defines the standard operations to be performed on a model object (or objects).
- *DAO implementation*: This class provides a concrete implementation for the DAO interface. Typically this uses a JDBC connection or data source to handle model object (or objects).
- *Model objects also called data objects or entities*: This is a simple POJO mapping to a table record.

Let's create a `SingerDao` interface to implement for the sample, as shown here:

```
package com.apress.prospring5.ch6.dao;

public interface SingerDao {
    String findNameById(Long id);
}
```

For the simple implementation, first we will add a `dataSource` property to the `JdbcSingerDao` implementation class. The reason we want to add the `dataSource` property to the implementation class rather than the interface should be quite obvious: the interface does not need to know how the data is going to be retrieved and updated. By adding `DataSource` mutator methods to the interface, in the best-case scenario this forces the implementations to declare the getter and setter stubs. Clearly, this is not a very good design practice. Take a look at the simple `JdbcSingerDao` class shown here:

```
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");
        }
```

```
    }
...
}
```

We can now instruct Spring to configure our singerDao bean by using the JdbcSingerDao implementation and set the dataSource property as shown in the following EmbeddedJdbcConfig configuration class:

```
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;
    }
}
```

Spring supports quite a few embedded databases, but they have to be added as dependencies to the projects. Here you can see a few database-specific libraries being configured for use in 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"
    ]
}
```

Spring now creates the singerDao bean by instantiating the JdbcSingerDao class with the dataSource property set to the dataSource bean. It is good practice to make sure that all required properties on a bean have been set. The easiest way to do this is to implement the InitializingBean interface and provide an implementation for the afterPropertiesSet() method. This way, we make sure that all required properties have been set on our JdbcSingerDao. For further discussion of bean initialization, refer to Chapter 4.

The code we have looked at so far uses Spring to manage the data source and introduces the SingerDao interface and its JDBC implementation. We also set the dataSource property on the JdbcSingerDao class in the Spring ApplicationContext file. Now we expand the code by adding the actual DAO operations to the interface and implementation.

Exception Handling

Because Spring advocates using runtime exceptions rather than checked exceptions, you need a mechanism to translate the checked SQLException into a runtime Spring JDBC exception. Because Spring's SQL exceptions are runtime exceptions, they can be much more granular than checked exceptions. By definition, this is not a feature of runtime exceptions, but it is inconvenient to have to declare a long list of checked exceptions in the throws clause; hence, checked exceptions tend to be much more coarse-grained than their runtime equivalents.

Spring provides a default implementation of the SQLExceptionTranslator interface, which takes care of translating the generic SQL error codes into Spring JDBC exceptions. In most cases, this implementation is sufficient enough, but you can extend Spring's default implementation and set your new SQLExceptionTranslator implementation to be used in JdbcTemplate, as shown in the following code sample:

```
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;
    }
}
```

At the same time, we need to add the dependency on spring-jdbc into the project. Here you can see spring-jdbc being configured for use in pro-spring-15/gradle.build:

```
ext {
    springVersion = '5.0.0.M4'
    ...

    spring = [
        jdbc   : "org.springframework:spring-jdbc:$springVersion",
        ...
    ]
}
```

This library is added as a dependency to all JDBC projects used as samples for this chapter.

```
//chapter06/spring-jdbc-embedded/build.gradle
dependencies {
    compile spring.jdbc
    compile db.h2,  db.derby
}
```

To use the custom translator, we need to pass it into JdbcTemplate in the DAO classes. The following code snippet represents a piece of the enhanced JdbcSingerDao.setDataSource() method to illustrate its usage:

```
...
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;
    }
...
}
```

Having the custom SQL exception translator in place, Spring will invoke it upon SQL exceptions detected when executing SQL statements against the database, and custom exception translation will happen when the error code is -12345. For other errors, Spring will fall back to its default mechanism for exception translation. Obviously, nothing can stop you from creating SQLExceptionTranslator as a Spring-managed bean and using the JdbcTemplate bean in your DAO classes. Don't worry if you don't remember reading about the JdbcTemplate class; we are going to discuss it in more detail.

The JdbcTemplate Class

This class represents the core of Spring's JDBC support. It can execute all types of SQL statements. In the most simplistic view, you can classify the data definition and data manipulation statements. Data definition statements cover creating various database objects (tables, views, stored procedures, and so on). Data manipulation statements manipulate the data and can be classified as select and update statements. A select statement generally returns a set of rows; each row has the same set of columns. An update statement modifies the data in the database but does not return any results.

The `JdbcTemplate` class allows you to issue any type of SQL statement to the database and return any type of result.

In this section, we will go through several common use cases for JDBC programming in Spring with the `JdbcTemplate` class.

Initializing JdbcTemplate in a DAO Class

Before discussing how to use `JdbcTemplate`, let's take a look at how to prepare `JdbcTemplate` for use in the DAO class. It's straightforward; most of the time you just need to construct the class by passing in the data source object (which should be injected by Spring into the DAO class). The following code snippet shows the code snippet that will initialize the `JdbcTemplate` object:

```
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);
    }
    ...
}
```

The general practice is to initialize `JdbcTemplate` within the `setDataSource` method so that once the data source is injected by Spring, `JdbcTemplate` will also be initialized and ready for use.

Once configured, `JdbcTemplate` is thread-safe. That means you can also choose to initialize a single instance of `JdbcTemplate` in Spring's configuration and have it injected into all DAO beans. A configuration like this is depicted here:

```
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;
    }
}
```

Retrieving a Single-Value by Using JdbcTemplate

Let's start with a simple query that returns a single value. For example, we want to be able to retrieve the name of a singer by its ID. Using `jdbcTemplate`, we can retrieve the value easily. The following code snippet shows the implementation of the *findNameById()* method in the *JdbcSingerDao* class. For other methods, empty implementations were created.

```
...
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");
    }

...
}
```

In the previous listing, we use the queryForObject() method of JdbcTemplate to retrieve the value of the first name. The first argument is the SQL string, and the second argument consists of the parameters to be passed to the SQL for parameter binding in object array format. The last argument is the type to be returned, which is String in this case. Besides Object, you can also query for other types such as Long and Integer. Let's take a look at the outcome. The following code snippet shows the testing program. Again, a JUnit test class will be used because this allows us to run test methods separately, and as tests are run when executing glade build, we are also ensuring that our build remains stable.

```
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));
    }
}
```

When executing the test method testH2DB(), we expect the John Mayer string to be returned by the singerDao.findNameById(1l) call, and we test this assumption using the asertTrue method. If there were any problems when initializing the database, this test will fail.

Using Named Parameters with NamedParameterJdbcTemplate

In the previous example, we are using the normal placeholder (the ? character) as a query parameter, and we need to pass the parameter values as an Object array. When using a normal placeholder, the order is important, and the order that you put the parameters into the array should be the same as the order of the parameters in the query.

Some developers prefer to use named parameters to ensure that each parameter is being bound exactly as intended. In Spring, an extension of the JdbcTemplate class, called NamedParameterJdbcTemplate (under the package org.springframework.jdbc.core.namedparam), provides support for this.

The initialization of NamedParameterJdbcTemplate is the same as JdbcTemplate, so we just need to declare a bean of type NamedParameterJdbcTemplate and inject it in the Dao class. In the following code, you can see the new and improved 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");
        }
    }
    ...
}
```

You will see that instead of the ? placeholder, the named parameter (prefixed by a semicolon) is used instead: :singerId. The following code snippet can used to test the new 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();
    }
}
```

When executing the test method `testCfg()`, we expect the `John Mayer` string to be returned by the `singerDao.findNameById(1l)` call, and we test this assumption using the `asertTrue` method. If there were any problems when initializing the database, this test will fail.

Retrieving Domain Objects with RowMapper<T>

Rather than retrieving a single value, most of the time you will want to query one or more rows and then transform each row into the corresponding domain object or entity. Spring's `RowMapper<T>` interface (under the package `org.springframework.jdbc.core`) provides a simple way for you to perform mapping from a JDBC result set to POJOs. Let's see it in action by implementing the `findAll()` method of the `SingerDao` using the `RowMapper<T>` interface. In the following code snippet, you can see the implementation of the `findAll()` method:

```
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");
      }
   }
}
```

In the previous code snippet, we define a static inner class called SingerMapper that implements the RowMapper<Singer> interface. The class needs to provide the mapRow() implementation, which transforms the values in a specific record of the ResultSet into the domain object you want. Making it a static inner class allows you to share the RowMapper<Singer> among multiple finder methods.

The class SingerMapper explicit implementation can be skipped altogether using Java 8 lambda expressions; thus, the findAll() method can be refactored like this:

```
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;
    });
}
```

Afterward, the findAll() method just needs to invoke the query method and pass in the query string and the row mapper. If the query requires parameters, the query() method provides an overloaded method that accepts the query parameters. The following test class contains a test method for the findAll() method:

```
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();
    }
}
```

If you run the testRowMapper method, the test must pass, and the following must be produced:

```
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
```

The albums are not printed because the RowMapper<Singer> implementation does not actually set them on the returned Singer instances.

Retrieving Nested Domain Objects with ResultSetExtractor

Let's proceed to a somewhat more complicated example, in which we need to retrieve the data from the parent (SINGER) and child (ALBUM) tables with a join and then transform the data back into the nested object (Album within Singer) accordingly.

The previously mentioned RowMapper<T> is suitable only for row mapping to a single domain object. For a more complicated object structure, we need to use the ResultSetExtractor interface. To demonstrate its use, let's add one more method, findAllWithAlbums(), into the SingerDao interface. The method should populate the list of singers with their album details.

The following code snippet shows the addition of the findAllWithAlbums() method to the interface and the implementation of the method using 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");
        }
    }
}
```

The code looks quite like the RowMapper sample, but this time we declare an inner class that implements ResultSetExtractor. Then we implement the extractData() method to transform the result set into a list of Singer objects accordingly. For the findAllWithDetail() method, the query uses a left join to join the two tables so that singers with no albums will also be retrieved. The result is a Cartesian product of the two tables. Finally, we use the JdbcTemplate.query() method, passing in the query string and the result set extractor.

Of course, the SingerWithDetailExtractor inner class is not actually necessary because lambda expressions are used. Here you can see the findAllWithAlbums() version that makes use of Java 8 lambda expressions:

```
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());
        });

}
```

The following test class contains a test method for the `findAllWithAlbums()` method:

```
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();
    }
}
```

If you run the testResultSetExtractor() method, the test must pass, and the following must be produced:

```
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
```

You can see the singers and their album details are listed accordingly. The data is based on the data population script that you can find in resources/db/test-data.sql for each of the JDBC sample projects. So far, you have seen how to use JdbcTemplate to perform some common query operations. JdbcTemplate (and the NamedParameterJdbcTemplate class too) provides a number of overloading update() methods that support data update operations, including insert, update, delete, and so on. However, the update() method is quite self-explanatory, so we leave it as an exercise for you to explore. On the other side, as you will see in later sections, we will use the Spring-provided SqlUpdate class to perform data update operations.

Spring Classes That Model JDBC Operations

In the preceding section, you saw how JdbcTemplate and the related data mapper utility classes greatly simplify the programming model in developing data access logic with JDBC. Built on top of JdbcTemplate, Spring also provides a number of useful classes that model JDBC data operations and let developers maintain the query and transformation logic from ResultSet to domain objects in a more object-oriented fashion. Specifically, this section presents the following classes:

- MappingSqlQuery<T>: The MappingSqlQuery<T> class allows you to wrap the query string together with the mapRow() method into a single class.
- SqlUpdate: The SqlUpdate class allows you to wrap any SQL update statement into it. It also provides a lot of useful functions for you to bind SQL parameters, retrieve the RDBMS-generated key after a new record is inserted, and so on.
- BatchSqlUpdate: As its name indicates, this class allows you to perform batch update operations. For example, you can loop through a Java List object and have the BatchSqlUpdate queue up the records and submit the update statements for you in a batch. You can set the batch size and flush the operation at any time.
- SqlFunction<T>: The SqlFunction<T> class allows you to call stored functions in the database with argument and return types. Another class, StoredProcedure, also exists that helps you invoke stored procedures.
- Setting up JDBC DAO by using annotations

First let's take a look at how to set up the DAO implementation class by using annotations. The following example code implements the SingerDao interface method by method until we have a full SingerDao implementation. The following code snippet shows the SingerDao interface class with a complete listing of the data access services it provides:

```
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);
}
```

At the beginning of the book, stereotype annotations were introduced, and `@Repository` was introduced as a specialization of the `@Component` annotation, which is designed to be used for beans handling database operations.[4] The following code snippet shows the initial declaration and injection of the data source property into a `@Repository` annotated DAO class using the JSR-250 annotation:

```
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;
    }
    ...
}
```

In the previous listing, we use `@Repository` to declare the Spring bean with a name of `singerDao`, and since the class contains data access code, `@Repository` also instructs Spring to perform database-specific SQL exceptions to the more application-friendly `DataAccessException` hierarchy in Spring.

We also declare the log variable by using SL4J logging to log the message within the program. We use JSR-250's `@Resource` for the `dataSource` property to let Spring inject the data source with a name of `dataSource`.

---

[4]This indicates that the annotated class is a repository, originally defined by *Domain-Driven Design* (Evans, 2003) as "a mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects."

In the following code sample, you can see the Java configuration class for the situation when annotations are used to declare DAO handling beans:

```
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;
        }
    }
}
```

In this configuration, we declare a MySQL database, which is being accessed using a poolable `BasicDataSource`, and use component scanning for automatic Spring bean discovery. You were instructed at the beginning of the chapter how to install and set up a MySQL database and create the `musicdb` schema. Having the infrastructure in place, we can now proceed to the implementation of JDBC operations.

Querying Data by Using MappingSqlQuery<T>

Spring provides the MappingSqlQuery<T> class for modeling query operations. Basically, we construct a MappingSqlQuery<T> class by using the data source and the query string. We then implement the mapRow() method to map each ResultSet record into the corresponding domain object.

Let's begin by creating the SelectAllSingers class (which represents the query operation for selecting all singers) that extends the MappingSqlQuery<T> abstract class. The SelectAllSingers class is shown here:

```
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;
    }
}
```

Within the SelectAllSingers class, the SQL for selecting all singers is declared. In the class constructor, the super() method is called to construct the class, using the DataSource as well as the SQL statement. Moreover, the MappingSqlQuery<T>.mapRow() method is implemented to provide the mapping of the result set to the Singer domain object.

Having the SelectAllSingers class in place, we can implement the findAll() method in the JdbcSingerDao class. The following code snippet depicts a section of the JdbcSingerDao class:

```
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();
    }

    ...
}
```

In the setDataSource() method, upon the injection of the DataSource, an instance of the SelectAllSingers class is constructed. In the findAll() method, we simply invoke the execute() method, which is inherited from the SqlQuery<T> abstract class indirectly. That's all we need to do. The following code snippet shows the method used to test the findAll() method implemented this way:

```
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();
    }
}
```

Running the testing method, if the test passes, yields the following output:

```
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
```

If DEBUG logging is enabled for the `org.springframework.jdbc` package by editing the `logback-test.xml` configuration file and adding the following element:

```
<logger name="org.springframework.jdbc" level="debug"/>
```

then in the console you will also see the query that was submitted by Spring, as shown here:

```
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]
```

Let's proceed to implement the `findByFirstName()` method, which takes one named parameter. As in the previous sample, we create the class `SelectSingerByFirstName` for the operation, which is depicted here:

```
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;
    }
}
```

The `SelectSingerByFirstName` class is similar to the `SelectAllSingers` class. First, the SQL statement is different and carries a named parameter called `first_name`. In the constructor method, the `declareParameter()` method is called (which is inherited from the `org.springframework.jdbc.object.RdbmsOperation` abstract class indirectly). Let's proceed to implement the `findByFirstName()` method in the `JdbcSingerDao` class. Here you can see the updated code:

```
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);
    }
...
}
```

Upon data source injection, an instance of `SelectSingerByFirstName` is constructed. Afterward, in the `findByFirstName()` method, a `HashMap` is constructed with the named parameters and values. Finally, the `executeByNamedParam()` method (inherited from the `SqlQuery<T>` abstract class indirectly) is called. Let's test this method by executing the `testFindByFirstName()` test method shown here:

```
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();
    }
}
```

Running the testing method, if the test passes, yields the following output:

```
Singer - Id: 1, First name: John, Last name: Mayer, Birthday: 1977-10-15
```

One point worth noting here is that `MappingSqlQuery<T>` is suitable only for mapping a single row to a domain object. For a nested object, you still need to use `JdbcTemplate` with `ResultSetExtractor`, like the example method `findAllWithAlbums()` presented in the `JdbcTemplate` class section.

Updating Data by Using SqlUpdate

For updating data, Spring provides the `SqlUpdate` class. The following code snippet shows the `UpdateSinger` class that extends the `SqlUpdate` class for update operations:

```
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));
    }
}
```

The preceding listing should be familiar to you now. An instance of the SqlUpdate class is constructed with the query, and the named parameters are declared too. The following code snippet shows the implementation of the update() method in the JdbcSingerDao class:

```
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);
    }
    ...
}
```

Upon data source injection, an instance of UpdateSinger is constructed. In the update() method, a HashMap of named parameters is constructed from the passed-in Singer object, and then updateByNamedParam() is called to update the contact record. To test the operation, let's add a new method to the AnnotationJdbcTest.

```
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 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();
    }
}
```

Here we simply construct a `Singer` object and then invoke the `update()` method. Running the program produces the following output from the last `listSingers()` method. Running the testing method, if the test passes, yields the following output:

```
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
```

Inserting Data and Retrieving the Generated Key

For inserting data, we can also use the SqlUpdate class. One interesting point is how the primary key is generated (which is typically the id column). This value is available only after the insert statement has completed; that's because the RDBMS generates the identity value for the record on insert. The column ID is declared with the attribute AUTO_INCREMENT and is the primary key, and this value will be assigned by the RDBMS during the insert operation. If you are using Oracle, you will probably get a unique ID first from an Oracle sequence and then execute an insert statement with the query result.

In old versions of JDBC, the method is a bit tricky. For example, if we are using MySQL, you need to execute the SQL select last_insert_id() and select @@IDENTITY statements for Microsoft SQL Server.

Luckily, starting from JDBC version 3.0, a new feature was added that allows the retrieval of an RDBMS-generated key in a unified fashion. The following code snippet shows the implementation of the insert() method, which also retrieves the generated key for the inserted contact record. It will work in most databases (if not all); just make sure you are using a JDBC driver that is compatible with JDBC 3.0 or newer.

We start by creating the InsertSinger class for the insert operation, which extends the SqlUpdate class.

```
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);
    }
}
```

The InsertSinger class is almost the same as the UpdateSinger class; we need to do just two more things. When constructing the InsertSinger class, we call the method SqlUpdate.setGeneratedKeysColumnNames() to declare the name of the ID column. The method SqlUpdate.setReturnGeneratedKeys() then instructs the underlying JDBC driver to retrieve the generated key. In the following code, you can see the implementation of the insert() method in the JdbcSingerDao class:

```
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);
    }
...
}
```

Upon data source injection, an instance of InsertSinger is constructed. In the insert() method, we also use the SqlUpdate.updateByNamedParam() method. Additionally, we pass in an instance of KeyHolder to the method, which will have the generated ID stored in it. After the data is inserted, we can then retrieve the generated key from the KeyHolder. Let's see how the test method looks for 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();
    }
}
```

Running the testing method, if the test passes, yields the following output:

```
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
```

Batching Operations with BatchSqlUpdate

For batch operations, we use the `BatchSqlUpdate` class. The new `insertWithAlbum()` method will insert both the singer and its released album into the database. To be able to insert the album record, we need to create the `InsertSingerAlbum` class, which is shown here:

```
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);
    }
}
```

Note that in the constructor, we call the `BatchSqlUpdate.setBatchSize()` method to set the batch size for the JDBC insert operation. Here you can see the implementation of the `insertWithAlbum()` method in the `JdbcSingerDao` class:

```
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.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, 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());
        }
    }
...
}
```

Each time the insertWithAlbum() method is called, a new instance of InsertSingerAlbum is constructed because the BatchSqlUpdate class is not thread safe. Then we use it just like SqlUpdate. The main difference is that the BatchSqlUpdate class will queue up the insert operations and submit them to the database in batch. Every time the number of records equals the batch size, Spring will execute a bulk insert operation to the database for the pending records. On the other hand, upon completion, we call the BatchSqlUpdate.flush() method to instruct Spring to flush all pending operations (that is, the insert operations being queued that still haven't reached the batch size yet). Finally, we loop through the list of

Album objects in the Singer object and invoke the BatchSqlUpdate.updateByNamedParam() method. To facilitate testing, the insertWithAlbum() method is also implemented. As this implementation is pretty big, it can be reduced by making use of Java 8 lambda expressions.

```
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());
        });
}
```

Let's see how the test method looks for 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();
    }
}
```

Running the testing method, if the test passes, yields the following output:

```
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
```

Calling Stored Functions by Using SqlFunction

Spring also provides classes to simplify the execution of stored procedures/functions using JDBC. In this section, we show you how to execute a simple function by using the SqlFunction class. We show how to use MySQL for the database, create a stored function, and call it by using the SqlFunction<T> class.

We're assuming you have a MySQL database with a schema called musicdb, with a username and password both equaling prospring5 (the same as the example in the section "Exploring the JDBC Infrastructure"). Let's create a stored function called getFirstNameById(), which accepts the contact's ID and returns the first name of the contact. The following code shows the script to create the stored function in MySQL (stored-function.sql). Run the script against the MySQL database.

```
DELIMITER //
CREATE FUNCTION getFirstNameById(in_id INT)
    RETURNS VARCHAR(60)
    BEGIN
        RETURN (SELECT first_name FROM singer WHERE id = in_id);
    END //
DELIMITER ;
```

The stored function simply accepts the ID and returns the first name of the singer record with the ID. Next we create a StoredFunctionFirstNameById class to represent the stored function operation, which extends the SqlFunction<T> class. You can see the content of the class in the following code snippet:

```
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();
    }
}
```

Here the class extends `SqlFunction<T>` and passes in the type `String`, which indicates the return type of the function. Then we declare the SQL to call the stored function in MySQL. Afterward, in the constructor, the parameter is declared, and we compile the operation. The class is now ready for use in the implementation class. The following code snippet shows the updated `JdbcSingerDao` class to use the stored function:

```
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);
    }
    ...
}
```

Upon data source injection, an instance of `StoredFunctionFirstNameById` is constructed. Then in the `findFirstNameById()` method, its `execute()` method is called, passing in the contact ID. The method will return a list of `String`s, and we need only the first one, because there should be only one record returned in the result set. Testing this functionality is quite simple.

```
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();
    }
}
```

In the program, we pass an ID of 2 into the stored function. This will return `Eric`, which is the first name of the record with ID equal to 2 if you run the `test-data.sql` against the MySQL database. Running the program produces the following output:

```
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
```

You can see that the first name is retrieved correctly. What is presented here is just a simple sample to demonstrate the Spring JDBC module's function capabilities. Spring also provides other classes (for example, `StoredProcedure`) for you to invoke complex stored procedures that return complex data types. We recommend you refer to Spring's reference manual in case you need to access stored procedures using JDBC.

Spring Data Project: JDBC Extensions

In recent years, database technology has evolved so quickly with the rise of so many purpose-specific databases, nowadays an RDBMS is not the only choice for an application's back-end database. In response to this database technology evolution and the developer community's needs, Spring created the Spring Data project (`http://springsource.org/spring-data`). The major objective of the project is to provide useful extensions on top of Spring's core data access functionality to interact with databases other than traditional RDBMSs.

The Spring Data project comes with various extensions. One that we would like to mention here is JDBC Extensions (`http://springsource.org/spring-data/jdbc-extensions`). As its name implies, the extension provides some advanced features to facilitate the development of JDBC applications using Spring. The main features that JDBC Extensions provides are listed here:

- *QueryDSL support*: QueryDSL (`http://querydsl.com`) is a domain-specific language that provides a framework for developing type-safe queries. Spring Data's JDBC Extensions provides `QueryDslJdbcTemplate` to facilitate the development of JDBC applications using QueryDSL instead of SQL statements.
- *Advanced support for Oracle Database*: The extension provides advanced features for Oracle Database users. On the database connection side, it supports Oracle-specific session settings, as well as Fast Connection Failover technology when working with Oracle RAC. Also, classes that integrate with Oracle Advanced Queueing are provided. On the data type side, native support for Oracle's XML types, `STRUCT` and `ARRAY`, and so on, are provided.

If you are developing JDBC applications using Spring with Oracle Database, JDBC Extensions is really worth a look.

Considerations for Using JDBC

With this rich feature set, you can see how Spring can make your life much easier when using JDBC to interact with the underlying RDBMS. However, there is still quite a lot of code you need to develop, especially when transforming the result set into the corresponding domain objects.

On top of JDBC, a lot of open source libraries have been developed to help close the gap between the relational data structure and Java's OO model. For example, iBATIS is a popular `DataMapper` framework that is also based on SQL mapping. iBATIS lets you map objects with stored procedures or queries to an XML

descriptor file. Like Spring, iBATIS provides a declarative way to query object mapping, greatly saving you the time it takes to maintain SQL queries that may be scattered around various DAO classes.

There are also many other ORM frameworks that focus on the object model, rather than the query. Popular ones include Hibernate, EclipseLink (also known as TopLink), and OpenJPA. All of them comply with the JCP's JPA specification.

In recent years, these ORM tools and mapping frameworks have become much more mature so that most developers will settle on one of them, instead of using JDBC directly. However, in cases where you need to have absolute control over the query that will be submitted to the database for performance purposes (for example, using a hierarchical query in Oracle), Spring JDBC is really a viable option. And when using Spring, one great advantage is that you can mix and match different data access technologies. For example, you can use Hibernate as the main ORM and then JDBC as a supplement for some of the complex query logic or batch operations; you can mix and match them in a single business operation and then wrap them under the same database transaction. Spring will help you handle those situations easily.

Spring Boot JDBC

As we've already introduced Spring Boot for web and simple console applications, it is only logical to cover a Spring Boot starter library for JDBC in this book. It helps you remove boilerplate configurations and jump directly into implementation.

When `spring-boot-starter-jdbc` is added as a dependency to a project, a set of libraries is added to the project. What is not added is a database driver. That decision must be taken by the developer. The project that will be covered in this section is `spring-boot-jdbc` and is a submodule of the `chapter06` project. Its Gradle dependencies and versions are specified in the following parent `build.gradle` file:

```
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",
        ..
     ]
    ...
}
```

They are declared as dependencies in the `spring-boot-jdbc\build.gradle` configuration file, only by using their property names.

```
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
}
```

The autoconfigured libraries are depicted in Figure 6-3 in the IntelliJ IDEA Gradle Projects view. The `spring-boot-starter-jdbc` library uses `tomcat-jdbc` to configure the `DataSource` bean. Thus, if there is no `DataSource` bean explicitly configured and there is an embedded database driver in the classpath, Spring Boot will automatically register the `DataSource` bean using in-memory database settings. Spring Boot also registers the following beans automatically:

- A `JdbcTemplate` bean
- A `NamedParameterJdbcTemplate` bean
- A `PlatformTransactionManager (DataSourceTransactionManager)` bean

:chapter06:spring-boot-jdbc
Source Sets
main
Dependencies
plain-jdbc:main (Compile)
org.springframework.boot:spring-boot-starter-jdbc:2.0.0.BUILD-SNAPSHOT (Compile)
org.springframework:spring-jdbc:5.0.0.M4 (Compile)
org.springframework.boot:spring-boot-starter:2.0.0.BUILD-SNAPSHOT (Compile)
org.apache.tomcat:tomcat-jdbc:8.5.13 (Compile)
com.h2database:h2:1.4.194 (Compile)
test
Tasks

Figure 6-3. *Spring Boot JDBC starter dependencies*

Here are other things that might be interesting that reduce the amount of environment setup work:

- Spring Boot looks for embedded database initialization files under `src/main/resources`. It expects to find a file named `schema.sql` that contains SQL DDL statements (for example, `CREATE TABLE` statements) and a file named `data.sql` that contains DML statements (for example, `INSERT` statements). It uses this file to initialize a database at boot time.
- The location and names for these files can be configured in the `application.properties` files, which is located under `src/main/resources` as well. A sample configuration file that would allow the Spring Boot application to use SQL files is shown here:

```
spring.datasource.schema=db/schema.sql
spring.datasource.data=db/test-data.sql
```

- By default Spring Boot initializes the database at boot time, but this could be changed as well by adding the property `spring.datasource.initialize=false` to the `application.properties` file.

Aside from all that was mentioned, what is left to do with Spring Boot is to provide some domain classes or entities and a DAO bean. The `Singer` bean is the same as the one used everywhere in this chapter, and its implementation resides in the `chapter06/plain-jdbc` project, which is added as a dependency everywhere. The `JdbcSingerDao` class to use is shown here:

```
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);
    }
    ...
}
```

The Spring Boot entrypoint class, the one annotated with `@SpringBootApplication`, is shown in the following code sample. Notice how how simple it is.

```
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();
        }
}
```

Summary

This chapter showed you how to use Spring to simplify JDBC programming. You learned how to connect to a database and perform selects, updates, deletes, and inserts, as well as call database stored functions. Using the core Spring JDBC class, `JdbcTemplate`, was discussed in detail. We covered other Spring classes that are built on top of `JdbcTemplate` and that help you model various JDBC operations. We also showed how to use the new lambda expressions from Java 8 where appropriate. In addition, Spring Boot JDBC was covered because whatever helps you focus more on the implementation of the business logic of an application and less on the configurations is a great tool to know. In the next couple of chapters, we discuss how to use Spring with popular ORM technologies when developing data access logic.

CHAPTER 7

Using Hibernate in Spring

In the previous chapter, you saw how to use JDBC in Spring applications. However, even though Spring goes a long way toward simplifying JDBC development, you still have a lot of code to write. In this chapter, we cover one of the object-relational mapping (ORM) libraries called Hibernate.

If you have experience developing data access applications using EJB entity beans (prior to EJB 3.0), you may remember the painful process. Tedious configuration of mappings, transaction demarcation, and much boilerplate code in each bean to manage its life cycle greatly reduced productivity when developing enterprise Java applications.

Just like Spring was developed to embrace POJO-based development and declarative configuration management rather than EJB's heavy and clumsy setup, the developer community realized that a simpler, lightweight, and POJO-based framework could ease the development of data access logic. Since then, many libraries have appeared; they are generally referred to as *ORM libraries*. The main objectives of an ORM library are to close the gap between the relational data structure in the relational database management system (RDBMS) and the object-oriented (OO) model in Java so that developers can focus on programming with the object model and at the same time easily perform actions related to persistence.

Out of the ORM libraries available in the open source community, Hibernate is one of the most successful. Its features, such as a POJO-based approach, ease of development, and support of sophisticated relationship definitions, have won the heart of the mainstream Java developer community.

Hibernate's popularity has also influenced the JCP, which developed the Java Data Objects (JDO) specification as one of the standard ORM technologies in Java EE. Starting from EJB 3.0, the EJB entity bean was even replaced with the Java Persistence API (JPA). JPA has a lot of concepts that were influenced by popular ORM libraries such as Hibernate, TopLink, and JDO. The relationship between Hibernate and JPA is also very close. Gavin King, the founder of Hibernate, represented JBoss as one of the JCP expert group members in defining the JPA specification. Starting from version 3.2, Hibernate provides an implementation of JPA. That means when you develop applications with Hibernate, you can choose to use either Hibernate's own API or the JPA API with Hibernate as the persistence service provider.

Having offered a brief history of Hibernate, this chapter will cover how to use Spring with Hibernate when developing data access logic. Hibernate is such an extensive ORM library that covering every aspect in just one chapter is simply not possible, and numerous books are dedicated to discussing Hibernate.

This chapter covers the basic ideas and main use cases of Hibernate in Spring. In particular, we discuss the following topics:

- *Configuring the Hibernate SessionFactory*: The core concept of Hibernate revolves around the `Session` interface, which is managed by `SessionFactory`. We show you how to configure Hibernate's session factory to work in a Spring application.
- *Major concepts of ORMs using Hibernate*: We go through the major concepts of how to use Hibernate to map a POJO to the underlying relational database structure. Some commonly used relationships, including one-to-many and many-to-many, are also discussed.

© Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho 2017
I. Cosmina et al., *Pro Spring 5*, https://doi.org/10.1007/978-1-4842-2808-1_7

- *Data operations*: We present examples of how to perform data operations (query, insert, update, delete) by using Hibernate in the Spring environment. When working with Hibernate, its `Session` interface is the main interface that you will interact with.

When defining object-to-relational mappings, Hibernate supports two configuration styles. One is configuring the mapping information in XML files, and the other is using Java annotations within the entity classes (in the ORM or JPA world, a Java class that is mapped to the underlying relational database structure is called an entity class). This chapter focuses on using the annotation approach for object-relational mapping. For the mapping annotation, we use the JPA standards (for example, under the `javax.persistence` package) because they are interchangeable with Hibernate's own annotations and will help you with future migrations to a JPA environment.

Sample Data Model for Example Code

Figure 7-1 shows the data model used in this chapter.

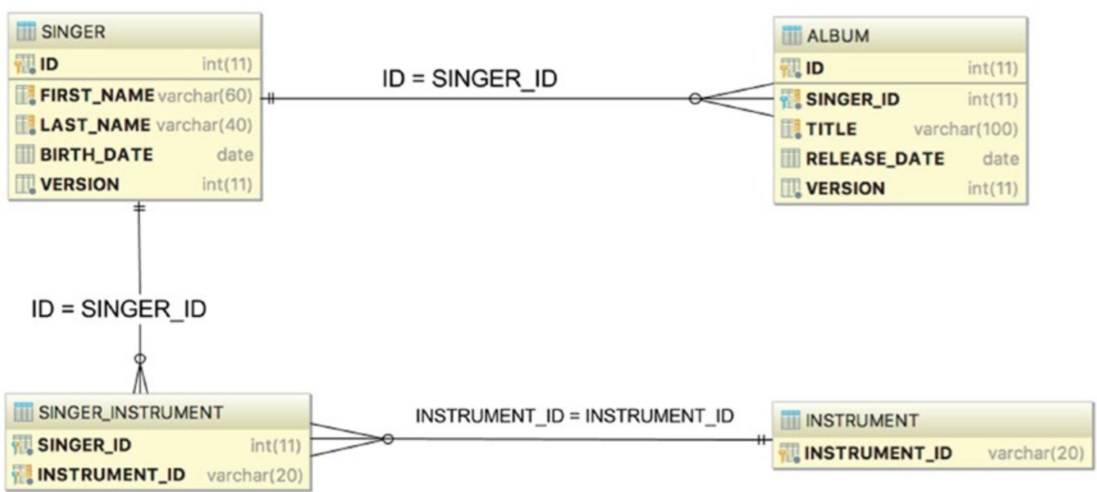

Figure 7-1. *Sample data model*

As shown in this data model, two new tables were added, namely, `INSTRUMENT` and `SINGER_INSTRUMENT` (the join table). `SINGER_INSTRUMENT` models the many-to-many relationships between the `SINGER` and `INSTRUMENT` tables. A `VERSION` column was added to the `SINGER` and `ALBUM` tables for optimistic locking, which will be discussed in detail later. In the examples in this chapter, we will use the embedded H2 database, so the database name is not required. Here are the scripts to create the tables needed for the examples in this chapter:

```
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)
);
```

The following SQL is the script for data population:

```
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');
```

Configuring Hibernate's SessionFactory

As mentioned earlier in this chapter, the core concept of Hibernate is based on the `Session` interface, which is obtained from `SessionFactory`. Spring provides classes to support the configuration of Hibernate's session factory as a Spring bean with the desired properties. To use Hibernate, you must add the Hibernate dependency as a dependency to the project. Here is the Gradle configuration used for the projects in this chapter:

```
//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"
  ]
  ...
}
//chapter07.gradle
dependencies {
  //we specify these dependencies for all submodules,
    except the boot module, that defines its own
  if !project.name.contains"boot" {
   compile spring.contextSupport, spring.orm,
   misc.slf4jJcl, misc.logback, db.h2, misc.lang3, [hibernate.em]
  }
  testCompile testing.junit
}
```

In the following configuration, you can see the XML elements needed to configure the application sample for this chapter:

```
<?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>
```

The equivalent Java configuration class is depicted next, and the components for these two configurations are explained in parallel after the code snippet:

```
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());
   }
}
```

In the previous configuration, several beans were declared to be able to support Hibernate's session factory. The main configurations are listed here:

- *The dataSource bean*: The database used is an H2 embedded one, declared as previously explained in Chapter 6.
- *The transactionManager bean*: The Hibernate session factory requires a transaction manager for transactional data access. Spring provides a transaction manager specifically for Hibernate 5 declared in package `org.springframework.orm.hibernate5.HibernateTransactionManager`. The bean was declared with the ID `transactionManager` assigned. By default, when using XML configuration, Spring will look up the bean with the name `transactionManager` within its `ApplicationContext` whenever transaction management is required. Java configuration is a little more flexible when the bean is being searched by its type, not by its name. We discuss transactions in detail in Chapter 9. In addition, we declare the tag `<tx:annotation-driven>` to support the declaration of transaction demarcation requirements using annotations. The equivalent for the Java configuration is the `@EnableTransactionManagement` annotation.
- *Component scan*: This tag and the `@ComponentScan` annotation should be familiar to you. We instruct Spring to scan the components under the package `com.apress.prospring5.ch7` to detect the beans annotated with `@Repository`.
- *Hibernate SessionFactory bean*: The `sessionFactory` bean is the most important part. Within the bean, several properties are provided. First, we need to inject the `dataSource` bean into the session factory. Second, we instruct Hibernate to scan for the domain objects under the package `com.apress.prospring5.ch.entities`. Finally, the `hibernateProperties` property provides configuration details for Hibernate. There are many configuration parameters, and we define only a few important properties that should be provided for every application. Table 7-1 lists the main configuration parameters for the Hibernate session factory.

Table 7-1. Hibernate Properties

Property	Description
hibernate.dialect	Specifies the database dialect for the queries that Hibernate should use. Hibernate supports the SQL dialects for many databases. Those dialects are subclasses of org.hibernate.dialect.Dialect. Major dialects include H2Dialect, Oracle10gDialect, PostgreSQLDialect, MySQLDialect, SQLServerDialect, and so on.
hibernate.max_fetch_depth	Declares the "depth" for outer joins when the mapping objects have associations with other mapped objects. This setting prevents Hibernate from fetching too much data with a lot of nested associations. A commonly used value is 3.
hibernate.jdbc.fetch_size	Specifies the number of records from the underlying JDBC ResultSet that Hibernate should use to retrieve the records from the database for each fetch. For example, a query was submitted to the database, and ResultSet contains 500 records. If the fetch size is 50, Hibernate will need to fetch 10 times to get all the data.
ibernate.jdbc.batch_size	Instructs Hibernate on the number of update operations that should be grouped together into a batch. This is useful for performing batch job operations in Hibernate. Obviously, when we are doing a batch job updating hundreds of thousands of records, we would like Hibernate to group the queries in batches, rather than submit the updates one by one.
hibernate.show_sql	Indicates whether Hibernate should output the SQL queries to the log file or console. You should turn this on in a development environment, which can greatly help in the testing and troubleshooting process.
hibernate.format_sql	Indicates whether SQL output in the log or console should be formatted.
hibernate..use_sql_comments	If set to true, Hibernate generates comments inside the SQL for easier debugging.

For the full list of properties that Hibernate supports, please refer to Hibernate's ORM user guide, specifically, Section 23, at https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html.

ORM Mapping Using Hibernate Annotations

Having the configuration in place, the next step is to model the Java POJO entity classes and their mapping to the underlying relational data structure.

There are two approaches to the mapping. The first one is to design the object model and then generate the database scripts based on the object model. For example, for the session factory configuration, you can pass in the Hibernate property hibernate.hbm2ddl.auto to have Hibernate automatically export the schema DDL to the database. The second approach is to start with the data model and then model the POJOs with the desired mappings. We prefer the latter approach because we can have more control over the data model, which is useful in optimizing the performance of data access. But the first will be covered later in the chapter to depict a different way of configuring a Spring application with Hibernate. Based on the data model, Figure 7-2 shows the corresponding OO model with a class diagram.

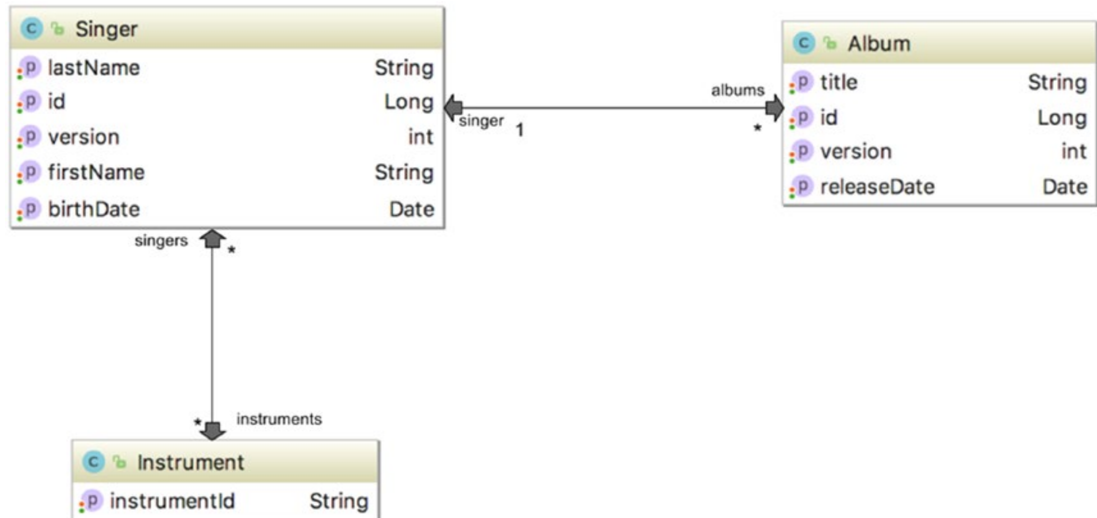

Figure 7-2. *Class diagram for the sample data model*

You can see there is a one-to-many relationship between `Singer` and `Album`, while there's a many-to-many relationship between the `Singer` and `Instrument` objects.

Simple Mappings

First let's start by mapping the simple attributes of the class. The following code snippet shows the `Singer` class with the mapping annotations:

```
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;
    }
}
```

First we annotate the type with `@Entity`, which means that this is a mapped entity class. The `@Table` annotation defines the table name in the database that this entity is being mapped to. For each mapped attribute, you annotate it with the `@Column` annotation, with the column names provided.

> **⚠** Table and column names can be skipped if the type and attribute names are the same as the table and column names.

About the mappings, we would like to highlight a few points.

- For the `birthDate` attribute, we annotate it with `@Temporal`, using the `TemporalType.DATE` value as an argument. This means we would like to map the data type from the Java date type (`java.util.Date`) to the SQL date type (`java.sql.Date`). This allows us to access the attribute `birthDate` in the `Singer` object by using `java.util.Date` as usual in our application.
- For the `id` attribute, we annotate it with `@Id`. This means it's the primary key of the object. Hibernate will use it as the unique identifier when managing the contact entity instances within its session. Additionally, the `@GeneratedValue` annotation tells Hibernate how the `id` value was generated. The `IDENTITY` strategy means that the `id` value was generated by the back end during insert.
- For the `version` attribute, we annotate it with `@Version`. This instructs Hibernate that we would like to use an optimistic locking mechanism, using the `version` attribute as a control. Every time Hibernate updates a record, it compares the version of the entity instance to that of the record in the database. If both versions are the same, it means that no one updated the data before, and Hibernate will update the data and increment the version column. However, if the version is not the same, it means that someone has updated the record before, and Hibernate will throw a `StaleObjectStateException` exception, which Spring will translate to `HibernateOptimisticLockingFailureException`. The example we used an integer for version control. In addition to an integer, Hibernate supports using a timestamp. However, using an integer for version control is recommended since Hibernate will always increment the version number by 1 after each update. When using a timestamp, Hibernate will update the latest timestamp after each update.
  A timestamp is slightly less safe because two concurrent transactions may both load and update the same item in the same millisecond.

Another mapped object is `Album`, as shown here:

```
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;
    }
}
```

Here is the third entity class used in the examples for this chapter:

```
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();
   }
}
```

One-to-Many Mappings

Hibernate has the capability to model many kinds of associations. The most common associations are one-to-many and many-to-many. Each Singer will have zero or more albums, so it's a one-to-many association (in ORM terms, the one-to-many association is used to model both zero-to-many and one-to-many relationships within the data structure). The following code snippet depicts the properties and the methods necessary to define the one-to-many relationship between the Singer and Album entities:

```
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;
    }
    ...
}
```

The getter method of the attribute contactTelDetails is annotated with @OneToMany, which indicates the one-to-many relationship with the Album class. Several attributes are passed to the annotation. The mappedBy attribute indicates the property in the Album class that provides the association (that is, linked up by the foreign-key definition in the FK_ALBUM_SINGER table). The cascade attribute means that the update operation should “cascade” to the child. The orphanRemoval attribute means that after the albums have been updated, those entries that no longer exist in the set should be deleted from the database. The following code snippet shows the updated code in the Album class for the association mapping:

```
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;
    }
    ...
}
```

We annotate the getter method of the `singer` attribute with `@ManyToOne`, which indicates it's the other side of the association from `Singer`. We also specify the `@JoinColumn` annotation for the underlying foreign-key column name. Finally, the `toString()` method is overridden to facilitate testing by printing its output to the console in the example code later.

Many-to-Many Mappings

Every singer can play zero or more instruments, and each instrument is also associated with zero or more singers, which means it's a many-to-many mapping. A many-to-many mapping requires a join table, which is `SINGER_INSTRUMENT`. The following code sample shows the code that needs to be added to the `Singer` class to implement this relationship:

```
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;
    }
    ...
}
```

The getter method of the attribute `instruments` in the `Singer` class is annotated with `@ManyToMany`. We also provide `@JoinTable` to indicate the underlying join table that Hibernate should look for. The name is the join table's name, `joinColumns` defines the column that is the foreign key to the `SINGER` table, and `inverseJoinColumns` defines the column that is the foreign key to the other side of the association (that is, the `INSTRUMENT` table). The following is the `Instrument` class with the code to implement the other side of this relationship added:

```
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();
    }
}
```

The mapping is more or less the same as the one for Singer, but the joinColumns and inverseJoinColumns attributes are reversed to reflect the association.

The Hibernate Session Interface

In Hibernate, when interacting with the database, the main interface you need to deal with is the Session interface, which is obtained from SessionFactory.

The following code snippet shows the SingerDaoImpl class used in the samples in this chapter and has the configured Hibernate SessionFactory injected into the class:

```
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;
   }
   ...
}
```

As usual, we declare the DAO class as a Spring bean by using the @Repository annotation. The @Transactional annotation defines the transaction requirements that we discuss further in Chapter 9. The sessionFactory attribute is injected by using the @Resource annotation.

```
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);
}
```

The interface is simple; it has just three finder methods, one save method, and one delete method. The `save()` method will perform both the insert and update operations.

Querying Data by Using the Hibernate Query Language

Hibernate, together with other ORM tools such as JDO and JPA, is engineered around the object model. So, after the mappings are defined, we don't need to construct SQL to interact with the database. Instead, for Hibernate, we use the Hibernate Query Language (HQL) to define our queries. When interacting with the database, Hibernate will translate the queries into SQL statements on our behalf.

When coding HQL queries, the syntax is quite like SQL. However, you need to think on the object side rather than database side. We will take you through several examples in the following sections.

Simple Querying with Lazy Fetching

Let's begin by implementing the `findAll()` method, which simply retrieves all the contacts from the database. The following code sample shows the updated code for this functionality:

```
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();
   }
   ...
}
```

The method SessionFactory.getCurrentSession() gets hold of Hibernate's Session interface. Then, the Session.createQuery() method is called, passing in the HQL statement. The statement from Singer s simply retrieves all contacts from the database. An alternative syntax for the statement is select s from Singer s. The @Transactional(readOnly=true) annotation means we want the transaction to be set as read-only. Setting that attribute for read-only methods will result in better performance.

The following code snippet shows a simple testing program for 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());
    }
   }
}
```

Running the previous class yields the following output:

```
---- Listing singers:
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
Singer - Id: 2, First name: Eric, Last name: Clapton, Birthday: 1945-03-30
```

Although the singer records were retrieved, what about the albums and instruments? Let's modify the testing class to print the details information. In the following code snippet, you can see the method `listSingers()` being replaced with `listSingersWithAlbum()`:

```
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());
     }
    }
    }
   }
}
```

If you run the program again, you will see the following exception:

```
---- Listing singers with instruments:
Singer - Id: 1, First name: John, Last name: Mayer, Birthday: 1977-10-16
org.hibernate.LazyInitializationException: failed to lazily initialize a
  collection of role: com.apress.prospring5.ch7.entities.Singer.albums,
  could not initialize proxy - no Session
```

You will see Hibernate throw the `LazyInitializationException` when you try to access the associations.

This is because, by default, Hibernate will fetch the associations *lazily*, which means that Hibernate will not join the association tables (that is, `ALBUM`) for records. The rationale behind this is for performance; as you can imagine, if a query is retrieving thousands of records and all the associations are retrieved, the massive amount of data transfer will degrade performance.

Querying with Associations Fetching

To have Hibernate fetch the data from associations, there are two options. First, you can define the association with the fetch mode `EAGER`, for example, `@ManyToMany(fetch=FetchType.EAGER)`. This tells Hibernate to fetch the associated records in every query. However, as discussed, this will impact data retrieval performance.

The other option is to force Hibernate to fetch the associated records in the query when required. If you use the `Criteria` query, you can call the function `Criteria.setFetchMode()` to instruct Hibernate to eagerly fetch the association. When using `NamedQuery`, you can use the `fetch` operator to instruct Hibernate to fetch the association eagerly.

Let's take a look at the implementation of the `findAllWithAlbum()` method, which will retrieve all contact information together with their telephone details and hobbies. This example will use the `NamedQuery` approach. `NamedQuery` can be externalized into an XML file or declared using an annotation on the entity class. Here you can see the revised `Singer` domain object with the named query defined using annotations:

```
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 {
   ...
}
```

First you define a NamedQuery instance called Singer.findAllWithAlbum. Then we define the query in HQL. Pay attention to the left join fetch clause, which instructs Hibernate to fetch the association eagerly. You also need to use select distinct; otherwise, Hibernate will return duplicate objects (two singer objects will be returned if a single singer has two albums associated with him).

Here is the implementation of the findAllWithAlbum() method:

```
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();
    }
}
```

This time we use the Session.getNamedQuery() method, passing in the name of the NamedQuery instance. Modifying the testing program (SpringHibernateDemo) to call singerDao.findAllWithAlbum() will yield the following output:

```
---- 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
```

Now all the singers with details were retrieved correctly. Let's see another example with NamedQuery with parameters. This time, we will implement the findById() method and would like to fetch the associations as well. The following code snippet shows the Singer class with the new named query added:

```
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 {
   ...
}
```

From the named query with the name `Singer.findById`, we declare a named parameter: `id`. Here you can see the implementation of the `findById()` method in `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();
   }
   ...
}
```

In this listing, we use the same `Session.getNameQuery()` method. But then we also call the `setParameter()` method, passing in the named parameter with its value. For multiple parameters, you can use the `setParameterList()` or `setParameters()` method of the `Query` interface.

There are also some more advanced query methods, such as native query and criteria query, which we discuss in the next chapter when we talk about JPA. To test the method, the `SpringHibernateDemo` class must be modified accordingly.

```
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();
   }
}
```

Running the program produces the following output:

```
Singer - Id: 1, First name: John, Last name: Mayer, Birthday: 1977-10-16
```

Inserting Data

Inserting data with Hibernate is simple. One other fancy thing is retrieving the database-generated primary key. In the previous chapter on JDBC, we needed to explicitly declare that we wanted to retrieve the generated key, pass in the `KeyHolder` instance, and get the key back from it after executing the insert statement. With Hibernate, all those actions are not required. Hibernate will retrieve the generated key and populate the domain object after the insert operation. The following code snippet shows the implementation of the `save()` method:

```
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;
   }
   ...
}
```

We just need to invoke the `Session.saveOrUpdate()` method, which can be used for both insert and update operations. We also log the ID of the saved singer object that will be populated by Hibernate after the object is persisted. The following code snippet shows the code for inserting a new singer record in the `SINGER` table with two child records in the `ALBUM` table and testing that the insertion succeeded. Also, because now we are modifying the contents of the tables, a JUnit class is more suitable to test each operation in isolation.

```
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();
   }
}
```

As shown in the previous code, in the testInsert() method, we add two albums, and save the object. Afterward, we list all the singers again by calling listSingersWithAlbum. Running the testInsert() method yields the following output:

```
...
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
```

The logging configuration has been modified so that more detailed Hibernate information is printed. From the INFO log record, we can see that the ID of the newly saved contact was populated correctly. Hibernate will also show all the SQL statements being executed against the database so you know what is happening behind the scenes.

Updating Data

Updating a record is as easy as inserting data. Suppose for the singer with an ID of 1 we want to update the first name and remove one album. To test the update operation, the following code snippet shows the testUpdate() method:

```
package com.apress.prospring5.ch7;
...
public class SingerDaoTest {

   private GenericApplicationContext ctx;
   private SingerDao singerDao;
   ...
```

```
    @Test
    public void testUpdate(){
      Singer singer = singerDao.findById(1L);
      //making sure such singer exists
      assertNotNull(singer);

      //making sure we got expected singer
      assertEquals("Mayer", singer.getLastName());

      //retrieve the album
      Album album = singer.getAlbums().stream().filter(
         a -> a.getTitle().equals("Battle Studies")).findFirst().get();

      singer.setFirstName("John Clayton");
      singer.removeAlbum(album);
      singerDao.save(singer);

      // test the update
      listSingersWithAlbum(singerDao.findAllWithAlbum());
    }
    ...
}
```

As shown in the previous code sample, we first retrieve the record with an ID of 1. Afterward, the first name is changed. We then loop through the album objects, retrieve the one with the title *Battle Studies*, and remove it from the singer's albums property. Finally, we call the singerDao.save() method again. When you run the program, you will see the following output:

```
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
```

You will see the first name is updated, and the *Battle Studies* album is removed. The album can be removed because of the orphanRemoval=true attribute we pass into the one-to-many association, which instructs Hibernate to remove all orphan records that exist in the database but are no longer found in the object when persisted.

Deleting Data

Deleting data is simple as well. Just call the Session.delete() method and pass in the contact object. The following code snippet shows the code for deletion:

```
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());
    }
    ...
}
```

The delete operation will delete the singer record, together with all its associated information, including albums and instruments, as we defined cascade=CascadeType.ALL in the mapping. The following code snippet shows the code for testing the delete method, testDelete():

```
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);
        //making sure such singer exists
        assertNotNull(singer);
        singerDao.delete(singer);

        listSingersWithAlbum(singerDao.findAllWithAlbum());
    }
}
```

The previous listing retrieves the singer with an ID of 2 and then calls the delete method to delete the singer information. Running the program will produce the following output:

```
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
```

You can see that the singer with an ID of 2 was deleted together with its child records in the ALBUM and SINGER_INSTRUMENT tables.

Configuring Hibernate to Generate Tables from Entities

In startup applications using Hibernate, it is common behavior to first write the entity classes and then generate the database tables based on their contents. This is done by using the hibernate.hbm2ddl.auto Hibernate property. When the application is started the first time, this property value is set to create; this will make Hibernate scan the entities and generate tables and keys (primary, foreign, unique) according to the relationships defined using JPA and Hibernate annotations.

If the entities are configured correctly and the resulting database objects are exactly as expected, the value of the property should be changed to update. This will tell Hibernate to update the existing database with any changes performed later on entities and keep the original database and any data that has been inserted into it.

In production applications, it is practical to write unit and integration tests that run on a pseudo-database that is discarded after all test cases are executed. Usually the test database is an in-memory database and Hibernate is told to create the database and discard it after the execution of the tests by setting the hibernate.hbm2ddl.auto value to create-drop.

You can find the full list of values for the hibernate.hbm2ddl.auto property in the Hibernate official documentation.[1]

The following code snippet shows the Java configuration AdvancedConfig class. As you can see, hibernate.hbm2ddl.auto is introduced, and the data source used is a DBCP pooled data source.

```
package com.apress.prospring5.ch7.config;

import com.apress.prospring5.ch6.CleanUp;
import org.apache.commons.dbcp2.BasicDataSource;
import org.hibernate.SessionFactory;
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 {
```

[1]See Table 3.7 at https://docs.jboss.org/hibernate/orm/5.0/manual/en-US/html/ch03.html.

```
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());
    }
}
```

The jdbc.properties file contains the properties necessary to access an in-memory database.

```
driverClassName=org.h2.Driver
url=jdbc:h2:musicdb
username=prospring5
password=prospring5
```

But in this case how do we initially populate the data? You can use a DatabasePopulator instance, a library like DbUnit,[2] or a custom populator bean similar to the DbIntializer bean, as shown here:

```
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);
        ...
```

[2]We will find the official DbUnit site at http://dbunit.sourceforge.net/.

```
        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);

        singerDao.save(singer);
        ...
        logger.info("Database initialization finished.");
    }
}
```

DbIntializer is just a simple bean, in which the repositories get injected as dependencies and that has an initialization method defined by the annotation @PostConstruct in which objects are created and persisted to the database. The bean was annotated with the @Service annotation to mark it as a bean providing the service of initializing the contents of a database. This bean will be created when ApplicationContext is created, and the initialization method will be executed, which will ensure the database will be populated before the context is used.

Using the AdvancedConfig configuration class, the same sets of tests run previously will pass.

Annotating Methods or Fields?

In the previous example, the entities had JPA annotations on their getters. But JPA annotations can be used directly on the fields, which has a few advantages.

- Entity configuration is clearer and located in the fields section, instead of being scattered in the whole class content. This is obviously true only if the code was written following clean code recommendations to keep all field declarations in a class in the same continuous section.
- Annotating entity fields does not enforce providing setter/getters. This is useful for the @Version annotated field, which should never be modified manually; only Hibernate should have access to it.
- Annotating fields allows to do extra processing in setters (for example, encrypting/calculating the value after loading it from the database). The problem with the property access is that the setters are also called when the object is loaded.

There are a lot of discussions on the Internet over which one is better. From a performance point of view, there isn't any difference. The decision is eventually up to the developer because there might be some valid cases when annotating accessors makes more sense. But keep in mind in the database the state of the objects is actually saved, and the state of the object is defined by the values of its fields, not the values returned by accessors. This also means that an object can be accurately re-created from the database exactly the way it was persisted. So, in a way, setting the annotations on the getters can be seen as breaking encapsulation.

Here you can see the `Singer` entity class that was rewritten to have annotated fields and that extends the abstract class `AbstractEntity`, which contains the two fields common to all Hibernate entity classes in an application:

```
// 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;
    }
}

//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<>();
    ...
}
```

Considerations When Using Hibernate

As shown in the examples of this chapter, once all the object-to-relational mapping, associations, and queries are properly defined, Hibernate can provide an environment for you to focus on programming with the object model, rather than composing SQL statements for each operation. In the past few years, Hibernate has been evolving quickly and has been widely adopted by Java developers as the data access layer library, both in the open source community and in enterprises.

However, there are some points you need to bear in mind. First, because you don't have control over the generated SQL, you should be careful when defining the mappings, especially the associations and their fetching strategy. Second, you should observe the SQL statements generated by Hibernate to verify that all perform as you expect.

Understanding the internal mechanism of how Hibernate manages its session is also important, especially in batch job operations. Hibernate will keep the managed objects in session and will flush and clear them regularly. Poorly designed data access logic may cause Hibernate to flush the session too frequently and greatly impact the performance. If you want absolute control over the query, you can use a native query, which we discuss in the next chapter.

Finally, the settings (batch size, fetch size, and so forth) play an important role in tuning Hibernate's performance. You should define them in your session factory and adjust them while load testing your application to identify the optimal value.

After all, Hibernate, and its excellent JPA support that we discuss in the next chapter, is a natural decision for Java developers looking for an OO way to implement data access logic.

Summary

In this chapter, we discussed the basic concepts of Hibernate and how to configure it within a Spring application. Then we covered common techniques for defining ORM mappings, and we covered associations and how to use the `HibernateTemplate` class to perform various database operations. With regard to Hibernate, we covered only a small piece of its functionality and features. For those interested in using Hibernate with Spring, we highly recommend you study Hibernate's standard documentation. Also, numerous books discuss Hibernate in detail. We recommend *Beginning Hibernate: For Hibernate 5* by Joseph Ottinger, Jeff Linwood, and Dave Minter (Apress, 2016),[3] as well as *Pro JPA 2* by Mike Keith and Merrick Schincariol (Apress, 2013).[4] In the next chapter, you will take a look at JPA and how to use it when using Spring. Hibernate provides excellent support for JPA, and we will continue to use Hibernate as the persistence provider for the examples in the next chapter. For query and update operations, JPA act likes Hibernate. In the next chapter, we discuss advanced topics including native and criteria query and how we can use Hibernate as well as its JPA support.

[3]Download the e-book from the Apress official site: `http://apress.com/us/book/9781484223185`.
[4]Download the e-book from the Apress official site: `http://apress.com/us/book/9781430249269`.

CHAPTER 8

Data Access in Spring with JPA2

In the previous chapter, we discussed how to use Hibernate with Spring when implementing data access logic with the ORM approach. We demonstrated how to configure Hibernate's `SessionFactory` in Spring's configuration and how to use the `Session` interface for various data access operations. However, that is just one way Hibernate can be used. Another way of adopting Hibernate in a Spring application is to use Hibernate as a persistence provider of the standard Java Persistence API (JPA).

Hibernate's POJO mapping and its powerful query language (HQL) have gained great success and also influenced the development of data access technology standards in the Java world. After Hibernate, the JCP developed the Java Data Objects (JDO) standard and then JPA.

At the time of this writing, JPA has reached version 2.1 and provides concepts that were standardized such as `PersistenceContext`, `EntityManager`, and the Java Persistence Query Language (JPQL). These standardizations provide a way for developers to switch between JPA persistence providers such as Hibernate, EclipseLink, Oracle TopLink, and Apache OpenJPA. As a result, most new JEE applications are adopting JPA as the data access layer.

Spring also provides excellent support for JPA. For example, a number of `EntityManagerFactoryBean` implementations are provided for bootstrapping a JPA entity manager with support for all of the JPA providers mentioned earlier. The Spring Data project also provides a subproject called Spring Data JPA, which provides advanced support for using JPA in Spring applications. The main features of the Spring Data JPA project include concepts of a repository and specification and support for the Query Domain-Specific Language (QueryDSL).

This chapter covers how to use JPA 2.1 with Spring, using Hibernate as the underlying persistence provider. You will learn how to implement various database operations by using JPA's `EntityManager` interface and JPQL. Then you will see how Spring Data JPA can further help simplify JPA development. Finally, we present advanced topics related to ORM, including native queries and criteria queries.

Specifically, we discuss the following topics:

- *Core concepts of Java Persistence API (JPA)*: We cover some of the major concepts of JPA.
- *Configuring the JPA entity manager*: We discuss the types of `EntityManagerFactory` that Spring supports and how to configure the most commonly used one, `LocalContainerEntityManagerFactoryBean`, in Spring's XML configuration.
- *Data operations*: We show how to implement basic database operations in JPA, which is much like the concepts when using Hibernate on its own.
- *Advanced query operations*: We discuss how to use native queries in JPA and the strongly typed criteria API in JPA for more flexible query operations.
- *Introducing Spring Data Java Persistence API (JPA)*: We discuss the Spring Data JPA project and demonstrate how it can help simplify the development of data access logic.

© Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho 2017

I. Cosmina et al., *Pro Spring 5*, https://doi.org/10.1007/978-1-4842-2808-1_8

- *Tracking entity changes and auditing*: In database update operations, it's a common requirement to keep track of the date an entity was created or last updated and who made the change. Also, for critical information such as a customer, a history table that stores each version of the entity is usually required. We discuss how Spring DataJPA and Hibernate Envers (Hibernate Entity Versioning System) can help ease the development of such logic.

Like Hibernate, JPA supports the definition of mappings either in XML or in Java annotations. This chapter focuses on the annotation type of mapping because its usage tends to be much more popular than the XML style.

Introducing JPA 2.1

Like other Java specification requests (JSRs), the objective of the JPA 2.1 specification (JSR-338) is to standardize the ORM programming model in both the JSE and JEE environments. It defines a common set of concepts, annotations, interfaces, and other services that a JPA persistence provider should implement. When programming to the JPA standard, developers have the option of switching the underlying provider at will, just like switching to another JEE-compliant application server for applications developed on the JEE standards.

Within JPA, the core concept is the `EntityManager` interface, which comes from factories of the type `EntityManagerFactory`. The main job of `EntityManager` is to maintain a persistence context, in which all the entity instances managed by it will be stored. The configuration of `EntityManager` is defined as a persistence unit, and there can be more than one persistence unit in an application. If you are using Hibernate, you can think of the persistence context in the same way as the `Session` interface, while `EntityManagerFactory` is the same as `SessionFactory`. In Hibernate, the managed entities are stored in the session, which you can directly interact with via Hibernate's `SessionFactory` or `Session` interface. In JPA, however, you can't interact with the persistence context directly. Instead, you need to rely on `EntityManager` to do the work for you.

JPQL is similar to HQL, so if you have used HQL before, JPQL should be easy to pick up. However, in JPA 2, a strongly typed Criteria API was introduced, which relies on the mapped entities' metadata to construct the query. Given this, any errors will be discovered at compile time rather than runtime.

For a detailed discussion of JPA 2, we recommend the book *Pro JPA 2* by Mike Keith and Merrick Schincariol (Apress, 2013).[1] In this section, we discuss the basic concepts of JPA, the sample data model that will be used in this chapter, and how to configure Spring's `ApplicationContext` to support JPA.

Sample Data Model for Example Code

In this chapter, we use the same data model as used in Chapter 7. However, when we discuss how to implement the auditing features, we will add a few columns and a history table for demonstration. To get started, we will begin with the same database creation scripts used in the previous chapter. If you skipped Chapter 7, take a look at the data model presented in that chapter's "Sample Data Model for Example Code" section, which can help you understand the sample code in this chapter.

---

[1]Get it online from at `www.apress.com/us/book/9781430249269`.

Configuring JPA's EntityManagerFactory

As mentioned earlier in this chapter, to use JPA in Spring, we need to configure EntityManagerFactory, just like SessionFactory used in Hibernate. Spring supports three types of EntityManagerFactory configurations.

The first one uses the LocalEntityManagerFactoryBean class. It's the simplest one, which requires only the persistence unit name. However, since it doesn't support the injection of DataSource and hence isn't able to participate in global transactions, it's suitable only for simple development purposes.

The second option is for use in a JEE-compliant container, in which the application server bootstraps the JPA persistence unit based on the information in the deployment descriptors. This allows Spring to look up the entity manager via JNDI lookup. The following configuration snippet depicts the element needed for looking up an entity manager via JNDI:

```
<beans ...>
    <jee:jndi-lookup  id="prospring5Emf"
       jndi-name="persistence/prospring5PersistenceUnit"/>
</beans>
```

In the JPA specification, a persistence unit should be defined in the configuration file META-INF/persistence.xml. However, as of Spring 3.1, a new feature has been added that eliminates this need; we show you how to use it later in this chapter.

The third option, which is the most common and is used in this chapter, is the LocalContainerEntityManagerFactoryBean class that supports the injection of DataSource and can participate in both local and global transactions. The following configuration snippet shows the corresponding XML configuration file (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: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>
```

You probably expect that there is an equivalent configuration using Java configuration classes. There is, as shown here:

```
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();
    }
}
```

In the previous configurations, several beans are declared in order to be able to support the configuration of LocalContainerEntityManagerFactoryBean with Hibernate as the persistence provider. The main configurations are as follows:

- *The dataSource bean*: We declared the data source with an embedded database using H2. Because it's an embedded database, the database name is not required.
- *The transactionManager bean*: EntityManagerFactory requires a transaction manager for transactional data access. Spring provides a transaction manager specifically for JPA (org.springframework.orm.jpa.JpaTransactionManager). The bean is declared with an ID of transactionManager assigned. We discuss transactions in detail in Chapter 9. We declare the tag <tx:annotation-driven> to support a declaration of the transaction demarcation requirements using annotations. Its equivalent annotation is @EnableTransactionManagement, which must be placed on a class annotated with @Configuration.
- *Component scan*: The tag should be familiar to you. We instruct Spring to scan the components under the package com.apress.prospring5.ch8.
- *JPA EntityManagerFactory bean*: The emf bean is the most important part. First, we declare the bean to use LocalContainerEntityManagerFactoryBean. Within the bean, several properties are provided. First, as you might have expected, we need to inject the DataSource bean. Second, we configure the property jpaVendorAdapter with the class HibernateJpaVendorAdapter because we are using Hibernate. Third, we instruct the entity factory to scan for the domain objects with ORM annotations under the package com.apress.prospring5.ch8 (specified by the <property name="packagesToScan"> tag). Note that this feature has been available only since Spring 3.1, and with the support of domain class scanning, you can skip the definition of the persistence unit in the META-INF/persistence.xml file. Finally, the jpaProperties property provides configuration details for the persistence provider Hibernate. You will see that the configuration options are the same as those we used in Chapter 7, so we can skip the explanation here.

Using JPA Annotations for ORM Mapping

Hibernate influenced the design of JPA in many ways. For the mapping annotations, they are so close that the annotations we used in Chapter 7 for mapping the domain objects to the database are the same in JPA. If you take a look at the domain classes' source code in Chapter 7, you will see that all mapping annotations are under the package javax.persistence, which means those annotations are already JPA compatible.

Once EntityManagerFactory has been properly configured, injecting it into your classes is simple. The following code snippet shows the code for the SingerServiceImpl class, which we will use as the sample for performing database operations using 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");
    }
}
```

Several annotations are applied to the class. The `@Service` annotation is used to identify the class as being a Spring component that provides business services to another layer and assigns the Spring bean the name `jpaSingerService`. The `@Repository` annotation indicates that the class contains data access logic and instructs Spring to translate the vendor-specific exceptions to Spring's `DataAccessException` hierarchy. As you are already familiar with, the `@Transactional` annotation is used for defining transaction requirements.

To inject `EntityManager`, we use the `@PersistenceContext` annotation, which is the standard JPA annotation for entity manager injection. It may be questionable as to why we're using the name `@PersistenceContext` to inject an entity manager, but if you consider that the persistence context itself is managed by `EntityManager`, the annotation naming makes perfect sense. If you have multiple persistence units in your application, you can also add the `unitName` attribute to the annotation to specify which persistence unit you want to be injected. Typically, a persistence unit represents an individual back-end `DataSource`.

Performing Database Operations with JPA

This section covers how to perform database operations in JPA. The following code snippet shows the `SingerService` interface, which indicates the singer information services we are going to provide:

```
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();
}
```

The interface is very simple; it has just three finder methods, one save method, and one delete method. The save method will serve both the insert and update operations.

Using the Java Persistence Query Language to Query Data

The syntax for JPQL and HQL is similar, and in fact, all the HQL queries that we used in Chapter 7 are reusable to implement the three finder methods within the `SingerService` interface. To use JPA and Hibernate, you need to add the following dependencies to the project:

```
//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"
    ]
}

//chapter08.gradle
dependencies {
       //we specify these dependencies for all submodules,
        //except the boot module, that defines its own
       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
}
```

The following code snippet recaps the code for the Singer domain object model classes from Chapter 7:

```
//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<>();
```

```
   //setters and getters

    @Override
    public String toString() {
        return "Singer - Id: " + id + ", First name: " + firstName
           + ", Last name: " + lastName + ", Birthday: " + birthDate;
    }
}
// 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() {
    //needed  byJPA
    }

    public Album(String title, Date releaseDate) {
       this.title = title;
       this.releaseDate = releaseDate;
    }
```

```
    //setters and getters
}
//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<>();

    //setters and getters
}
```

If you analyze the queries defined using `@NamedQuery`, you will see that there seems to be no difference between HQL and JPQL. Let's begin with the `findAll()` method, which simply retrieves all the singers from the database.

```
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();
    }
    ...
}
```

As shown in this listing, we use the `EntityManager.createNamedQuery()` method, passing in the name of the query and the expected return type. In this case, `EntityManager` will return a `TypedQuery<X>` interface. The method `TypedQuery.getResultList()` is then called to retrieve the singers. To test the implementation of the method, we will use a test class that will contain a test method for each JPA method that will be implemented.

```
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();
    }
}
```

If assertEquals does not throw an exception (the test failed), running the testFindAll() test method will produce the following output:

```
----  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
```

For associations, the JPA specification states that, by default, the persistence providers must fetch the association eagerly. However, for Hibernate's JPA implementation, the default fetching strategy is still lazy. So, when using Hibernate's JPA implementation, you don't need to explicitly define an association as lazy fetching. The default fetching strategy of Hibernate is different from the JPA specification.

Now let's implement the findAllWithAlbum() method, which will fetch all the associated albums and instruments. The implementation is shown here:

```
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;
    }
    ...
}
```

findAllWithAlbum() is the same as the findAll() method, but it uses a different named query with left join fetch enabled. The method used to test it and print the entries is shown here:

```
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();
    }
}
```

If assertEquals does not throw an exception (the test failed), running the testFindAllWithAlbum() test method will produce the following output:

```
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
```

If logging is enabled for Hibernate, you can also see the native query that was generated to extract all the data from the database.

Now let's see the `findById()` method, which demonstrates how to use a named query with named parameters in JPA. The associations will be fetched as well. The following code snippet shows the implementation:

```
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();

    }
    ...
}
```

`EntityManager.createNamedQuery(java.lang.String name, java.lang.Class<T> resultClass)` was called to get an instance of the `TypedQuery<T>` interface, which ensures that the result of the query must be of type `Singer`. Then the `TypedQuery<T>.setParameter()` method was used to set the values of the named parameters within the query and to invoke the `getSingleResult()` method, since the result should contain only a single `Singer` object with the specified ID. We will leave the testing of the method as an exercise for you.

```
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();
    }    ...
}
```

Querying with Untyped Results

In many cases, you want to submit a query to the database and manipulate the results at will, instead of storing them in a mapped entity class. One typical example is a web-based report that lists only a certain number of columns across multiple tables. For example, say you have a web page that shows the singer information and his most recently released album title. The summary information contains the complete name of the singer and his most recently released album title. Singers without albums will not be listed. In this case, we can implement this use case with a query and then manually manipulate the ResultSet object.

Let's create a new class called SingerSummaryUntypeImpl and name the method displayAllSingerSummary(). The following code snippet shows a typical implementation of the method:

```
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]);
        }
    }
}
```

As shown in the previous code sample, we use the `EntityManager.createQuery()` method to create `Query`, passing in the JPQL statement, and then get the result list.

When we explicitly specify the columns to be selected within JPQL, JPA will return an iterator, and each item within the iterator is an array of objects. We loop through the iterator, and for each element in the object array, the value is displayed. Each object array corresponds to a record within the `ResultSet` object. The following code snippet shows the testing program:

```
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();
    }
}
```

Running the testing program produces the following output:

```
1: John, Mayer, The Search For Everything
2: Eric, Clapton, From The Cradle
```

In JPA, there is a more elegant solution, rather than playing around with the object array returned from the query, that is discussed in the next section.

Querying for a Custom Result Type with a Constructor Expression

In JPA, when querying for a custom result like the one in the previous section, you can instruct JPA to directly construct a POJO from each record for you. This POJO is also called a *view* because it contains data from multiple tables. For the example in the previous section, let's create a POJO called `SingerSummary` that stores the results of the query for the singer summary. The following code snippet shows the class:

```
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;
    }
}
```

The previous SingerSummary class has the properties for each singer summary, with a constructor method that accepts all the properties. Having the SingerSummary class in place, we can revise the findAll() method and use a constructor expression within the query to instruct the JPA provider to map the ResultSet to the SingerSummary class. Let's create an interface for the SingerSummary service first. The following code snippet shows the interface:

```
package com.apress.prospring5.ch8.service;

import com.apress.prospring5.ch8.view.SingerSummary;

import java.util.List;

public interface SingerSummaryService {
    List<SingerSummary> findAll();
}
```

Here you can see the implementation of the SingerSummaryImpl.findAll() method, using the constructor expression for the ResultSet mapping:

```
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;
    }
}
```

In the JPQL statement, the new keyword was specified, together with the fully qualified name of the POJO class that will store the results and pass in the selected attributes as the constructor argument of each SingerSummary class. Finally, the SingerSummary class was passed into the createQuery() method to indicate the result type. The following code snippet shows the testing program:

```
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();
    }
}
```

Executing the `testFindAll` method class again produces the output for each `SingerSummary` object within the list, as shown here (other output was omitted):

```
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
```

As you can see, the constructor expression is useful for mapping the result of a custom query into POJOs for further application processing.

Inserting Data

Inserting data by using JPA is simple. Like Hibernate, JPA also supports retrieving a database-generated primary key. The following code snippet shows the `save()` method:

```
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;
    }
    ...
}
```

As shown here, the `save()` method first checks whether the object is a new entity instance, by checking the `id` value. If `id` is `null` (that is, not yet assigned), the object is a new entity instance, and the `EntityManager.persist()` method will be invoked. When calling the `persist()` method, `EntityManager` persists the entity and makes it a managed instance within the current persistence context. If the `id` value exists, then we're carrying out an update, and the `EntityManager.merge()` method will be called instead. When the `merge()` method is called, the `EntityManager` merges the state of the entity into the current persistence context.

The following code snippet shows the code to insert a new singer record. It's all done in a test method because we want to test that the insert succeed.

```
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();
    }
}
```

As shown here, we create a new singer, add two albums, and save the object. Then we list all the singers again, after we test that we have the correct number of records in the table. Running the program yields the following output:

```
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
```

From the `INFO` log record, you can see that the `id` of the newly saved singer was populated correctly. Hibernate will also show all the SQL statements being fired to the database.

Updating Data

Updating data is as easy as inserting data. Let's go through an example. Suppose for a singer with an ID of 1, we want to update its first name and remove an album. To test the update operation, the following code snippet shows the `testUpdate()` method:

```
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);
       //making sure such singer exists assertNotNull(singer);
       //making sure we got expected record assertEquals("Mayer", singer.getLastName());
       //retrieve the  album
       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();
    }
}
```

We first retrieve the record with an ID of 1 and we change the first name. Then we loop through the album objects and retrieve the one with title *Battle Studies* and remove it from the singer's `albums` property. Finally, we call the `SingerService.save()` method again. When you run the program, you will see the following output (other output was omitted):

```
---- 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
```

You will see that the first name was updated and the album was removed. The album can be removed because of the rphanRemoval=true attribute that was defined in the one-to-many association, which instructs the JPA provider (Hibernate) to remove all orphan records that exist in the database but are no longer found in the object when persisted.

```
@OneToMany(mappedBy = "singer", cascade=CascadeType.ALL, orphanRemoval=true)
```

Deleting data

Deleting data is just as simple. Simply call the EntityManager.remove() method and pass in the singer object. The following code snippet shows the updated code to delete a singer:

```
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");
    }
       ...
}
```

First the EntityManager.merge() method is invoked to merge the state of the entity into the current persistence context. The merge() method returns the managed entity instance. Then EntityManager.remove() is called, passing in the managed singer entity instance. The remove operation deletes the singer record, together with all its associated information, including albums and instruments, as we defined the cascade=CascadeType.ALL in the mapping. To test the delete operation, the testDelete() method can be used, which is depicted in the following code snippet:

```
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);
        //making sure such singer exists
        assertNotNull(singer);
        singerService.delete(singer);

        listSingersWithAlbum(singerService.findAllWithAlbum());
    }
    ...

    @After
    public void tearDown(){
        ctx.close();
    }
}
```

The previous listing retrieves the singer with an ID of 2 and then calls the `delete()` method to delete the singer information. Running the program produces the following output:

```
 ---- 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
```

You can see that the singer with an ID of 1 was deleted.

Using a Native Query

Having discussed performing trivial database operations by using JPA, now let's move on to some more advanced topics. Sometimes you may want to have absolute control over the query that will be submitted to the database. One example is using a hierarchical query in an Oracle database. This kind of query is database-specific and referred to as a *native query*.

JPA supports the execution of native queries; `EntityManager` will submit the query to the database as is, without any mapping or transformation performed. One main benefit of using JPA native queries is the mapping of `ResultSet` back to the ORM-mapped entity classes. The following two sections discuss how to use a native query to retrieve all singers and directly map `ResultSet` back to the `Singer` objects.

Using a Simple Native Query

To demonstrate how to use a native query, let's implement a new method to retrieve all the singers from the database. The following snippet shows the new method that must be added to the SingerServiceImpl:

```
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();
    }
    ...
}
```

You can see that the native query is just a simple SQL statement to retrieve all the columns from the SINGER table. To create and execute the query, EntityManager.createNativeQuery() was first called, passing in the query string as well as the result type. The result type should be a mapped entity class (in this case the Singer class). The createNativeQuery() method returns a Query interface, which provides the getResultList() operation to get the result list. The JPA provider will execute the query and transform the ResultSet object into the entity instances, based on the JPA mappings defined in the entity class. Executing the previous method produces the same result as the findAll() method.

Native Querying with SQL ResultSet Mapping

Besides the mapped domain object, you can pass in a string, which indicates the name of a SQL ResultSet mapping. A SQL ResultSet mapping is defined at the entity class level by using the @SqlResultSetMapping annotation. A SQL ResultSet mapping can have one or more entity and column mappings.

```
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 {
 ...
}
```

A SQL ResultSet mapping called singerResult is defined for the entity class, with the entityClass attribute in the Singer class itself. JPA supports more complex mapping for multiple entities and supports mapping down to column-level mapping.

After the SQL ResultSet mapping is defined, the findAllByNativeQuery() method can be invoked using the ResultSet mapping's name. The following code snippet shows the updated findAllByNativeQuery() method:

```
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();
    }
    ...
}
```

As you can see, JPA also provides strong support for executing native queries, with a flexible SQL ResultSet mapping facility provided.

Using the JPA 2 Criteria API for a Criteria Query

Most applications provide a front end for users to search for information. Most likely, a large number of searchable fields are displayed, and the users enter information in only some of them and do the search. It's difficult to prepare a large number of queries, with each possible combination of parameters that users may choose to enter. In this situation, the criteria API query feature comes to the rescue.

In JPA 2, one major new feature introduced was a strongly typed Criteria API query. In this new Criteria API, the criteria being passed into the query is based on the mapped entity classes' meta-model. As a result, each criteria specified is strongly typed, and errors will be discovered at compile time, rather than runtime.

In the JPA Criteria API, an entity class meta-model is represented by the entity class name with a suffix of an underscore (_). For example, the meta-model class for the Singer entity class is Singer_. The following code snippet shows the Singer_ class:

```
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;

}
```

The meta-model class is annotated with @StaticMetamodel, and the attribute is the mapped entity class. Within the class are the declaration of each attribute and its related types.

It would be tedious to code and maintain those meta-model classes. However, tools can help generate those meta-model classes automatically based on the JPA mappings within the entity classes. The one provided by Hibernate is called Hibernate Metamodel Generator (www.hibernate.org/subprojects/jpamodelgen.html).

The way you go about generating your meta-model classes depends on what tools you are using to develop and build your project. We recommend reading the "Usage" section of the documentation (http://docs.jboss.org/hibernate/jpamodelgen/1.3/reference/en-US/html_single/#chapter-usage) for specific details. The sample code that comes as part of this book uses Gradle to generate the meta-classes. The required dependency for meta-model class generation is the hibernate-jpamodelgen library. This dependency is configured with its version in the pro-spring-15/build.gradle file.

```
ext {
    ...

    //persistency libraries
    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"
    ]
    ...
```

This is the main library for generating meta-model classes. It is used in `chapter08/jpa-criteria/build.gradle` by the `generateQueryDSL` Gradle task to generate the meta-model classes, before compiling the module. The `chapter08/jpa-criteria/build.gradle` configuration is shown here:

```
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
```

With the class generation strategy set up, let's define a query that accepts both the first name and last name for searching singers. The following code snippet shows the definition of the new method `findByCriteriaQuery()` in the `SingerService` interface:

```
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);
}
```

The next code snippet shows the implementation of the `findByCriteriaQuery()` method using a JPA 2 criteria API query:

```
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();
    }
}
```

Let's break down the Criteria API usage.

- `EntityManager.getCriteriaBuilder()` is called to retrieve an instance of `CriteriaBuilder`.
- A typed query is created using `CriteriaBuilder.createQuery()`, passing in `Singer` as the result type.
- The `CriteriaQuery.from()` method is invoked, passing in the entity class. The result is a query root object (the `Root<Singer>` interface) corresponding to the specified entity. The query root object forms the basis for path expressions within the query.
- The two `Root.fetch()` method calls enforce the eager fetching of the associations relating to albums and instruments. The `JoinType.LEFT` argument specifies an outer join. Calling the `Root.fetch()` method with `JoinType.LEFT` as the second argument is equivalent to specifying the left join fetch join operation in JPQL.
- The `CriteriaQuery.select()` method is called and passes the root query object as the result type. The `distinct()` method with true means that duplicate records should be eliminated.
- A `Predicate` instance is obtained by calling the `CriteriaBuilder.conjunction()` method, which means that a conjunction of one or more restrictions is made. A `Predicate` can be a simple or compound predicate, and a predicate is a restriction that indicates the selection criteria defined by an expression.
- The first- and last-name arguments are checked. For each not `null` argument, a new `Predicate` will be constructed using the `CriteriaBuilder()` method (that is, the `CriteriaBuilder.and()` method). The method `equal()` is to specify an equal restriction, within which `Root.get()` is called, passing in the corresponding attribute of the entity class meta-model to which the restriction applies. The constructed predicate is then "conjunct" with the existing predicate (stored by the variable criteria) by calling the `CriteriaBuilder.and()` method.
- The `Predicate` is constructed with all the criteria and restrictions and passed as the `where` clause to the query by calling the `CriteriaQuery.where()` method.

- Finally, CriteriaQuery is passed to EntityManager. EntityManager then constructs the query based on the CriteriaQuery value passed in, executes the query, and returns the result.

To test the criteria query operation, the following code snippet shows the updated SingerJPATest class:

```
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();
    }
}
```

Running the program produces the following output (other output was omitted, but the generated query was kept):

```
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
```

You can try a different combination or pass a `null` value to either of the arguments to observe the output.

Introducing Spring Data JPA

The Spring Data JPA project is a subproject under the Spring Data umbrella project. The main objective of the project is to provide additional features for simplifying application development with JPA.

Spring Data JPA provides several main features. In this section, we discuss two. The first one is the `Repository` abstraction, and the other one is the entity listener for keeping track of basic audit information of entity classes.

Adding Spring Data JPA Library Dependencies

To use Spring Data JPA, we need to add the dependency to the project. Here you can see the Gradle configuration necessary to use Spring Data JPA:

```
//pro-spring-15/build.gradle
ext {
    //spring libs
    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",
            ...
    ]
     ...
}

//chapter08/spring-data-jpa/build.gradle
dependencies {
    compile spring.aop, spring.data, misc.guava
}
```

Using Spring Data JPA Repository Abstraction for Database Operations

One of the main concepts of Spring Data and all its subprojects is the `Repository` abstraction, which belongs to the Spring Data Commons project (https://github.com/spring-projects/spring-data-commons). At the time of this writing, it's at version 2.0.0 M2. In Spring Data JPA, the repository abstraction wraps the underlying JPA `EntityManager` and provides a simpler interface for JPA-based data access. The central interface within Spring Data is the `org.springframework.data.repository.Repository<T,ID extends Serializable>` interface, which is a marker interface belonging to the Spring Data Commons distribution. Spring Data provides various extensions of the `Repository` interface; one of them is the `org.springframework.data.repository.CrudRepository` interface (which also belongs to the Spring Data Commons project), which we discuss in this section.

The `CrudRepository` interface provides a number of commonly used methods. The following code snippet shows the interface declaration, which is extracted from Spring Data Commons project source code:

```
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);
}
```

Although the method naming is self-explanatory, it's better to show how the `Repository` abstraction works by going through a simple example. Let's revise the `SingerService` interface a bit, down to just three finder methods. The following code snippet shows the revised `SingerService` interface:

```
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);
}
```

The next step is to prepare the SingerRepository interface, which extends the CrudRepository interface. The following code snippet shows the SingerRepository interface:

```
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);
}
```

We just need to declare two methods in this interface, as the findAll() method is already provided by the CrudRepository.findAll() method. As shown in the preceding listing, the SingerRepository interface extends the CrudRepository interface, passing in the entity class (Singer) and the ID type (Long). One fancy aspect of Spring Data's Repository abstraction is that when you use the common naming convention of findByFirstName and findByFirstNameAndLastName, you don't need to provide Spring Data JPA with the named query. Instead, Spring Data JPA will "infer" and construct the query for you based on the method name. For example, for the findByFirstName() method, Spring Data JPA will automatically prepare the query select s from Singer s where s.firstName = :firstName for you and set the named parameter firstName from the argument.

To use the Repository abstraction, you have to define it in Spring's configuration. The following code snippet shows the configuration file (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>
```

First, we need to add the `jpa` namespace in the configuration file. Then, the `<jpa:repositories>` tag is used to configure Spring Data JPA's `Repository` abstraction. We instruct Spring to scan the package `com.apress.prospring5.ch8` for repository interfaces and to pass in `EntityManagerFactory` and the transaction manager, respectively.

In case you haven't noticed, there is an `<context:exclude-filter>` in the `<context:component-scan>` definition that specified classes annotated with `@Configuration`. That element is introduced to exclude from scanning the Java configuration class that can be used instead of the XML configuration previously depicted. That class is shown here:

```
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();
    }
}
```

The only configuration element used here to enable support of Spring Data JPA repositories is the `@EnableJpaRepositories` annotation. Using the `basePackages` attribute, the packages where the custom `Repository` extensions are scanned and the repository beans are created. The rest of the dependencies (the `emf` and the `transactionManager` beans) are injected automatically by the Spring container.

The following code snippet shows the implementation of the three finder methods of the `SingerService` interface:

```
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);
    }
}
```

You can see that instead of `EntityManager`, we just need to inject the `singerRepository` bean, generated by Spring based on the `SingerRepository` interface, into the service class, and Spring Data JPA will do all the low-level work for us. In the following code snippet, you can see a testing class, and by now you should already be familiar with its content:

```
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();
    }

}
```

Running the test class, all tests should pass, and the expected data will be printed in the console.

Using JpaRepository

Besides `CrudRepository`, there is an even more advanced Spring interface that can make creating custom repositories easier; it's called the `JpaRepository` interface, and it provides batch, paging, and sorting operations. Figure 8-1 shows the relationship between `JpaRepository` and the `CrudRepository` interface. Depending on the complexity of the application, you can choose to use `CrudRepository` or `JpaRepository`. As you can see from Figure 8-1, `JpaRepository` extends `CrudRepository`; thus, it provides all the functionalities this interface does.

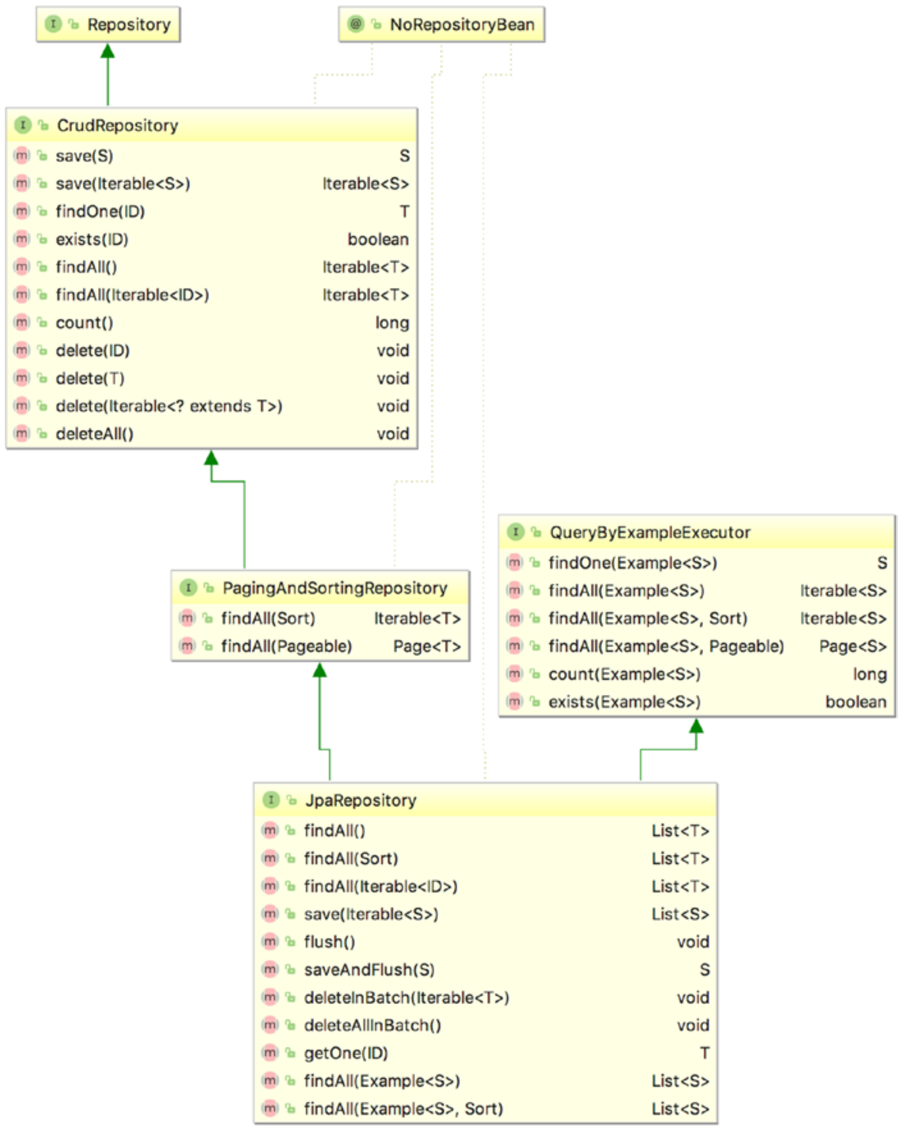

Figure 8-1. *Spring Data JPA Repository interfaces hierarchy*

Spring Data JPA with Custom Queries

In complex applications, you might need custom queries that cannot be “inferred” by Spring. In this case, the query must be defined explicitly using the `@Query` annotation. Let’s use this annotation to search for all music albums containing *The* in their title. The following code snippet depicts the `AlbumRepository` interface:

```
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);
}
```

The previous query has a named parameter called `title`. When the name of the named parameter is the same as the name of the argument in the method annotated with `@Query`, the `@Param` annoation is not needed. But if the method argument has a different name, the `@Param` annotation is needed to tell Spring that the value of this argument is to be injected in the named parameter in the query.

`AlbumServiceImpl` is quite simple and only uses the `albumRepository` bean to call its methods.

```
//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);
}

//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);
    }
}
```

To test the `findByTitle(..)` method, you can use the following test class:

```
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();
    }

}
```

If you run the previous test class, the `testFindByTitle` passes, and the two album details are printed in the console.

```
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
```

Keeping Track of Changes on the Entity Class

In most applications, we need to keep track of basic audit activities for the business data being maintained by users. The audit information typically includes the user who creates the data, the date it was created, the date it was last modified, and the user who last modified it.

The Spring Data JPA project provides this function in the form of a JPA entity listener, which helps you keep track of the audit information automatically. To use the feature, until Spring 4, the entity class needed to implement the `Auditable<U, ID extends Serializable, T extends TemporalAccessor> extends Persistable<ID>` interface (belonging to Spring Data Commons) or extend any class that implements this interface. The following code snippet shows the `Auditable` interface that was extracted from Spring Data's reference documentation:

```
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);
}
```

To show how it works, let's create a new table called SINGER_AUDIT in the database schema, which is based on the SINGER table, with four audit-related columns added. The following code snippet shows the table creation script (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)
);
```

As you can see from the definition of the Auditable interface previously shown, the date types columns are restricted to types extending java.time.temporal.TemporalAccessor.

Starting with Spring 5, implementing Auditable<U,ID extends Serializable> is no longer necessary because everything can be replaced by annotations. The four underlined columns indicate the audit-related columns. Notice the @CreatedBy, @CreatedDate, @LastModifiedBy, and @LastModifiedDate annotations. Using these annotations, the type restriction for the date columns no longer applies. The next step is to create the entity class called SingerAudit. The following code snippet shows the SingerAudit class:

```
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;
    }
}
```

The @Column annotations are applied on the auditing fields to map to the actual column in the table. The @EntityListeners(AuditingEntityListener.class) annotation registers AuditingEntityListener to be used only for this entity in the persistence context. In more complex examples when more than one entity class is needed, the auditing functionality is isolated into an abstract class annotated with @MappedSuperclass, and this will be the class also annotated with @EntityListeners(AuditingEntityListener.class). If SingerAudit were part of such hierarchy, it would have to extend the following class:

```
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;
    }
}
```

This would greatly reduce the size of SingerAudit, becoming this:

```
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;
    }
}
```

The following code snippet shows the SingerAuditService interface, where we define only a few methods to demonstrate the auditing feature:

```
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);
}
```

The SingerAuditRepository interface just extends CrudRepository, which has already implemented all the methods that we are going to use for SingerAuditService. The findById() method is implemented by the CrudRepository.findOne() method. The following code snippet shows the service implementation class SingerAuditServiceImpl:

```
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);
    }
}
```

We also need to do a little configuration work. Using the XML configuration, the AuditingEntityListener<T> JPA entity listener that provides the auditing service needs to be declared into a file called /src/main/resources/META-INF/orm.xml (it's mandatory to use this file name, which is indicated by the JPA specification) under the project root folder and declare the listener, like the one shown here:

```
<?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>
```

Using annotation configuration, this file is replaced by the `@EntityListeners (AuditingEntityListener.class)` annotation. The JPA provider will discover this listener during persistence operations (save and update events) for audit fields processing. The rest of the Spring configuration is almost identical to what you've seen so far, with one exception: a new configuration annotation to enable auditing, of course.

```
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 {
    // same content as  the configuration in the previous section
    ...
}
```

The @EnableJpaAuditing(auditorAwareRef = "auditorAwareBean") annoation is the equivalent of the <jpa:auditing auditor-aware-ref="auditorAwareBean"/> element that is used to enable the JPA auditing feature using XML configuration. The auditorAwareBean bean provides the user information. The following code snippet shows the AuditorAwareBean class:

```
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");
    }
}
```

AuditorAwareBean implements the AuditorAware<T> interface, passing in the type String. In real situations, this should be an instance of user information, for example, a User class, which represents the logged-in user who is performing the data update action. We use String here just for simplicity. In the AuditorAwareBean class, the method getCurrentAuditor() is implemented, and the value is hard-coded to prospring5. In real situations, the user should be obtained from the underlying security infrastructure. For example, in Spring Security, the user information can be retrieved from the SecurityContextHolder class.

Now that all the implementation work is completed, the following code snippet shows the SpringAuditJPADemo testing program:

```
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(4l);
        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();
        }
    }
}
```

In the main() method, we list the singer audit information both after a new singer is inserted and after it's later updated. Running the program produces the following output:

```
Add new singer
Listing singers without details:
// other singers ...
Singer - Id: 4, First name: BB, Last name: King, Birthday: 1940-09-16,
   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
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
```

In the previous output, you can see that after the new singer is created, the create date and last-modified dates are the same. However, after the update, the last-modified date is updated. Auditing is another handy feature that Spring Data JPA provides so that you don't need to implement the logic yourself.

Keeping Entity Versions by Using Hibernate Envers

In an enterprise application, for business-critical data, it is always a requirement to keep versions of each entity. For example, in a customer relationship management (CRM) system, each time a customer record is inserted, updated, or deleted, the previous version should be kept in a history or auditing table to fulfill the firm's auditing or other compliance requirements.

To accomplish this, there are two common options. The first one is to create database triggers that will clone the pre-update record into the history table before any update operations. The second is to develop the logic in the data access layer (for example, by using AOP). However, both options have their drawbacks. The trigger approach is tied to the database platform, while implementing the logic manually is quite clumsy and error prone.

Hibernate Envers (short for "entity versioning system") is a Hibernate module specifically designed to automate the versioning of entities. In this section, we discuss how to use Envers to implement the versioning of the `SingerAudit` entity.

Hibernate Envers is not a feature of JPA. We mention it here because we believe it's more appropriate to cover this after we have discussed some basic auditing features that you can use with Spring Data JPA.

Envers supports two auditing strategies, which are shown in Table 8-1.

Table 8-1. *Envers Auditing Strategies*

Auditing Strategy	Description
Default	Envers maintains a column for the revision of the record. Every time a record is inserted or updated, a new record will be inserted into the history table with the revision number retrieved from a database sequence or table.
Validity audit	This strategy stores both the start and end revisions of each history record. Every time a record is inserted or updated, a new record will be inserted into the history table with the start revision number. At the same time, the previous record will be updated with the end revision number. It's also possible to configure Envers to record the timestamp at which the end revision was updated into the previous history record.

In this section, we demonstrate the validity audit strategy. Although it will trigger more database updates, retrieving the history records becomes much faster. Because the end revision timestamp is also written to the history records, it will be easier to identify the snapshot of a record at a specific point in time when querying the data.

Adding Tables for Entity Versioning

To support entity versioning, we need to add a few tables. First, for each table that the entity (in this case, the SingerAudit entity class) will be versioning, we need to create the corresponding history table. For the versioning of records in the SINGER_AUDIT table, let's create a history table called SINGER_AUDIT_H. The following code snippet shows the table creation script (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)
);
```

To support the validity audit strategy, we need to add four columns for each history table, shown underlined in the previous script snippet. Table 8-2 shows the columns and their purposes. Hibernate Envers requires another table for keeping track of the revision number and the timestamp at which each revision was created. The table name should be REVINFO.

Table 8-2. *Envers Auditing Strategies*

Auditing Strategy	Data Type	Description
AUDIT_REVISION	INT	The start revision of the history record.
ACTION_TYPE	INT	The action type, with these possible values: 0 for add, 1 for modify, and 2 for delete
AUDIT_REVISION_END	INT	The end revision of the history record
AUDIT_REVISION_END_TS	TIMESTAMP	The timestamp at which the end revision was updated

The following code snippet shows the table creation script (schema.sql):

```
CREATE TABLE REVINFO (
    REVTSTMP BIGINT NOT NULL
    , REV INT NOT NULL AUTO_INCREMENT
    , PRIMARY KEY (REVTSTMP, REV)
);
```

The REV column is for storing each revision number, which will be auto-incremented when a new history record is created. The REVTSTMP column stores the timestamp (in a number format) when the revision was created.

Configuring EntityManagerFactory for Entity Versioning

Hibernate Envers is implemented in the form of EJB listeners. We can configure those listeners in the LocalContainerEntityManagerFactory bean. Here you can see the Java configuration class. There is no point in showing an XML configuration because the only difference for this section is that we have a lot of extra Hibernate-specific properties.

```
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);
        //Properties for 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();
    }
}
```

The Envers audit event listener org.hibernate.envers.event.AuditEventListener is attached to various persistence events. The listener intercepts the events post-insert, post-update, or post-delete and clones the pre-update snapshot of the entity class into the history table. The listener is also attached to those association update events (pre-collection-update, pre-collection-remove, and pre-collection-recreate) for handling the update operations of the entity class's associations. Envers is capable of keeping the history of the entities within an association (for example one-to-many or many-to-many). A few properties are also defined for Hibernate Envers, which are summarized in Table 8-3 (the prefix of the properties, org.hibernate.envers, is omitted for clarity).[2]

Table 8-3. *Hibernate Envers Properties Table*

Method	Description
audit_table_suffix	The table name suffix for the versioned entity. For example, for the entity class SingerAudit, which is mapped to the SINGER_AUDIT table, Envers will keep the history in the table SINGER_AUDIT_H, since we defined the value _H for the property.
revision_field_name	The history table's column for storing the revision number for each history record.
revision_type_field_name	The history table's column for storing the update action type.
audit_strategy	The audit strategy to use for entity versioning.
audit_strategy_validity_end_rev_field_name	The history table's column for storing the end revision number for each history record. Required only when using the validity audit strategy.
audit_strategy_validity_store_revend_timestamp	Whether to store the timestamp when the end revision number for each history record is updated. Required only when using the validity audit strategy.
audit_strategy_validity_revend_timestamp_field_name	The history table's column for storing the timestamp when the end revision number for each history record is updated. Required only when using the validity audit strategy and the previous property is set to true.

[2]You can find the full list of Hibernate properties in the Hibernate official documentation at https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html#envers-configuration.

Enabling Entity Versioning and History Retrieval

To enable versioning of an entity, just annotate the entity class with `@Audited`. This annotation can be used at the class level and then the changes on all the fields will be audited. If you want to escape certain fields from auditing, `@NotAudited` can be used on those fields. Here you can see a snippet of the `SingerAudit` entity class with the annotation applied:

```
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 {
    ...
}
```

The entity class is annotated with `@Audited`, which Envers listeners will check for and perform versioning of the updated entities. By default, Envers will also try to keep a history of the associations; if you want to avoid this, you should use the `@NotAudited` annotation mentioned earlier.

To retrieve the history records, Envers provides the `org.hibernate.envers.AuditReader` interface, which can be obtained from the `AuditReaderFactory` class. Let's add a new method called `findAuditByRevision()` into the `SingerAuditService` interface for retrieving the `SingerAudit` history record by the revision number. The following code snippet shows the `SingerAuditService` interface:

```
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);
}
```

To retrieve a history record, one option is to pass in the entity's ID and the revision number. The following code snippet shows the implementation of the method extracting revisions:

```
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);
    }
}
```

The EntityManager is injected into the class, which is passed to AuditReaderFactory to retrieve an instance of AuditReader. Then we can call the AuditReader.find() method to retrieve the instance of the SingerAudit entity at a particular revision.

Testing Entity Versioning

Let's take a look at how entity versioning works. The following code snippet shows the testing code snippet; the code for bootstrapping ApplicationContext and the listSingers() function is the same as the code in the SpringJpaDemo class.

```
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(4l, 1);
        System.out.println("");
        System.out.println("Old Singer with id 1 and rev 1:" +  oldSinger);
        System.out.println("");
        oldSinger = singerAuditService.findAuditByRevision(4l, 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();
```

```
        }
    }
}
```

We first create a new singer and then update it. Then we retrieve the `SingerAudit` entities with revisions 1 and 2, respectively. Running the code produces the following output:

```
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
```

From the previous output, you can see that after the update operation, the `SingerAudit`'s first name is changed to `Riley B.`. However, when looking at the history, at revision 1, the first name is `BB`. At revision 2, the first name becomes `Riley B.`. Also notice that the last-modified date of revision 2 reflects the updated date and time correctly.

Spring Boot JPA

Until now we have configured everything including entities, database, repositories, and services. As you probably expect by now, there should be a Spring Boot starter artifact to help you develop your project faster and minimize the effort of configuration. A Spring Boot JPA starter depends on Spring Boot JPA, so it comes with preconfigured embedded database; all it's needed is for the dependency to be on the classpath. It also comes with Hibernate to abstract the persistence layer. Spring `Repository` interfaces are automatically detected. So, all that is left for the developer to do is to provide entities, repository extensions, and an `Application` class to use them all together. Eventually, you can also develop a class to populate the database. All this will be done and explained in this section.

First we need to add the Spring Boot starter JPA as a dependency. This is done like all other libraries before it. Add the version, group ID, and artifact ID in the root `build.gradle` file and define the dependencies in the `chapter08/boot-jpa/build.gradle` file. The two configuration snippets are shown here:

```
//pro-spring-15/build.gradle
ext {
    //spring libs
    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"
    ]
}

//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
    }
}
```

After adding these dependencies and refreshing the project, the entire tree of `spring-boot-starter-data-jpa` dependencies should be visible in the IntelliJ IDEA Gradle Projects view, as shown in Figure 8-2.

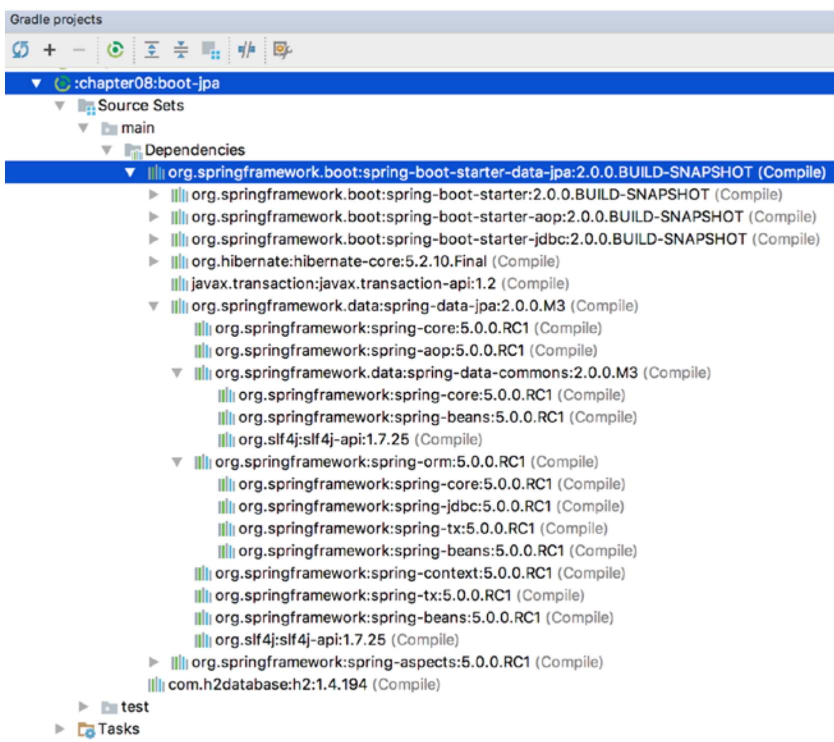

Figure 8-2. *Dependency tree for the Spring Boot Starter artifact*

The entities will be the same as used until now (`Singer`, `Album`, and `Instrument`), so there is no need to depict them again. The bean to initialize them is of type `DBInitializer`, and it is a service class that uses the `SingerRepository` and `InstrumentRepository` beans, provided by Spring Boot to save a set of objects to the database. Its contents are shown here:

```
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.");
    }

}
```

The SingerRepository and InstrumentRepository interfaces are quite simple for this example and are shown here:

```
//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>  {

}

//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);
}
```

SingerRepository will be directly injected into the Spring Boot annotated Application class and used to retrieve all singer records and their children records from the database and log them in the console. The Application class is shown here:

```
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()));
            }
        });
    }
}
```

This Application class introduces something new; it implements the CommandLineRunner interface. This interface is used to tell Spring Boot that the run() method should be executed when this bean is contained within a Spring application.

The scanBasePackages = "com.apress.prospring5.ch8.config" attribute is used to enable component scanning for the package (or packages) specified as an argument so that the beans contained there are created and added to the application context. This is needed when the beans are defined in a different package than the Application class.

Another thing you should notice is that no other configuration class is needed; you don't need any SQL scripts to initialize the database nor any other annotations on the Application class. Clearly, if you want to focus on the logic of an application, Spring Boot with its starter dependencies are quite handy.

If you run the previous class, you will get the expected result. (Note that the application waits for a key to be pressed before exiting).

```
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)
```

Considerations When Using JPA

Although we covered a fair amount, this chapter discussed only a small portion of JPA. For example, using JPA to call a database stored procedure was not covered. JPA is a complete and powerful ORM data access standard, and with the help of third-party libraries such as Spring Data JPA and Hibernate Envers, you can implement various crosscutting concerns relatively easily.

JPA is a JEE standard that is supported by most major open source communities as well as commercial vendors (JBoss, GlassFish, WebSphere, WebLogic, and so on). So, it's a compelling choice for adopting JPA as the data access standard. If you require absolute control over the query, you can use JPA's native query support instead of using JDBC directly.

In conclusion, for developing JEE applications with Spring, we recommend using JPA to implement the data access layer. When desired, you can still mix in JDBC for some special data access needs. Always remember that Spring allows you to mix and match data access technologies easily, with the transaction management transparently handled for you. And if you want to simplify the development even more, Spring Boot makes that happen with its preconfigured beans and custom configurations.

Summary

In this chapter, we covered the basic concepts of JPA and how to configure JPA's `EntityManagerFactory` in Spring by using Hibernate as the persistence service provider. Then we discussed using JPA to perform basic database operations. Advanced topics included native queries and the strongly typed JPA Criteria API.

Additionally, we demonstrated how Spring Data JPA's `Repository` abstraction can help simplify JPA application development, as well as how to use its entity listener to keep track of basic auditing information for entity classes. For full versioning of entity classes, using Hibernate Envers to fulfill the requirement was also covered.

Spring Boot for JPA applications was also covered, as it simplifies configuration a lot and keeps the focus on developing the required functionality.

In the next chapter, we discuss transaction management in Spring.

CHAPTER 9

Transaction Management

Transactions are one of the most critical parts of building a reliable enterprise application. The most common type of transaction is a database operation. In a typical database update operation, a database transaction begins, data is updated, and then the transaction is committed or rolled back, depending on the result of the database operation. However, in many cases, depending on the application requirements and the back-end resources that the application needs to interact with (such as an RDBMS, message-oriented middleware, an ERP system, and so on), transaction management can be much more complicated.

In the early days of Java application development (after JDBC was created but before the JEE standard or an application framework like Spring was available), developers programmatically controlled and managed transactions within application code. When JEE and, more specifically, the EJB standard became available, developers were able to use container-managed transactions (CMTs) to manage transactions in a declarative way. But the complicated transaction declaration in the EJB deployment descriptor was difficult to maintain and introduced unnecessary complexity for transaction processing. Some developers favored having more control over the transaction and chose bean-managed transactions (BMTs) to manage transactions in a programmatic way. However, the complexity of programming with the Java Transaction API (JTA) also hindered developers' productivity.

As discussed in Chapter 5, transaction management is a crosscutting concern and should not be coded within the business logic. The most appropriate way to implement transaction management is to allow developers to define transaction requirements in a declarative way and have frameworks such as Spring, JEE, or AOP weave in the transaction processing logic on our behalf. In this chapter, we discuss how Spring helps simplify the implementation of transaction-processing logic. Spring provides support for both declarative and programmatic transaction management.

Spring offers excellent support for declarative transactions, which means you do not need to clutter your business logic with transaction management code. All you have to do is declare those methods (within classes or layers) that must participate in a transaction, together with the details of the transaction configuration, and Spring will take care of handling the transaction management. To be more specific, this chapter covers the following:

- *Spring transaction abstraction layer*: We discuss the base components of Spring transaction abstraction classes and explain how to use these classes to control the properties of the transactions.
- *Declarative transaction management*: We show you how to use Spring and just plain Java objects to implement declarative transactional management. We offer examples for declarative transaction management using the XML configuration files as well as Java annotations.

© Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho 2017
I. Cosmina et al., *Pro Spring 5*, https://doi.org/10.1007/978-1-4842-2808-1_9

- *Programmatic transaction management*: Even though programmatic transaction management is not used very often, we explain how to use the Spring-provided `TransactionTemplate` class, which gives you full control over the transaction management code.
- *Global transactions with JTA*: For global transactions that need to span multiple back-end resources, we show how to configure and implement global transactions in Spring by using JTA.

Exploring the Spring Transaction Abstraction Layer

When developing your applications, no matter whether you choose to use Spring or not, you have to make a fundamental choice when using transactions about whether to use global or local transactions. Local transactions are specific to a single transactional resource (a JDBC connection, for example), whereas global transactions are managed by the container and can span multiple transactional resources.

Transaction Types

Local transactions are easy to manage, and if all operations in your application need to interact with just one transactional resource (such as a JDBC transaction), using local transactions will be sufficient. However, if you are not using an application framework such as Spring, you have a lot of transaction management code to write, and if in the future the scope of the transaction needs to be extended across multiple transactional resources, you have to drop the local transaction management code and rewrite it to use global transactions.

In the Java world, global transactions were implemented with the JTA. In this scenario, a JTA-compatible transaction manager connects to multiple transactional resources via respective resource managers, which are capable of communicating with the transaction manager over the XA protocol (an open standard defining distributed transactions), and the 2 Phase Commit (2PC) mechanism was used to ensure that all back-end data sources were updated or rolled back altogether. If either of the back-end resources fails, the entire transaction will roll back, and hence the updates to other resources will be rolled back too.

Figure 9-1 shows a high-level view of global transactions with JTA. As shown in Figure 9-1, four main parties participate in a global transaction (also generally referred to as a *distributed transaction*). The first party is the back-end resource, such as an RDBMS, messaging middleware, an enterprise resource planning (ERP) system, and so on.

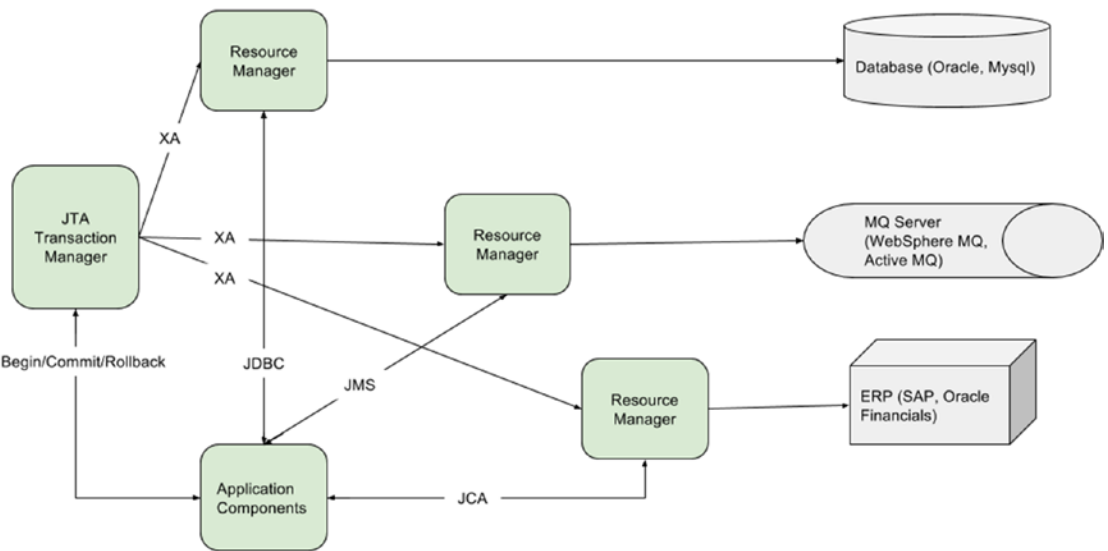

Figure 9-1. *Overview of global transactions with JTA*

The second party is the resource manager, which is generally provided by the back-end resource vendor and is responsible for interacting with the back-end resource. For example, when connecting to a MySQL database, we need to interact with the `MysqlXADataSource` class provided by MySQL's Java connector. Other back-end resources (for example, MQ, ERP, and so on) provide their resource managers too.

The third party is the JTA transaction manager, which is responsible for managing, coordinating, and synchronizing the transaction status with all resource managers that are participating in the transaction. The XA protocol is used, which is an open standard widely used for distributed transaction processing. The JTA transaction manager also supports 2PC so that all changes will be committed together, and if any resource update fails, the entire transaction will be rolled back, resulting in none of the resources being updated. The entire mechanism was specified by the Java Transaction Service (JTS) specification.

The final component is the application. Either the application itself or the underlying container or Spring Framework that the application runs on manages the transaction (begin, commit, roll back a transaction, and so on). At the same time, the application interacts with the underlying back-end resources via various standards defined by JEE. As shown in Figure 9-1, the application connects to the RDBMS via JDBC, MQ via JMS, and an ERP system via Java EE Connector Architecture (JCA).

JTA is supported by all full-blown JEE-compliant application servers (for example, JBoss, WebSphere, WebLogic, and GlassFish), within which the transaction is available via JNDI lookup. As for stand-alone applications or web containers (for example, Tomcat and Jetty), there also exist open source and commercial solutions that provide support for JTA/XA in those environments (for example, Atomikos, Java Open Transaction Manager [JOTM], and Bitronix).

Implementations of the PlatformTransactionManager

In Spring, the `PlatformTransactionManager` interface uses the `TransactionDefinition` and `TransactionStatus` interfaces to create and manage transactions. The actual implementation of these interfaces must have detailed knowledge of the transaction manager.

Figure 9-2 shows the implementations of PlatformTransactionManager in Spring.

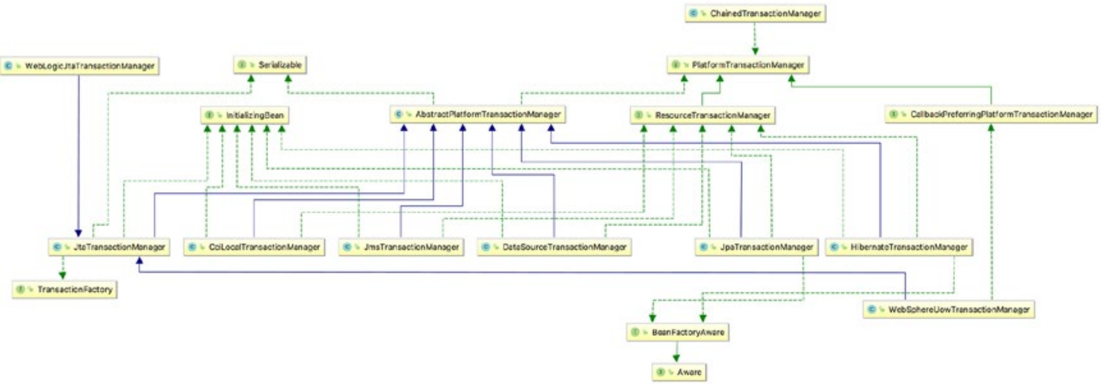

Figure 9-2. *PlatformTransactionManager implementations as of Spring*

Spring provides a rich set of implementations for the PlatformTransactionManager interface. The CciLocalTransactionManager class supports JEE, JCA, and the Common Client Interface (CCI). The DataSourceTransactionManager class is for generic JDBC connections. For the ORM side, there are a number of implementations, including JPA (the JpaTransactionManager class)[1] and Hibernate 5 (HibernateTransactionManager).[2] For JMS, the implementations support JMS 2.0 through the JmsTransactionManager class.[3] For JTA, the generic implementation class is JtaTransactionManager. Spring also provides several JTA transaction manager classes that are specific to particular application servers. Those classes provide native support for WebSphere (the WebSphereUowTransactionManager class), WebLogic (the WebLogicJtaTransactionManager class), and Oracle OC4J (the OC4JJtaTransactionManager class).

Analyzing Transaction Properties

In this section, we discuss the transaction properties that Spring supports, focusing on interacting with RDBMS as the back-end resource.

Transactions have the four notoriously known ACID properties (atomicity, consistency, isolation, and durability), and it is up to the transactional resources to maintain these aspects of a transaction. You cannot control the atomicity, consistency, and durability of a transaction. However, you can control the transaction propagation and timeout, as well as configure whether the transaction should be read-only and specify the isolation level.

Spring encapsulates all these settings in a TransactionDefinition interface. This interface is used in the core interface of the transaction support in Spring, which is the PlatformTransactionManager interface, whose implementations perform transaction management on a specific platform, such as JDBC or JTA. The core method, PlatformTransactionManager.getTransaction(), takes a TransactionDefinition interface as an argument and returns a TransactionStatus interface. The TransactionStatus interface is used to control the transaction execution, more specifically to set the transaction result and to check whether the transaction is completed or whether it is a new transaction.

---

[1]JDO support was dropped in Spring 5; thus, JdoTransactionManager is missing from the class diagram.
[2]Spring 5 works only with Hibernate 5; implementations for Hibernate 3 and Hibernate 4 have been removed.
[3]Support for JMS 1.1 was dropped in Spring 5.

The TransactionDefinition Interface

As we mentioned earlier, the `TransactionDefinition` interface controls the properties of a transaction. Let's take a more detailed look at the `TransactionDefinition` interface, shown here, and describe its methods:

```
package org.springframework.transaction;

import java.sql.Connection;

public interface TransactionDefinition {

    // Variable declaration statements omitted

    ...

    int getPropagationBehavior();

    int getIsolationLevel();

    int getTimeout();

    boolean isReadOnly();

    String getName();

}
```

The simple and obvious methods of this interface are `getTimeout()`, which returns the time (in seconds) in which the transaction must complete, and `isReadOnly()`, which indicates whether the transaction is read-only. The transaction manager implementation can use this value to optimize the execution and check to make sure that the transaction is performing only read operations. The `getName()` method returns the name of the transaction.

The other two methods, `getPropagationBehavior()` and `getIsolationLevel()`, need to be discussed in more detail. We begin with `getIsolationLevel()`, which controls what changes to the data other transactions see. Table 9-1 lists the transaction isolation levels you can use and explains what changes made in the current transaction other transactions can access. The isolation levels are represented as static values defined in the `TransactionDefinition` interface.

Table 9-1. *Transaction Isolation Levels*

Isolation Level	Description
`ISOLATION_DEFAULT`	Default isolation level of the underlying data store.
`ISOLATION_READ_UNCOMMITTED`	Lowest level of isolation; it is barely a transaction at all because it allows this transaction to see data modified by other uncommitted transactions.
`ISOLATION_READ_COMMITTED`	Default level in most databases; it ensures that other transactions are not able to read data that has not been committed by other transactions. However, the data that was read by one transaction can be updated by other transactions.
`ISOLATION_REPEATABLE_READ`	Stricter than `ISOLATION_READ_COMMITTED`; it ensures that once you select data, you can select at least the same set again. However, if other transactions insert new data, you can still select the newly inserted data.
`ISOLATION_SERIALIZABLE`	The most expensive and reliable isolation level; all transactions are treated as if they were executed one after another.

Choosing the appropriate isolation level is important for the consistency of the data, but making these choices can have a great impact on performance. The highest isolation level, `ISOLATION_SERIALIZABLE`, is particularly expensive to maintain.

The `getPropagationBehavior()` method specifies what happens to a transactional call, depending on whether there is an active transaction. Table 9-2 describes the values for this method. The propagation types are represented as static values defined in the `TransactionDefinition` interface.

Table 9-2. *Transaction Isolation Levels*

Propagation Type	Description
PROPAGATION_REQUIRED	Supports a transaction if one already exists. If there is no transaction, it starts a new one.
PROPAGATION_SUPPORTS	Supports a transaction if one already exists. If there is no transaction, it executes nontransactionally.
PROPAGATION_MANDATORY	Supports a transaction if one already exists. Throws an exception if there is no active transaction.
PROPAGATION_REQUIRES_NEW	Always starts a new transaction. If an active transaction already exists, it is suspended.
PROPAGATION_NOT_SUPPORTED	Does not support execution with an active transaction. Always executes nontransactionally and suspends any existing transaction.
PROPAGATION_NEVER	Always executes nontransactionally even if an active transaction exists. Throws an exception if an active transaction exists.
PROPAGATION_NESTED	Runs in a nested transaction if an active transaction exists. If there is no active transaction, the execution is executed as if `PROPAGATION_REQUIRED` is set.

The TransactionStatus Interface

The `TransactionStatus` interface, shown next, allows a transactional manager to control the transaction execution. The code can check whether the transaction is a new one or whether it is a read-only transaction, and it can initiate a rollback.

```
package org.springframework.transaction;

public interface TransactionStatus extends SavepointManager {

    boolean isNewTransaction();

    boolean hasSavepoint();

    void setRollbackOnly();

    boolean isRollbackOnly();

    void flush();

    boolean isCompleted();

}
```

The methods of the `TransactionStatus` interface are fairly self-explanatory; the most notable one is `setRollbackOnly()`, which causes a rollback and ends the active transaction.

The `hasSavePoint()` method returns whether the transaction internally carries a save point (that is, the transaction was created as a nested transaction based on a save point). The `flush()` method es the underlying session to a data store if applicable (for example, when using with Hibernate). The `isCompleted()` method returns whether the transaction has ended (that is, committed or rolled back).

Sample Data Model and Infrastructure for Example Code

This section provides an overview of the data model and the infrastructure used in our examples of transaction management. We use JPA with Hibernate as the persistence layer for implementing data access logic. In addition, the Spring Data JPA and its repository abstraction are used to simplify the development of basic database operations.

Creating a Simple Spring JPA Project with Dependencies

Let's start by creating the project. Because we are using JPA, we also need to add the required dependencies to the project for the examples in this chapter.

```
//pro-spring-15/build.gradle
ext {
    //spring libs
    springVersion = '5.0.0.RC1'
    bootVersion = '2.0.0.BUILD-SNAPSHOT'
    springDataVersion = '2.0.0.M3'

    //logging libs
    slf4jVersion = '1.7.25'
    logbackVersion = '1.2.3'
    guavaVersion = '21.0'
    junitVersion = '4.12'

    aspectjVersion = '1.9.0.BETA-5'

    //database library
    h2Version =  '1.4.194'

    //persistency libraries
    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"
    ]
}

//chapter09/build.gradle
dependencies {
    //we specify these dependencies for all submodules, except
    // the boot module, that defines its own
    if !project.name.contains"boot" {
        //we exclude transitive dependencies, because spring-data
        //will take care of these
           compile spring.contextSupport {
           exclude  module: 'spring-context'
           exclude  module: 'spring-beans'
           exclude  module: 'spring-core'
        }
        //we exclude the 'hibernate' transitive dependency
        //to have control over the version used
```

```
        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
}
```

To observe the detailed behavior of the example code as we modify the transaction attributes, let's also turn on the DEBUG-level logging in logback. The following snippet shows the logback.xml file:

```
<?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>
```

Sample Data Model and Common Classes

To keep things simple, we will use just two tables, namely, the SINGER and ALBUM tables that we used throughout the chapters about data access. No SQL script to create the tables is needed because you can use the Hibernate property hibernate.hbm2ddl.auto and set it to create-drop so we'll always have a clean run every time you test something. The tables will be generated based on the *Singer* and Album entities. Snippets with the annotated fields of each are depicted here:

```
//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<>();

    ...
}
/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;
    ...
}
```

These two classes will be isolated in a project named `base-dao` that will be a dependency for all transaction projects. Aside from the entities, repository interfaces are defined in this project also. They will be depicted a little bit later in the session. A configuration class to define the `DataSource` bean is also needed. The configuration class is shown here, and because of practical and educational purposes, the database credentials, driver, and URL will be used directly and not read from an external file. (In production you will never encounter this, though. We hope.)

```
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();
    }
}
```

An embedded H2 database is defined. The credentials are set directly in the code, and the DataSource implementation is SimpleDriverDataSource, which is designed to be used only in simple, testing, or educational applications.

Currently, using annotations is the most common way to define transaction requirements in Spring. The main benefit is that the transaction requirement together with the detail transaction properties (timeout, isolation level, propagation behavior, and so on) are defined within the code itself, which makes the application easier to trace and maintain. The configuration is also done using annotations and Java configuration classes. To enable annotation support for transaction management in Spring using XML configuration, we need to add the <tx:annotation-driven> tag in the XML configuration file. In the following configuration snippet, you can see a snippet of transactional configuration and the transactional matching namespaces. You can find the full configuration in the project in case you are interested.

```
<?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>
```

Because we are using JPA, you define the JpaTransactionManager bean. The <tx:annotation-driven> tag specifies that we are using annotations for transaction management. This simple definition instructs Spring to look for a bean named transactionManager of type PlatformTransactionManager. If the transaction bean is named differently, let's say customTransactionManager, the element definition must be declared with the transaction-manager attribute that must receive as a value the name of the transaction management bean.

```
<tx:annotation-driven transaction-manager="customTransactionManager"/>
```

The EntityManagerFactory bean is then defined, followed by the <context:component-scan> tag to scan the service-layer classes. Finally, the <jpa:repositories> tag is used to enable Spring Data JPA's repository abstraction. This element is replaced in the DataJpaConfiguration class with the @EnableJpaRepositories annotation.

In professional environments, it is a common practice to separate the persistence configuration (DAO) from the transactional configuration (service). That is why the contents of the XML introduced before were split in the Java configuration into two configuration classes. DataJpaConfig, which was introduced before, contains only data access beans, and ServicesConfig, which is depicted next, contains only transactional management–related beans:

```
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);
    }
}
```

For the implementation of the `SingerService` interface, we begin by creating the class with an empty implementation of all the methods in the `SingerService` interface. Let's implement the `SingerService.findAll()` method first. The following code snippet shows the `SingerServiceImpl` class with the `findAll()` method implemented:

```
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());
    }
}
```

When using annotation-based transaction management, the only annotation that we need to deal with is `@Transactional`. In the previous code snippet, the `@Transactional` annotation is applied at the class level, which means that, by default, Spring will ensure that a transaction is present before the execution of each method within the class. The `@Transactional` annotation supports a number of attributes that you can provide to override the default behavior. Table 9-3 shows the available attributes, together with the possible and default values.

Table 9-3. Attributes for the @Transactional Annotation

Attribute Name	Default Value	Possible Values
propagation	Propagation.REQUIRED	Propagation.REQUIRED Propagation.SUPPORTS Propagation.MANDATORY Propagation.REQUIRES_NEW Propagation.NOT_SUPPORTED Propagation.NEVER Propagation.NESTED
isolation	Isolation.DEFAULT (default isolation level of the underlying resource)	Isolation.DEFAULT Isolation.READ_UNCOMMITTED Isolation.READ_COMMITTED Isolation.REPEATABLE_READ Isolation.SERIALIZABLE
timeout	TransactionDefinition.TIMEOUT_DEFAULT (default transaction timeout in seconds of the underlying resource)	An integer value larger than zero; indicates the number in seconds for timeout
readOnly	false	{true, false}
rollbackFor	Exception classes for which the transaction will be rolled back	N/A
rollbackForClassName	Exception class names for which the transaction will be rolled back	N/A
noRollbackFor	Exception classes for which the transaction will not be rolled back	N/A
noRollbackForClassName	Exception class names for which the transaction will not be rolled back	N/A
value	"" (a qualifier value for the specified transaction)	N/A

As a result, based on Table 9-3, the @Transactional annotation without any attribute means that the transaction propagation is required, the isolation is the default, the timeout is the default, and the mode is read-write. For the findAll() method introduced previously, the method is annotated with @Transactional(readOnly=true). This will override the default annotation applied at the class level, with all other attributes unchanged, but the transaction is set to read-only. The following code snippet shows the testing program for the findAll() method:

```
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();
    }
}
```

With the proper logging enabled, in other words, `<logger name="org.springframework.orm.jpa" level="debug"/>`, you will be able to see in the log messages related to transaction handling. Running the program produces the following reduced output (see the debug log in the console for full details):

```
DEBUG o.s.o.j.JpaTransactionManager - Creating new transaction with name
  [com.apress.prospring5.ch9.services.SingerServiceImpl.findAll]:
     PROPAGATION_REQUIRED,ISOLATION_DEFAULT,readOnly; ''
DEBUG o.s.o.j.JpaTransactionManager - Participating in existing transaction
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

DEBUG o.s.o.j.JpaTransactionManager - Initiating transaction commit

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
```

As shown in the previous output, the irrelevant output statements were removed for clarity. First, before the `findAll()` method is run, Spring's `JpaTransactionManager` creates a new transaction (the name is equal to the fully qualified class name with the method name) with default attributes, but the transaction is set to read-only, as defined at the method-level `@Transactional` annotation. Then, the query is submitted, and upon completion and without any errors, the transaction is committed. The `JpaTransactionManager` handles the creation and commit operations of the transaction.

Let's proceed to the implementation of the update operation. We need to implement both the `findById()` and `save()` methods in the `SingerServiceImpl` interface. The following code snippet shows the implementation:

```
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);
    }
}
```

The `findById()` method is also annotated with `@Transactional(readOnly=true)`. Generally, the `readOnly=true` attribute should be applied to all finder methods. The main reason is that most persistence providers will perform a certain level of optimization on read-only transactions. For example, Hibernate will not maintain the snapshots of the managed instances retrieved from the database with read-only turned on.

For the `save()` method, we simply invoke the `CrudRepository.save()` method and don't provide any annotation. This means the class-level annotation will be used, which is a read-write transaction. Let's modify the `TxAnnotationDemo` class for testing the `save()` method, as shown in the following code:

```
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();
    }
}
```

The Singer object with an ID of 1 is retrieved, and then the first name is updated and saved to the database. Running the code produces the following relevant output:

```
Singer saved successfully: Singer - Id: 1, First name: John Clayton,
   Last name: Mayer, Birthday: 1977-10-16
```

The save() method gets the default attributes that are inherited from the class-level @Transactional annotation. Upon completion of the update operation, Spring's JpaTransactionManager fires a transaction commit, which causes Hibernate to flush the persistence context and commit the underlying JDBC connection to the database. Finally, let's take a look at the countAll() method. We will investigate two transaction configurations for this method. Although the CrudRepository.count() method can fulfill the purpose, we will not use that method. Instead, we will implement another method for demonstration purposes, mainly because the methods defined by the CrudRepository interface in Spring Data are already marked with the appropriate transaction attributes.

The following code snippet shows the new method countAllSingers() defined in the SingerRepository interface:

```
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();
}
```

For the new countAllSingers() method, the @Query annotation is applied, with the value equaling the JPQL statement that counts the number of contacts. The following code snippet shows the implementation of the countAll() method in the SingerServiceImpl class:

```
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();
    }
}
```

The annotation is the same as other finder methods. To test this method, just add `System.out.println("Singer count: " + contactService.countAll());` in the `main()` method of the `TxAnnotationDemo` class and watch the console. If you see a message like `Singer count: 3`, the method was executed correctly.

In this output, you can see that the transaction for `countAll()` was created with read-only equaling true, as expected. But for the `countAll()` function, we don't want it to be enlisted in a transaction at all. We don't need the result to be managed by the underlying JPA `EntityManager`. Instead, we just want to get the count and forget about it. In this case, we can override the transaction propagation behavior to `Propagation.NEVER`. The following method shows the revised `countAll()` method:

```
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();
    }
}
```

Run the testing code again, and you will find that the transaction will not be created for the `countAll()` method in the debug output.

This section covered some major configurations that you will deal with when processing transactions on a day-to-day basis. For special cases, you may need to define the timeout, isolation level, rollback (or not) for specific exceptions, and so on.

⚠ Spring's `JpaTransactionManager` doesn't support a custom isolation level. Instead, it always uses the default isolation level for the underlying data store. If you are using Hibernate as the JPA service provider, you can use a workaround: extend the `HibernateJpaDialect` class to support a custom isolation level.

Using AOP Configuration for Transaction Management

Another common approach of declarative transaction management is to use Spring's AOP support. Before Spring version 2, we needed to use the `TransactionProxyFactoryBean` class to define transaction requirements for Spring beans. However, ever since version 2, Spring provides a much simpler way by introducing the `aop` namespace and using the common AOP configuration technique for defining transaction requirements. Of course, after introducing annotations, this way of configuring transaction management is deprecated as well. But is useful to know it exists, just in case you might need to wrap in a transaction code that is not part of your project and you cannot edit it to add `@Transaction` annotations in it.

In the following configuration snippet, the example in the previous section is configured using XML and makes use of the `aop` namespace:

```
<?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>
```

The configuration is quite similar to the XML one introduced at the beginning of the section. Basically, the `<tx:annotation-driven>` tag is removed, and the `<context:component-scan>` tag is modified for the package name we used for declarative transaction management. The most important tags are `<aop:config>` and `<tx:advice>`.

Under the `<aop:config>` tag, a pointcut is defined for all operations within the service layer (that is, all implementation classes under the `com.apress.prospring5.ch9.services` package). The advice is referencing the bean with an ID of `txAdvice`, which is defined by the `<tx:advice>` tag. In the `<tx:advice>` tag, we configure the transaction attributes for various methods that we want to participate in a transaction. As shown in the tag, you specify that all finder methods (methods with the prefix `find`) will be read-only, and we specify that the count methods (methods with the prefix `count`) will not participate in the transaction. For the rest of the methods, the default transaction behavior will be applied. This configuration is the same as the one in the annotation example.

Because transaction management is done explicitly through `aop`, the `@Transactional` annotation is no longer needed on the `SingerServiceImpl` class or methods within.

To test the previous configuration, you can use the following class:

```
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);

    // Testing  findAll()
        List<Singer> singers = singerService.findAll();
        singers.forEach(s -> System.out.println(s));

    // Testing save()
        Singer singer = singerService.findById(1L);
        singer.setFirstName("John Clayton");
        singerService.save(singer);
        System.out.println("Singer saved successfully: " + singer);

        // Testing  countAll()
        System.out.println("Singer count: " + singerService.countAll());

        ctx.close();
    }
}
```

We will leave you to test the program and observe the output for transaction-related operations that Spring and Hibernate have performed. Basically, they are the same as the annotation example.

Using Programmatic Transactions

The third option is to control the transaction behavior programmatically. In this case, we have two options. The first one is to inject an instance of `PlatformTransactionManager` into the bean and interact with the transaction manager directly. Another option is to use the Spring-provided `TransactionTemplate` class, which simplifies your work a lot. In this section, we demonstrate using the `TransactionTemplate class`. To make it simple, we focus on implementing the `SingerServiceImpl.countAll()` method. The following code snippet depicts the `ServiceConfig` class, modified for using programmatic transactions:

```
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);
    }
}
```

Here the AOP transaction advice is removed. In addition, a `transactionTemplate` bean is defined, using the `org.springframework.transaction.support.TransactionTemplate` class, with a few transaction attributes. Also, the `@EnableTransactionManagement` was removed, because transaction management is not now done explicitly. Let's take a look at the implementation of the `countAll()` method, which is shown here:

```
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());
    }
}
```

Here the `TransactionTemplate` class is injected from Spring. And then in the `countAll()` method, the `TransactionTemplate.execute()` method is invoked, passing in a declaration of an inner class that implements the `TransactionCallback<T>` interface. Then `doInTransaction()` is overridden with the desired logic. The logic will run within the attributes as defined by the `transactionTemplate` bean. The reason you are not clearly seeing all that in the previous code snippet is because Java 8 lambda expression are used. The following code is the expanded version (because it was written before lambda expressions were introduced) of the previous method:

```
public long countAll() {
       return transactionTemplate.execute(new TransactionCallback<Long>() {
           public Long doInTransaction(TransactionStatus transactionStatus) {
               return em.createNamedQuery(Singer.COUNT_ALL,
                       Long.class).getSingleResult();
           }
       });
   }
```

The following code snippet shows the testing program:

```
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();
    }
}
```

We will leave it to you to run the program and observe the result. Try to tweak the transaction attributes and see what happens in the transaction processing of the `countAll()` method.

Considerations on Transaction Management

So, having discussed the various ways for implementing transaction management, which one should you use? The declarative approach is recommended in all cases, and you should avoid implementing transaction management within your code as much as possible. Most of the time, when you find it necessary to code transaction control logic in the application, it is because of bad design, and in this case, you should consider refactoring your logic into manageable pieces and have the transaction requirements defined on those pieces declaratively.

For the declarative approach, using XML and using annotations both have their own pros and cons. Some developers prefer not to declare transaction requirements in code, while others prefer using annotations for easy maintenance, because you can see all the transaction requirement declarations within the code. Again, let the application requirements drive your decision, and once your team or company has standardized on the approach, stay consistent with the configuration style.

Global Transactions with Spring

Many enterprise Java applications need to access multiple back-end resources. For example, a piece of customer information received from an external business partner may need to update the databases for multiple systems (CRM, ERP, and so on). Some may even need to produce a message and send it to an MQ server via JMS for all other applications within the company that are interested in customer information. Transactions that span multiple back-end resources are referred to as *global* (or *distributed*) transactions.

A main characteristic of a global transaction is the guarantee of atomicity, which means that involved resources are all updated, or none is updated. This includes complex coordination and synchronization logic that should be handled by the transaction manager. In the Java world, JTA is the de facto standard for implementing global transactions.

Spring supports JTA transactions equally well as local transactions and hides that logic from the business code. In this section, we demonstrate how to implement global transactios by using JTA with Spring.

Infrastructure for Implementing the JTA Sample

We are using the same tables as for the previous samples in this chapter. However, the embedded H2 database doesn't fully support XA (at least at the time of writing), so in this example, we use MySQL as the back-end database.

We also want to show how to implement global transactions with JTA in a stand-alone application or web container environment. So, in this example, we use Atomikos (`www.atomikos.com/Main/TransactionsEssentials`), which is a widely used open source JTA transaction manager for use in a non-JEE environment.

To show how global transactions work, we need at least two back-end resources. To make things simple, we will use one MySQL database but two JPA entity managers to simulate the use case. The effect is the same because you have multiple JPA persistence units to distinct back-end databases.

In the MySQL database, we create two schemas and corresponding users, as shown in the following DDL script:

```
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;
```

After the setup has completed, we can proceed to the Spring configuration and implementation.

Implementing Global Transactions with JTA

First let's take a look at Spring's configuration. The following code snippet depicts the `XAJpaConfig` configuration class that declares beans needed to access the two databases:

```
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");
        // required by Hibernate 5
        hibernateProp.put("hibernate.transaction.coordinator_class", "jta");
        hibernateProp.put("hibernate.dialect",
            "org.hibernate.dialect.MySQL5Dialect");
        // this will work only if users/schemas are created first,
        // use ddl.sql script for this
        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();
    }
}
```

The configuration is long but not too complex. First, two `DataSource` beans are defined to indicate the two database resources. The bean names are `dataSourceA` and `dataSourceB`, which connect to the schemas `musicdb_a` and `musicdb_a`, respectively. Both `DataSource` beans use the class `com.atomikos.jdbc.AtomikosDataSourceBean`, which supports an XA-compliant `DataSource`, and within the two beans' definitions, MySQL's XA `DataSource` implementation class is defined: `com.mysql.cj.jdbc.MysqlXADataSource`, which is the resource manager for MySQL. Then, the database connection information is provided. Note that the `poolSize` attribute defines the number of connections within the connection pool that Atomikos needs to maintain. It's not mandatory. However, if the attribute is not provided, Atomikos will use the default value 1.

Then, two `EntityManagerFactory` beans are defined, named `emfA` and `emfB`. The common JPA properties are wrapped together in the `hibernateProperties` bean. The only difference between the two beans is that they were injected with the corresponding data source (that is, `dataSourceA` injected into `emfA`, and `dataSourceB` injected into `emfB`). Consequently, `emfA` will connect to MySQL's `prospring5_a` schema via the `dataSourceA` bean, while `emfB` will connect to the `prospring5_b` schema via the `dataSourceB` bean. Take a look at the properties `hibernate.transaction.factory_class` and `hibernate.transaction.jta.platform` in the `emfBase` bean. These two properties are very important because they are used by Hibernate to look up the underlying `UserTransaction` and `TransactionManager` beans to participate in the persistence context that it's managing into the global transaction. Also important is the `hibernate.transaction.coordinator_class` that is needed to make the Atomikos classes for Hibernate 4 work with Hibernate 5.[4]

The following codes snippet depicts `ServicesConfig`, which declares beans used for implementing the global transactions management:

```
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;
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;
```

[4]This configuration was created with help from the official Atomikos documentation for integration with Spring at https://www.atomikos.com/Documentation/SpringIntegration and with help from the Stack Overflow community at https://stackoverflow.com/questions/33127854/hibernate-5-with-spring-jta.

```
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;
    }
}
```

For the Atomikos part, two beans, the atomikosTransactionManager and atomikosUserTransaction beans, are defined. The implementation classes are provided by Atomikos, which implements the standard Spring org.springframework.transaction.PlatformTransactionManager and javax.transaction.UserTransaction interfaces, respectively. Those beans provide the transaction coordination and synchronization services required by JTA and communicate with the resource managers over the XA protocol in supporting 2PC. Then, Spring's transactionManager bean (with org.springframework.transaction.jta.JtaTransactionManager as the implementation class) is defined, injecting the two transaction beans provided by Atomikos. This instructs Spring to use Atomikos JTA for transaction management. Also, notice the UserTransactionService bean that is used to configure the Atomikos transaction service to administer pending transactions.[5]

The following code snippet shows the SingerServiceImpl class for JTA. Note that for simplicity, only the save() method is implemented.

```
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");
    }
```

[5]This configuration is an annotation configuration adaptation of the XML configuration given as an example in the Atomikos documentation at https://www.atomikos.com/Documentation/SpringIntegration#The_Advanced_Case_40As_of_3.3_41.

```
    @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;
    }
}
```

The two entity managers defined are injected into the `SingerServiceImpl` class. In the `save()` method, we persist the contact object to the two schemas, respectively. Ignore the `throw exception` statement at the moment; we will use it later to verify that the transaction was rolled back when saving to the schema `prospring5_b` fails. The following code snippet shows the testing program:

```
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();
    }
}
```

The program creates a new contact object and calls the `SingerService.save()` method. The implementation will try to persist the same object to two databases. Providing all went well, running the program produces the following output (the other output was omitted):

```
--> Singer saved successfully
```

Atomikos creates a composite transaction, communicates with the XA `DataSource` (MySQL, in this case), performs synchronization, commits the transaction, and so on. From the database, you will see that the new contact is persisted to both schemas of the database, respectively. But if you want to check the saving in the code, you can provide an implementation to the `findAll()` method that does that for you.

```
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()
    {
        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();
    }
    ...
}
```

So, the code to test the singer being saved in both databases can be modified like this:

```
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!!");
        }
```

```
        // check saving in  both databases
        List<Singer> singers = singerService.findAll();
        if (singers.size()!= 2) {
            logger.error("--> Something went wrong.");
        } else {
            logger.info("--> Singers form both DBs: " + singers);
        }

        ctx.close();
    }
}
```

Now let's see how the rollback works. As shown in the next code snippet, instead of calling `emB.persist()`, we just throw an exception to simulate that something went wrong and the data could not be saved in the second database.

```
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;
    }
}
```

Running the program again produces the following results:

```
...
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.
```

As shown in the previous output, the first singer is persisted (note the `insert` statement). However, when saving to the second `DataSource`, because an exception is thrown, Atomikos will roll back the entire transaction. You can take a look at the schema `musicdb_a` to check that the new singer was not saved.

Spring Boot JTA

The Spring Boot for JTA starter comes out of the box with a set of preconfigured beans designed to help you focus on the business functionality of the code and not on the environment setup. Then again, this is what all Spring Boot starter libraries do, no matter the component, so the previous sentence might seem a little redundant. Spring Boot for JTA contains a library for using Atomikos, which pulls the appropriate libraries and configures Atomikos components for you. Migrating the previous example to Spring Boot would mean basically importing the `DataSource` and transaction manager configurations into the Spring Boot application. But as the purpose of this section is to show how Spring Boot can help speed up development for applications involving global transaction management by the provided preconfigured beans, a different example is

necessary. We'll assume we want to transmit a message to a messaging queue signaling the creation of a new Singer instance. Obviously, if saving the Singer record to the database fails, we want to roll back the transaction and prevent the message from being sent. To wrap up this example, we need to do the following:

- Configure the Spring Boot Gradle project for JTA and JMS usage. The configuration is as follows:

```
//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"
   ]
}
//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'
   }
}
```

In Figure 9-3 you can see the Spring Boot starter libraries declared earlier as dependencies for the project, and you can see the dependencies they add to the project.

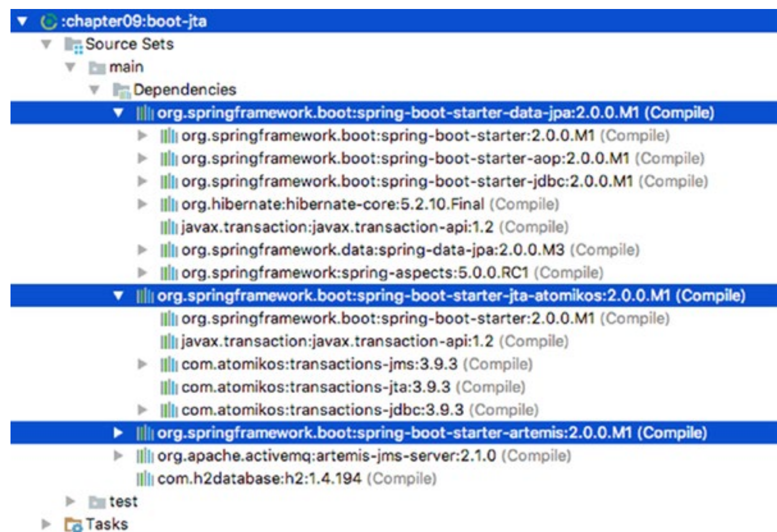

Figure 9-3. *Spring Boot Starter libraries and their dependencies*

- Define the `Singer` entity class and the repository handling it. The `Singer` entity class has the same structure as mentioned earlier in the section, but without any related entities. And the `SingerRepository` will be left empty as only methods already provided by `CrudRepository` will be used in this example: `save(..)` and `count()`.
- Define a service class that will save the `Singer` record and send the confirmation message.

```
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();
    }
}
```

No `@Autowired` annotation is needed to inject the repository bean, nor for the `JmsTemplate`. Spring Boot is magic like that and injects the required beans, if they are the only ones declared.

- A bean that will listen for messages being delivered to the `singers` queue and print them.

```
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);
   }
}
```

- Configure the Artemis JMS server to create an embedded queue named `singers`. This is done by setting the `spring.artemis.embedded.queues` property with the value `singers` in the `application.properties` file, which is the file that can be used to configure a Spring Boot application.

```
spring.artemis.embedded.queues=singers
spring.jta.log-dir=out
```

The previous configuration snippet depicts the content of the `application.properties` file. Besides the `spring.artemis.embedded.queues` property, `spring.jta.log-dir` is used to set where the JTA log should be written by Atomikos, and in this case the out directory was set.

- Here is the application class to wrap it all together and test it:

```
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());
                }
             }
         }
```

If you run Application, you will see output similar to this:

```
...
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
...//etc
```

From the log you can clearly see the global transaction being created and reused for each of the operations. If you exit the application normally by pressing any key and then Enter, be patient, because it takes a while for the application to shut down gracefully.

Here are some conclusions about creating JTA applications with Spring Boot: although it seems easy, when working with multiple data sources, configuring the environment is something you cannot get out of. Also, if the JTA provider is provided by a JEE server, things get quite complicated. But for sample applications used for educational purposes and testing, it is quite practical.

Considerations on Using JTA Transaction Manager

Whether to use JTA for global transaction management is under hot debate. For example, the Spring development team generally does not recommend using JTA for global transactions.

As a general principle, when your application is deployed to a full-blown JEE application server, there is no point in not using JTA because all the vendors of the popular JEE application servers have optimized their JTA implementation for their platforms. That's one major feature you are paying for.

For stand-alone or web container deployment, let the application requirements drive your decision. Perform load testing as early as possible to verify that performance is not being impaired by using JTA.

One piece of good news is that Spring works seamlessly with both local and global transactions in most major web and JEE containers, so code modification is generally not required when you switch from one transaction management strategy to another. If you decide to use JTA within your application, make sure you use Spring's `JtaTransactionManager`.

Summary

Transaction management is a key part of ensuring data integrity in almost any type of application. In this chapter, we discussed how to use Spring to manage transactions with almost no impact on your source code. You also learned how to use local and global transactions.

We provided various examples of transaction implementation, including declarative ways of using XML configuration and annotation, as well as the programmatic approach.

Local transactions are supported inside/outside a JEE application server, and only simple configuration is required to enable local transaction support in Spring. However, setting up a global transaction environment involves more work and greatly depends on which JTA provider and corresponding back-end resources your application needs to interact with.

CHAPTER 10

Validation with Type Conversion and Formatting

In an enterprise application, validation is critical. The purpose of validation is to verify that the data being processed fulfills all predefined business requirements as well as ensures the data integrity and usefulness in other layers of the application.

In application development, data validation is always mentioned alongside conversion and formatting. The reason is that the format of the source of data most likely is different from the format being used in the application. For example, in a web application, a user enters information in the web browser front end. When the user saves that data, it is sent to the server (after the local validation has completed). On the server side, a data-binding process is performed, in which the data from the HTTP request is extracted, converted, and bound to corresponding domain objects (for example, a user enters singer information in an HTML form that is then bound to a `Singer` object in the server), based on the formatting rules defined for each attribute (for example, the date format pattern is `yyyy-MM-dd`). When the data binding is complete, validation rules are applied to the domain object to check for any constraint violation. If everything runs fine, the data is persisted, and a success message is displayed to the user. Otherwise, validation error messages are populated and displayed to the user.

In the first part of this chapter, you will learn how Spring provides sophisticated support for type conversion, field formatting, and validation. Specifically, this chapter covers the following topics:

- *The Spring type conversion system and the Formatter service provider interface (SPI)*: We present the generic type conversion system and Formatter SPI. We cover how the new services can be used to replace the previous `PropertyEditor` support and how they convert between any Java types.
- *Validation in Spring*: We discuss how Spring supports domain object validation. First, we provide a short introduction to Spring's own `Validator` interface. Then, we focus on the JSR-349 (Bean Validation) support.

© Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho 2017
I. Cosmina et al., *Pro Spring 5*, https://doi.org/10.1007/978-1-4842-2808-1_10

Dependencies

As in prior chapters, the samples presented in this chapter require some dependencies, which are depicted in the following configuration snippet. One dependency you may notice is `joda-time`. If you are running Java 8, Spring 5 also supports JSR-310, which is the `javax.time` API.

```
//pro-spring-15/build.gradle
ext {
    //spring libs
    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",
      ...
    ]
    ...
}
//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
}
```

Spring Type Conversion System

In Spring 3, a new type conversion system was introduced, providing a powerful way to convert between any Java types within Spring-powered applications. This section shows how this new service can perform the same functionality provided by the previous `PropertyEditor` support, as well as how it supports the conversion between any Java types. We also demonstrate how to implement a custom type converter by using the Converter SPI.

Conversion from a String Using PropertyEditors

Chapter 3 covered how Spring handles the conversion from a `String` in the properties files into the properties of POJOs by supporting `PropertyEditors`. Let's do a quick review here and then cover how Spring's Converter SPI (available since 3.0) provides a more powerful alternative.

Consider this simpler version of the `Singer` class:

```
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;

    //getters and setters
    ...

    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);
    }
}
```

For the `birthDate` attribute, we use JodaTime's `DateTime` class. In addition, there is a `URL` type field that indicates the singer's personal web site, if applicable. Now suppose we want to construct `Singer` objects in Spring's `ApplicationContext`, with values stored either in Spring's configuration file or in a properties file. The following configuration snippet shows the Spring XML configuration file (`prop-editor-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: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>
```

Here we construct two different beans of the `Singer` class. The `eric` bean is constructed with values provided in the configuration file, while for the `countrysinger` bean, the attributes are externalized into a properties file. In addition, a custom editor is defined for converting from a `String` to JodaTime's `DateTime` type, and the date-time format pattern is externalized in the properties file too. The following snippet shows the properties file (`application.properties`):

```
date.format.pattern=yyyy-MM-dd

countrySinger.firstName=John
countrySinger.lastName=Mayer
countrySinger.birthDate=1977-10-16
countrySinger.personalSite=http://johnmayer.com/
```

The following code snippet shows the custom editor for converting `String` values into the JodaTime `DateTime` type:

```
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));
        }
    }
}
```

DateTimeEditorRegistrar implements the PropertyEditorRegister interface to register our custom PropertyEditor. We then create an inner class called DateTimeEditor that handles the conversion of the String to a DateTime. We use an inner class in this sample, as it's accessed by only the PropertyEditorRegistrar implementation. Now let's test it. The next code snippet shows the testing program:

```
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();
    }
}
```

As you can see, the two Singer beans are retrieved from ApplicationContext and printed. Running the program produces the following output:

```
[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/}
```

As shown in the output, the properties are converted and applied to the Singer beans. The reason why XML is used here instead of a Java configuration class is that the values to be injected are declared as text values and Spring does the conversion in the background, transparently.

Introducing Spring Type Conversion

With Spring 3.0, a general type conversion system was introduced, which resides under the package org.springframework.core.convert. In addition to providing an alternative to PropertyEditor support, the type conversion system can be configured to convert between any Java types and POJOs (while PropertyEditor is focused on converting String representations in the properties file into Java types).

Implementing a Custom Converter

To see the type conversion system in action, let's revisit the previous example and use the same Singer class. Suppose this time we want to use the type conversion system to convert the date in String format into the Singer's birthDate property, which is of JodaTime's DateTime type. To support the conversion, instead of creating a custom PropertyEditor, we create a custom converter by implementing the org.springframework.core.convert.converter.Converter<S,T> interface. The following code snippet shows the custom converter:

```
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);
    }
}
```

We implement the interface `Converter<String, DateTime>`, which means the converter is responsible for converting a `String` (the source type `S`) to a `DateTime` type (the target type `T`). The injection of the date-time pattern is optional and can be done by calling the setter `setDatePattern`. If not injected, the default pattern `yyyy-MM-dd` is used. Then, in the initialization method (the `init()` method annotated with `@PostConstruct`), an instance of JodaTime's `DateTimeFormat` class is constructed, which will perform the conversion based on the specified pattern. Finally, the `convert()` method is implemented to provide the conversion logic.

Configuring ConversionService

To use the conversion service instead of `PropertyEditor`, we need to configure an instance of the `org.springframework.core.convert.ConversionService` interface in Spring's `ApplicationContext`. The following code snippet shows the Java configuration class:

```
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;
    }
}
```

The values are read from a property file with contents identical to the file introduced in the previous section and are injected into the created bean using `@Value` annotations.

Here we instruct Spring to use the type conversion system by declaring a `conversionService` bean with the class `ConversionServiceFactoryBean`. If no conversion service bean is defined, Spring will use the `PropertyEditor`-based system.

By default, the type conversion service supports conversion between common types including strings, numbers, enums, collections, maps, and so on. In addition, the conversion from Strings to Java types within the PropertyEditor-based system is supported.

For the conversionService bean, a custom converter is configured for conversion from a String to DateTime. The testing program is shown here:

```
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();
    }
}
```

Running the testing program produces the following output:

```
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/}
```

As you can see, the john bean's property conversion result is the same as when we use PropertyEditors.

Converting Between Arbitrary Types

The real strength of the type conversion system is the ability to convert between arbitrary types. To see it in action, suppose we have another class, called AnotherSinger, that is the same as the Singer class. The code is shown here:

```
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;
```

```
    //seters and  getters
    ...

    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);
    }
}
```

We want to be able to convert any instance of the `Singer` class to the `AnotherSinger` class. When converted, the `firstName` and `lastName` values of `Singer` will become `lastName` and `firstName` of `AnotherSinger`, respectively. Let's implement another custom converter to perform the conversion. The following code snippet shows the custom converter:

```
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;
    }
}
```

The class is simple; just swap the `firstName` and `lastName` property values between the `Singer` and `AnotherSinger` classes. To register the custom converter into `ApplicationContext`, replace the definition of the `conversionService` bean in the `AppConfig` class with the code snippet in the following code snippet:

```
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();
    }
}
```

The order of the beans within the converter property is not important. To test the conversion, we use the following testing program, which is the `MultipleConvServDemo` class shown here:

```
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);
    }
}
```

A handle to the ConversionService interface is obtained from ApplicationContext. Because we already registered ConversionService in ApplicationContext with our custom converters, we can use it to convert the Singer object, as well as convert between other types that the conversion service already supports. As shown in the listing, examples of converting from a String (delimited by a comma character) to an Array and from a List to a Set were also added for demonstration purposes. Running the program produces the following output:

```
[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
```

In the output, you will see that Singer and AnotherSinger are converted correctly, as well as the String to Array and the List to Set. With Spring's type conversion service, you can create custom converters easily and perform conversion at any layer within your application. One possible use case is that you have two systems with the same singer information that you need to update. However, the database structure is different (for example, the last name in system A means the first name in system B, and so on). You can use the type conversion system to convert the objects before persisting to each individual system.

Starting with Spring 3.0, Spring MVC makes heavy use of the conversion service (as well as the Formatter SPI discussed in the next section). In the web application context configuration, the declaration of the tag <mvc:annotation-driven/>, or in the Java configuration class the use of @EnableWebMvc introduced in Spring 3.1, will automatically register all default converters (for example, StringToArrayConverter, StringToBooleanConverter, and StringToLocaleConverter, all residing under the org.springframework.core.convert.support package) and formatters (for example, CurrencyFormatter, DateFormatter, and NumberFormatter, all residing under various subpackages within the org.springframework.format package). More is covered in Chapter 16, when we discuss web application development in Spring.

Field Formatting in Spring

Besides the type conversion system, another great feature that Spring brings to developers is the Formatter SPI. As you might expect, this SPI can help configure the field-formatting aspects.

In the Formatter SPI, the main interface for implementing a formatter is the org.springframework.format.Formatter<T> interface. Spring provides a few implementations of commonly used types, including CurrencyFormatter, DateFormatter, NumberFormatter, and PercentFormatter.

Implementing a Custom Formatter

Implementing a custom formatter is easy too. We will use the same Singer class and implement a custom formatter for converting the DateTime type of the birthDate attribute to and from a String.

However, this time we will take a different approach; we will extend Spring's org.springframework.format.support.FormattingConversionServiceFactoryBean class and provide our custom formatter. The FormattingConversionServiceFactoryBean class is a factory class that provides convenient access to the underlying FormattingConversionService class, which supports the type conversion system, as well as field formatting according to the formatting rules defined for each field type.

The following code snippet shows a custom class that extends the FormattingConversionServiceFactoryBean class, with a custom formatter defined for formatting JodaTime's DateTime type.

```
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);
      }
          };
    }
}
```

In the preceding listing, the custom formatter is underlined. It implements the Formatter<DateTime> interface and implements two methods defined by the interface. The parse() method parses the String format into the DateTime type (the locale was also passed for localization support), while the logger.info() method is to format a DateTime instance into a String. The date pattern can be injected into the bean (or the default will be yyyy-MM-dd). Also, in the init() method, the custom formatter is registered by calling the setFormatters() method. You can add as many formatters as required.

Configuring ConversionServiceFactoryBean

Declaring a bean of type FormattingConversionServiceFactoryBean greatly reduces in size the AppConfig configuration class.

```
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;
    }
}
```

The testing program is shown here:

```
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();
    }
}
```

Running the program produces the following output:

```
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
```

In the output, you can see that Spring uses our custom formatter's `parse()` method to convert the property from a `String` to the `DateTime` type of the `birthDate` attribute. When we call the `ConversionService.convert()` method and pass in the `birthDate` attribute, Spring will call the `logger`'s `info` method to format the output.

Validation in Spring

Validation is a critical part of any application. Validation rules applied on domain objects ensure that all business data is well structured and fulfills all the business definitions. The ideal case is that all validation rules are maintained in a centralized location, and the same set of rules are applied to the same type of data, no matter which source the data comes from (for example, from user input via a web application, from a remote application via web services, from a JMS message, or from a file).

When talking about validation, conversion and formatting are important too, because before a piece of data can be validated, it should be converted to the desired POJO according to the formatting rules defined for each type. For example, a user enters some singer information via a web application within a browser and then submits that data to a server. On the server side, if the web application was developed in Spring MVC, Spring will extract the data from the HTTP request and perform the conversion from a `String` to the

desired type based on the formatting rule (for example, a String representing a date will be converted into a Date field, with the formatting rule yyyy-MM-dd). The process is called *data binding*. When the data binding is complete and the domain object constructed, validation will then be applied to the object, and any errors will be returned and displayed to the user. If validation succeeds, the object will be persisted to the database.

Spring supports two main types of validation. The first one is provided by Spring, within which custom validators can be created by implementing the org.springframework.validation.Validator interface. The other one is via Spring's support of JSR-349 (Bean Validation). We present both of them in the coming sections.

Using the Spring Validator Interface

Using Spring's Validator interface, we can develop some validation logic by creating a class to implement the interface. Let's see how it works. For the Singer class that we've worked with so far, suppose the first name cannot be empty. To validate Singer objects against this rule, we can create a custom validator. The following code snippet shows the validator class:

```
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");
    }
}
```

The validator class implements the Validator interface and implements two methods. The supports() method indicates whether validation of the passed-in class type is supported by the validator. The validate() method performs validation on the passed-in object. The result will be stored in an instance of the org.springframework.validation.Errors interface. In the validate() method, we perform a check only on the firstName attribute and use the convenient ValidationUtils.rejectIfEmpty() method to ensure that the first name of the singer is not empty. The last argument is the error code, which can be used for looking up validation messages from resource bundles to display localized error messages.

The following code snippet depicts the configuration class:

```
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 {
}
```

The following code snippet contains the testing program:

```
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();
    }
}
```

A Singer object is constructed with the first name set to null. Then, the validator is retrieved from ApplicationContext. To store the validation result, an instance of the BeanPropertyBindingResult class is constructed. To perform the validation, the ValidationUtils.invokeValidator() method is called. Then we check for validation errors. Running the program produces the following output:

```
[main] INFO    c.a.p.c.SpringValidatorDemo - No of validation errors: 1
[main] INFO    c.a.p.c.SpringValidatorDemo - firstName.empty
```

The validation produces one error, and the error code is displayed correctly.

Using JSR-349 Bean Validation

As of Spring 4, full support for JSR-349 (Bean Validation) has been implemented. The Bean Validation API defines a set of constraints in the form of Java annotations (for example, @NotNull) under the package javax.validation.constraints that can be applied to the domain objects. In addition, custom validators (for example, class-level validators) can be developed and applied by using annotation.

Using the Bean Validation API frees you from coupling to a specific validation service provider. By using the Bean Validation API, you can use standard annotations and the API for implementing validation logic to your domain objects, without knowing the underlying validation service provider. For example, the Hibernate Validator version 5 (http://hibernate.org/subprojects/validator) is the JSR-349 reference implementation.

Spring provides seamless support for the Bean Validation API. The main features include support for JSR-349 standard annotations for defining validation constraints, custom validators, and configuration of JSR-349 validation within Spring's ApplicationContext. Let's go through them one by one in the following sections. Spring still seamlessly provides support for JSR-303 when using Hibernate Validator version 4 and the 1.0 version of the validation API on your classpath.

Defining Validation Constraints on Object Properties

Let's begin with applying validation constraints to domain object properties. The following code snippet shows a more advanced Singer class with validation constraints applied to the firstName and genre attributes:

```
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;

    //setters and getters
    ...
}
```

Here the validation constraints applied are shown underlined. For the firstName attribute, two constraints are applied. The first one is governed by the @NotNull annotation, which indicates that the value should not be null. Moreover, the @Size annotation governs the length of the firstName attribute. The @NotNull constraint is applied to the genre attribute too.

The following code sample shows the Genre and Gender enum classes, respectively:

```
//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;
   }
}

//Genfer.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;
   }
}
```

The genre indicates the music genre a singer belongs to, while the gender is not really that relevant for a musical career, so it could be null.

Configuring Bean Validation Support in Spring

To configure the support of the Bean Validation API in Spring's ApplicationContext, we define a bean of type org.springframework.validation.beanvalidation.LocalValidatorFactoryBean in Spring's configuration. The following code snippet depicts the configuration class:

```
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();
   }
}
```

The declaration of a bean with the type `LocalValidatorFactoryBean` is all that is required. By default, Spring will search for the existence of the Hibernate Validator library in the classpath. Now, let's create a service class that provides a validation service for the `Singer` class. The validator class is shown here:

```
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);
   }
}
```

An instance of the `javax.validation.Validator` was injected (note the difference from the Spring-provided `Validator` interface, which is `org.springframework.validation.Validator`). Once the `LocalValidatorFactoryBean` is defined, you can create a handle to the `Validator` interface anywhere in your application. To perform validation on a POJO, the `Validator.validate()` method is called. The validation results will be returned as a `List` of the `ConstraintViolation<T>` interface.

The testing program is shown here:

```
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());
      }
   }
}
```

As shown in this listing, a Singer object is constructed with firstName and genre violating the constraints. In the validateSinger() method, the SingerValidationService.validateSinger() method is called, which in turn will invoke JSR-349 (Bean Validation). Running the program produces the following output:

```
[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
```

As you can see, there are two violations, and the messages are shown. In the output, you will also see that Hibernate Validator had already constructed default validation error messages based on the annotation. You can also provide your own validation error message, which we demonstrate in the next section.

Creating a Custom Validator

Besides property-level validation, we can apply class-level validation. For example, for the Singer class, for country singers, we want to make sure that the lastName and gender attributes are not null, not that it really matters, but just for educational purposes. In this case, we can develop a custom validator to perform the check. In the Bean Validation API, developing a custom validator is a two-step process. First create an Annotation type for the validator, as shown in the following code snippet. The second step is to develop the class that implements the validation logic.

```
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 {};
}
```

The @Target(ElementType.TYPE) annotation means that the annotation should be applied only at the class level. The @Constraint annotation indicates that it's a validator, and the validatedBy attribute specifies the class providing the validation logic. Within the body, three attributes are defined (in the form of a method), as follows:

- The message attribute defines the message (or error code) to return when the constraint is violated. A default message can also be provided in the annotation.
- The groups attribute specifies the validation group if applicable. It's possible to assign validators to different groups and perform validation on a specific group.
- The payload attribute specifies additional payload objects (of the class implementing the javax.validation.Payload interface). It allows you to attach additional information to the constraint (for example, a payload object can indicate the severity of a constraint violation).

The following code snippet shows the CountrySingerValidator class that provides the validation logic:

```
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;
    }
}
```

CountrySingerValidator implements the ConstraintValidator<CheckCountrySinger, Singer> interface, which means that the validator checks the CheckCountrySinger annotation on the Singer classes. The isValid() method is implemented, and the underlying validation service provider (for example, Hibernate Validator) will pass the instance under validation to the method. In the method, we verify that if the singer is a country music singer, the lastName and gender properties should not be null. The result is a Boolean value that indicates the validation result.

To enable the validation, apply the `@CheckCountrySinger` annotation to the `Singer` class, as shown here:

```
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;

      //getters and  setter
      ...

      public boolean isCountrySinger() {
      return genre == Genre.COUNTRY;
      }
}
```

To test the custom validation, the following `Singer` instance will be created in the testing class `Jsr349CustomDemo`:

```
public class Jsr349CustomDemo {
      ...

      Singer singer = new Singer();
      singer.setFirstName("John");
      singer.setLastName("Mayer");
      singer.setGenre(Genre.COUNTRY);
      singer.setGender(null);

      validateSinger(singer, singerValidationService);

      ...
}
```

Running the program produces the following output (the other output was omitted):

```
[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
```

In the output, you can see that the value being checked (which is the `Singer` instance) violates the validation rule for country singers, because the `gender` attribute is `null`. Note also that in the output, the property path is empty because it's a class-level validation error.

Using AssertTrue for Custom Validation

Besides implementing a custom validator, another way to apply custom validation in the Bean Validation API is to use the `@AssertTrue` annotation. Let's see how it works. For the `Singer` class, the `@CheckCountrySinger` annotation is removed, and the `isCountrySinger()` method is modified as follows:

```
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;
   }
}
```

As you can probably infer, the `@CheckCountrySinger` annotation and the `CountrySingerValidator` class are no longer necessary.

The `isCountrySinger()` method is added to the `Singer` class and annotated with `@AssertTrue` (which is under the package `javax.validation.constraints`). When invoking validation, the provider will invoke the checking and make sure that the result is true. JSR-349 also provides the `@AssertFalse` annotation to check for some condition that should be false. Now run the testing program `Jsr349AssertTrueDemo`, and you will get the same output as produced by the custom validator.

Considerations for Custom Validation

So, for custom validation in JSR-349, which approach should you use: the custom validator or the `@AssertTrue` annotation? Generally, the `@AssertTrue` method is simpler to implement, and you can see the validation rules right in the code of the domain objects. However, for validators with more complicated logic (for example, say you need to inject a service class, access a database, and check for valid values), then implementing a custom validator is the way to go because you never want to inject service-layer objects into your domain objects. Also, custom validators can be reused across similar domain objects.

Deciding Which Validation API to Use

Having discussed Spring's own `Validator` interface and the Bean Validation API, which one should you use in your application? JSR-349 is definitely the way to go. The following are the major reasons:

- JSR-349 is a JEE standard and is broadly supported by many front-end/back-end frameworks (for example, Spring, JPA 2, Spring MVC, and GWT).
- JSR-349 provides a standard validation API that hides the underlying provider, so you are not tied to a specific provider.
- Spring tightly integrates with JSR-349 starting with version 4. For example, in the Spring MVC web controller, you can annotate the argument in a method with the `@Valid` annotation (under the package `javax.validation`), and Spring will invoke JSR-349 validation automatically during the data-binding process. Moreover, in a Spring MVC web application context configuration, a simple tag called `<mvc:annotation-driven/>` will configure Spring to automatically enable the Spring type conversion system and field formatting, as well as support of JSR-349 (Bean Validation).
- If you are using JPA 2, the provider will automatically perform JSR-349 validation to the entity before persisting, providing another layer of protection.

For detailed information about using JSR-349 (Bean Validation) with Hibernate Validator as the implementation provider, please refer to Hibernate Validator's documentation page: http://docs.jboss.org/hibernate/validator/5.1/reference/en-US/html.

Summary

In this chapter, we covered the Spring type conversion system as well as the field Formatter SPI. You saw how the new type conversion system can be used for arbitrary type conversion, in addition to the `PropertyEditors` support.

We also covered validation support in Spring, Spring's `Validator` interface, and the recommended JSR-349 (Bean Validation) support in Spring.

CHAPTER 11

Task Scheduling

Task scheduling is a common feature in enterprise applications. Task scheduling is composed mainly of three parts: the task (which is the piece of business logic needed to run at a specific time or on a regular basis), the trigger (which specifies the condition under which the task should be executed), and the scheduler (which executes the task based on the information from the trigger). Specifically, this chapter covers the following topics:

- *Task scheduling in Spring*: We discuss how Spring supports task scheduling, focusing on the `TaskScheduler` abstraction introduced in Spring 3. We also cover scheduling scenarios such as fixed-interval scheduling and `cron` expressions.
- *Asynchronous task execution*: We show how to use the `@Async` annotation in Spring to execute tasks asynchronously.
- *Task execution in Spring*: We briefly discuss Spring's `TaskExecutor` interface and how tasks are executed.

Dependencies for the Task Scheduling Samples

You can see the dependencies needed for this chapter in the following Gradle configuration snippet:

```
//pro-spring-15/build.gradle
ext {
   springDataVersion = '2.0.0.M3'

   //logging libs
   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",
         ...
   ]
```

© Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho 2017
I. Cosmina et al., *Pro Spring 5*, https://doi.org/10.1007/978-1-4842-2808-1_11

```
   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",
         ...
   ]
   ...
}
...
//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
}
```

Task Scheduling in Spring

Enterprise applications often need to schedule tasks. In many applications, various tasks (such as sending e-mail notifications to customers, running day-end jobs, doing data housekeeping, and updating data in batches) need to be scheduled to run on a regular basis, either in a fixed interval (for example, every hour) or at a specific schedule (for example, at 8 p.m. every night, from Monday to Friday). As mentioned, task scheduling consists of three parts: the schedule definition (trigger), the task execution (scheduler), and the task itself.

There are many ways to trigger the execution of a task in a Spring application. One way is to trigger a job externally from a scheduling system that already exists in the application deployment environment. For example, many enterprises use commercial systems, such as Control-M or CA AutoSys, for scheduling tasks. If the application is running on a Linux/Unix platform, the `crontab` scheduler can be used. The job triggering can be done by sending a RESTful-WS request to the Spring application and having Spring's MVC controller trigger the task.

Another way is to use the task scheduling support in Spring. Spring provides three options in terms of task scheduling.

- *Support of JDK Timer*: Spring supports JDK's `Timer` object for task scheduling.
- *Integrates with Quartz*: The Quartz Scheduler[1] is a popular open source scheduling library.

[1]You can find the official page at www.quartz-scheduler.org.

- *Spring's own Spring TaskScheduler abstraction*: Spring 3 introduces the `TaskScheduler` abstraction, which provides a simple way to schedule tasks and supports most typical requirements.

This section focuses on using Spring's `TaskScheduler` abstraction for task scheduling.

Introducing the Spring TaskScheduler Abstraction

Spring's `TaskScheduler` abstraction has mainly three participants.

- *The Trigger interface*: The `org.springframework.scheduling.Trigger` interface provides support for defining the triggering mechanism. Spring provides two `Trigger` implementations. The `CronTrigger` class supports triggering based on a `cron` expression, while the `PeriodicTrigger` class supports triggering based on an initial delay and then a fixed interval.
- *The task*: The task is the piece of business logic that needs to be scheduled. In Spring, a task can be specified as a method within any Spring bean.
- *The TaskScheduler interface*: The `org.springframework.scheduling.TaskScheduler` interface provides support for task scheduling. Spring provides three implementation classes of the `TaskScheduler` interface. The `TimerManagerTaskScheduler` class (in package `org.springframework. scheduling.commonj`) wraps CommonJ's `commonj.timers.TimerManager` interface, which is commonly used in commercial JEE application servers such as WebSphere and WebLogic. The `ConcurrentTaskScheduler` and `ThreadPoolTaskScheduler` classes (both under the package `org.springframework.scheduling.concurrent`) wrap the `java.util.concurrent.ScheduledThreadPoolExecutor` class. Both classes support task execution from a shared thread pool.

Figure 11-1 shows the relationships between the `Trigger` interface, the `TaskScheduler` interface, and the task implementation that implements the `java.lang.Runnable` interface. To schedule tasks by using Spring's `TaskScheduler` abstraction, you have two options. One is to use `task-namespace` in Spring's XML configuration, and the other is to use annotations. Let's go through each of them.

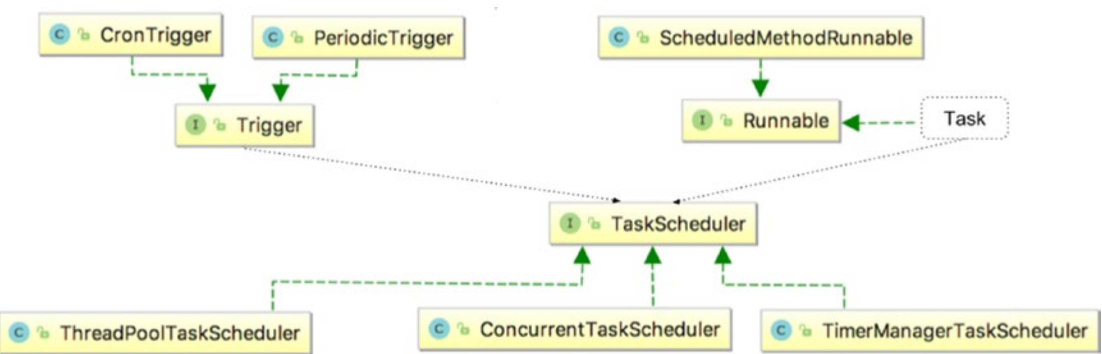

***Figure 11-1.** Relationship between trigger, task, and scheduler*

Exploring a Sample Task

To demonstrate task scheduling in Spring, let's implement a simple job first, namely, an application maintaining a database of car information. The following code snippet shows the Car class, which is implemented as a JPA entity class:

```
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;

    //getters and setters
    ...
```

```
     @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);
    }
}
```

This entity class is used as a model for the CAR table generated by Hibernate. The configuration for the data access and services layer, all merged into one for this chapter, is provided by the DataServiceConfig class depicted in the following code snippet:

```
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();
    }
}
```

A class named `DBInitializer` is responsible for populating the `CAR` table.

```
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.");
    }
}
```

Let's define a DAO layer for the `Car` entity. We will use Spring Data's JPA and its repository abstraction support. Here you can see the `CarRepository` interface, which is a simple extension of `CrusRepository` because as we are not interested in any special DAO operations.

```
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> {
}
```

The service layer is represented by the CarService interface and its implementation CarServiceImpl.

```
package com.apress.prospring5.ch11.services;
//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();
}

//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;
    }
}
```

Four methods were provided, as shown here:

- One retrieves the information about all cars: `List<Car> findAll()`.
- One persists an updated `Car` object: `Car save(Car car)`.
- The third method, `void updateCarAgeJob()`, is the job that needs to be run regularly to update the age of the car based on the manufacture date of the car and the current date.
- The fourth method, `boolean isDone()`, is a utility method designed to be used to know when the job ended, so the application can be shut down gracefully.

Like the support for other namespaces in Spring, `task-namespace` provides a simplified configuration for scheduling tasks by using Spring's `TaskScheduler` abstraction. The following XML configuration snippet shows the contents of the `task-namespace-app-context.xml` file and shows the configuration for a Spring application that contains scheduled tasks. Using `task-namespace` for task scheduling is very simple.

```
<?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>
```

When it encounters the `<task:scheduler>` tag, Spring instantiates an instance of the `ThreadPoolTaskScheduler` class, while the attribute `pool-size` specifies the size of the thread pool that the scheduler can use. Within the `<task:scheduled-tasks>` tag, one or more tasks can be scheduled. In the `<task:scheduled>` tag, a task can reference a Spring bean (the `carService` bean, in this case) and a specific method within the bean (in this case, the `updateCarAgeJob()` method). The attribute `fixed-delay` instructs Spring to instantiate `PeriodicTrigger` as the `Trigger` implementation for `TaskScheduler`.

The task scheduling configuration is combined with the data access configuration by declaring a new configuration class and importing the two configurations using `@Import` for configuration classes and `@ImportResource` for XML configurations.

```
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 {}
```

This configuration class, the AppConfig class, is used to create a Spring ApplicationContext to test the Spring scheduling capabilities:

```
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();
    }
}
```

Running the program produces the following batch job output:

```
[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
```

In the previous example, the application is stopped after only one run of the scheduled task. As we've declared we want the task to run every 10 seconds, by setting the `fixed-delay="10000"` attribute; we should allow the repeated run of the task by letting the application run until the user presses a key. Modify `ScheduleTaskDemo` as follows:

```
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();
    }
}
```

From the output, you can see that the cars' age attributes are updated. Besides a fixed interval, a more flexible scheduling mechanism is to use a `cron` expression. In the XML configuration file, change the line from this:

```
<task:scheduled ref="carService" method="updateCarAgeJob" fixed-delay="10000"/>
```

to the following:

```
<task:scheduled ref="carService" method="updateCarAgeJob" cron="0 * * * * *"/>
```

After the change, run the `ScheduleTaskDemo` class again, and let the application run for more than a minute. You will see that the job will run every minute.

Using Annotations for Task Scheduling

Another option for scheduling tasks with Spring's `TaskScheduler` abstraction is to use an annotation. Spring provides the `@Scheduled` annotation for this purpose. To enable annotation support for task scheduling, we need to provide the `<task:annotation-driven>` tag in Spring's XML configuration. Or, if a configuration class is used, it has to be annotated with `@EnableScheduling`. Let's go for this approach and remove the XML altogether.

```
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 {
}
```

Yep, that is all that is needed. You do not even need to declare the scheduler yourself anymore, because Spring will take care of it. The @EnableScheduling annotation used on a @Configuration class enables detection of @Scheduled annotations on any Spring-managed bean in the container or their methods. The fun part is that methods annotated with @Scheduled may even be declared directly within @Configuration classes. This annotation tell Spring to look for an associated scheduler definition: either a unique TaskScheduler bean in the context or a TaskScheduler bean named taskScheduler or a ScheduledExecutorService bean. If none is found, a local single-threaded default scheduler will be created and used within the registrar.

To schedule a specific method in a Spring bean, the method must be annotated with @Scheduled and pass in the scheduling requirements. In the following code snippet, the CarServiceImpl class was extended and used to declare a new bean with a scheduled method, which overrides the updateCarAgeJob() method in the parent class to make use of the @Scheduled annotation:

```
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");
    }
}
```

The testing program is shown here:

```
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();
    }
}
```

Running the program produces almost the same output as using `task-namespace`. You can try different triggering mechanisms by changing the attribute within the `@Scheduled` annotation (that is, `fixedDelay`, `fixedRate`, `cron`). Feel free to test it yourself.

```
[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
... // more stacktrace here
[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
... // more stacktrace here
[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
```

Also, you can define your own TaskScheduler bean if you want. The following example declares a ThreadPoolTaskScheduler bean that is equivalent to the one declared in the XML configuration from the previous section:

```
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;
    }
}
```

If you run the testing example now, you will see that the exceptions are not printed in the log anymore and that the name of the scheduler executing the method has changed because the TaskScheduler bean is named 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
```

Asynchronous Task Execution in Spring

Since version 3.0, Spring also supports using annotations to execute a task asynchronously. To do this, you just need to annotate the method with `@Async`.

```
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);
    }
}
```

`AsyncService` defines two methods. The `asyncTask()` method is a simple task that logs information to the logger. The method `asyncWithReturn()` accepts a `String` argument and returns an instance of the `java.util.concurrent.Future<V>` interface. Upon completion of `asyncWithReturn()`, the result is

stored in an instance of the `org.springframework.scheduling.annotation.AsyncResult<V>` class, which implements the `Future<V>` interface and can be used by the caller to retrieve the result of the execution later. The `@Async` annotation is picked up by enabling Spring's asynchronous method execution capability, and that is done by annotating a Java configuration class wth `@EnableAsync`.[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
@EnableAsync
@ComponentScan(basePackages = {"com.apress.prospring5.ch11"} )
public class AppConfig {
}
```

The testing program is shown here:

```
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();
        }
```

[2]In XML just declaring `<task:annotation-driven />` will do it.

```
        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();
    }
}
```

We call the asyncTask() method five times and then asyncWithReturn() three times with different arguments and then retrieve the result after sleeping for six seconds. Running the program produces the following output:

```
...
17:55:31.851 [SimpleAsyncTaskExecutor-1] INFO  c.a.p.c.AsyncServiceImpl -
  Start execution of async. task
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
17:55:31.852 [SimpleAsyncTaskExecutor-7] INFO  c.a.p.c.AsyncServiceImpl -
  Start execution of async. task with return for Eric Clapton
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
17:55:41.852 [SimpleAsyncTaskExecutor-4] INFO  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
```

From the output, you can see that all the calls were started at the same time. The three calling with return values complete first and are displayed on the console output. Finally, the five `asyncTask()` methods called are completed too.

Task Execution in Spring

Since Spring 2.0, the framework provided an abstraction for executing tasks by way of the `TaskExecutor` interface. A `TaskExecutor` does as exactly as it sounds: it executes a task represented by a Java `Runnable` implementation. Out of the box, Spring provides a number of `TaskExecutor` implementations suited for different needs. You can find a full list of TaskExecutor implementations at `http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/core/task/TaskExecutor.html`.

Some commonly used `TaskExecutor` implementations are listed here:

- `SimpleAsyncTaskExecutor`: Creates new threads on each invocation; does not reuse existing threads
- `SyncTaskExecutor`: Does not execute asynchronously; invocation occurs in the calling thread
- `SimpleThreadPoolTaskExecutor`: Subclass of Quartz's `SimpleThreadPool`; used when you need to share a thread pool by both Quartz and non-Quartz components
- `ThreadPoolTaskExecutor`: `TaskExecutor` implementation providing the ability to configure `ThreadPoolExecutor` via bean properties and expose it as a Spring `TaskExecutor`

Each `TaskExecutor` implementation serves its own purpose, and the calling convention is the same. The only variation is in the configuration, when defining which `TaskExecutor` implementation you want to use and its properties, if any. Let's take a look at a simple example that prints out a number of messages. The `TaskExecutor` implementation that we will use is `SimpleAsyncTaskExecutor`. First let's create a bean class that holds the task execution logic, as shown here:

```
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()));
        }
    }
}
```

This class is just a regular bean that needs `TaskExecutor` to be injected as a dependency and defines a method `executeTask()`. The `executeTask()` method calls the `execute` method of the provided `TaskExecutor` by creating a new `Runnable` instance containing the logic we would like want to execute for this task. This might not be obvious here as a lambda expression is used to create the `Runnable` instance. The configuration is quite simple; it is similar to the configuration depicted in the previous section. The only thing we have to take into account here is that we need to provide a declaration for a `TaxExecutor` bean, which needs to be injected in the `TaskToExecute` bean.

```
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();
    }
}
```

A simple bean called `taskExecutor` of type `SimpleAsyncTaskExecutor` is declared in the previous configuration. That bean is injected by the Spring IoC container into the `TaskToExecute` bean. To test the execution, you can use the following program:

```
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();
    }
}
```

When the example is run, it should print output similar to the following:

```
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
```

As you can see from the output, each task (the message we are printing) is displayed as it's executed. We print out the message plus the thread name, which is by default the class name SimpleAsyncTaskExecutor, and the thread number.

Summary

In this chapter, we covered Spring's support for task scheduling. We focused on Spring's built-in TaskScheduler abstraction and demonstrated how to use it to fulfill task scheduling needs with a sample batch data update job. We also covered how Spring supports annotation for executing tasks asynchronously. Additionally, we briefly covered Spring's TaskExecutor and common implementations.

A Spring Boot section was not needed as the scheduling and asynchronous execution of tasks annotations are part of the spring-context library and they have to be used with a Spring Boot configuration as well. Plus, configuring scheduled and asynchronous tasks is already as easy as it can be with Spring; there is not much Spring Boot could do to improve on this topic.[3]

[3]But if you are curios and want to convert the provided project to Spring Boot, you can find a small tutorial here: https://spring.io/guides/gs/scheduling-tasks/

CHAPTER 12

Using Spring Remoting

An enterprise application typically needs to communicate with other applications. Take, for example, a company selling products; when a customer places an order, an order-processing system processes that order and generates a transaction. During order processing, an inquiry is made to the inventory system to check whether the product is in stock. Upon order confirmation, a notification is sent to the fulfillment system to deliver the product to the customer. Finally, the information is sent to the accounting system, an invoice is generated, and the payment is processed.

Most of the time, this business process is not fulfilled by a single application but by a number of applications working together. Some of the applications may be developed in-house, and others may be purchased from external vendors. Moreover, the applications may be running on different machines in different locations and implemented with different technologies and programming languages (for example, Java, .NET, or C++). Performing the handshaking between applications in order to build an efficient business process is always a critical task when architecting and implementing an application. As a result, remoting support via various protocols and technologies is needed for an application to participate well in an enterprise environment.

In the Java world, remoting support has existed since Java was first created. In the early days (Java 1.*x*), most remoting requirements were implemented by using traditional TCP sockets or Java Remote Method Invocation (RMI). After J2EE came on the scene, EJB and JMS became common choices for interapplication server communications. The rapid evolution of XML and the Internet gave rise to remote support using XML over HTTP, including the Java API for XML-based RPC (JAX-RPC), the Java API for XML Web Services (JAX-WS), and HTTP-based technologies (for example, Hessian and Burlap). Spring also offers its own HTTP-based remoting support, called the Spring HTTP invoker. In recent years, to cope with the explosive growth of the Internet and more responsive web application requirements (for example, via Ajax), more lightweight and efficient remoting support of applications has become critical for the success of an enterprise. Consequently, the Java API for RESTful Web Services (JAX-RS) was created and quickly gained popularity. Other protocols, such as Comet and HTML5 WebSocket, also attracted a lot of developers. Needless to say, remoting technologies keep evolving at a rapid pace.

In terms of remoting, as mentioned, Spring provides its own support (via the Spring HTTP invoker), as well as supporting a lot of technologies mentioned earlier (for example, RMI, EJB, JMS, Hessian, Burlap,[1] JAX-RPC, JAX- WS, and JAX-RS). It's not possible to cover all of them in this chapter. So, here we focus on those that are most commonly used. Specifically, this chapter covers the following topics:

- *Spring HTTP invoker*: If both applications that need to communicate are Spring based, the Spring HTTP invoker provides a simple and efficient way to invoke the services exposed by other applications. We show you how to use the Spring HTTP invoker to expose a service within its service layer, as well as invoking the services provided by a remote application.

[1]When Spring 4 was released, it was mentioned that Burlap is no longer under active development and support will be dropped entirely in the future.

© Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho 2017

I. Cosmina et al., *Pro Spring 5*, https://doi.org/10.1007/978-1-4842-2808-1_12

- *Using JMS in Spring*: The Java Message Service (JMS) provides another asynchronous and loosely coupled way of exchanging messages between applications. We show you how Spring simplifies application development with JMS.
- *Using RESTful web services in Spring*: Designed specifically around HTTP, RESTful web services are the most commonly used technology for providing remote support for an application, as well as supporting highly interactive web application front ends using Ajax. We show how Spring MVC provides comprehensive support for exposing services using JAX-RS and how to invoke services by using the `RestTemplate` class. We also discuss how to secure the services from unauthorized access.
- *Using AMQP in Spring*: The sister project Spring Advanced Message Queuing Protocol (AMQP) provides a typical Spring-like abstraction around AMQP along with a RabbitMQ implementation. This project offers a rich set of capabilities, but in this chapter we focus on its remoting capabilities through the projects RPC supports.

Using a Data Model for Samples

In the samples in this chapter, we will use a simple data model, which contains only the `SINGER` table for storing information. The table is generated by Hibernate, based on the `Singer` class shown next. The class and its properties are decorated with standard JPA annotations.

```
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;

    //setters and getters
    ...
}
```

To populate the table, you use an initializer bean. The class is shown here:

```
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.");
    }
}
```

Adding Required Dependencies for the JPA Back End

We need to add the required dependencies to the project. The following configuration snippet shows the dependencies required for implementing a service layer with JPA 2 and Hibernate as the persistence provider. Also, Spring Data JPA will be used.

```
//pro-spring-15/build.gradle
ext {
    //spring libs
    springVersion = '5.0.0.RC1'
    springDataVersion = '2.0.0.M3'

    //logging libs
    slf4jVersion = '1.7.25'
    logbackVersion = '1.2.3'

    junitVersion = '4.12'

    //database library
    h2Version = '1.4.194'

    //persistency libraries
    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"
    ]
}

//chapter12/spring-invoker/build.gradle
dependencies {
    //we specify these dependencies for all submodules, except
    // the boot module, that defines its own
    if (!project.name.contains("boot")) {
        //we exclude transitive dependencies, because spring-data
        //will take care of these
           compile (spring.contextSupport) {
           exclude  module: 'spring-context'
           exclude  module: 'spring-beans'
           exclude  module: 'spring-core'
        }
        //we exclude the 'hibernate' transitive dependency
        //to have control over the version used
        compile (hibernate.tx) {
       exclude group: 'org.hibernate', module: 'hibernate'
    }
        compile misc.slf4jJcl, misc.logback, misc.lang3,
           hibernate.em, misc.guava
     }
     testCompile testing.junit
}
```

Implementing and Configuring SingerService

With the dependencies outlined, we start showing how to implement and configure the service layer for the samples in this chapter. In the following sections, we discuss the implementation of SingerService using JPA 2, Spring Data JPA, and Hibernate as the persistence service provider. Then, we cover how to configure the service layer in the Spring project.

Implementing SingerService

In the samples, we show how to expose the services for various operations on the singer information to remote clients; the SingerService interface is shown here:

```
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);
}
```

The methods should be self-explanatory. Because we will use Spring Data JPA's repository support, we implement the SingerRepository interface, as shown here:

```
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);
}
```

By extending the CrudRepository<T,ID extends Serializable> interface, for the methods in SingerService, we need to explicitly declare only the findByFirstName() method.

The next code snippet shows the implementation class of the SingerService interface:

```
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);
    }
}
```

The implementation is basically completed, and the next step is to configure the service in Spring's `ApplicationContext` within the web project, which is discussed in the next section.

Configuring SingerService

For the data access and transactions, you use a simple Java configuration class, as introduced before and shown here:

```
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();
    }
}
```

Since we are exposing the HTTP invoker via Spring MVC, we need to configure the web application. To configure a Spring Web MVC application without using any XML, two configuration classes are needed, as listed here:

- One configuration class implements the `WebMvcConfigurer` interface. This interface was introduced in Spring 3.1 and defines callback methods to customize the Java-based configuration for Spring MVC enabled using `@EnableWebMvc`. Because we only need to expose an HTTP service (a web interface is not necessary), an empty implementation is sufficient in this case. An implementation of this interface that is annotated with `@EnableWebMvc` replaces a Spring XML configuration using the `mvc` namespace. A more complex sample configuration will be covered in detail in Chapter 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 {

}
```

- The other configuration class implements `org.springframework.web.WebApplicationInitializer` or extends one of the out-of-the-box Spring implementations. This interface needs to be implemented in Spring 3.0+ environments to configure the `ServletContext` programmatically. This removes the necessity of providing a `web.xml` file to configure a web application. This class imports the configuration for data access and transactions and creates the root application context based on it. The web application context is created using the `WebConfig` class and the configuration class that defines the configuration for the HTTP invoker service. These classes can also be combined into one, but good practice when using Spring is to keep custom services and infrastructure beans in different classes.

```
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/*"};
    }
}
```

As this is a Spring MVC web application, a WAR file needs to be created and deployed to a servlet container. There are multiple ways to do this, for example, a stand-alone container such as Tomcat, an IDE-launched instance of Tomcat, or an embedded Tomcat instance that runs with a build tool such as Maven. Which option you choose is up to your needs, but for a local development environment, an embedded instance launched from your build tool or directly from your IDE is recommended. In the code for this book, we use Tomcat Server version 9.*x*, and a launcher is set up in Intellij IDEA to start the web application. See the book source code for more details. At this point, you should build the web application and deploy via the method of your choice. If you try loading the `http://localhost:8080/` URL in the browser, you will see the following message:

```
Spring Remoting: Simplifying Development of Distributed Applications

RMI services over HTTP should be correctly exposed when this page is visible.
```

This means the web application has been deployed correctly, and now the `singerService` bean should be accessible at the `http://localhost:8080/invoker/httpInvoker/singerService` URL.

Exposing the Service

If the application you are going to communicate with is also Spring-powered, using the Spring HTTP invoker is a good choice. It provides an extremely simple way to expose the services within the Spring `WebApplicationContext` to remote clients also using the Spring HTTP invoker to invoke the service. The procedures for exposing and accessing the service are elaborated in the following sections. The `HttpInvokerConfig` class contains a single bean used to expose the HTTP invoker service.

```
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;
    }

}
```

An `httpInvokerServiceExporter` bean is defined with the `HttpInvokerServiceExporter` class, which is for exporting any Spring bean as a service via the HTTP invoker. Within the bean, two properties are defined. The first one is the `service` property, indicating the bean providing the service. For this property, the `singerService` bean is injected. The second property is the interface type to expose, which is the `com.apress.prospring5.ch12.serviced.SingerService` interface.

Now, the service layer is completed and ready to be exposed and used by remote clients.

Invoking the Service

Invoking a service via the Spring HTTP invoker is simple. First we configure a Spring ApplicationContext, as shown in the following configuration class:

```
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();
    }
}
```

As shown previously for the client side, a bean of type HttpInvokerProxyFactoryBean is declared. Two properties are set. serviceUrl specifies the location of the remote service, which is http://localhost:8080/invoker/httpInvoker/singerService. The second property is the interface of the service (that is, SingerService). If you are developing another project for the client, you need to have the SingerService interface and the Singer entity class within your client application's classpath.

The following code snippet shows a test class for invoking the remote service. We are using a test class that uses the RmiClientConfig class to create a test context. The SpringRunner class is an alias for SpringJUnit4ClassRunner that is needed to run Junit tests in a Spring context. You will learn more about this in Chapter 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()));
    }
}
```

The test class should be executed after deploying the web application. The tests should pass and list the Singer instances returned by the singerService bean. The expected output is shown here:

```
//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 - all singers with 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
```

Using JMS in Spring

Using message-oriented middleware (generally referred to as an MQ server) is another popular way to support communication between applications. The main benefits of a message queue (MQ) server are that it provides an asynchronous and loosely coupled way for application integration. In the Java world, JMS is the standard for connecting to an MQ server for sending or receiving messages. An MQ server maintains a list of queues and topics for which applications can connect to and send and receive messages. The following is a brief description of the difference between a queue and a topic:

- *Queue*: A queue is used to support a point-to-point message exchange model. When a producer sends a message to a queue, the MQ server keeps the message within the queue and delivers it to one, and only one, consumer the next time the consumer connects.
- *Topic*: A topic is used to support the publish-subscribe model. Any number of clients can subscribe to the message within a topic. When a message arrives for that topic, the MQ server delivers it to all clients that have subscribed to the message. This model is particularly useful when you have multiple applications that will be interested in the same piece of information (for example, a news feed).

In JMS, a producer connects to an MQ server and sends a message to a queue or topic. A consumer also connects to the MQ server and listens to a queue or topics for messages of interest. In JMS 1.1, the API was unified, so the producer and consumer didn't need to deal with different APIs for interacting with queues and topics. In this section, we focus on the point-to-point style for using queues, which is a more commonly used pattern within an enterprise.

As of Spring Framework 4.0, support for JMS 2.0 has been implemented. JMS 2.0 functionality is usable simply by having the JMS 2.0 JAR in your classpath while retaining backward compatibility for 1.*x*. As of Spring Framework 5.0, JMS 1.1 is no longer supported; thus, in this book the examples will be relevant only to JMS 2.*x*.

At the time of this writing, ActiveMQ does not support JMS 2.0; thus, we will utilize HornetQ (containing JMS 2.0 support starting with 2.4.0.Final) as the message broker in this sample and will use a stand-alone server. Downloading and installing HornetQ is outside the scope of this book; please refer to the documentation at http://docs.jboss.org/hornetq/2.4.0.Final/docs/quickstart-guide/html/index.html.[2]

Several new dependencies are required, and the Gradle configurations needed are shown here:

```
//pro-spring-15/build.gradle
ext {
   jmsVersion = '2.0'
   hornetqVersion = '2.4.0.Final'

   spring = [
    ...
    jms            : "org.springframework:spring-jms:$springVersion"
   ]
```

[2]If you prefer Apache products, Apache ActiveMQ Artemis is a JMS 2.0 implementation with a nonblocking architecture. It delivers outstanding performance. You can find more details at https://activemq.apache.org/artemis/. Also, a project using Artemis ActiveMQ is provided in the code samples.

```
   misc = [
      ...
      Hornetq       : "org.hornetq:hornetq-jms-client:$hornetqVersion"
   ]
...
}
//chapter12/jms-hornetq/build.gradle
dependencies {
    compile spring.jms, misc.jms
}
```

After installing the server, we need to create a queue within the HornetQ JMS configuration file. This file resides under the directory that you extracted HornetQ in. The location of the file is `config/stand-alone/non-clustered hornetq-jms.xml`, and we need to add the queue definition, as shown here:

```
<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>
```

Now, start the HornetQ server by running the `run.sh` script (depending on your operating system) and ensure that the server starts without any errors. Just make sure you see something similar in the log, with no exceptions, and make sure the underlined line is there.

```
...
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:
// the queue configured in the previous configuration sample
     trying to deploy queue jms.queue.prospring5
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
```

Now a Spring configuration must be provided to connect to this server and access the prospring5 queue configured previously. Usually, there should be two configuration classes, one for the message sender and one for the message listener, but because Spring JMS configuration using configuration classes is really practical and not many beans are needed, we put all the configuration in a single class, as shown here:

```
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
@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;
    }
}
```

The `javax.jms.ConnectionFactory` interface implementation is provided by the HornetQ Java library (the `HornetQJMSConnectionFactory` class) and is used to create a connection with a JMS provider. Then, a bean of type `JmsListenerContainerFactory` is declared, which will create message listener containers that use plain JMS client APIs to receive JMS messages. The `jmsTemplate` bean will be used to send JMS messages to the `prospring5` queue.

To receive JMS messages, a message listener component must be declared, providing the destination (that is, the `prospring5` queue) and the JMS container factory `jmsListenerContainerFactory`.

Implementing a JMS Listener in Spring

Before Spring 4.1, to develop a message listener, we needed to create a class that implements the `javax.jms.MessageListener` interface and implements its `onMessage()` method. In Spring 4.1 the `@JmsListener` annotation was added. This annotation is used on bean methods to mark them to be the target of a JMS message listener on the specified destination (queue or topic). The following code snippet depicts the `SimpleMessageListener` class and bean declaration:

```
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;

@Component("messageListener")
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);
        }
    }
}
```

The method is named onMessage here to make its purpose obvious. In onMessage(), which is annotated with the @JmsListener method, an instance of the javax.jms.Message interface will be passed upon message arrival. Within the method, the message is cast to an instance of the javax.jms.TextMessage interface, and the message body in text is retrieved using the TextMessage.getText() method. For a list of possible message formats, please refer to the current JEE online documentation.

Processing of the @JmsListener annotation is done by using @EnableJms on the configuration class or by using the equivalent XML element declaration, which is <jms:annotation-driven/>.

Now let's see how to send a message to the propring5 queue.

Sending JMS Messages in Spring

Let's see how to send messages by using JMS in Spring. We will use the handy bean jmsTemplate of type org.springframework.jms.core.JmsTemplate for this purpose. First we will develop a MessageSender interface and its implementation class, SimpleMessageSender. The following code snippet shows the interface and the class, respectively:

```
//MessageSender.java
package com.apress.prospring5.ch12;

public interface MessageSender {
    void sendMessage(String message);
}

//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;
            }
        });
    }
}
```

As you can see, an instance of `JmsTemplate` is injected. In the `sendMessage()` method, we call the `JmsTemplate.send()` method, with an in-place construction of an instance of the `org.springframework.jms.core.MessageCreator` interface. In the `MessageCreator` instance, the `createMessage()` method is implemented to create a new instance of `TextMessage` that will be sent to HornetQ.

Both message listener and sender bean declarations are picked up using component scanning.

Now let's tie both sending and receiving together to see JMS in action. The following code snippet shows the main testing program for sending messages and receiving messages:

```
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();
    }
}
```

The program is simple. Running the program sends messages to the queue. The `SimpleMessageListener` class receives those messages, and you can see the following output in your console:

```
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
```

Spring Boot Artemis Starter

The possibility of using Spring Boot to make development of JMS application more practical was hinted in Chapter 9 where a distributed transaction example involving a database and a queue was covered.

Spring Boot can autoconfigure a `javax.jms.ConnectionFactory` bean when it detects Artemis is available in the classpath. Am embedded JMS broker is started and configured automatically. There are multiple modes that Artemis can be used in, and it is configurable using special Artemis properties that can be set in the `application.properties` file.

Artemis can be used in a `native` mode, with the connection to the broker being provided by the `netty` protocol. The `application.properties` file can look something like this:

```
spring.artemis.mode=native
spring.artemis.host=0.0.0.0
spring.artemis.port=61617
spring.artemis.user=prospring5
spring.artemis.password=prospring5
```

The easiest way to use Spring Boot and Artemis to create a JMS application is to use an embedded server; all that is needed is a name for the queue that will hold the messages. Thus, the `application.properties` file looks like this:

```
spring.artemis.mode=embedded
spring.artemis.embedded.queues=prospring5
```

This is the approach that will be used in the sources for this section because it requires the least configuration customization. Using this approach, all that is needed is the `application.properties` configuration file and the `Application` class, as shown here:

```
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);
        }
```

```
        System.in.read();
        ctx.close();
    }

    @JmsListener(destination = "prospring5", containerFactory = "connectionFactory")
    public void onMessage(Message message) {
        TextMessage textMessage = (TextMessage) message;

        try {
            logger.info(">>> Received: " + textMessage.getText());
        } catch (JMSException ex) {
            logger.error("JMS error", ex);
        }
    }
}
```

Of course, to make this work, the Spring Boot JMS starter library must be used as a dependency, and the Artemis server must be in the classpath. The Gradle configuration is shown here:

```
//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"

    ]
...
}
//boot-jms/build.gradle
buildscript {
    repositories {
        ...
     }

    dependencies {
        classpath boot.springBootPlugin
    }
}
```

```
apply plugin: 'org.springframework.boot'

dependencies {
    compile boot.starterJms, misc.artemisServer
}
```

Having the `spring-boot-starter-artemis` declared as a dependency removes the need to use `@EnableJms` to process methods annotated with `@JmsListener`. The `jmsTemplate` bean is created by Spring Boot with the default configuration provided by properties set in the `application.properties` file and can be used to not only send but receive messages as well, using the `receive()` method, but this is done synchronously, which means the `jmsTemplate` will block. That is why an explicitly configured `JmsListenerContainerFactory` bean is used to create a `DefaultMessageListenerContainer` that will consume messages asynchronously and with maximum connection efficiency.

If you run the `Application` class, you will get some output in the console, which is really similar to the HornetQ output.

Using RESTful-WS in Spring

Nowadays, RESTful-WS is perhaps the most widely used technology for remote access. From remote service invocation via HTTP to supporting an Ajax-style interactive web front end, RESTful-WS is being adopted intensively. RESTful web services are popular for several reasons.

- *Easy to understand*: RESTful web services are designed around HTTP. The URL, together with the HTTP method, specifies the intention of the request. For example, the URL `http://somedomain.com/restful/customer/1` with an HTTP method of GET means that the client wants to retrieve the customer information, where the customer ID equals 1.
- *Lightweight*: RESTful is much more lightweight when compared to SOAP-based web services, which include a large amount of metadata to describe which service the client wants to invoke. For a RESTful request and response, it's simply an HTTP request and response, as with any other web application.
- *Firewall friendly*: Because RESTful web services are designed to be accessible via HTTP (or HTTPS), the application becomes much more firewall friendly and easily accessed by remote clients.

In this section, we present the basic concepts of RESTful-WS and Spring's support of RESTful-WS through its Spring MVC module.

Introducing RESTful Web Services

The REST in RESTful-WS is short for REpresentational State Transfer, which is an architectural style. REST defines a set of architectural constraints that together describe a uniform interface for accessing resources. The main concepts of this uniform interface include the identification of resources and the manipulation of resources through representations. For the identification of resources, a piece of information should be accessible via a uniform resource identifier (URI). For example, the URL `http://somedomain.com/api/singer/1` is a URI that represents a resource, which is a piece of singer information with an identifier of 1. If the singer with an identifier of 1 does not exist, the client will get a 404 HTTP error, just like a "page not found" error on a web site. Another example, `http://somedomain.com/api/singers`, is a URI that represents a resource that is a list of singer information. Those identifiable resources can be managed through various representations, as shown in Table 12-1.

***Table 12-1.** Representations for Manipulating Resources*

HTTP Method	Description
GET	GET retrieves a representation of a resource.
HEAD	Identical to GET, without the response body. Typically used for getting a header.
POST	POST creates a new resource.
PUT	PUT updates a resource.
DELETE	DELETE deletes a resource.
OPTIONS	OPTIONS retrieves allowed HTTP methods.

For a detailed description of RESTful web services, we recommend *Ajax and REST Recipes: A Problem-Solution Approach* by Christian Gross (Apress, 2006).

Adding Required Dependencies for Samples

To build a Spring REST application, we need to add a few new dependencies. Because we will be sending objects from a server to a client, we need a library to serialize and deserialize them. And we also want to show you that multiple types of serialization can be used in the same application, in this case XML and JSON, so libraries for each of them are needed. Table 12-2 lists the dependencies and their purpose.

***Table 12-2.** Dependencies for RESTful Web Services*

GroupId:ModuleId	Version	Purpose
`org.springframework:spring-oxm`	5.0.0.RC1	Spring object-to-XML mapping module.
`org.codehaus.jackson:jacksonDatabind`	2.9.0.pr3	Jackson JSON processor to support data in JSON format.
`org.codehaus.castor:castor-xml`	1.4.1	The Castor XML library will be used for the marshalling and unmarshalling of XML data.
`org.apache.httpcomponents:httpclient`	4.5.3	Apache HTTP Components project. The HTTP client library will be used for RESTful-WS invocation.

Designing the Singer RESTful Web Service

When developing a RESTful-WS application, the first step is to design the service structure, which includes what HTTP methods will be supported, together with the target URLs for different operations. For the singer RESTful web services, we want to support query, create, update, and delete operations. For querying, we want to support retrieving all singers or a single singer by ID.

The services will be implemented as a Spring MVC controller. The name is the `SingerController` class, under the package `com.apress.prospring5.ch12`. Table 12-3 shows the URL pattern, HTTP method, description, and corresponding controller methods. For the URLs, all will be relative to `http://localhost:8080`. In terms of data format, both XML and JSON are supported. The corresponding format will be provided according to the accept media type of the client's HTTP request header.

Table 12-3. *XMLHttpRequest Methods and Properties Table*

URL	HTTP Method	Description	Controller Method
`/singer/listdata`	GET	Retrieves all singers	`listData`
`/singer/id`	GET	Retrieves a singer by ID	`findBySingerId(...)`
`/singer`	POST	Creates a new singer	`create(...)`
`/singer/id`	PUT	Updates a singer by ID	`update(...)`
`/singer/id`	DELETE	Deletes a singer by ID	`delete(...)`

Using Spring MVC to Expose RESTful Web Services

In this section, we show you how to use Spring MVC to expose the singer services as RESTful web services, as designed in the previous section. This sample builds upon some of the `SingerService` classes that were used in the Spring HTTP invoker sample.

You are already familiar with the `Singer` class, so we won't show the code here again. But to serialize and deserialize a list of singers, we need to encapsulate it in a container.[3] Here you can see the `Singers` class. It has a single property, which is a list of `Singer` objects. The purpose is to support the transformation from a list of singers (returned by the `listData()` method within the `SingerController` class) into XML or JSON format.

```
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;
    }
}
```

[3]This is needed for XML serialization; JSON works without the container class.

Configuring Castor XML

To support the transformation of the returned singer information into XML format, we will use the Castor XML library (http://castor.codehaus.org). Castor supports several modes between POJO and XML transformation, and in this sample, we use an XML file to define the mapping. The following XML snippet shows the mapping file (oxm-mapping.xml):

```
<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>
```

Two mappings are defined. The first <class> tag maps the Singers class, within which its singers property (a List of singers objects) is mapped using the <bind-xml name="singer"/> tag. The Singer object is then mapped (with the <map-to xml="singer" /> tag within the second <class> tag). In addition, to support the transformation from java.util.Date type (for Singer's birthDate attribute), we implement a custom Castor field handler. The following code snippet shows the field handler:

```
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;
    }
}
```

We extend Castor's `org.exolab.castor.mapping.GeneralizedFieldHandler` class and implement the `convertUponGet()`, `convertUponSet()`, and `getFieldType()` methods. Within the methods, we implement the logic to perform the transformation between `Date` and `String` for use by Castor.

In addition, we also define a properties file for use with Castor. The following snippet shows the contents of the file (`castor.properties`):

```
org.exolab.castor.indent=true
```

The property instructs Castor to generate XML with an indent, which is much easier to read when testing.

Implementing SingerController

The next step is to implement the controller class, `SingerController`. The following code snippet shows the contents of this class, which has all the methods in Table 12-3 implemented:

```
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");
    }
}
```

The main points about the previous class are as follows:

- The class is annotated with `@Controller`, indicating that it's a Spring MVC controller.
- The class-level annotation `@RequestMapping(value="/singer")` indicates that this controller will be mapped to all URLs under the main web context. In this sample, all URLs under `http://localhost:8080/singer` will be handled by this controller.
- The `SingerService` within the service layer implemented earlier in this chapter is autowired into the controller.

- The `@RequestMapping` annotation for each method indicates the URL pattern and the corresponding HTTP method that it will be mapped to. For example, the `listData()` method will be mapped to the `http://localhost:8080/singer/listdata` URL, with an HTTP GET method. For the `update()` method, it will be mapped to the URL `http://localhost:8080/singer/\protect\T1\textbraceleftid\ protect\T1\ textbraceright`, with an HTTP PUT method.
- The `@ResponseBody` annotation is applied to all methods. This instructs that all the return values from the methods should be written to the HTTP response stream directly and not matched to a view.
- For methods that accept path variables (for example, the `findSingerById()` method), the path variable is annotated with `@PathVariable`. This instructs Spring MVC to bind the path variable within the URL (for example, `http://localhost:8080/singer/1`) into the `id` argument of the `findSingerById()` method. Note that for the `id` argument, the type is `Long`, while Spring's type conversion system will automatically handle the conversion from `String` to `Long` for us.
- For the `create()` and `update()` method, the `Singer` argument is annotated with `@RequestBody`. This instructs Spring to automatically bind the content within the HTTP request body into the `Singer` domain object. The conversion will be done by the declared instances of the `HttpMessageConverter<Object>` interface (under the package `org.springframework.http.converter`) for supporting formats, which will be discussed later in this chapter.

Starting with Spring 4.0, a controller annotation dedicated for REST usage was introduced, the `@RestController`. This is a convenience annotation that is itself annotated with `@Controller` and `@ResponseBody`. When used on a controller class, all methods that are annotated with `@RequestMapping` are automatically annotated with `@ResponseBody`. A version of the `SingerController` written using this annotation will be covered later in the chapter.

Configuring a Spring Web Application

A Spring web application is needed to resolve the REST requests sent by a client, so it needs to be configured. Earlier in the chapter we covered a simple web application configuration. That configuration now must be enriched with HTTP message converter beans for XML and JSON.

Spring web applications follow the Front Controller design pattern,[4] where all requests are received by a single controller, which later dispatches them to the appropriate handlers (controller classes). This central dispatcher is an instance of `org.springframework.web.servlet.DispatcherServlet` and is registered by an `AbstractAnnotationConfigDispatcherServletInitializer` class, which needs to be extended to replace the `web.xml` configuration. The class `WebInitializer` that does this in the samples for this section is shown here:

```
package com.apress.prospring5.ch12.init;

import com.apress.prospring5.ch12.config.DataServiceConfig;
import org.springframework.web.servlet.support.
    AbstractAnnotationConfigDispatcherServletInitializer;
```

[4]You can find a good explanation of this design pattern here: `www.oracle.com/technetwork/java/frontcontroller-135648.html`.

```
public class WebInitializer extends
   AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[]{
                DataServiceConfig.class
        };
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[]{
                WebConfig.class
        };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[]{"/"};
    }
}
```

In Spring MVC, each `DispatchServlet` will have its own `WebApplicationContext` (however, all service-layer beans defined in `DataServiceConfig.class`, which is called the root `WebApplicationContext`, will be available for each servlet's own `WebApplicationContext` too).

The `getServletMappings()` method instructs the web container (for example, Tomcat) that all URLs under the pattern `/` (for example, `http://localhost:8080/singer`) will be handled by the RESTful servlet. Of course, we could have added a context, such as `/ch12`, there, but for the samples needed in this section, we wanted to keep the URLs as short and as obvious in purpose as possible.

The Spring MVC configuration class (the `WebConfig` class) with HTTP message converters is shown here:

```
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;
    }
}
```

The important points for the previous class are as follows:

- The `@EnableWebMvc` annotation[5] enables the annotation support for Spring MVC (that is, the `@Controller` annotation), as well as registers Spring's type conversion and formatting system. In addition, JSR-303 validation support is enabled under the definition of this annotation.
- The `configureMessageConverters(...)` method[6] declares the instances of `HttpMessageConverter` that will be used for media conversion for supported formats. Because we will support both JSON and XML as the data format, two converters are declared. The first one is `MappingJackson2HttpMessageConverter`, which is Spring's support for the Jackson JSON library.[7] The other one is `MarshallingHttpMessageConverter`, which is provided by the `spring-oxm` module for XML marshalling/unmarshalling. Within `MarshallingHttpMessageConverter`, we need to define the marshaller and unmarshaller to use, which is the one provided by Castor in this case.
- For the `castorMarshaller` bean, we use the Spring-provided class `org.springframework.oxm.castor.CastorMarshaller`, which integrates with Castor, and we provide the mapping location that Castor requires for its processing.
- The `@ComponentScan annotation`[8] instructs Spring to scan for the specified package for controller classes.

Now, the server-side service is complete. At this point, you should build the WAR file containing the web application, or if you are using an IDE such as IntelliJ IDEA or STS, launch the Tomcat instance.

---

[5]This is the equivalent of the `<mvc:annotation-driven>` tag/.
[6]This is the equivalent of the `<mvc:message-converters>` tag introduced in Spring 3.1.
[7]You can find the Jackson JSON library official site at http://jackson.codehaus.org.
[8]This is the equivalent of the `<context:component-scan>` tag.

Using curl to Test RESTful-WS

Let's do a quick test of the RESTful web services that we implemented. One easy way is to use `curl`,[9] which is a command-line tool for transporting data with URL syntax. To use the tool, just download it from the web site and extract it onto your computer.

For example, to test the retrieval of all singers, open a command prompt in Windows or a terminal in Unix/Linux, deploy the WAR onto the server, and fire the following command:

```
$ 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: 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
```

This command sends an HTTP request to the server's RESTful web service; in this case, it invokes the `listData()` method in `SingerController` to retrieve and return all singer information. Also, the `-H` option declares an HTTP header attribute, meaning that the client wants to receive data in JSON format. Running

[9] See `http://curl.haxx.se`.

the command produces output in JSON format for the initially populated singer information that was returned. Now let's take a look at the XML format; the command and results are shown here:

```
$ 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
```

As you can see, there is only one difference between the two samples. The accept media was changed from JSON to XML. Running the command produces XML output instead. This is because of the `HttpMessageConverter` beans that were defined in the RESTful servlet's `WebApplicationContext`, while Spring MVC will invoke the corresponding message converter based on the client's HTTP header's accept media information and will write to the HTTP response accordingly.

Using RestTemplate to Access RESTful-WS

For Spring-based applications, the `RestTemplate` class is designed to access RESTful web services. In this section, we show how to use the class to access the singer service on the server. First let's take a look at the basic `ApplicationContext` configuration for Spring's `RestTemplate`, as shown in the following code snippet:

```
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;
    }
}
```

You declare a restTemplate bean using the RestTemplate class. The class uses Castor to inject the property messageConverters with an instance of MarshallingHttpMessageConverter, the same as the one on the server side. The mapping file will be shared between both the server and client sides. In addition, for the restTemplate bean, within the anonymous class MarshallingHttpMessageConverter, the property supportedMediaTypes is injected with a MediaType instance, indicating that the only supported media is XML. As a result, the client is always expecting XML as the return data format, and Castor will help perform the conversion between POJO and XML.

To test all REST URLs supported by the web application, a JUnit class is more appropriate, executed within a Spring application context defined by RestClientConfig. The code is shown next, and in a smart editor such as IntelliJ IDEA or STS, each method can be executed individually:

```
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()));
    }
}
```

The URLs for accessing various operations are declared, which will be used in later samples. The instance of `RestTemplate` is injected, and then in the `testFindAll` method the `RestTemplate.getForObject()` method is called (which corresponds to the HTTP GET method), passing in the URL and the expected return type, which is the `Singers` class that contains the full list of singers.

Make sure the application server is running. Running the testFindAll test method, the test should pass and produce the following output:

```
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
```

As you can see, the MarshallingHttpMessageConverter bean registered within RestTemplate converts the message into a POJO automatically. Next, let's try to retrieve a singer by ID. In this method we use a variant of the RestTemplate.getForObject() method, which also passes in the ID of the singer we want to retrieve as the path variable within the URL (the {id} path variable in URL_GET_CONTACT_BY_ID). If the URL has more than one path variable, you can use an instance of Map<String,Object> or use the varargs support of the method to pass in the path variables. With varargs, you need to follow the order of the path variable as declared in the URL. Run the testFindbyId() test method. The test should pass, and you should see the following output:

```
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
```

As you can see, the correct singer is retrieved. Now it's update's turn. First we retrieve the singer we want to update. After the singer object is updated, we then use the RestTemplate.put() method, which corresponds to the HTTP PUT method, passing in the update URL, the updated singer object, and the ID of the singer to update. Running the testUpdate() produces the following output (other output has been omitted):

```
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
```

Next is the delete operation. The RestTemplate.delete() method is called, which corresponds to the HTTP DELETE method, passing in the URL and the ID. Then, all singers are retrieved and displayed again to verify the deletion. Running the testDelete() test method produces the following output (other output has been omitted):

```
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
```

As you can see, the singer with an ID of 3 is deleted. Finally, let's try the insert operation. A new instance of the Singer object is constructed. Then the RestTemplate.postForObject() method is called, which corresponds to the HTTP POST method, passing in the URL, the Singer object we want to create, and the class type. Running the program again produces the following output:

```
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
 //listing all singers
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
```

The server is created on the server and returned to the client.

Securing RESTful-WS with Spring Security

Any remoting service requires security to restrict unauthorized parties from accessing the service and retrieving business information or acting on it. RESTful-WS is no exception. In this section, we demonstrate how to use the Spring Security project to secure RESTful-WS on the server. In this example, we are using Spring Security 5.0.0.M2 (the latest stable version at the time of writing), which provides some useful support for RESTful-WS.

Using Spring Security to secure RESTful-WS is a three-step process. First, in the web application deployment descriptor (web.xml), a security filter named springSecurityFilterChain needs to be added, but as we are not configuring the application using XML, the filter is replaced by a class extending AbstractSecurityWebApplicationInitializer. This class registers DelegatingFilterProxy to use springSecurityFilterChain before any other registered Filter. The implementation is shown here, and the class is empty because we are not doing any customization to it:

```
package com.apress.prospring5.ch12.init;

import org.springframework.security.web.
    context.AbstractSecurityWebApplicationInitializer;

public class SecurityWebApplicationInitializer
   extends AbstractSecurityWebApplicationInitializer {
}
```

We need now to add a Spring configuration class for security where we will declare who can access the application and what they are allowed to do. In the case of this application, things are easy: we are using in-memory authentication for teaching purposes, so add a user named prospring5 with the password prospring5 and the role 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();
   }
}
```

The class is annotated with the @EnableWebSecurity annotation to enable secured behavior in a Spring web application. In the configure(...) method, we declare that the resources under the URL /rest/** should be protected. The sessionCreationPolicy() method is used to allow us to configure whether the HTTP session will be created upon authentication. Since the RESTful-WS we are using is stateless, we set the value to SessionCreationPolicy.STATELESS, which instructs Spring Security not to create an HTTP session for all RESTful requests. This can help improve the performance of the RESTful services.

Next, in antMatchers("/rest/**"), we set that only users with the REMOTE role assigned can access the RESTful service. The httpBasic() method specifies that only HTTP basic authentication is supported for RESTful services.

The configureGlobal(AuthenticationManagerBuilder auth) method defines the authentication information. Here we define a simple authentication provider with a hard-coded user and password (both set to remote) with the REMOTE role assigned. In an enterprise environment, most likely the authentication will be done by either a database or an LDAP lookup.

The .formLogin() method is used to tell Spring to generate a basic login form that can be used to test that the application has been secured correctly. The login form is accessible at http://localhost:8080/login.

The filter springSecurityFilterChain is used to enable Spring Security to intercept the HTTP request for an authentication and authorization check. Because we want to secure only RESTful-WS, the filter is applied only to the URL pattern /rest/* (see the antMatchers(...) methods). We want to secure all the REST URLs but allow the user to see the main page of the application (a simple HTML file that is displayed when http://localhost:8080/ is accessed in the browser), so this is the moment that you add the rest application context, besides adding SecurityConfig to the root context application.

```
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/**"};
    }
}
```

Now the security setup is complete. If you redeploy the project and run any of the test methods under `RestClientTest`, you will have the following output (other output has been omitted):

```
Exception in thread "main" org.springframework.web.cient.HttpClientErrorException:
    401 Unauthorized
```

You will get the HTTP status code 401, which means you are not authorized to access the service. Now let's configure the client's `RestTemplate` to provide the credential information to the server.

```
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 = n
            ew 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;
    }
}
```

In the restTemplate bean, a constructor argument with a reference to the httpRequestFactory bean is injected. For the httpRequestFactory bean, the HttpComponentsClientHttpRequestFactory class is used, which is Spring's support for the Apache HttpComponents HttpClient library, and we need the library to construct an instance of CloseableHttpClient that stores the credentials for our client. To support the injection of credentials, you create a simple bean of type UsernamePasswordCredentials. The UsernamePasswordCredentials class is constructed with the prospring5 username and password. With the httpRequestFactory constructed and injected into the RestTemplate, all RESTful requests fired using this template will carry the credential provided. Now we can simply run the test methods in the RestClientTest class again, and you will see that the services are invoked as usual.

RESTful-WS with Spring with Spring Boot

Because Spring Boot makes everything easier to develop, we needed to add a section about how Spring Boot can make the development of a Spring RESTful service easier as well. The `Singer` entity, repository, and service classes are the same as before; there's no need to change anything. To make things simple and to make use of as much of the default Spring Boot default configuration as possible, the XML serialization will be removed as well. JSON serialization is supported by default. As the application is a web application, the configuration is the same as for the Spring Boot web application introduced earlier, so we won't go over it again. The `Application` class and entry point for a Spring Boot application are as simple as shown here:

```
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();
   }
}
```

And as promised earlier, here is the new and improved `SingerController` rewritten using `@RestController` and HTTP method–specific mapping annotations introduced in 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");
   }
}
```

Spring version 4.3 introduced some customization of the @RequestMapping annotations that match basic HTTP methods. Table 12-4 lists the equivalence between the new annotations and old-style @RequestMapping.

Table 12-4. *Annotations for Mapping HTTP Method Requests onto Specific Handler Methods Introduced in Spring 4-3*

Annotation	Old-Style Equivalent
@GetMapping	@RequestMapping(method = RequestMethod.GET)
@PostMapping	@RequestMapping(method = RequestMethod.POST)
@PutMapping	@RequestMapping(method = RequestMethod.PUT)
@DeleteMapping	@RequestMapping(method = RequestMethod.DELETE)

Also, because we are using JSON, which supports lists and arrays, the class Singers is no longer needed.

Testing the application is simple because RestTemplate does not need any configuration. All that is needed to create a RestTemplate instance by calling the default constructor. The test methods are the same as before.

```
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);
    }
    ...
}
```

Just run the Application class and then execute the test methods one by one.

And if you want to make sure that the application actually works and the singer instances are serialized using the JSON format, you can use curl to test this Spring Boot application.

```
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}]
```

If the output troubles you, just keep in mind that without declaring an explicit JSON message converter, the Date fields will be shown as numbers and the response won't be formatted.

Using Spring Boot, you can also secure resources really easily, but this is a subject that will be covered more in detail in Chapter 16.

Using AMQP in Spring

Remoting can also be accomplished by using remote procedure call (RPC) communication with Advanced Message Queuing Protocol (AMQP) as a transport. AMQP is an open standard protocol for implementing message-oriented middleware (MOM).

A JMS application works in any OS environment, but it supports only the Java platform. So, all communicating applications must be developed in Java. The AMQP standard can be used to develop applications in multiple languages that can easily communicate.

Similar to using JMS, AMQP also uses a message broker to exchange messages through. In this example, we use RabbitMQ[10] as the AMQP server. Spring itself does not provide remoting capabilities in the core framework. Instead, they are handled by a sister project called Spring AMQP,[11] which we use as the underlying communication API. The Spring AMQP project provides a base abstraction around AMQP and an implementation for communicating with RabbitMQ. In this chapter, we won't cover all of AMQP or Spring AMQP's features, just the remoting functionality via RPC communication.

The Spring AMQP project consists of two parts: `spring-amqp` is the base abstraction, and `spring-rabbit` is the RabbitMQ implementation. The stable version of Spring AMQP at the time of writing is 2.0.0.M4.

First, you will need to obtain RabbitMQ from `www.rabbitmq.com/download.html` and start the server. RabbitMQ will work fine out of box for our needs, and no configuration changes are needed. Once RabbitMQ is running, the next thing we need to do is create a service interface. In this example, we create a simple weather service that returns a forecast for the provided state code. Let's get started by creating the `WeatherService` interface shown here:

```
package com.apress.prospring5.ch12;

public interface WeatherService {
    String getForecast(String stateCode);
}
```

Next, let's create an implementation of `WeatherService` that will simply reply with a weather forecast for the provided state, or an unavailable message if no forecast is available, as shown here:

```
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";
        }

        return "Not available at this time";
    }
}
```

With the weather service code in place, let's build the configuration file (`amqp-rpc-app-context.xml`) that will configure the AMQP connection and expose `WeatherService`, as shown here:

```
<?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"
```

[10]See `www.rabbitmq.org`.
[11]See `http://projects.spring.io/spring-amqp`.

```
       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>
```

We configure the RabbitMQ connection along with exchange and queue information. We then create a bean by using the AmqpProxyFactoryBean class, which the client uses as a proxy to make an RPC request. For the response, we use the AmqpInvokerServiceExporter class, which gets wired into a listener container. The listener container is responsible for listening for AMQP messages and handing them off to the weather service. As you can see, the configuration is similar to JMS in terms of connections, queues, listener containers, and so on. While similar in configuration, JMS and AMQP are very different transports, and it's recommended that you visit the AMQP web site[12] for full details on the protocol.

[12]See the AMQP web site at www.amqp.org.

With the configuration in place, let's create a sample class to execute RPC calls.

```
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");
        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();
    }
}
```

Now let's run the sample, and you should get the following output:

```
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
```

Of course, the XML configuration can be easily transformed into Java configuration classes. But a few changes need to be made to the other involved classes as well. The `WeatherServiceImpl` doesn't need to implement an interface anymore as it will just declare a listener method, which will listen to messages written on the `forecasts` queue.

```
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");
        }
    }
}
```

The rabbitListenerContainerFactory bean is of type RabbitListenerContainerFactory and is used to create a regular SimpleMessageListenerContainer. But let's see the full Java configuration.

```
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;
    }

}
```

All beans in the previous configuration can be easily matched with their XML counterparts. The new element is the `@EnableRabbit` annotation. When used on a class annotated with `@Configuration`, it enables Rabbit listener annotated endpoints that are created behind the scenes by the `RabbitListenerContainerFactory` bean.

To test the new weather service, we also have to modify the test program, and `amqpTemplate` is used to send messages to the `forecasts` queue, where `WeatherServiceImpl.getForecast(...)` will read them and print out the forecast output.

```
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();
    }
}
```

If you run the previous program and the RabbitMQ server is up, you will see the following output:

```
[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
```

Using AMQP with Spring Boot

Spring Boot helps you develop AMQP applications as well; its starter artifact `spring-boot-starter-amqp` if just for that. The configuration is simplified much. You don't need to define `RabbitTemplate`, `RabbitAdmin`, and `SimpleRabbitListenerContainerFactory` beans anymore because these beans are automatically configured and created by Spring Boot. The implementation of `WeatherServiceImpl` does not change much, but as the `SimpleRabbitListenerContainerFactory` bean is handled by Spring Boot, there is no need to add it as a value for the `@RabbitListener` annotation anymore.

```
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");
      }
   }
}
```

The Application class, annotated with @SpringBootApplication, is used as a configuration class and runner class as well.

```
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();
    }
}
```

As you can see, the @EnableRabbit annotation is not needed either, and although the configuration is not reduced by much, it's still an improvement. If you run the previous class, you will get a similar result, as you have seen in the previous examples.

```
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
```

Summary

In this chapter, we covered the most commonly used remoting techniques in Spring-based applications.

If both applications are built with Spring, then using the Spring HTTP invoker is a viable option. If an asynchronous mode or loosely coupled mode of integration is required, JMS is a commonly used approach. We discussed how to use RESTful-WS in Spring for exposing services or accessing services with the RestTemplate class. Finally, we discussed how to use Spring AMQP for RPC-style remoting via RabbitMQ.

Spring Boot for each of the technologies, remote, REST, JMS, etc was covered, as is always something you should look for.

In the next chapter, we discuss using Spring for testing applications; it's about time that we elaborated on some test techniques to make your life easier.

CHAPTER 13

Spring Testing

When developing applications for enterprise use, testing is an important way to ensure that the completed application performs as expected and fulfills all kinds of requirements (architectural, security, user requirements, and so on). Every time a change is made, you should ensure that the changes that were introduced don't impact the existing logic. Maintaining an ongoing build and test environment is critical for ensuring high-quality applications. Reproducible tests with high coverage for all your code allow you to deploy new applications and changes to applications with a high level of confidence. In an enterprise development environment, there are many kinds of testing that target each layer within an enterprise application, and each kind of testing has its own characteristics and requirements. In this chapter, we discuss the basic concepts involved in the testing of various application layers, especially in the testing of Spring-powered applications. We also cover the ways in which Spring makes implementing the test cases of various layers easier for developers. Specifically, this chapter covers the following topics:

- *Enterprise testing framework*: We briefly describe an enterprise-testing framework. We discuss various kinds of testing and their purposes. We focus on unit testing, targeting various application layers.
- *Logic unit test*: The finest unit test is to test only the logic of the methods within a class, with all other dependencies being "mocked" with the correct behavior. In this chapter, we discuss the implementation of logic unit testing for the Spring MVC controller classes, with the help of a Java mock library to perform the mocking of a class's dependencies.
- *Integration unit test*: In an enterprise-testing framework, integration testing refers to testing the interaction of a group of classes within different application layers for a specific piece of business logic. Typically, in an integration-testing environment, the service layer should test with the persistence layer, with the back-end database available. However, as application architecture evolves and the maturity of lightweight in-memory databases evolves, it's now a common practice to "unit test" the service layer with the persistence layer and back-end database as a whole. For example, in this chapter, we use JPA 2, with Hibernate and Spring Data JPA as the persistence provider and with H2 as the database. In this architecture, it's of less importance to "mock" Hibernate and Spring Data JPA when testing the service layer. As a result, in this chapter, we discuss testing of the service layer together with the persistence layer and the H2 in-memory database. This kind of testing is generally referred to as *integration unit testing*, which sits in the middle of unit testing and full-blown integration testing.

© Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho 2017
I. Cosmina et al., *Pro Spring 5*, https://doi.org/10.1007/978-1-4842-2808-1_13

- *Front-end unit test*: Even if you test every layer of the application, after the application is deployed, you still need to ensure that the entire application works as expected. More specifically, for a web application, upon deployment to the continuous build environment, you should run front-end testing to ensure that the user interface is working properly. For example, for a singer application, you should ensure that each step of the normal functionality works properly, and you also should test exceptional cases (for example, how the application functions when information doesn't pass the validation phase). In this chapter, we briefly discuss a front-end testing framework.

Introducing Testing Categories

An enterprise testing framework refers to testing activities in the entire application's life cycle. In various phases, different testing activities are performed to verify that the functionalities of the application are working as expected, according to the defined business and technical requirements.

In each phase, different test cases are executed. Some are automated, while others are performed manually. In each case, the result is verified by the corresponding personnel (for example, business analysts, application users, and so on). Table 13-1 describes the characteristics and objectives of each type of testing, as well as common tools and libraries that are used for implementing the test cases.

Table 13-1. *Different Testing Categories Used in Practice*

Test Category	Description	Common Tools
Logic unit test	A logic unit test takes a single object and tests it by itself, without worrying about the role it plays in the surrounding system.	Unit test: JUnit, TestNG Mock objects: Mockito, EasyMock
Integration unit test	An integration unit test focuses on testing the interaction between components in a "near real" environment. These tests will exercise the interactions with the container (embedded database, web container, and so on).	Embedded database: H2 Database testing: DbUnit In-memory web container: Jetty
Front-end unit test	A front-end unit test focuses on testing the user interface. The objective is to ensure that each user interface reacts to users' actions and produces output to users as expected.	Selenium
Continuous build and code quality test	The application code base should be built on a regular basis to ensure that the code quality complies with the standard (for example, comments are all in place, no empty exception catch block, and so on). Also, test coverage should be as high as possible to ensure that all developed lines of codes are tested.	Code quality: PMD, Check-style, FindBugs, Sonar Test coverage: Cobertura, EclEmma Build tool: Gradle, Maven Continuous build: Hudson, Jenkins

(*continued*)

Table 13-1. (*continued*)

Test Category	Description	Common Tools
System integration test	A system integration test verifies the accuracy of communication among all programs in the new system and between the new system and all external interfaces. The integration test must also prove that the new system performs according to the functional specifications and functions effectively in the operating environment without adversely affecting other systems.	IBM Rational Functional Tester, HP Unified Functional Testing
System quality test	A system quality test is to ensure that the developed application meets those nonfunctional requirements. Most of the time, this tests the performance of the application to ensure that the target requirements for concurrent users of the system and workload are met. Other nonfunctional requirements include security, high-availability features, and so on.	Apache JMeter, HP LoadRunner
User acceptance test	A user acceptance test simulates the actual working conditions of the new system, including the user manuals and procedures. Extensive user involvement in this stage of testing provides the user with invaluable training in operating the new system. It also benefits the programmer or designer to see the user experience with the new programs. This joint involvement encourages the user and operations personnel to approve the system conversion.	IBM Rational TestManager, HP Quality Center

In this chapter, we focus on the implementation of the three kinds of unit test (logic unit test, integration unit test, and front-end unit test) and show how the Spring TestContext framework and other supporting tools and libraries can help in developing those test cases.

Instead of presenting the full details and list of classes that the Spring Framework provides in the testing area, we cover the most commonly used patterns and the supporting interfaces and classes within the Spring TestContext framework as we show how to implement the sample test cases in this chapter.

Using Spring Test Annotations

Before moving on to logic and integration tests, it's worth noting that Spring provides testing-specific annotations in addition to the standard annotations (such as `@Autowired` and `@Resource`). These annotations can be used in your logic and unit tests, providing various functionality such as simplified context file loading, profiles, test execution timing, and much more. Table 13-2 outlines the annotations and their uses.

***Table 13-2.** Description of Enterprise Testing Framework*

Annotation	Description
@ContextConfiguration	Class-level annotation used to determine how to load and configure an ApplicationContext for integration tests.
@WebAppConfiguration	Class-level annotation used to indicate the ApplicationContext loaded should be a WebApplicationContext.
@ContextHierarchy	Class-level annotation indicating which bean profile should be active.
@DirtiesContext	Class and method-level annotation used to indicate that the context has been modified or corrupted in some way during the execution of the test and should be closed and rebuilt for subsequent tests.
@TestExecutionListeners	Class-level annotation for configuring TestExecutionListeners that should be registered with the TestContextManager.
@TransactionConfiguration	Class-level annotation used to indicate transaction configuration such as rollback settings and a transaction manager (if your desired transaction manager does not have a bean name of transactionManager).
@Rollback	Class and method-level annotation used to indicate whether the transaction should be rolled back for the annotated test method. Class-level annotations are used for testing class default settings.
@BeforeTransaction	Method-level annotation indicating that the annotated method should be called before a transaction is started for test methods marked with the @Transactional annotation.
@AfterTransaction	Method-level annotation indicating that the annotated method should be called after a transaction has ended for test methods marked with the @Transactional annotation.
@IfProfileValue	Class-and method-level annotation used to indicate that the test method should be enabled for a specific set of environmental conditions.
@ProfileValueSource Configuration	Class-level annotation used to specify the ProfileValueSource used by @IfProfileValue. If this annotation is not declared on the test, SystemProfileValueSource is used as the default.
@Timed	Method-level annotation used to indicate that the test must finish in the specified time period.
@Repeat	Method-level annotation used to indicate that the annotated test method should be repeated the specified number of times.

Implementing Logic Unit Tests

As previously discussed, a logic unit test is the finest level of testing. The objective is to verify the behavior of an individual class, with all the class's dependencies being "mocked" with expected behavior. In this section, we demonstrate a logic unit test by implementing the test cases for the SingerController class, with the service layer being mocked with expected behavior. To help mock the behavior of the service layer, we will show how to use Mockito (http://site.mockito.org/), which is a popular mocking framework.

The Spring Framework provides first-class support for integration testing in the `spring-test` module. To provide a test context for the integration tests that will be created for this section, you will use the `spring-test.jar` library. This library contains valuable classes for integration testing with a Spring container.

Adding Required Dependencies

First we need to add the dependencies into the project, as shown in the following configuration sample. We will also be building upon classes and interfaces created in prior chapters such as `Singer`, `SingerService`, and so on.

```
\\pro-spring-15\build.gradle
ext {
   //spring libs
   springVersion = '5.0.0.RC1'
   bootVersion = '2.0.0.M1'

   //testing libs
   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"
   ]
   ...
}
```

Unit Testing Spring MVC Controllers

In the presentation layer, controller classes provide the integration between the user interface and the service layer.

Methods in controller classes will be mapped to the HTTP requests. Within the method, the request will be processed, will bind to model objects, and will interact with the service layer (which was injected into the controller classes via Spring's DI) to process the data. Upon completion, depending on the result, the controller class will update the model and the view state (for example, user messages, objects for REST services, and so on) and return the logical view (or the model with the view together) for Spring MVC to resolve the view to be displayed to the user.

For unit testing controller classes, the main objective is to make sure that the controller methods update the model and other view states properly and return the correct view. As we want to test only the controller classes' behavior, we need to "mock" the service layer with the correct behavior.

For the `SingerController` class, we would like to develop the test cases for the `listData()` and `create(Singer)` methods. In the following sections, we discuss the steps for this.

Testing the listData( ) Method

Let's create the first test case for the `singerController.listData()` method. In this test case, we want to make sure that when the method is called, after the list of singers is retrieved from the service layer, the information is saved correctly into the model, and the correct objects are returned. The following code snippet shows the test case:

```
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(1l);
        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());
    }
}
```

First, the test case calls the `initSingers()` method, which is applied with the `@Before` annotation, which indicates to JUnit that the method should be run before running each test case (if you want to run some logic before the entire test class, use the `@BeforeClass` annotation). In the method, a list of singers is initialized with hard-coded information.

Second, the `testList()` method is applied with the `@Test` annotation, which indicates to JUnit that it's a test case that JUnit should run. Within the test case, the private variable `singerService` (of type `SingerService`) is mocked by using Mockito's `Mockito.mock()` method (note the `import static` statement). The `when()` method is also provided by Mockito to mock the `SingerService.findAll()` method, which will be used by the `SingerController` class.

Third, an instance of the `SingerController` class is created, and then its `singerService` variable, which will be injected by Spring in normal situations, is set with the mocked instance by using the Spring-provided `ReflectionTestUtils` class's `setField()` method. `ReflectionTestUtils` provides a collection of reflection-based utility methods for use in unit and integration testing scenarios. In addition, an instance of the `ExtendedModelMap` class (which implements the `org.springframework.ui.Model` interface) is constructed.

Next, the `SingerController.listData()` method is called. Upon invocation, the result is verified by calling the various assert methods (provided by JUnit) to ensure that the list of singer information is saved correctly in the model used by the view.

Now we can run the test case, and it should run successfully. You can verify this via your build system or IDE. We can now proceed with the `create()` method.

Testing the create( ) Method

The following code snippet shows the code snippet for testing the `create()` method:

```
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(999l), singer.getId());
        assertEquals("BB", singer.getFirstName());
        assertEquals("King", singer.getLastName());

        assertEquals(2, singers.size());
    }
}
```

The `SingerService.save()` method is mocked to simulate the addition of a new `Singer` object in the list of singers. Note the use of the `org.mockito.stubbing.Answer<T>` interface, which mocks the method with the expected logic and returns a value.

Then, the `SingerController.create()` method is called, and assert operations are invoked to verify the result. Run the result again, and note the test case results. For the `create()` method, we should create more test cases to test various scenarios. For example, we need to test when data access errors are encountered during the save operation.

Everything that was covered until now can be done with JMock (www.jmock.org/), and a version of the SingerControllerTest class using this library is part of the code sample for this section. We won't cover this here because the idea of mocking dependencies is the focus, not the library that this can be done with.[1]

Implementing an Integration Test

In this section, we will implement the integration test for the service layer. In the singer application, the core service is the class SingerServiceImpl, which is the JPA implementation of the SingerService interface.

When unit testing the service layer, you will use the H2 in-memory database to host the data model and testing data, with the JPA providers (Hibernate and Spring Data JPA's repository abstraction) in place. The objective is to ensure that the SingerServiceImpl class is performing the business functions correctly.

In the following sections, we show how to test some of the finder methods and the save operation of the SingerServiceImpl class.

Adding Required Dependencies

For implementing test cases with the database in place, we need a library that can help populate the desired testing data in the database before executing the test case and that can perform the necessary database operations easily. Moreover, to make it easier to prepare the test data, we will support the preparation of test data in Microsoft Excel format.

To fulfill these purposes, additional libraries are required. On the database side, DbUnit (http://dbunit.sourceforge.net) is a common library that can help implement database-related testing. In addition, the Apache POI (http://poi.apache.org) project's library will be used to help parse the test data that was prepared in Microsoft Excel.

Configuring the Profile for Service-Layer Testing

The bean definition profiles feature introduced in Spring 3.1 is useful for implementing a test case with the appropriate configuration of the testing components. To facilitate the testing of the service layer, we will also use the profile feature for the ApplicationContext configuration. For the Singer application, we would like to have two profiles, as follows:

- *Development profile (dev)*: Profile with configuration for the development environment. For example, in the development system, the back-end H2 database will have both the database creation and the initial data population scripts executed.
- *Testing profile (test)*: Profile with configuration for the testing environment. For example, in the testing environment, the back-end H2 database will have only the database creation script executed, while the data will be populated by the test case.

Let's configure the profile environment for the singer application. For the singer application, the back-end configuration (that is, data source, JPA, transaction, and so on) was defined in the configuration XML file datasource-tx-jpa.xml. We would like to configure the data source in the file for the dev profile only.

[1]Another alternative is EasyMock: http://easymock.org/.

To do this, we need to wrap the data source bean with the profile configuration. The following configuration snippet shows the change required:

```
<?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>
```

As shown in the configuration snippet, the dataSource bean is wrapped with the <beans> tag and given the profile attribute with the value dev, which indicates that the data source is applicable only for the development system. Remember, profiles can be activated by, for example, passing -Dspring.profiles.active=dev to the JVM as a system parameter.

Java Configuration Version

Starting with the introduction of Java configuration classes, XML is slowly losing ground. Because of that, the focus of this book is the usage of Java configuration classes; XML configuration is covered just to show the evolution of Spring configuration over time. The XML configuration shown previously can be split in two: one to cover the data source configuration, which will be profile specific, and a transaction configuration, which is common for development and test configuration. The two classes are depicted next. One improvement added in the Java configuration is the “automagical” generation of the database schema, which was done by setting the Hibernate property `hibernate.hbm2ddl.auto` to `create-drop`.

```
//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;
      }
   }

}

//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();
   }
}
```

Implementing the Infrastructure Classes

Before implementing the individual test case, we need to implement some classes to support the population of test data in the Excel file. Moreover, to ease the development of the test case, we want to introduce a custom annotation called @DataSets, which accepts the Excel file name as the argument. We will develop a custom test execution listener (a feature supported by the Spring testing framework) to check for the existence of the annotation and load the data accordingly.

Implementing Custom TestExecutionListener

In the spring-test module, the org.springframework.test.context.TestExecutionListener interface defines a listener API that can intercept the events in the various phases of the test case execution (for example, before and after the class under test, before and after the method under test, and so on). In testing the service layer, we will implement a custom listener for the newly introduced @DataSets annotation. The objective is to support the population of test data with a simple annotation on the test case. For example, to test the SingerService.findAll() method, we would like to have the code look like the following code snippet:

```
@DataSets(setUpDataSet= "/com/apress/prospring5/ch13/SingerServiceImplTest.xls")
@Test
public void testFindAll() throws Exception {
    List<Singer> result = singerService.findAll();
    ...
}
```

The application of the @DataSets annotation to the test case indicates that before running the test, testing data needs to be loaded into the database from the specified Excel file. First we need to define the custom annotation, which is shown here:

```
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 "";
}
```

The custom annotation @DataSets is a method-level annotation. In addition, implementing the TestExecutionListener interface, which is shown in the following code snippet, will develop the custom test listener class:

```
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 {
    }
}
```

After implementing the TestExecutionListener interface, a number of methods need to be implemented. However, in this case, we are interested only in the methods beforeTestMethod() and afterTestMethod(), in which the population and cleanup of the testing data before and after the execution of each test method will be performed. Note that within each method, Spring will pass in an instance of the TestContext class so the method can access the underlying testing ApplicationContext bootstrapped by the Spring Framework.

The method beforeTestMethod() is of particular interest. First, it checks for the existence of the @DataSets annotation for the test method. If the annotation exists, the test data will be loaded from the specified Excel file. In this case, the IDatabaseTester interface (with the implementation class org.dbunit.DataSourceDatabaseTester, which we will discuss later) is obtained from TestContext. The IDatabaseTester interface is provided by DbUnit and supports database operations based on a given database connection or data source.

Second, an instance of the XlsDataFileLoader class is obtained from TestContext. The XlsDataFileLoader class is DbUnit's support of loading data from the Excel file. It uses the Apache POI library behind the scenes for reading files in Microsoft Office format. Then, the XlsDataFileLoader.load() method is called to load the data from the file, which returns an instance of the IDataSet interface, representing the set of data loaded.

Finally, the IDatabaseTester.setDataSet() method is called to set the testing data, and the IDatabaseTester.onSetup() method is called to trigger the population of data.

In the afterTestMethod() method, the IDatabaseTester.onTearDown() method is called to clean up the data.

Implementing the Configuration Class

Let's proceed to implement the configuration class for the testing environment. The following code snippet shows the code using Java Config–style configuration:

```
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();
    }
}
```

The `ServiceTestConfig` class defines the `ApplicationContext` implementation for service-layer testing. The `@ComponentScan` annotation is applied to instruct Spring to scan the service-layer beans that we want to test.

The `excludeFilters` attribute is used to make sure the test database is not being initialized with the development entries.

The `@Profile` annotation specifies that the beans defined in this class belong to the test profile.

Second, within the class, another `dataSource` bean is declared that executes only the `schema.sql` script to the H2 database without any data. The custom test execution listener for loading test data from the Excel file used the `databaseTester` and `xlsDataFileLoader` beans. Note that the `dataSourceDatabaseTester` bean was constructed using the `dataSource` bean defined for the testing environment.

Unit Testing the Service Layer

Let's begin with unit testing the finder methods, including the `SingerService.findAll()` and `SingerService.findByFirstNameAndLastName()` methods. First we need to prepare the testing data in Excel format. A common practice is to put the file into the same folder as the test case class, with the same name. So, in this case, the file name is `/src/test/java/com/apress/prospring5/ch13/SingerServiceImplTest.xls`.

The testing data is prepared in a worksheet. The worksheet's name is the table's name (`SINGER`), while the first row is the column names within the table. Starting with the second row, data is entered for the first and last names along with the birth date. We specify the ID column, but not a value. This is because the ID will be populated by the database. See the book source code for an example Excel file.

The following code snippet shows the test class with test cases for the two finder methods:

```
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);
    }
}
```

The @RunWith annotation is the same as when testing the controller class. The @ContextConfiguration annotation specifies that the ApplicationContext configuration should be loaded from the ServiceTestConfig, ServiceConfig, and DataConfig classes. The DataConfig class should have not been there, but it was used, just to make it obvious that Spring profiles actually work. The @TestExecutionListeners annotation indicates that the ServiceTestExecutionListener class should be used for intercepting the test case execution life cycle. The @ActiveProfiles annotation specifies the profile to use. So, in this case, the dataSource bean defined in the ServiceTestConfig class will be loaded, instead of the one defined in the datasource-tx-jpa.xml file, since it belongs to the dev profile.

In addition, the class extends Spring's AbstractTransactionalJUnit4SpringContextTests class, which is Spring's support for JUnit, with Spring's DI and transaction management mechanism in place. Note that in Spring's testing environment, Spring will roll back the transaction upon execution of each test method so that all database update operations will be rolled back. To control the rollback behavior, you can use the @Rollback annotation at the method level.

There is one test case for the findAll() method and two test cases for the testFindByFirstNameAndLastName() method (one retrieves a result and one doesn't). All the finder methods are applied with the @DataSets annotation with the Singer test data file in Excel. In addition, the SingerService is autowired into the test case from ApplicationContext. The rest of the code should be self-explanatory. Various assert statements are applied in each test case to make sure that the result is as expected.

Run the test case and ensure that it passes. Next, let's test the save operation. In this case, we would to test two scenarios. One is the normal situation in which a valid Singer is saved successfully, and the other is a Singer error that should cause the correct exception to be thrown. The following code snippet shows the additional snippet for the two test cases:

```
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());
    }
}
```

In the preceding listing, take a look at the `testAddSinger()`method. Within the method, to ensure that no data exists in the `Singer` table, we call the convenient method `deleteFromTables()` provided by the `AbstractTransactionalJUnit4SpringContextTests` class to clean up the table. Note that after calling the save operation, we need to explicitly call the `EntityManager.flush()` method to force Hibernate to flush the persistence context to the database so that the `findAll()` method can retrieve the information from the database correctly.

In Spring 4.3 an alias for `SpringJUnit4ClassRunner.class` was introduced, `SpringRunner.class`.

In the second test method, the `testAddSingerWithJSR349Error()` method, we test the save operation of a `Singer` with a validation error. Note that in the `@Test` annotation, an expected attribute is passed, which specifies that this test case is expected to throw an exception with the specified type, which in this case is the `ConstraintViolationException` class.

Run the test class again and verify that the result is successful.

Note that we covered only the most commonly used classes within Spring's testing framework. Spring's testing framework provides a lot of support classes and annotations that allow us to apply fine control during the execution of the test case life cycle. For example, the @BeforeTransaction and @AfterTransaction annotations allow certain logic to be executed before Spring initiates a transaction or after a transaction is completed for the test case. For a more detailed description of the various aspects of Spring's testing framework, kindly refer to Spring's reference documentation.

Dropping DbUnit

DbUnit can be considered difficult to use because it requires extra dependencies and extra configuration classes. A Spring approach would be nice, wouldn't it? Luckily, a lot of useful annotations were introduced in Spring versions after 4.0. One of them will be used in the following example: @Sql. This annotation is used to annotate a test class or test method to configure SQL scripts and statements to be executed against a given database during integration tests. This means that test data can be prepared without using DbUnit. Because of this, the test configuration gets simplified too, and test classes do not have to extend anything.

Another addition to this section will be the use of JUnit 5 (http://junit.org/junit5/), also called JUnit Jupiter. Support for it has been provided since Spring 4.3, even before the first stable version was released. At the time of writing, the current version is 5.0.0-M4. JUnit 5 is the next generation of JUnit. The goal is to create an up-to-date foundation for developer-side testing on the JVM. This includes focusing on Java 8 and newer, as well as enabling many different styles of testing.[2]

Let's see how the configuration gets modified. The test data source configuration class for the test profile becomes a lot simpler because all we need now is an empty database.

```
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();
```

[2]This is a quote from the official site.

```
            return dbBuilder.setType(EmbeddedDatabaseType.H2).build();
        } catch (Exception e) {
            logger.error("Embedded DataSource bean cannot be created!", e);
            return null;
        }
    }

}
```

The data and queries necessary for the test case will be provided using SQL script files. The data will be provided by a file called `test-data.sql`. Its contents are shown here:

```
insert into singer (first_name, last_name, birth_date,version)
 values ('John', 'Mayer', '1977-10-16', 0);
```

The cleanup of the test database will be done using the `clean-up.sql` script. This script is used to empty the database so that the data necessary for a test method does not pollute the data for another test method. Its contents are shown here:

```
delete from singer;
```

The test class will make use of a few JUnit5 annotations just to show how they can be used. Every annotation will be explained after the code section.

```
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);
    }
}
```

ApplicationContext is created using the SpringJUnitJupiterConfig annotation. This is a composed annotation that combines @ExtendWith(SpringExtension.class) from JUnit Jupiter with @ContextConfiguration from the Spring TestContext Framework.

The @DisplayName annotation is a typical JUnit Jupiter annotation used to declare a custom display value for the annotated test class or test method. In an editor that supports JUnit 5, this can look very pretty, as shown in Figure 13-1.

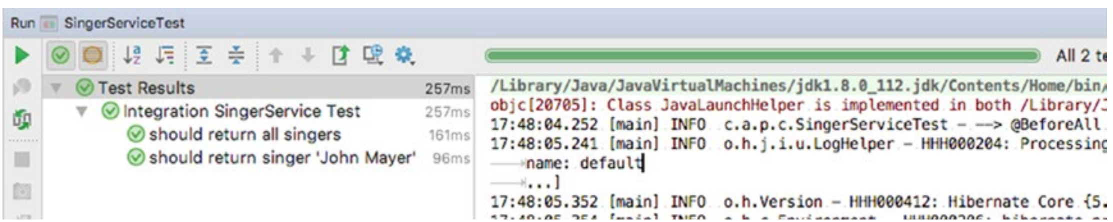

Figure 13-1. IntelliJ IDEA Jnit 5 test run view

This makes it easier to see that SingerService is working as expected, right?

The annotations @BeforeAll and @AfterAll have names that are self-explanatory, and they replace JUnit 4's @BeforeClass and @AfterClass.

The same can be said about @BeforeEach and @AfterEach; they replace JUnit 4 @Before and @After. The @SqlGroup annotation is used to group multiple @Sql annotations.

Implementing a Front-End Unit Test

Another testing area of particular interest is testing the front-end behavior as a whole, upon the deployment of the web application to a web container like Apache Tomcat.

The main reason is that even though we test every layer within the application, we still need to make sure that the views behave correctly with different actions from users. Automating front-end testing is important in saving time for developers and users when repeating the actions on the front end for a test case.

However, developing a test case for a front end is a challenging task, especially for those web applications with a lot of interactive, rich, and Ajax-based components.

Introducing Selenium

Selenium is a powerful and comprehensive tool and framework target for automating web-based front-end testing. The main feature is that by using Selenium, we can "drive" the browsers, simulating user interactions with the application, and perform verification of the view status.

Selenium supports common browsers including Firefox, IE, and Chrome. In terms of languages, Java, C#, PHP, Perl, Ruby, and Python are supported. Selenium is also designed with Ajax and rich Internet applications (RIAs) in mind, making automated testing of modern web applications possible.

If your application has a lot of front-end user interfaces and needs to run a large number of front-end tests, the `selenium-server` module provides built-in grid functionality that supports the execution of front-end tests among a group of computers.

The Selenium IDE is a Firefox plug-in that can help "record" user interactions with the web application. It also supports replay and exports the scripts into various formats that can help simplify the development of test cases.

Starting from version 2.0, Selenium integrates the WebDriver API, which addresses a number of limitations and provides an alternative, and simpler, programming interface. The result is a comprehensive object-oriented API that provides additional support for a larger number of browsers along with improved support for modern advanced web application testing problems.

Front-end web testing is a complex subject and beyond the scope of this book. From this brief overview, you can see how Selenium can help automate the user interaction with the web application front end with cross-browser compatibility. For more details, please refer to Selenium's online documentation (http://seleniumhq.org/docs).

Summary

In this chapter, we covered how to develop various kinds of unit testing in Spring-based applications with the help of commonly used frameworks, libraries, and tools including JUnit, DbUnit, and Mockito.

First, we presented a high-level description of an enterprise-testing framework, which shows what tests should be executed in each phase of the application development life cycle. Second, we developed two types of tests, including a logic unit test and integration unit test. We then briefly touched on the front-end testing framework Selenium.

Testing an enterprise application is a huge topic, and if you want to have a more detailed understanding of the JUnit library, we recommend the book *JUnit in Action* by Petar Tahchiev (Manning, 2011) that covers JUnit 4.8.

If you are interested in more Spring testing approaches, you can find more about this in the *Pivotal Certified Professional Spring Developer Exam* (www.apress.com/us/book/9781484208120, Apress, 2016), which has a dedicated chapter covering more testing libraries and working with Spring Boot Test to test Spring Boot applications.

CHAPTER 14

Scripting Support in Spring

In previous chapters, you saw how the Spring Framework can help Java developers create JEE applications. By using the Spring Framework's DI mechanism and its integration with each layer (via libraries within the Spring Framework's own modules or via integration with third-party libraries), you can simplify implementing and maintaining business logic.

However, all the logic we have developed so far was with the Java language. Although Java is one of the most successful programming languages in history, it is still criticized for some weaknesses, including its language structure and its lack of comprehensive support in areas such as massive parallel processing.

For example, one feature of the Java language is that all variables are statically typed. In other words, in a Java program, each variable declared should have a static type associated with it (`String`, `int`, `Object`, `ArrayList`, and so on). However, in some scenarios, dynamic typing may be preferred, which is supported by dynamic languages such as JavaScript.

To address those requirements, many scripting languages have been developed. Some of the most popular include JavaScript, Groovy, Scala, Ruby, and Erlang. Almost all of these languages support dynamic typing and were designed to provide the features that are not available in Java, as well as targeting other specific purposes. For example, Scala (`www.scala-lang.org`) combines functional programming patterns with OO patterns and supports a more comprehensive and scalable concurrent programming model with concepts of actors and message passing. In addition, Groovy (`http://groovy.codehaus.org`) provides a simplified programming model and supports the implementation of domain-specific languages (DSLs) that make the application code easier to read and maintain.

One other important concept that these scripting languages bring to Java developers is closures (which we discuss in more detail later in this chapter). Simply speaking, a *closure* is a piece (or block) of code wrapped in an object. Like a Java method, it's executable and can receive parameters and return objects and values. In addition, it's a normal object that can be passed with a reference around your application, like any POJO in Java.

In this chapter, we cover some of the main concepts behind scripting languages, with the primary focus on Groovy; you'll see how the Spring Framework can work with scripting languages seamlessly to provide specific functionality to Spring-based applications. Specifically, this chapter covers the following topics:

- *Scripting support in Java*: In JCP, JSR-223 (Scripting for the Java Platform) enables the support of scripting languages in Java; it has been available in Java since SE 6. We provide an overview of scripting support in Java.
- *Groovy*: We present a high-level introduction to the Groovy language, which is one of the most popular scripting languages being used with Java.
- *Using Groovy with Spring*: The Spring Framework provides comprehensive support for scripting languages. Since version 3.1, out-of-the-box support for Groovy, JRuby, and BeanShell is provided.

© Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho 2017

I. Cosmina et al., *Pro Spring 5*, https://doi.org/10.1007/978-1-4842-2808-1_14

This chapter is not intended to serve as a detailed reference on using scripting languages. Each language has one or more books of their own that discuss their design and usage in detail. The main objective of this chapter is to describe how the Spring Framework supports scripting languages, with a sound example showing the benefits of using a scripting language in addition to Java in a Spring-based application.

Working with Scripting Support in Java

Starting with Java 6, the Scripting for the Java Platform API (JSR-223) is bundled into the JDK. Its objective is to provide a standard mechanism for running logic written in other scripting languages on the JVM. Out of the box, JDK 6 comes bundled with the engine called Mozilla Rhino, which is able to evaluate JavaScript programs. This section introduces you to the JSR-223 support in JDK 6.

In JDK 6, the scripting support classes reside in the `javax.script` package. First let's develop a simple program to retrieve the list of script engines. The following code snippet shows the class content:

```
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);
      });
   }
}
```

An instance of the `ScriptEngineManager` class is created, which will discover and maintain a list of engines (in other words, classes implementing the `javax.script.ScriptEngine` interface) from the classpath. Then, a list of `ScriptEngineFactory` interfaces is retrieved by calling the `ScriptEngineManager.getEngineFactories()` method. The `ScriptEngineFactory` interface is used to describe and instantiate script engines. From each `ScriptEngineFactory` interface, information about the scripting language support can be retrieved. Running the program may produce varied output, depending on your setup, and you should see something similar to the following in your console:

```
INFO: Engine name: AppleScriptEngine Language: AppleScript Version: 2.5
INFO: Engine name: Oracle Nashorn Language: ECMAScript Version: ECMA - 262 Edition 5.1
```

Let's write a simple program to evaluate a basic JavaScript expression. The program is shown here:

```
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);
      }
   }
}
```

Here an instance of the ScriptEngine interface is retrieved from the ScriptEngineManager class, using the name JavaScript. Then, the ScriptEngine.eval() method is called, passing in a String argument, which contains a JavaScript expression. Note that the argument can also be a java.io.Reader class, which can read JavaScript from a file.

Running the program produces the following result:

```
Hello JavaScript in Java
```

This should give you an idea of how to run scripts in Java. However, it's not of much interest to just dump some output using another language. In the next section, we introduce Groovy, a powerful and comprehensive scripting language.

Introducing Groovy

Started by James Strachan in 2003, the main objective of Groovy is to provide an agile and dynamic language for the JVM, with features inspired from other popular scripting languages including Python, Ruby, and Smalltalk. Groovy is built on top of Java, extends Java, and addresses some of the shortcomings in Java.

In the following sections, we discuss some main features and concepts behind Groovy and how it supplements Java to address specific application needs. Note that many features mentioned here also are available in other scripting languages (for example, Scala, Erlang, Python, and Clojure).

Dynamic Typing

One main difference between Groovy (and many other scripting languages) and Java is the support of dynamic typing of variables. In Java, all properties and variables should be statically typed. In other words, the type should be provided with the `declare` statement. However, Groovy supports the dynamic typing of variables. In Groovy, dynamic typing variables are declared with the keyword `def`.

Let's see this in action by developing a simple Groovy script. The file suffix of a Groovy class or script is `groovy`. The following code snippet shows a simple Groovy script with dynamic typing in action:

```
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
```

This Groovy script can be run directly in an IDE, executed without compilation (Groovy provides a command-line tool called `groovy` that can execute Groovy scripts directly) or can be compiled to Java bytecode and then executed just like other Java classes. Groovy scripts don't require a `main()` method for execution. Also, a class declaration that matches the file name is not required.

In this example, a class `Singer` is defined, with the properties set to dynamic typing with the `def` keyword. Three properties are declared. Then, the `toString()` method is overridden with a closure that returns a string.

Next, two instances of the `Singer` object are constructed, with shorthand syntax provided by Groovy to define the properties. For the first `Singer` object, the `firstName` attribute is supplied with a `String`, while an integer is provided for the second `Singer` object. Finally, the `println` statement (the same as calling `System.out.println()` method) is used for printing the two singer objects. To show how Groovy handles dynamic typing, two `println` statements are defined to print the output for the operation `firstName + 39`. Note that in Groovy, when passing an argument to a method, the parentheses are optional.

Running the program produces the following output:

```
John,Mayer,Sun Oct 16 00:00:00 EET 1977
39,Mayer,Sun Oct 16 00:00:00 EET 1977
John39
78
```

From the output, you can see that since firstName is defined with dynamic typing, the object constructs successfully when passing in either a String or an Integer as the type. In addition, in the last two println statements, the add operation was correctly applied to the firstName property of both objects. In the first scenario, since firstName is a String, the string 39 is appended to it. For the second scenario, since firstName is an integer, the integer 39 is added to it, resulting in 78.

Dynamic typing support of Groovy provides greater flexibility for manipulating class properties and variables in application logic.

Simplified Syntax

Groovy also provides simplified syntax so that the same logic in Java can be implemented in Groovy with less code. Some of the basic syntax is as follows:

- A semicolon is not required for ending a statement.
- In methods, the return keyword is optional.
- All methods and classes are public by default. So, you don't need to declare the public keyword for method declaration, unless required.
- Within a class, Groovy will automatically generate the getter/setter methods for the declared properties. So in a Groovy class, you just need to declare the type and name (for example, String firstName or def firstName), and you can access the properties in any other Groovy/Java classes by using the getter/setter methods automatically. In addition, you can simply access the property without the get/set prefix (for example, singer.firstName = 'John'). Groovy will handle them for you intelligently.

Groovy also provides simplified syntax and many useful methods to the Java Collection API. The following code snippet shows some of the commonly used Groovy operations for list manipulation:

```
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']
```

The previous code shows only a small portion of the features that Groovy offers. For a more detailed description, please refer to the Groovy online documentation at http://groovy.codehaus.org/JN1015-Collections.

Closure

One of the most important features that Groovy adds to Java is the support of closures. A closure allows a piece of code to be wrapped as an object and to be passed freely within the application. Closure is a powerful feature that enables smart and dynamic behavior. The addition of closure support to the Java language has

been requested for a long time. JSR-335 (Lambda Expressions for the Java Programming Language), which aims to support programming in a multicore environment by adding closures and related features to the Java language, has been added to Java 8 and supported by the new Spring Framework 4.

The following code snippet shows a simple example of using closures (the file name is `Runner.groovy`) in Groovy:

```
def names = ['John', 'Clayton', 'Mayer']

names.each {println 'Hello: ' + it}
```

Here a list is declared. Then, the convenient `each()` method is used for an operation that will iterate through each item in the list. The argument to the `each()` method is a closure, which is enclosed in curly braces in Groovy. As a result, the logic in the closure will be applied to each item within the list. Within the closure is a special variable used by Groovy to represent the item currently in context. So, the closure will prefix each item in the list with the string `"Hello: "` and then print it. Running the script produces the following output:

```
Hello: John
Hello: Clayton
Hello: Mayer
```

As mentioned, a closure can be declared as a variable and used when required. Another example is shown here:

```
def map = ['a': 10, 'b': 50]

Closure square = {key, value -> map[key] = value * value}

map.each square

println map
```

In this example, a map is defined. Then, a variable of type `Closure` is declared. The closure accepts the key and value of a map's entry as its arguments, and the logic calculates the square of the value of the key. Running the program produces the following output:

```
[a:100, b:2500]
```

This is just a simple introduction to closures. In the next section, we will develop a simple rule engine by using Groovy and Spring; closures are used also. For a more detailed description of using closures in Groovy, please refer to the online documentation at http://groovy.codehaus.org/JN2515-Closures.

Using Groovy with Spring

The main benefit that Groovy and other scripting languages bring to Java-based applications is the support of dynamic behavior. By using a closure, business logic can be packaged as an object and passed around the application like any other variables.

Another main feature of Groovy is the support for developing DSLs by using its simplified syntax and closures. As the name implies, a DSL is a language targeted for a particular domain with very specific goals in design and implementation. The objective is to build a language that is understandable not only by the

developers but the business analysts and users as well. Most of the time, the domain is a business area. For example, DSLs can be defined for customer classification, sales charge calculation, salary calculation, and so on.

In this section, we demonstrate how to use Groovy to implement a simple rule engine with Groovy's DSL support. The implementation is referencing the sample from the excellent article on this topic at www.pleus.net/articles/grules/grules.pdf, with modifications. In addition, we discuss how Spring's support of refreshable beans enables the update of the underlying rules on the fly without the need to compile, package, and deploy the application.

In this sample, we implement a rule used for classifying a specific singer into different categories based on their age, which is calculated based on their date-of-birth property.

Developing the Singer Domain

As mentioned, a DSL targets a specific domain, and most of the time the domain is referring to some kind of business data. For the rule we are going to implement, it was designed to be applied to the domain of singer information.

So, the first step is to develop the domain object model we want the rule to apply to. This sample is simple and contains only one `Singer` entity class, as shown next. Note that it's a POJO class, like those we used in previous chapters.

```
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;

    ... //getters and setter

    @Override
    public String toString() {
    return "Singer - Id: " + id + ", First name: " + firstName
        + ", Last name: " + lastName + ", Birthday: " + birthDate
        + ", Age category: " + ageCategory;
    }
}
```

Here the `Singer` class consists of simple singer information. For the `ageCategory` property, we want to develop a dynamic rule that can be used to perform classification. The rule will calculate the age based on the `birthDate` property and then assign the `ageCategory` property (for example, kid, youth, or adult) based on the rule.

Implementing the Rule Engine

The next step is to develop a simple rule engine for applying the rules on the domain object. First we need to define what information a rule needs to contain. The following code snippet shows the Rule class, which is a Groovy class (the file name is 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()
}
```

Each rule has several properties. The conditions property defines the various conditions that the rule engine should check for with the domain object under processing. The actions property defines the actions to take when a match on the condition is hit. The parameters property defines the behavior of the rule, which is the outcome of the action for different conditions. Finally, the singlehit property defines whether the rule should end its execution immediately whenever a match of condition is found.

The next step is the engine for rule execution. The following code snippet shows the RuleEngine interface (note it's a Java interface):

```
package com.apress.prospring5.ch14;

public interface RuleEngine {
    void run(Rule rule, Object object);
}
```

The interface defines only a method run(), which is to apply the rule to the domain object argument.

We will provide the implementation of the rule engine in Groovy. The following code snippet shows the Groovy class RuleEngineImpl (the file name is 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
        }
        }
        }
    }
    }
}
```

First RuleEngineImpl implements the RuleEngine Java interface, and Spring's annotation is applied like any other POJO. Within the run() method, the parameters defined in the rule are passed into a closure for processing one by one. For each parameter (which is a list of values), the conditions (each condition is a closure) are checked one by one with the corresponding item within the parameter's list and the domain object. The success indicator becomes true only when all the conditions result in a positive match. In this case, the actions (each action is a closure too) defined in the rule will be performed on the object, with the corresponding value within the parameter's list. Finally, if a match is found for a specific parameter and the singlehit variable is true, the rule execution will be stopped and will exit immediately.

To allow the retrieval of a rule in a more flexible way, let's define a RuleFactory interface, as shown next. Note that it's a Java interface.

```
package com.apress.prospring5.ch14;

public interface RuleFactory {
    Rule getAgeCategoryRule();
}
```

Since there is only one rule for an age category classification for singers, the interface defines only a single method for retrieving the rule.

To make the rule engine transparent to the consumer, let's develop a simple service layer to wrap it up. The following code snippet shows the SingerService interface and the SingerServiceImpl class, respectively. Note that they are Java implementations.

```
//SingerService.java
package com.apress.prospring5.ch14;

public interface SingerService {
    void applyRule(Singer singer);
}
```

```
//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);
    }
}
```

As you can see, the required Spring beans are autowired into the service implementation class. In the `applyRule()` method, the rule is obtained from the rule factory and then applied to the `Singer` object. The result is that the `ageCategory` property for the `Singer` will be derived based on the rule's defined conditions, actions, and parameters.

Implementing the Rule Factory as a Spring Refreshable Bean

Now we can implement the rule factory and the rule for age category classification. We want to be able to update the rule on the fly and have Spring check for its changes and pick it up to apply the latest logic. The Spring Framework provides wonderful support for Spring beans written in scripting languages, called *refreshable beans*. We will see how to configure a Groovy script as a Spring bean and instruct Spring to refresh the bean on a regular interval later. First let's see the implementation of the rule factory in Groovy. To allow dynamic refresh, we put the class into an external folder so it can be modified. We will call this folder `rules`. The `RuleFactoryImpl` class (which is a Groovy class, with the name `RuleFactoryImpl.groovy`) will be placed into this folder. The following code snippet shows the class content:

```
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
    }
}
```

The class implements the RuleFactory interface, and the getAgeCategoryRule() method is implemented to provide the rule. Within the rule, a Closure called age is defined to calculate the age based on the birthDate property (which is of JodaTime's DateTime type) of a Singer object.

Within the rule, two conditions are defined. The first one is to check whether the age of a singer is larger than or equal to the provided parameter value, while the second check is for the smaller-than or equal-to condition.

Then, one action is defined to assign the value provided in the parameter to the ageCategory property of the Singer object.

The parameters define the values for both condition checking and action. For example, in the first parameter, it means that when the age is between 0 and 10, then the value Kid will be assigned to the ageCategory property of the Singer object, and so on. So, for each parameter, the first two values will be used by the two conditions to check for age range, while the last value will be used for assigning the ageCategory property.

The next step is to define the Spring ApplicationContext. The following configuration snippet shows the configuration file (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>
```

The configuration is simple. For defining Spring beans in a scripting language, we need to use lang-namespace. Then, the <lang:groovy> tag is used to declare a Spring bean with a Groovy script. The script-source attribute defines the location of the Groovy script that Spring will load from. For the refreshable bean, the attribute refresh-check-delay should be provided. In this case, we supplied the value of 5000 ms, which instructs Spring to check for file changes if the elapsed time from the last invocation is greater than five seconds. Note that Spring will not check the file every five seconds. Instead, it will check the file only when the corresponding bean is invoked.

Testing the Age Category Rule

Now we are ready to test the rule. The testing program is shown next, which is a Java class:

```
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();
    }
}
```

Upon initialization of Spring's GenericXmlApplicationContext, an instance of the Singer object is constructed. Then, the instance of the SingerService interface is obtained to apply the rule onto the Singer object and then output the result to the console. The program will be paused for user input, before the second application of the rule. During the pause, we can then modify the RuleFactoryImpl.groovy class so that Spring will refresh the bean and we can see the changed rule in action.

Running the testing program produces the following output:

```
00:34:24.814 [main] INFO  c.a.p.c.RuleEngineImpl - Executing rule
00:34:24.822 [main] INFO  c.a.p.c.RuleEngineImpl - Condition Param index: 0
00:34:24.851 [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
00:34:24.858 [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: 2
00:34:24.870 [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: Adult
```

From the logging statement in the output, since the age of the singer is 39, you can see that the rule will find a matching in the third parameter (in other words, [21,40,'Adult']). As a result, ageCategory is set to Adult.

Now the program is paused, so let's change the parameters within the RuleFactoryImpl.groovy class. You can see this modification in the following code snippet:

```
rule.parameters=[
   [0,10,'Kid'],
   [11,20,'Youth'],
   [21,30,'Adult'],
   [31,60,'Middle-aged'],
   [61,120,'Old']
]
```

Change and save the file as indicated. Now press Enter in the console area to trigger the second application of the rule to the same object. After the program continues, the following output is produced:

```
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 - 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
```

In the previous output, you can see that the rule execution stops at the fourth parameter (in other words, `[31,60,'Middleaged']`), and as a result, the value `Middle-aged` is assigned to the `ageCategory` property.

If you take a look at the article that was referred to when we prepared this sample (`http://pleus.net/articles/grules/grules.pdf`), it also shows how the rule parameter can be externalized into a Microsoft Excel file, so users can prepare and update the parameter file by themselves.

Of course, this rule is a simple one, but it shows how a scripting language such as Groovy can help supplement Spring-based Java EE applications in specific areas, for example, using a rule engine with DSL.

You may be asking, "Is it possible to go one step further by storing the rule into the database and then have Spring's refreshable bean feature detect the change from the database?" This can help further simplify the maintenance of the rule by providing a front end for users (or administrators) to update the rule into the database on the fly, instead of uploading the file.

Actually, there is a JIRA issue in the Spring Framework that discusses this (`https://jira.springsource.org/browse/SPR-5106`). Stay tuned with this feature. In the meantime, providing a user front end to upload the rule class is also a workable solution. Of course, extreme care should be taken in this case, and the rule should be tested thoroughly before you upload it to the production environment.

Inlining Dynamic Language Code

Not only can dynamic language code be executed from external source files, but you can inline this code directly into your bean configuration. While this practice may be useful in some scenarios such as quick proof of concepts and so on, from a maintainability standpoint, it would not be good practice to build an entire application by using this method. Using the previous `Rule` engine as an example, let's delete the file

RuleEngineImpl.groovy and move that code into an inline bean definition (in file app-context.xml), as shown in the following code snippet:

```
<?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>
```

As you can see, we added the lang:groovy tag with an ID of ruleFactory representing the bean name. We then used the lang:inline-script tag to encapsulate the Groovy code from RuleFactoryImpl.groovy. Surrounding the Groovy code is a CDATA tag to avoid the code being parsed by the XML parser. Now with that in place, go ahead and run the rule engine sample again. As you can see, it works the same way, except we inlined the Groovy code directly into the bean definition rather than having it reside in an external file. Using the code from RuleFactoryImpl.groovy was also done intentionally to show how unwieldy an application can become when inlining large amounts of code.

Summary

In this chapter, we covered how to use scripting languages in Java applications and demonstrated how the Spring Framework's support of scripting languages can help provide dynamic behavior to the application.

First we discussed JSR-223 (Scripting for the Java Platform), which was built into Java 6 and supports the execution of JavaScript out of the box. Then, we introduced Groovy, a popular scripting language within the Java developer communities. We also demonstrated some of its main features when compared to the traditional Java language.

Finally, we discussed the support of scripting languages in the Spring Framework. We saw it in action by designing and implementing a simple rule engine using Groovy's DSL support. We also discussed how the rule can be modified and have the Spring Framework pick up the changes automatically by using its refreshable bean feature, without the need to compile, package, and deploy the application. Additionally, we showed how to inline Groovy code directly into a configuration file to define a bean's implementation code.

CHAPTER 15

Application Monitoring

A typical JEE application contains a number of layers and components, such as the presentation layer, service layer, persistence layer, and back-end data source. During the development stage, or after the application had been deployed to the quality assurance (QA) or production environment, we will want to ensure that the application is in a healthy state without any potential problems or bottlenecks.

In a Java application, various areas may cause performance problems or overload server resources (such as CPU, memory, or I/O). Examples are inefficient Java code, memory leaks (for example, Java code that keeps allocating new objects without releasing the reference and prevents the underlying JVM from freeing up the memory during the garbage collection process), miscalculated JVM parameters, miscalculated thread pool parameters, too generous data source configurations (for example, too many concurrent database connections allowed), improper database setup, and long-running SQL queries.

Consequently, we need to understand an application's runtime behavior and identify whether any potential bottlenecks or problems exist. In the Java world, a lot of tools can help monitor the detailed runtime behavior of JEE applications. Most of them are built on top of the Java Management Extensions (JMX) technology.

In this chapter, we present common techniques for monitoring Spring-based JEE applications. Specifically, this chapter covers the following topics:

- *Spring support of JMX*: We discuss Spring's comprehensive support of JMX and demonstrate how to expose Spring beans for monitoring with JMX tools. In this chapter, we show how to use the `jvisualvm` Java executable (`https: //visualvm.github.io/?Java_VisualVM`) as the application-monitoring tool.
- *Monitoring Hibernate statistics*: Hibernate and many other packages provide support classes and infrastructure for exposing the operational status and performance metrics using JMX. We show how to enable the JMX monitoring of those commonly used components in Spring-powered JEE applications.
- *Spring Boot JMX support*: Spring Boot provides a starter library for JMX support that comes with full default configuration out of the box.

Remember that this chapter is not intended to be an introduction to JMX, and a basic understanding of JMX is assumed. For detailed information, please refer to Oracle's online resource at `http://oracle.com/technetwork/java/javase/tech/javamanagement-140525.html`.

JMX Support in Spring

In JMX, the classes that are exposed for JMX monitoring and management are called *managed beans* (generally referred to as *MBeans*). The Spring Framework supports several mechanisms for exposing MBeans. This chapter focuses on exposing Spring beans (which were developed as simple POJOs) as MBeans for JMX monitoring.

© Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho 2017
I. Cosmina et al., *Pro Spring 5*, https://doi.org/10.1007/978-1-4842-2808-1_15

In the following sections, we discuss the procedure for exposing a bean containing application-related statistics as an MBean for JMX monitoring. Topics include implementing the Spring bean, exposing the Spring bean as an MBean in Spring `ApplicationContext`, and using VisualVM to monitor the MBean.

Exporting a Spring Bean to JMX

As an example, we will use the REST sample from Chapter 12. Review that chapter for the sample application code or jump directly to the book's source companion, which provides the source code we will use to build upon. With the JMX additions, you want to expose the count of the singers in the database for JMX monitoring purposes. So, let's implement the interface and the class, as shown here:

```
//AppStatistics.java
package com.apress.prospring5.ch15;

public interface AppStatistics {
    int getTotalSingerCount();
}

//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();
    }
}
```

In this example, a method is defined to retrieve the total count of singer records in the database. To expose the Spring bean as JMX, we need to add configuration in Spring's `ApplicationContext`. The configuration of this Spring-secured web application was covered in Chapter 12. Now we must add to it two infrastructure beans of types `MBeanServer` and `MBeanExporter` to enable support for the JMX management beans.

```
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 {
    //other Web infrastructure specific beans
    ...
```

```
    @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;
    }
}
```

First, you declare the bean for the POJO with statistics, `AppStatisticsImpl`, that we want to expose. Second, you declare the `jmxExporter` bean with the implementation class `MBeanExporter`.

The `MBeanExporter` class is the core class within the Spring Framework's support for JMX. It's responsible for registering Spring beans with a JMX MBean server (a server that implements JDK's `javax.management.MBeanServer` interface, which exists in most commonly used web and JEE containers, such as Tomcat and WebSphere). When exposing a Spring bean as an MBean, Spring will attempt to locate a running `MBeanServer` instance within the server and register the MBean with it. For example, with Tomcat, an `MBeanServer` instance will be created automatically, so no additional configuration is required.

Within the `jmxExporter` bean, the property `beans` defines the Spring beans we want to expose. It's a `Map`, and any number of MBeans can be specified here. In this case, we would like to expose the `appStatisticsBean` bean, which contains information about the singer application we want to show to administrators. For the MBean definition, the key will be used as the `ObjectName` value (the `javax.management.ObjectName` class in JDK) for the Spring bean referenced by the corresponding entry value. In the previous configuration, `appStatisticsBean` will be exposed under `ObjectName bean:name=Prospring5SingerApp`. By default, all public properties of the bean are exposed as attributes, and all public methods are exposed as operations.

Now the MBean is available for monitoring via JMX. Let's proceed to set up VisualVM and use its JMX client for monitoring purposes.

Using Java VisualVM for JMX Monitoring

VisualVM is a useful tool that can help in monitoring Java applications in various aspects. It's a free tool and resides under the `bin` folder in the JDK installation folder. You can also download a stand-alone version from the project web site.[1] We will use the JDK installation version in this chapter.

VisualVM uses a plug-in system to support various monitoring functions. To support monitoring MBeans of Java applications, we need to install the MBeans plug-in. To install the plug-in, follow these steps:

1. From VisualVM's menu, choose Tools ➤ Plug-ins.
2. Click the Available Plug-ins tab.
3. Click the Check for Newest button.
4. Select the plug-in VisualVM-MBeans and then click the Install button.

[1] At the time of this writing, the current version of Java VisualVM is 1.3.9. You can find it here: `http://visualvm.java.net/download.html`.

Figure 15-1 depicts the Plugins dialog. After completing the installation, verify that Tomcat is up and that the sample application is running. Then in VisualVM's left Applications view, you should be able to see that the Tomcat process is running.

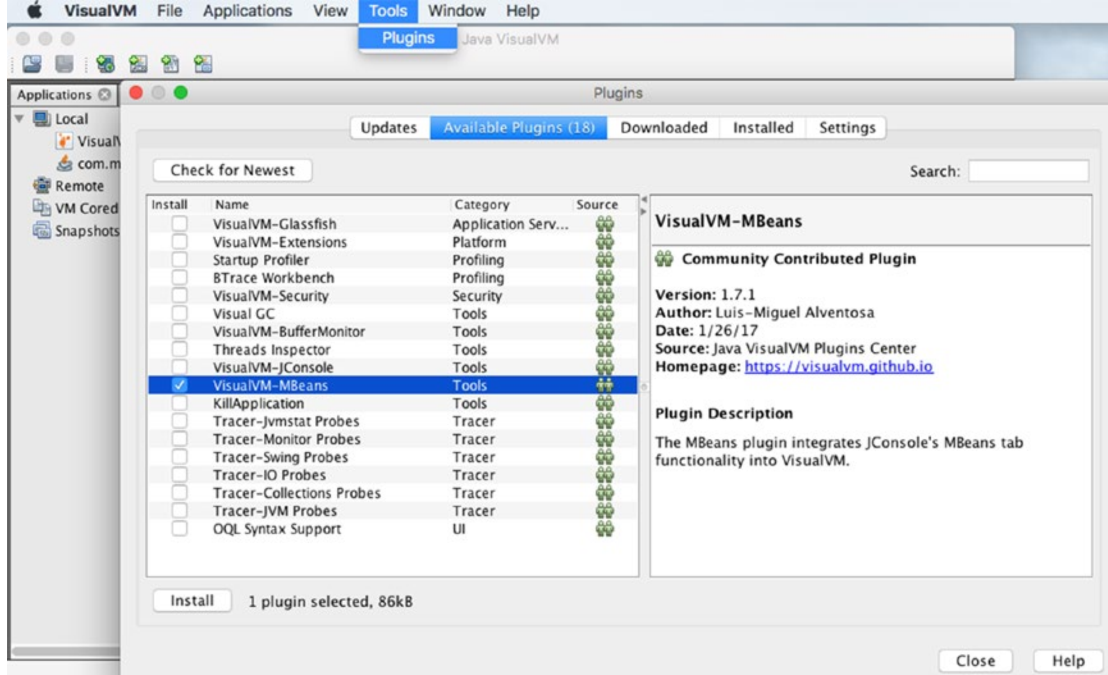

Figure 15-1. *Overview of global transactions with JTA*

By default, VisualVM scans for the Java applications that are running on the JDK platform. Double-clicking the desired node brings up the monitoring screen.

After the installation of the VisualVM-MBeans plug-in, you will able to see the MBeans tab. Clicking this tab shows the available MBeans. You should see the node called `bean`. When you expand it, it will show the `Prospring5SingerApp` MBean that was exposed.

On the right side, you will see the method that we implemented in the bean, with the attribute `TotalSingerCount` (which was automatically derived by the `getTotalSingerCount()` method within the bean). The value is 3, corresponding to the number of records we added in the database at application startup. In a regular application, this number would change based on the number of singers added during the application runtime.

Figure 15-2 depicts the MBeans window with the `Prospring5SingerApp` MBean exposed.

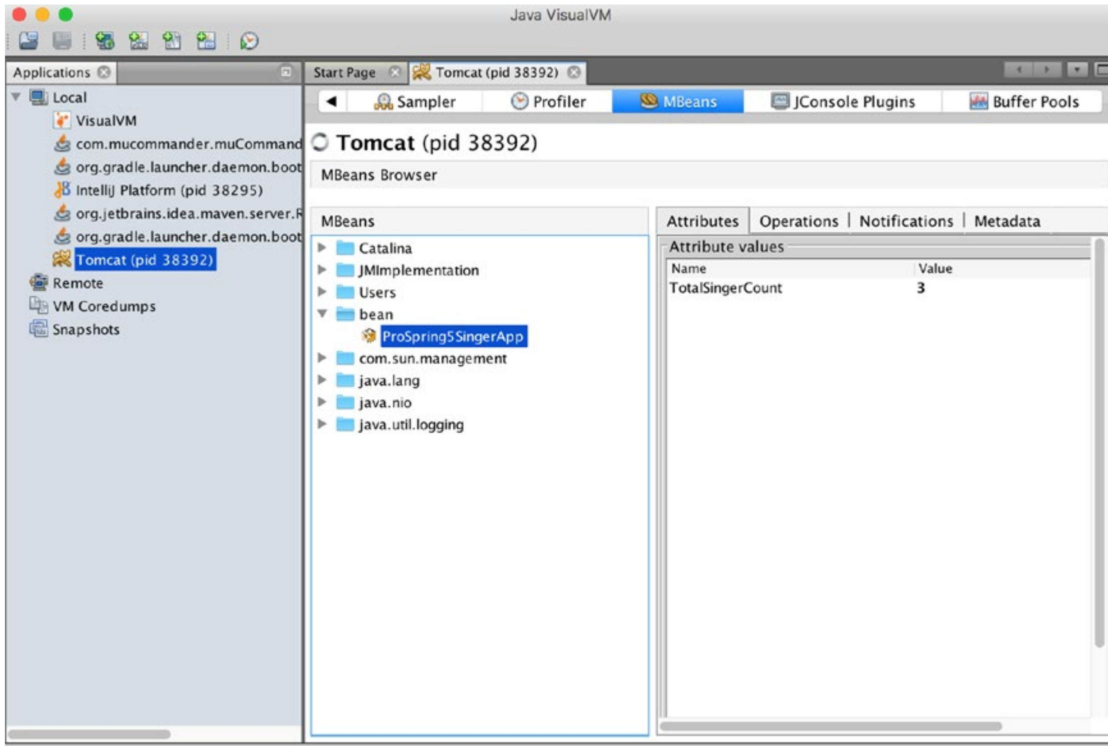

Figure 15-2. *The Prospring5SingerApp MBean exposed in VisualVM*

Monitoring Hibernate Statistics

Hibernate also supports the maintenance and exposure of persistence-related metrics to JMX. To enable this, in the JPA configuration, add three more Hibernate properties, as shown here:

```
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
//using components that were introduced in Chapter 12 project
@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;
    }
    ...
}
```

The property `hibernate.jmx.enabled` is used to enable Hibernate JMX behavior.

The property `hibernate.generate_statistics` instructs Hibernate to generate statistics for its JPA persistence provider, while the property `hibernate.session_factory_name` defines the name of the session factory required by the Hibernate statistics `MBean`.

Finally, we need to add the MBean into Spring's `MBeanExporter` configuration. The following configuration snippet shows the updated `MBean` configuration that we created earlier in the `WebConfig` class. The `CustomStatistics` class is a replacement of `org.hibernate.jmx.StatisticsService` that is no longer part of Hibernate 5.[2]

```
package com.apress.prospring5.ch15.init;

...

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = {"com.apress.prospring5.ch15"})
public class WebConfig implements WebMvcConfigurer {
    ...
    // JMX beans
```

[2]The code is still available on GitHub at https://github.com/manuelbernhardt/hibernate-core/blob/master/hibernate-core/src/main/java/org/hibernate/jmx/StatisticsService.java if you want to enrich the provided implementation.

```
    @Bean AppStatistics appStatisticsBean() {
        return new AppStatisticsImpl();
    }

    @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;
    }
}
```

The `statisticsBean()` method is declared, with Hibernate's `org.hibernate.stat.Statistics` implementation as the core component. This is how Hibernate supports exposing statistics to JMX.

Now the Hibernate statistics are enabled and available via JMX. Reload the application and refresh VisualVM; you will be able to see the Hibernate statistics MBean. Clicking the node displays the detail statistics on the right side. Note that for the information that is not of a Java primitive type (for example, a `List`), you can click in the field to expand it and show the content.

In VisualVM, you can see many other metrics, such as `EntityNames`, `SessionOpenCount`, `SecondCloseCount`, and `QueryExecutionMaxTime`. Those figures are useful for you to understand the persistence behavior within your application and can assist you in troubleshooting and performance-tuning exercises.

JMX with Spring Boot

Migrating the previous application to Spring Boot is easy, and the dependencies are provided and automatically configured. For JMX, there is no starter dependency needed, but you can add `spring-boot-starter-actuator.jar` as a dependency; it might help to monitor a Spring application in a smart editor and display the beans, health, and mappings in the application if you use the Spring-specific plug-in.

This application will be a web application without an interface (because that is a subject for the following chapter, Chapter 16), with an in-memory database and a full-blown JTA configuration using Atomikos. As this implementation was introduced in previous chapters, the focus will be here on the MBeans.

Let's upgrade the `AppStatisticsImpl` class by using `@ManagedResource` to register instances of this class with a JMX server. This is practical because Spring Boot by default will create an `MBeanServer` with a bean ID of `mbeanServer` and expose any of the beans that are annotated with Spring JMX annotations

(@ManagedResource, @ManagedAttribute, @ManagedOperation). The upgraded version of AppStatisticsImpl using all the previously mentioned annotations is shown here:

```
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";
    }
}
```

By default Spring Boot will expose management endpoints as JMX MBeans under the org.springframework.boot domain. In the previously depicted code snippet, the @ManagedResource annotation has an attribute called objectName, and its value represents the domain and name of the MBean. Because we wanted to easily find in VisualVM the managed beans that were created explicitly (Spring Boot provides its own autoconfigured MBeans for internal monitoring), we used the domain jmxDemo.

The @ManagedAttribute annotation is used to expose the given bean property as a JMX attribute. @ManagedOperation is used to expose a given method as a JMX operation. Because the two methods shown earlier are annotated differently, they will be shown in different tabs in VisualVM. The result of calling getTotalSingerCount will be visible on the Attributes tab. On the Operations tab, both methods will be depicted as clickable buttons to be called on the spot. The string provided as descriptions in each of the annotations can be viewed on the Metadata tab.

Figure 15-3 depicts the MBeans window with the Prospring5SingerApp MBean exposed under the jmxDemo domain. Under it, you can see the org.springframework.boot domain.

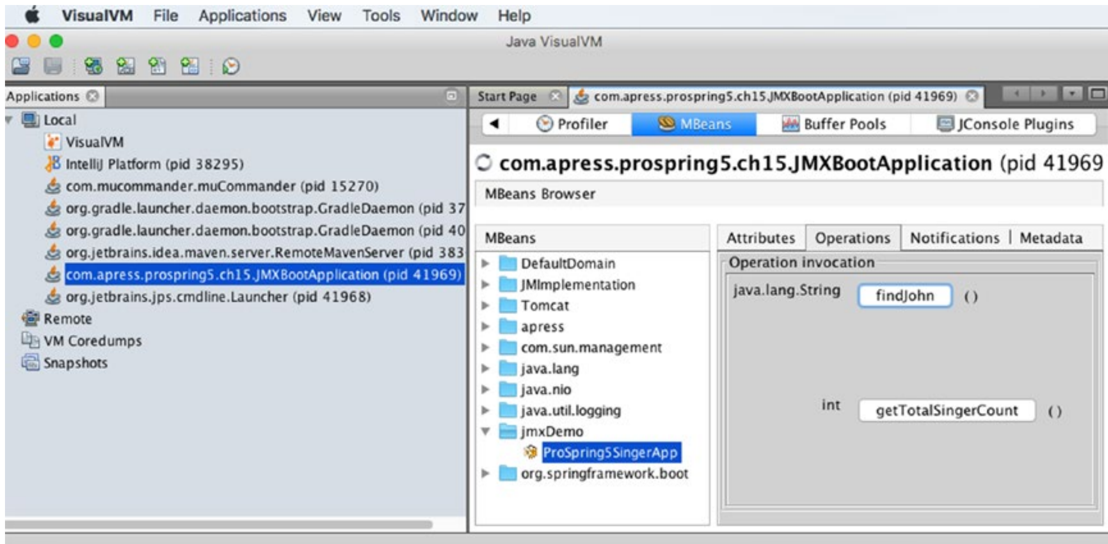

Figure 15-3. The Prospring5SingerApp MBean exposed in VisualVM

No, that's not it, though! Support of JMX must be enabled. This is done by annotating a configuration class with @EnableMBeanExport. This annotation enables the default exporting of all standard MBeans from the Spring context, as well as all @ManagedResource annotated beans. Basically, this annotation is what tells Spring Boot to create an MBeanExporter bean with the name of mbeanExporter. The configuration class for this Spring Boot application is shown here:

```
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();
    }
}
```

Now, we can say that's it, because this is all there is to say when using JMX in a Spring Boot application!

Summary

In this chapter, we covered the high-level topics of monitoring a Spring-powered JEE application. First, we discussed Spring's support of JMX, the standard in monitoring Java applications. We discussed implementing custom MBeans for exposing application-related information, as well as exposing statistics of common components such as Hibernate. Second, we showed how JMX can be used in a Spring Boot application and how special Spring is.

CHAPTER 16

Web Applications

In an enterprise application, the presentation layer critically affects the level of acceptance that users give the application. The presentation layer is the front door into your application. It lets users perform business functions provided by the application, as well as presents a view of the information that is being maintained by the application. How the user interface performs greatly contributes to the success of the application.

Because of the explosive growth of the Internet (especially these days), as well as the rise of different kinds of devices that people are using, developing an application's presentation layer is a challenging task. The following are some of the major considerations when developing web applications:

- *Performance*: Performance is always the top requirement of a web application. If users choose a function or click a link and it takes a long time to execute (in the world of the Internet, three seconds is like a century!), users will definitely not be happy with the application.
- *User-friendliness*: The application should be easy to use and easy to navigate, with clear instructions that don't confuse the user.
- *Interactivity and richness*: The user interface should be highly interactive and responsive. In addition, the presentation should be rich in terms of visual presentation, such as charting, a dashboard type of interface, and so on.
- *Accessibility*: Nowadays, users require that the application is accessible from anywhere via any device. In the office, they will use their desktop for accessing the application. On the road, users will use various mobile devices (including laptops, tablets, and smartphones) to access the application.

Developing a web application to fulfill the previous requirements is not easy, but they are considered mandatory for business users. Fortunately, many new technologies and frameworks have been developed to address those needs. Many web application frameworks and libraries—such as Spring MVC (Spring Web Flow), Struts, Tapestry, Java Server Faces (JSF), Google Web Toolkit (GWT), jQuery, and Dojo, to name a few—provide tools and rich component libraries that can help you develop highly interactive web front ends. In addition, many frameworks provide tools or corresponding widget libraries targeting mobile devices including smartphones and tablets. The rise of the HTML5 and CSS3 standards and the support of these latest standards by most web browsers and mobile device manufacturers also help ease the development of web applications that need to be available anywhere, from any device.

In terms of web application development, Spring provides comprehensive and intensive support. The Spring MVC module provides a solid infrastructure and Model View Controller (MVC) framework for web application development. When using Spring MVC, you can use various view technologies (for example, JSP or Velocity). In addition, Spring MVC integrates with many common web frameworks and toolkits (for example, Struts and GWT). Other Spring projects help address specific needs for web applications. For example, Spring MVC, when combined with the Spring Web Flow project and its Spring Faces module,

© Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho 2017

I. Cosmina et al., *Pro Spring 5*, https://doi.org/10.1007/978-1-4842-2808-1_16

provides comprehensive support for developing web applications with complex flows and for using JSF as the view technology. Simply speaking, there are many choices out there in terms of presentation layer development. This chapter focuses on Spring MVC and discusses how we can use the powerful features provided by Spring MVC to develop highly performing web applications. Specifically, this chapter covers the following topics:

- *Spring MVC*: We discuss the main concepts of the MVC pattern and introduce Spring MVC. We present Spring MVC's core concepts, including its `WebApplicationContext` hierarchy and the request-handling life cycle.
- *i18n, locale, and theming*: Spring MVC provides comprehensive support for common web application requirements including i18n (internationalization), locale, and theming. We discuss how to use Spring MVC to develop web applications that support those requirements.
- *View and Ajax support*: Spring MVC supports many view technologies. In this chapter, we focus on using JavaServer Pages (JSP) and Tiles as the view part of the web application. On top of JSP, JavaScript will be used to provide the richness part. There are many outstanding and popular JavaScript libraries, such as jQuery and Dojo. In this chapter, we focus on using jQuery, with its subproject jQuery UI library that supports the development of highly interactive web applications.
- *Pagination and file upload support*: When showing how to develop the samples in this chapter, we discuss how you can use Spring Data JPA and the front-end jQuery component to provide pagination support when browsing grid-based data. In addition, we cover how to implement file uploading in Spring MVC. Instead of integration with Apache Commons File Upload, we discuss how we can use Spring MVC with the Servlet 3.1 container's built-in multipart support for file upload.
- *Security*: Security is a big topic in web applications. We discuss how we can use Spring Security to help protect the application and handle logins and logouts.

Implementing the Service Layer for Samples

In the service layer for this chapter, we will still use the singer application as the sample. In this section, we discuss the data model and the implementation of the service layer that will be used throughout this chapter.

Using a Data Model for the Samples

You will use a simple data model for the samples in this chapter; it contains only a single `SINGER` table for storing singer information. The following SQL snippet shows the script for schema creation (`schema.sql`):

```
DROP TABLE IF EXISTS SINGER;

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)
);
```

As you can see, the SINGER table stores only a few basic fields of a singer's information. One thing worth mentioning is the PHOTO column, of the binary large object (BLOB) data type, which will be used to store the photo of a singer using file upload. To create the table you won't use this SQL script; instead, Hibernate will generate the SQL necessary to create the table based on the configuration of the Singer entity that is depicted by the following code snippet:

```
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;
 }
}
```

If a population script was used to populate the SINGER table, the script would look like this:

```
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';
```

But in the official code samples you will find all the previous data being inserted via the DBInitializer class. This time, we need more testing data so we can show you the pagination support later.

Implementing the DAO Layer

The entity class, the repo, and the database configuration make up the application layer called dao, which is in charge of database objects.

The entity class was introduced previously, and in it you might have noticed the typical JPA annotations. However, there are two new ones.

- A new transient property (by applying the @Transient annotation to the getter method) called birthDateString is added, which will be used for front-end presentation in later samples.
- For the photo attribute, we use a byte array as the Java data type, which corresponds to the BLOB data type in the RDBMS. In addition, the getter method is annotated with @Lob and @Basic(fetch=FetchType.LAZY). The former annotation indicates to the JPA provider that it's a large object column, while the latter indicates that the attribute should be fetched lazily to avoid a performance impact when loading a class that does not require photo information.

There are also some validation annotations that will be explained later in the chapter.

Because we will use Spring Data JPA's repository support, we will implement the SingerRepository interface, as shown here:

```
package com.apress.prospring5.ch16.repo;

import org.springframework.data.repository.PagingAndSortingRepository;

public interface SingerRepository extends
  PagingAndSortingRepository<Singer, Long> {
}
```

Instead of extending the CrudRepository interface, this example uses PagingAndSortingRepository, which is an advanced extension of CrudRepository that provides methods to retrieve entities using the pagination and sorting abstraction. This is quite useful as the queries return already sorted data that only needs to be shown in the interface, with no additional changes.

Implementing the Service Layer

In this section, we first discuss the implementation of SingerService by using JPA 2, Spring Data JPA, and Hibernate as the persistence service provider. Then we cover the configuration of the service layer in the Spring project. The following code snippet shows the SingerService interface with the services we would like to expose:

```
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);
}
```

The methods should be self-explanatory. The implementation of this interface is simple as well. Because the application is simple, no other changes need to be made to the data, so the role of `SingerServiceImpl` is to just forward the calls to the analogous repository methods.

```
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);
 }
}
```

The implementation is basically completed, and the next step is to configure the service in Spring's `ApplicationContext` within the web project, which is discussed in the next section.

Configuring SingerService

Obviously, there are two ways of doing this: XML and Java configurations. You can find a version of this project configured with XML in the source code for this chapter. If you are curious, you can analyze it, but in this section the focus will be on the Java configuration classes. To configure `SingerService`, the database access, and the transactions, you can use the following class (which you should be familiar with already from Chapter 9):

```
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();
  }
}
```

Now the service layer is completed and ready to be exposed and used by remote clients.

Introducing MVC and Spring MVC

Before moving on to implement the presentation layer, let's go through some major concepts of MVC as a pattern in web applications and how Spring MVC provides comprehensive support in this area.

In the following sections, we present these high-level concepts one by one. First, we give a brief introduction to MVC. Second, we present a high-level view of Spring MVC and its `WebApplicationContext` hierarchy. Finally, we discuss the request life cycle within Spring MVC.

Introducing MVC

MVC is a commonly used pattern in implementing the presentation layer of an application. The main principle of the MVC pattern is to define an architecture with clear responsibilities for different components. As its name implies, there are three participants within the MVC pattern.

- *Model*: A model represents the business data as well as the "state" of the application within the context of the user. For example, in an e-commerce web site, the model will include the user profile information, shopping cart data, and order data if users purchase goods on the site.
- *View*: This presents the data to the user in the desired format, supports interaction with users, and supports client-side validation, i18n, styles, and so on.
- *Controller*: The controller handles requests for actions performed by users in the front end, interacting with the service layer, updating the model, and directing users to the appropriate view based on the result of execution.

Because of the rise of Ajax-based web applications, the MVC pattern has been enhanced to provide a more responsive and rich user experience. For example, when using JavaScript, the view can "listen" to events or actions performed by the user and then submit an `XMLHttpRequest` to the server. On the controller side, instead of returning the view, the raw data (for example, in XML or JSON format) is returned, and the JavaScript application performs "partial" updates of the view with the received data. Figure 16-1 illustrates a commonly used web application pattern, which can be treated as an enhancement to the traditional MVC pattern. A normal view request is handled as follows:

1. *Request*: A request is submitted to the server. On the server side, most frameworks (for example, Spring MVC or Struts) have a dispatcher (in the form of a servlet) to handle the request.
2. *Invokes*: The dispatcher dispatches the request to the appropriate controller based on the HTTP request information and the web application configuration.
3. *Service call*: The controller interacts with the service layer.
4. *Model is populated*: The information obtained from the service layer is used by the controller to populate a model.

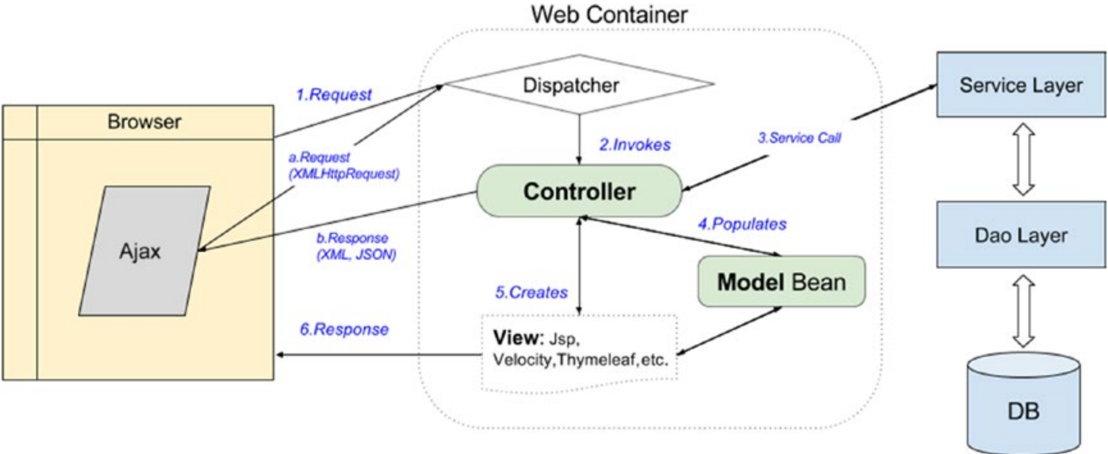

Figure 16-1. *The MVC pattern in a typical web application*

5. *View is created*: Based on the model, a view is created.

6. *Response*: The controller returns the corresponding view to the user.

In addition, within a view, Ajax calls will happen. For example, say the user is browsing data within a grid. When the user clicks the next page, instead of a full-page refresh, the following flow will happen:

a. *Request*: An XMLHttpRequest is prepared and submitted to the server. The dispatcher will dispatch the request to the corresponding controller.

b. *Response*: The controller interacts with the service layer, and the response data will be formatted and sent to the browser. No view is involved in this case. The browser receives the data and performs a partial update of the existing view.

Introducing Spring MVC

In the Spring Framework, the Spring MVC module provides comprehensive support for the MVC pattern, with support for other features (for example, theming, i18n, validation, and type conversion and formatting) that ease the implementation of the presentation layer.

In the following sections, we discuss the main concepts of Spring MVC. Topics include Spring MVC's WebApplicationContext hierarchy, a typical request-handling life cycle, and configuration.

Spring MVC WebApplicationContext Hierarchy

In Spring MVC, DispatcherServlet is the central servlet that receives requests and dispatches them to the appropriate controllers. In a Spring MVC application, there can be any number of DispatcherServlet instances for various purposes (for example, handling user interface requests and RESTful-WS requests), and each DispatcherServlet has its own WebApplicationContext configuration, which defines the servlet-level characteristics, such as controllers supporting the servlet, handler mapping, view resolving, i18n, theming, validation, and type conversion and formatting.

Underneath the servlet-level WebApplicationContext configurations, Spring MVC maintains a root WebApplicationContext, which includes the application-level configurations such as the back-end data source, security, and service and persistence layer configuration. The root WebApplicationContext will be available to all servlet-level WebApplicationContexts.

Let's look at an example. Say we have two DispatcherServlet instances in an application. One servlet supports the user interface (called the *application servlet*), and the other provides services in the form of RESTful-WS to other applications (called the *RESTful servlet*). In Spring MVC, we will define the configurations for both the root WebApplicationContext instance and the WebApplicationContext instance for the two DispatcherServlet instances. Figure 16-2 shows the WebApplicationContext hierarchy that will be maintained by Spring MVC for this scenario.

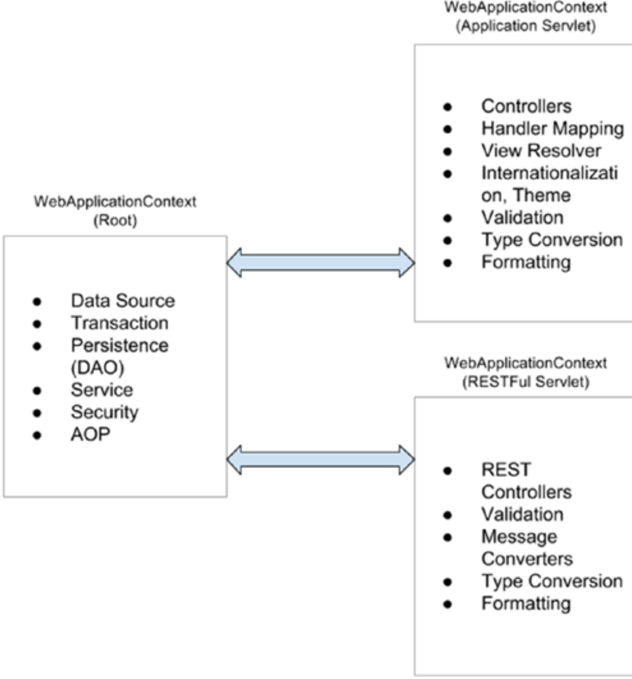

Figure 16-2. *Spring MVC WebApplicationContext hierarchy*

Spring MVC Request Life Cycle

Let's see how Spring MVC handles a request. Figure 16-3 shows the main components involved in handling a request in Spring MVC. The main components and their purposes are as follows:

- *Filter*: The filter applies to every request. Several commonly used filters and their purposes are described in the next section.
- *Dispatcher servlet*: The servlet analyzes the requests and dispatches them to the appropriate controller for processing.[1]
- *Common services*: The common services will apply to every request to provide supports including i18n, theme, and file upload. Their configuration is defined in the `DispatcherServlet`'s `WebApplicationContext`.

[1]If you are familiar with design patterns, you will recognize that `DispatcherServlet` is an expression of the Front Controller design pattern.

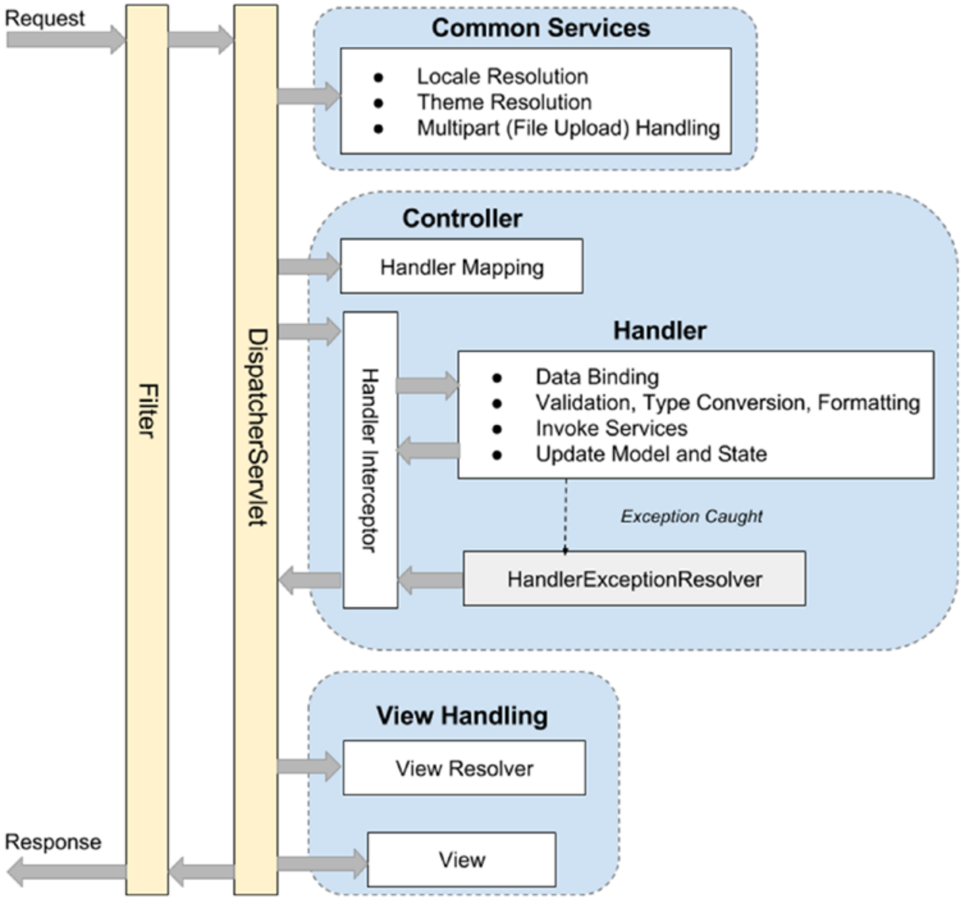

Figure 16-3. *Spring MVC request life cycle*

- *Handler mapping*: This maps incoming requests to handlers (a method within a Spring MVC controller class). Since Spring 2.5, in most situations the configuration is not required because Spring MVC will automatically register a `HandlerMapping` implementation out of the box that maps handlers based on HTTP paths expressed through the `@RequestMapping` annotation at the type or method level within controller classes.[2]
- *Handler interceptor*: In Spring MVC, you can register interceptors for the handlers for implementing common checking or logic. For example, a handler interceptor can check to ensure that only the handlers can be invoked during office hours.

[2]In Spring 2.5, `DefaultAnnotationHandlerMapping` was the default implementation. Starting with Spring 3.1, `RequestMappingHandlerMapping` has become the default implementation, which supports request mapping to handlers defined without annotations also, as long as the Spring conventions of naming controllers and methods are respected.

- *Handler exception resolver:* In Spring MVC, the HandlerExceptionResolver interface (defined in package org.springframework.web.servlet) is designed to deal with unexpected exceptions thrown during request processing by handlers. By default, DispatcherServlet registers the DefaultHandlerExceptionResolver class (from package org.springframework.web.servlet.mvc.support). This resolver handles certain standard Spring MVC exceptions by setting a specific response status code. You can also implement your own exception handler by annotating a controller method with the @ExceptionHandler annotation and passing in the exception type as the attribute.

- *View Resolver*: Spring MVC's ViewResolver interface (from package org.springframework.web.servlet) supports view resolution based on a logical name returned by the controller. There are many implementation classes to support various view-resolving mechanisms. For example, the UrlBasedViewResolver class supports direct resolution of logical names to URLs. The ContentNegotiatingViewResolver class supports dynamic resolving of views depending on the media type supported by the client (such as XML, PDF, and JSON). There also exists a number of implementations to integrate with different view technologies, such as FreeMarker (FreeMarkerViewResolver), Velocity (VelocityViewResolver), and JasperReports (JasperReportsViewResolver).

These descriptions cover only a few commonly used handlers and resolvers. For a full description, please refer to the Spring Framework reference documentation and its Javadoc.

Spring MVC Configuration

To enable Spring MVC within a web application, some initial configuration is required, especially for the web deployment descriptor web.xml. Since Spring 3.1, support has been available for code-based configuration within a Servlet 3.0 web container. This provides an alternative to the XML configuration required in the web deployment descriptor file (web.xml).

To configure Spring MVC support for web applications, we need to perform the following configurations in the web deployment descriptor:

- Configuring the root WebApplicationContext
- Configuring the servlet filters required by Spring MVC
- Configuring the dispatcher servlets within the application

The following configuration class does all three configurations in only a few lines of code:

```
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};
 }
}
```

To use code-based configuration, a class that implements the `org.springframework.web.WebApplicationInitializer` interface must be developed. To make things more practical, in the previous example the Spring class `AbstractAnnotationConfigDispatcherServletInitializer`, an implementation of `WebApplicationInitializer`, was extended because it contains concrete implementations of methods needed for the configuration of Spring web applications that use Java-based Spring configuration.

All classes implementing the `WebApplicationInitializer` interface will be automatically detected by the `org.springframework.web.SpringServletContainerInitializer` class (which implements Servlet 3.0's `javax.servlet.ServletContainerInitializer` interface), which bootstraps automatically in any Servlet 3.0 containers. As seen in the previous example, the following methods were overridden to plug in customized configurations:

- `getRootConfigClasses()`: A root application context of type `AnnotationConfigWebApplicationContext` will be created using the configuration classes returned by this method.
- `getServletConfigClasses()`: A web application context of type `AnnotationConfigWebApplicationContext` will be created using the configuration classes returned by this method.

- `getServletMappings()`: The `DispatcherServelt`'s mappings (context) are specified by the array of strings returned by this method.
- `getServletFilters()`: As the name of the methods says, this one will return an array of implementations of `javax.servlet.Filter` that will be applied to every request.

But wait, if you look at the example mentioned earlier, there is no mention of the security filter anywhere! How is this possible? This was covered in Chapter 12, but in case you skipped that one, here's the simple answer: there's a specialized Spring class for that.

```
package com.apress.prospring5.ch16.init;

import org.springframework.security.web.context.
 AbstractSecurityWebApplicationInitializer;
public class SecurityWebApplicationInitializer
  extends AbstractSecurityWebApplicationInitializer {
}
```

By providing an empty class that extends `AbstractSecurityWebApplicationInitializer`, you are basically telling Spring that you want `DelegatingFilterProxy` enabled, so `springSecurityFilterChain` will be used before any other registered `javax.servlet.Filter`.

Using this approach, when combined with the Java code-based configuration of Spring, it's possible to implement a pure Java code-based configuration of a Spring-based web application, without the need to declare any Spring configuration in `web.xml` or other Spring XML configuration files. And yes, it is that simple.

Returning to the filters, Table 16-1 describes each of the filters in the array returned by `getServletFilters()`.

Table 16-1. *Commonly Used Spring MVC Servlet Filters*

Filter Class Full Name	Description
`org.springframework.web.filter.CharacterEncodingFilter`	This filter is used to specify the character encoding for the request.
`org.springframework.web.filter.HiddenHttpMethodFilter`	This filter provides support for HTTP methods other than GET and POST (for example, PUT).

Although not needed here (and thus not used in the configuration), there is a filter implementation that should be mentioned: `org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter`. This implementation binds a JPA `EntityManager` to the thread for the entire processing of the request. It is intended for the Open EntityManager in View pattern, allowing for lazy loading in web views despite the original transactions already being completed. Although practical, it is quite dangerous, as multiple requests might end up consuming all database-allowed open connections. Also, if the data set to load is big, the application might freeze. That is why developers prefer not to use it, instead having specific handlers called via Ajax requests to load the data in web-specific view objects (instead of entities).

Creating the First View in Spring MVC

Having the service layer and Spring MVC configuration in place, we can start to implement our first view. In this section, we will implement a simple view to display all singers who were initially populated by the `DBInitializer` bean.

As mentioned earlier, we will use JSPX to implement the view. JSPX is JSP in a well-formed XML format. The main advantages of JSPX over JSP are as follows:

- JSPX forces the separation of code from the view layer more strictly. For example, you can't place Java "scriptlets" in a JSPX document.
- Tools might perform instant validation (on the XML syntax) so that mistakes can be caught earlier. We need to configure our project with the dependencies listed.

We need to configure our project with the dependencies listed in the following configuration snippet:

```
\\pro-spring-15\build.gradle
ext {
  //spring libs
  springVersion = '5.0.0.RC2'
  springSecurityVersion =  '5.0.0.M2'
  h2Version = '1.4.194'
  tilesVersion = '3.0.7'

  //persistency libraries
  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"
  ]
}
...
\\chapter-16\build.gradle
dependencies {
   // we exclude common transitive dependencies
   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
}
```

Most of the libraries listed in the previous configuration should be known to you by now. The ones that are used in this book for the first time are explained here:

- `spring-webmvc` is the Spring MVC module for MVC support.
- `spring-security-web` is the Spring web module for adding support for Spring Security. It is a direct dependency of `spring-security-web` that contains definitions of the security tags, to be used in JSP pages. `spring-security-web` contains the `AbstractSecurityWebApplicationInitializer` class and other related Spring components for securing web applications.

- `spring-security-config` is the Spring Security module, containing classes that are used to configure security in a Spring application.
- `tiles-jsp` is the Apache Tiles module, containing Java classes and tag definitions to create Java templates for web applications.[3]

Configuring DispatcherServlet

The next step is to configure `DispatcherServlet`. This is done by creating a configuration class that defines all infrastructure beans needed for a Spring web application. A snippet of the Java Config–based class is depicted next, containing minimal information:

```
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 {

 //Declare the static  resources.
 @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");
 }
```

[3]You can find more details at `https://tiles.apache.org/`.

```
 // <=>  <mvc:default-servlet-handler/>
 @Override
 public void configureDefaultServletHandling(
  DefaultServletHandlerConfigurer configurer) {
   configurer.enable();
 }
}
```

The interface `WebMvcConfigurer` defines callback methods to customize the Java-based configuration for Spring MVC enabled by using `@EnableWebMvc`. Although there can be more than one Java-based configuration class in a Spring application, only one is allowed to be annotated with `@EnableWebMvc`. In the previous configuration, you can observe that several methods are overridden to customize the configuration:

- The `addResourceHandlers()` method adds handlers that are used to serve static resources such as images, JavaScript, and CSS files from specific locations under the web application root, the classpath, and others. In this customized implementation, any request with the URL containing resources will be treated by a special handler that bypasses all filters.
- The `configureDefaultServletHandling(..)` method enables a handler for handling static resources.
- The `addViewControllers(..)` method defines simple automated controllers preconfigured with the response status code and/or a view to render the response body. These views have no controller logic and are used to render a welcome page, perform simple site URL redirects, return a 404 status, and more. In the configuration depicted previously, you are using this method to perform a redirection to the `singers/list` view.
- The `viewResolver(..)` method declares a view resolver of type `InternalResourceViewResolver` that matches symbolic view names to `*.jspx` templates under `/WEB-INF/views`.

Implementing SingerController

Having `DispatcherServlet`'s `WebApplicationContext` configured, the next step is to implement the controller class.

```
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;
 }
}
```

The annotation `@Controller` is applied to the class, indicating that it's a Spring MVC controller. The `@RequestMapping` annotation at the class level indicates the root URL that will be handled by the controller. In this case, all URLs with the prefix `/singers` will be dispatched to this controller. On the `list()` method, the `@RequestMapping` annotation is also applied, and at this time the method is mapped to the HTTP GET method. This means that the URL `/singers` with the HTTP GET method will be handled by this method. Within the `list()` method, the list of singers is retrieved and saved into the `Model` interface passed in to the method by Spring MVC. Finally, the logical view named `singers/list` is returned. In the `DispatcherServlet` configuration, `InternalResourceViewResolver` is configured as the view resolver, and the file has the prefix `/WEB-INF/views/` and the suffix `.jspx`. As a result, Spring MVC will pick up the file `/WEB-INF/views/singers/list.jspx` as the view.

Implementing the Singer List View

The next step is to implement the view page for displaying the singer information, which is the file `//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>
```

If you have developed with JSP before, the previous snippet should be familiar to you. But since this is a JSPX page, the page content is embedded under the `<div>` tag. In addition, the tag libraries being used are declared as XML namespaces.

First, the `<jsp:directive.page>` tag defines the attributes that apply to the entire JSPX page, while the `<jsp:output>` tag controls the properties of the output of the JSPX document.

Second, the tag `<c:if>` detects whether the model attribute `singers` is empty. Because we already populated some singer information in the database, the `singers` attribute should contain data. As a result, the `<c:forEach>` tag will render the singer information in the table within the page. Note the use of the `<fmt:formatDate>` tag to format the `birthDate` attribute, which is of the type `java.utilDate`.

Testing the Singer List View

Now we are ready to test the singer list view. First build and deploy the application; then, to test the singer list view, open a web browser and visit the URL `http://localhost:8080/singers`. You should be able to see the singer listing page.

Now we have our first view working. In the upcoming sections, we will enrich the application with more views and enable support of i18n, themes, and so on.

Understanding the Spring MVC Project Structure

Before diving into the implementation of the various aspects of a web application, let's take a look at what the project structure in the sample web application developed in this chapter looks like.

Typically, in a web application, a lot of files are required to support various features. For example, there are a lot of static resource files, such as style sheets, JavaScript files, images, and component libraries. Then there are files that support presenting the interface in various languages. And of course, there are the view pages that will be parsed and rendered by the web container, as well as the layout and definition files that will be used by the templating framework (for example, Apache Tiles) for providing a consistent look and feel of the application.

It's always good practice to store files that serve different purposes in a well-structured folder hierarchy to give you a clear picture of the various resources being used by the application and ease ongoing maintenance work.

Table 16-2 describes the folder structure of the web application that will be developed in this chapter. Note that the structure presented here is not mandatory but is commonly used in the developer community for web application development.

Table 16-2. *Sample Web Project Folder Structure Description*

Folder Name	Purpose
ckeditor	CKEditor (http://ckeditor.com) is a JavaScript component library that provides a rich-text editor in input form. We will use it to enable rich-text editing of a singer's description.
jqgrid	jqGrid (http://trirand.com) is a component built on top of jQuery that provides various grid-based components for data presentation. We will use this library for implementing the grid in order to display singers, as well as to support Ajax-style pagination.
scripts	This is the folder for all generic JavaScript files. For the samples in this chapter, the jQuery (http://jquery.org) and jQuery UI (http://jqueryui.com) JavaScript libraries will be used to implement a rich user interface. The scripts will be placed in this folder. In-house JavaScript libraries should be put here too.
styles	This folder stores the style sheet files and related images in supporting the styles.
WEB-INF/i18n	This folder stores files for supporting i18n. The file application*.properties stores the layout-related text (for example, page titles, field labels, and menu titles). The message*.properties file stores various messages (for example, success and error messages and validation messages). The sample will support both English (US) and Chinese (HK).
WEB-INF/layouts	This folder stores the layout view and definitions. Those files will be used by the Apache Tiles (http://tiles.apache.org) templating framework.
WEB-INF/views	This folder stores the views (in this case, JSPX files) that will be used by the application.

In the upcoming sections, we will need various files (for example, CSS files, JavaScript files, and images) to support the implementation. The source code of the CSS and JavaScript will not be shown here. Given that, we recommend you download a copy of the source code for this chapter and extract it to a temporary folder so that you can copy the files required into the project directly.

Enabling Internationalization (i18n)

When developing web applications, it's always good practice to enable i18n in the early stages. The main work is to externalize the user interface text and messages into properties files.

Even though you may not have i18n requirements on day one, it's good to externalize the language-related settings so that it will be easier later when you need to support more languages.

With Spring MVC, enabling i18n is simple. First, externalize the language-related user interface settings into various properties files within the /WEB-INF/i18n folder, as described in Table 16-2. Because we will support both English (US) and Chinese (HK), you will need four files. The application.properties and message.properties files store the settings for the default locale, which in this case is English (US). The application_zh_HK.properties and message_zh_HK.properties files store the settings in the Chinese (HK) language.

Configuring i18n in the DispatcherServlet Configuration

Having the language settings in place, the next step is to configure the `DispatcherServlet` instance's `WebApplicationContext` for i18n support. The following configuration snippet depicts the beans and methods for the `WebConfig` Java-based configuration class that are declared to enable and customize internationalization support.

```
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 {

  //Declare our static  resources.
   @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;
    }
  ...
}
```

In the previous configuration snippet, the resource definition is revised to reflect the new folder structure as presented in Table 16-2. The `addResourceHandlers(..)` method defines the locations of the static resource files, which enable Spring MVC to handle the files within those folders efficiently. Within the tag, the `location` attribute defines the folders for the static resources. The resource location, `/`, indicates the root folder for the web application, which is `/src/main/webapp`. The resource handler path, `/resources/**`, defines the URL for mapping to static resources; as an example, for the URL `http://localhost:8080/resources/styles/standard.css`, Spring MVC will retrieve the file `standard.css` from the folder `/src/main/webapp/styles`.

The `configureDefaultServletHandling(..)` method enables the mapping of `DispatcherServlet` to the web application's root context URL, while still allowing static resource requests to be handled by the container's default servlet.

Second, a Spring MVC interceptor with class `LocaleChangeInterceptor` is defined, which intercepts all the requests to `DispatcherServlet`. The interceptor supports locale switching with a configurable request parameter. From the interceptor configuration, the URL param with the name `lang` is defined for changing the locale for the application.

Then, a bean with class `ReloadableResourceBundleMessageSource` is defined. The `ReloadableResourceBundleMessageSource` class implements the `MessageSource` interface, which loads the messages from the defined files (in this case, it's the `messages*.properties` and `application*.properties` files in the `/WEB-INF/i18n` folder), to support i18n. Note the property `fallbackToSystemLocale`. This property instructs Spring MVC whether to fall back to the locale of the system that the application is running on when a special resource bundle for the client locale isn't found.

Finally, a bean with the `CookieLocaleResolver` class is defined. This class supports the storage and retrieval of locale settings from the user browser's cookie.

Modifying the Singer List View for i18n Support

Now we can change the JSPX page to display i18n messages. The following JSPX snippet shows the revised singer list view:[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"/>
 <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>
```

As shown in the previous snippet, first the Spring namespace is added into the page. Then, the `<spring:message>` tag is used to load the messages required by the view in the corresponding variables. Finally, the page title and the labels are changed to use the i18n messages.

Now build and redeploy the project, open your browser, and point to the URL `http://localhost:8080/singers?lang=zh_HK`. You will see the page in the Chinese (HK) locale.

---

[4]Keep in mind that we have not introduced Tiles or JavaScript in it yet.

Because we defined localeResolver in DispatcherServlet's WebApplicationContext, Spring MVC will store the locale setting in your browser's cookie (with the name locale), and by default, the cookie will be kept for the user session. If you want to persist the cookie for a longer time, in the localeResolver bean definition you can override the property cookieMaxAge, which is inherited from the class org.springframework.web.util.CookieGenerator, by calling setCookieMaxAge(...).

To switch to English (US), you can change the URL in your browser to reflect ?lang=en_US, and the page will switch back to English (US). Although we don't provide the properties file named application_en_US.properties, Spring MVC will fall back to use the file application.properties, which stores the properties in the default language of English.

Using Theming and Templating

Besides i18n, a web application requires an appropriate look and feel (for example, a business web site needs a professional look and feel, while a social web site needs a more vivid style), as well as a consistent layout so that users will not get confused while using the web application.

In addition, to provide a consistent layout, a templating framework is required. In this section, we will use Apache Tiles (http://tiles.apache.org), a popular page templating framework, for view templating support. Spring MVC tightly integrates with Apache Tiles in this aspect. Spring also supports Velocity and FreeMarker out of the box, which are more general templating systems and useful outside a web application as well for e-mail templates and so on.

In the following sections, we discuss how to enable theming support in Spring MVC, as well as how to use Apache Tiles to define the page layout.

Theming Support

Spring MVC provides comprehensive support for theming, and enabling it in web applications is easy. For example, in the sample singer application in this chapter, we want to create a theme and name it standard. First, in the folder /src/main/resources, create a file named standard.properties with the content shown here:

```
styleSheet=resources/styles/standard.css
```

This properties file contains a property named styleSheet, which points to the style sheet to use for the standard theme. This properties file is ResourceBundle for the theme, and you can add as many components for your theme as you want (for example, the logo image location and background image location).

The next step is to configure DispatcherServlet's WebApplicationContext for theming support by modifying the configuration class. First, in the addInterceptors(..) method, we need to add one more interceptor bean, as shown here:

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
  registry.addInterceptor(localeChangeInterceptor());
  registry.addInterceptor(themeChangeInterceptor());
}

@Bean
ThemeChangeInterceptor themeChangeInterceptor() {
  return new ThemeChangeInterceptor();
}
```

The new interceptor of type ThemeChangeInterceptor is added, and this class intercepts every request for changing the theme.

Second, the bean definitions are needed:

```
@Bean
ResourceBundleThemeSource themeSource()  {
  return new ResourceBundleThemeSource();
}

@Bean
CookieThemeResolver themeResolver() {
  CookieThemeResolver cookieThemeResolver = new CookieThemeResolver();
  cookieThemeResolver.setDefaultThemeName("standard");
  cookieThemeResolver.setCookieMaxAge(3600);
  cookieThemeResolver.setCookieName("theme");
  return  cookieThemeResolver;
}
```

Here, two beans are defined. The first bean, of type ResourceBundleThemeSource, is responsible for loading the ResourceBundle bean of the active theme. For example, if the active theme is called standard, the bean will look for the file standard.properties as the ResourceBundle bean of the theme. The second bean, of type CookieThemeResolver, is used to resolve the active theme for users. The property defaultThemeName defines the default theme to use, which is the standard theme. Note that as its name implies, the CookieThemeResolver class uses cookies to store the theme for the user. There also exists the SessionThemeResolver class that stores the *theme* attribute in a user's session.

Now the standard theme is configured and ready for use in our views. The following JSPX snippet shows the revised singer list view (/WEB-INF/views/singers/list.jspx) with theme support:

```
<?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>
```

A <head> section is added in the view, and the <spring:theme> tag is used to retrieve the styleSheet property from the theme's ResourceBundle, which is the style sheet file standard.css. Finally, the link to the style sheet is added into the view.

After rebuilding and redeploying the application to the server, open the browser and point to the singer list view's URL again (http://localhost:8080/singers), and you will see that the style defined in the standard.css file was applied.

Using Spring MVC's theme support, you can easily add new themes or change the existing theme within your application.

View Templating with Apache Tiles

For view templating using JSP technology, Apache Tiles (http://tiles.apache.org) is the most popular framework in use. Spring MVC tightly integrates with Tiles. To use Tiles and validation on data, the tiles-jsp, validation-api, and hibernate-validator libraries were added as dependencies.

In the following sections, we discuss how to implement page templates, including page layout design, definition, and implementation of the components within the layout.

Designing the Template Layout

First we need to define the number of templates required in our application and the layout for each template.

In the singer sample in this chapter, we require only one template. The layout is rather trivial, as shown in Figure 16-4. As you can see, the template requires the following page components:

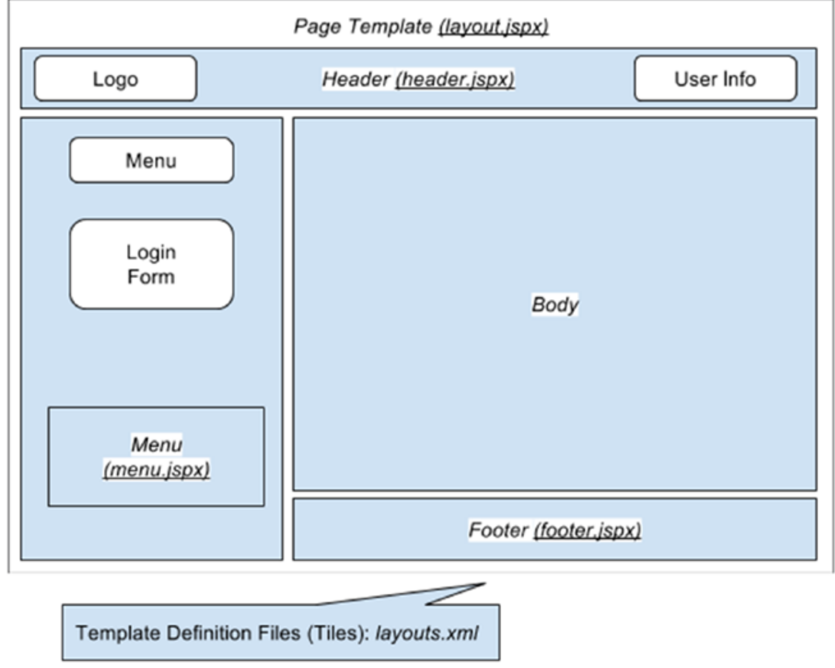

Figure 16-4. *Page template with layout components*

- `/WEB-INF/views/header.jspx`: This page provides the header area.
- `/WEB-INF/views/menu.jspx`: This page provides the left menu area, as well as the login form that will be implemented later in this chapter.
- `/WEB-INF/views/footer.jspx`: This page provides the footer area.

We will use Apache Tiles to define the template, and we need to develop the page template file as well as the layout definitions file, as listed here:

- `/WEB-INF/layouts/default.jspx`: This page provides the overall page layout for a specific template.
- `/WEB-INF/layouts/layouts.xml`: This file stores the page layout definitions required by Apache Tiles.

Implementing Page Layout Components

Having the layout defined, we can implement the page components. First we will develop the page template file and the layout definition files required by Apache Tiles.

```
<?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>
```

The file should be easy to understand. There is one page template definition, with the name `default`. The template code is in the file `default.jspx`. Within the page, three components are defined, named `header`, `menu`, and `footer`. The content of the components will be loaded from the file provided by the `value` attribute. For a detailed description of the Tiles definition, please refer to the project documentation page (http://tiles.apache.org/).

The following JSPX snippet shows the content of the `default.jspx` template file:

```
<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}" />

 <!-- Get  the user locale  from the page  context
 (it was  set by  Spring MVC's  locale resolver) -->
 <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>
```

The page is basically a JSP page. The highlights are as follows:

- The `<spring:theme>` tag is placed in the template, which supports theming at the template level.
- The `<tiles:insertAttribute>` tag is used to indicate the page components that need to be loaded from other files, as indicated in the `layouts.xml` file.

Now let's implement the `header`, `menu`, and `footer` components. The contents are shown here. The `header.jspx` file is quite simple and contains just this text:

```
<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>
```

The `menu.jspx` file is simple as well, as this application was designed with a minimal interface because the main focus is, after all, on 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>
```

The footer.jspx file contains the URLs to change the language in which the interface is displayed.

```
<?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>
```

Now for the singer list view, we can modify it to fit into the template. Basically, we just need to remove the <head> section, because it's now in the template page, default.jspx. The revised and improved version is shown here:

```
<?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>
```

Now the template, definition, and components are ready; the next step is to configure Spring MVC to integrate with Apache Tiles.

Configuring Tiles in Spring MVC

Configuring Tiles support in Spring MVC is simple. In the `DispatcherServlet` configuration (class `WebConfig`), we need to make a modification to replace `InternalResourceViewResolver` with the `UrlBasedViewResolver` class. The following code snippet contains only beans that are needed to configure Tiles support:

```
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;
 }
 ...
}
```

In the configuration snippet depicted previously, a `ViewResolver` bean with the class `UrlBasedViewResolver` is defined, with the property `viewClass` set to the `TilesView` class, which is Spring MVC's support for Tiles. Finally, a `tilesConfigurer` bean is defined that provides the layout configurations required by Tiles.

One final configuration file we need to prepare is the /WEB-INF/views/singers/views.xml file, which defines the views for the singer application in the sample. The file content is depicted here:

```
<?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>
```

As shown previously, the logical view name is mapped to the corresponding body attribute of the view to display. As in the SingerController class, the list() method returns the logical view name singers/list, so Tiles will be able to map the view name to the correct template and the view body to display.

We can now test the page. Make sure that the project was rebuilt and deployed to the server. Load the singer list view again (http://localhost:8080/singers), and the view based on the template will be displayed.

Implementing the Views for Singer Information

Now we can proceed to implement the views that allow users to view the details of a singer, create new singers, or update existing singer information.

In the following sections, we discuss the mapping of URLs to the various views, as well as how the views are implemented. We also discuss how to enable JSR-349 validation support in Spring MVC for the edit view.

Mapping URLs to the Views

First we need to design how the various URLs are to be mapped to the corresponding views. In Spring MVC, one of the best practices is to follow the RESTful-style URL for mapping views. Table 16-3 shows the URLs-to-views mapping, as well as the controller method name that will handle the action.

Table 16-3. *Mapping of URLs to Views*

URL	HTTP Method	Controller Method	Description
/singers	GET	list()	Lists the singer's information.
/singers/id	GET	show()	Displays a single singer's information.
/singers/id?form	GET	updateForm()	Displays the edit form for updating an existing singer.
/singers/id?form	POST	update()	Users update the singer information and submit the form. Data will be processed here.
/singers?form	GET	createForm()	Displays the edit form for creating a new singer.
/singers?form	POST	create()	Users enter singer information and submit the form. Data will be processed here.
/singers/photo/id	GET	downloadPhoto()	Downloads the photo of a singer.

Implementing the Show Singer View

Now we implement the view for showing a singer's information. Implementing the show view is a three-step process:

1. Implement the controller method.
2. Implement the show singer view (/views/singers/show.jspx).
3. Modify the view definition file for the view (/views/singers/views.xml).

The following code snippet shows the show() method implementation of the singerController class for displaying a singer's information:

```
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;
 }
...
}
```

On the show() method, the @RequestMapping annotation applied to it indicates that the method is to handle the URL /singers/{id} with the HTTP GET method. In the method, the @PathVariable annotation is applied to the argument id, which instructs Spring MVC to extract the ID from the URL into the argument.

Then the singer is retrieved and added to the model, and the logical view name `singers/show` is returned. The next step is to implement the show singer view, `/views/singers/show.jspx`, which is shown here:

```
<?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>
```

The page is simple; it displays the model attribute `singer` within the page.

The final step is to modify the view definition file `/views/singers/views.xml` for mapping the logical view name `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>
```

The show singer view is complete. Now we need to add an anchor link into the singer list view, `/views/singers/list.jspx`, for each singer to the show singer view. The revised file contents are shown here:

```
<?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>
```

As shown previously, we declare a URL variable by using the `<spring:url>` tag and add an anchor link for the `firstName` attribute. To test the show singer view, upon rebuild and deploy, open the singer list view again. The list should now include the hyperlink to the show singer view. Clicking any link will bring you to the show singer information view.

Implementing the Edit Singer View

Let's implement the view for editing a singer. It's the same as the show view; first we add the methods `updateForm()` and `update()` to the `SingerController` class. The following code snippet shows the revised controller for the two methods:

```
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;
  }
...
}
```

In the previous configuration, the highlights are as follows:

- The `MessageSource` interface is autowired into the controller for retrieving messages with i18n support.
- For the `updateForm()` method, the singer is retrieved and saved into the model, and then the logical view `singers/update` is returned, which will display the edit singer view.
- The `update()` method will be triggered when the user updates singer information and clicks the Save button. This method needs a bit of explanation. First, Spring MVC will try to bind the submitted data to the `Singer` domain object and perform the type conversion and formatting automatically. If binding errors are found (for example, the birth date was entered in the wrong format), the errors will be saved into the `BindingResult`interface (in the package `org.springframework.validation`), and an error message will be saved into the model, redisplaying the edit view. If the binding is successful, the data will be saved, and the logical view name will be returned for the display singer view by using `redirect:` as the prefix. Note that we want to display the message after the redirect, so we need to use the `RedirectAttributes.addFlashAttribute()` method (an interface in the package `org.springframework.web.servlet.mvc.support`) for displaying the success message in the show singer view. In Spring MVC, flash attributes are saved temporarily before the redirect (typically in the session) to be made available to the request after the redirect and removed immediately.

- The Message class is a custom class that stores the message retrieved from MessageSource and the type of message (that is, success or error) for the view to display in the message area. Here is the content of the Message class:

```
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;
  }
}
```

- UrlUtil is a utility class for encoding the URL for redirect. Its content is shown here:

```
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) {
 //
 }

 return  pathSegment;
  }
}
```

Next is the edit singer view, `/views/singers/edit.jspx`, and we will use it for both updating and creating a new singer.

```
<?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>
```

The highlights for the previous template are as follows:

- The `<spring:eval>` tag is used, which uses the Spring Expression Language to test whether the singer ID is `null`. If yes, then it's a new singer; otherwise, it's an update. The corresponding form title will be displayed.
- Various Spring MVC `<form>` tags are used within the form for displaying the label, the input field, and errors in case binding was not successful on form submission.

Next, add the view mapping to the view definition file `/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/update">
 <put-attribute name="body" value="/WEB-INF/views/singers/edit.jspx" />
  </definition>
  ...
</tiles-definitions>
```

The edit view is now completed. Let's rebuild and deploy the project. Upon clicking the edit link, the edit view will be displayed. Update the information and click the Save button. If binding was successful, then you will see the success message, and the show singer view will be displayed.

Implementing the Add Singer View

Implementing the add singer view is much like the edit view. Because we will reuse the `edit.jspx` page, we only need to add the methods in the `SingerController` class and the view definition. The following snippet of the `SingerController` class depicts the methods needed to implement the save operation for a new singer:

```
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;
  }
...
}
```

Next, add the view mapping to the view definition file, /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>
```

The add view is now complete. After you rebuild and deploy the project, click the New Singer link in the menu area. The add singer view will be displayed, allowing you to enter new singer details.

Enabling JSR-349 (Bean Validation)

Let's configure JSR-349 (Bean Validation) support for creating and updating singer actions. First, apply the validation constraints to the Singer domain object. In this sample, we define constraints only for the firstName and lastName attributes. The Singer class was already shown at the beginning of this chapter with the validation annotations in place, but in this section we explain them. Here we can see a snippet of this class with the annotated fields we are interested in:

```
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;

  ...
}
```

The constraints are applied to their respective fields. Note that for the validation message, you specify a message key by using the curly braces. This will cause the validation messages to be retrieved from ResourceBundle and hence support i18n.

To enable JSR-349 validation during the web data binding process, we just need to apply the @Valid annotation to the argument of the create() and update() methods in the SingerController class. The following code snippet shows the signatures for both methods:

```
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, ...
...
}
```

We also want the JSR-349 validation message to use the same ResourceBundle instance as for the views. To do this, we need to configure the validator in the DispatcherServlet configuration, in the WebConfig class.

```
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;
 }
 ...
}
```

First, a validator bean is defined, with the class `LocalValidatorFactoryBean`, for JSR-349 support. Note that we set the `validationMessageSource` property to reference the `messageSource` bean defined, which instructs the JSR-349 validator to look up the messages by the code from the `messageSource` bean. Then the `getValidator()` method is implemented to return the validator bean we defined.

That's all; we can test the validation now. Bring up the add singer view and just click the Save button. The returned page will now show us a validation error.

Switch to the Chinese (HK) language, and do the same thing. This time, the messages will be displayed in Chinese.

The views are basically complete, except the delete action. We leave that one to you as an exercise. Next, we will start to give our interface more richness.

Using jQuery and jQuery UI

Although the views for our singer application work well, the user interface is quite raw. For example, for the birth date field, it would be much better if we could add a date picker for entering the birth date of the singer, instead of having the user input the date string manually.

To provide a richer interface to the users of a web application, unless you are using rich Internet application (RIA) technologies that require special runtimes on the web browser client (for example, Adobe Flex requires Flash, JavaFX requires JRE, and Microsoft Silverlight requires Silverlight), you need to use JavaScript to implement the features.

However, developing web front ends with raw JavaScript is not easy. The syntax is very different from Java, and you also need to deal with cross-browser compatibility issues. As a result, a lot of open source JavaScript libraries are available that can make the process easier, such as jQuery and Dojo Toolkit.

In the following sections, we discuss how to use jQuery and jQuery UI to develop more responsive and interactive user interfaces. We also discuss some commonly used jQuery plug-ins for specific purposes, such as rich-text editing support, and discuss some grid-based components for browsing data.

Introducing jQuery and jQuery UI

jQuery (`http://jquery.org`) is one of the most popular JavaScript libraries being used for web front-end development. jQuery provides comprehensive support for main features including a robust "selector" syntax for selecting DOM elements within the document, a sophisticated event model, and powerful Ajax support.

Built on top of jQuery, the jQuery UI library (http://jqueryui.com) provides a rich set of widgets and effects. Main features include widgets for commonly used user interface components (a date picker, autocomplete, accordion, and so on), drag and drop, effects and animation, theming, and more.

There are a wealth of jQuery plug-ins developed by the jQuery community for specific purposes, and we discuss two of them in this chapter.

What we cover here only scratches the surface of jQuery. For more details on using jQuery, we recommend the books *jQuery Recipes: A Problem-Solution Approach* by B. M. Harwani (Apress, 2010) and *jQuery in Action* by Bear Bibeault and Yehuda Katz (Manning, 2010).

Enabling jQuery and jQuery UI in a View

To be able to use jQuery and jQuery UI components in our view, we need to include the required style sheets and JavaScript files. If we read the section "Understanding the Spring MVC Project Structure" earlier in this chapter, the required files should have been already copied into the project. These are the main files that we need to include in our view:

- `/src/main/webapp/scripts/jquery-1.12.4.js`: This is the core jQuery JavaScript library. The version we use in this chapter is 1.12.4. Note that it's the full source version. In production, you should use the minified version (that is, `jquery-1.12.4.min.js`), which is optimized and compressed to improve download and execution performance.
- `/src/main/webapp/scripts/jquery-ui.min.js`: This is the jQuery UI library bundled with a theme style sheet that can be customized and downloaded from the jQuery UI Themeroller page (http://jqueryui.com/themeroller). The jQuery UI version we are using is 1.12.1. Note that it's the minified version of JavaScript.
- `/src/main/webapp/styles/custom-theme/jquery-ui.theme.min.css`: This is the style sheet for the custom theme that will be used by jQuery UI for theming support.

To include the previous files, we only need to include them in the template page (that is, `/layouts/default.jspx`). The following code snippet needs to be added to the page:

```
<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>
```

First, the <spring:url> tag is used to define the URLs for the files and store them in variables. Then, in the <head> section, the reference to the CSS and JavaScript files is added. Note the use of the <jsp:text/> tag within the <script> tag. This is because JSPX will autocollapse tags without a body. So, the tag <script ..></script> in the file will end up as <script .. /> in the browser, which will cause undetermined behavior in the page. The addition of <jsp:text/> ensures that the <script> tag will not render in the page because it avoids unexpected issues.

With these scripts included, we can add some fancier stuff into our view. For the edit singer view, let's make the buttons look a bit better and enable the date picker component for the birth date field. The following code snippet shows the change that we need to add to /views/singers/edit.jspx for the button and date field:

```
<?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>
```

The $(function(){} syntax instructs jQuery to execute the script when the document is ready. Within the function, the birth date input field (with ID birthDate) is decorated using jQuery UI's datepicker() function. Second, various style classes are added to the buttons.

Now redeploy the application, and you will see the new button style, and when you click the birth date field, the date picker component will be displayed.

Rich-Text Editing with CKEditor

For the description field of the singer information, we use the Spring MVC `<form:textarea>` tag to support multiline input. Suppose we want to enable rich-text editing, which is a common requirement for long text inputs such as user comments.

To support this feature, we will use the rich-text component library CKEditor (http://ckeditor.com), which is a common rich-text JavaScript component with integration with jQuery UI. The files are in the folder `/src/main/we- bapp/ckeditor` of the sample source code.

First we need to include the required JavaScript files into the template page, `default.jspx`. The following code snippet shows you what you need to add to the page:

```
<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>
  <!-- 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>
```

The previous JSPX snippet includes two scripts: the core CKEditor script and the adapter with jQuery.

The next step is to enable CKEditor in the edit singer view. The next JSPX snippet shows the change required for the page `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>
```

The singer description field is decorated with CKEditor when the document is ready. Redeploy the application and go to the add singer page, and the description field will be enabled with rich-text editing support.

For complete documentation on using and configuring CKEditor, please refer to the project documentation site (http://docs.cksource.com/).

Using jqGrid for a Data Grid with Pagination

The current singer list view is fine if only a few singers exist in the system. However, as the data grows to thousands and even more records, performance will become a problem.

A common solution is to implement a data grid component, with pagination support, for data browsing so that the user browses only a certain number of records, which avoids a large amount of data transfer between the browser and the web container. This section demonstrates the implementation of a data grid with `jqGrid` (http://trirand.com/blog), a popular JavaScript-based data grid component. The version we are using here is 4.6.0.

For the pagination support, we will use jqGrid's built-in Ajax pagination support, which fires an `XMLHttpRequest` for each page and accepts the JSON data format for page data. To generate JSON data, you will use the `jackson-databind` library.

In the following sections, we discuss how to implement the pagination support on both the server and client sides. First, we cover implementing the `jqGrid` component in the singer list view. Then, we discuss how to implement pagination on the server side by using the Spring Data Commons module's comprehensive pagination support.

Enabling jqGrid in the Singer List View

To enable jqGrid in our views, first we need to include the required JavaScript and style sheet files in the template page `default.jspx`.

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
...
 <head>
 ...
   <!-- 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>
```

First, the grid-specific CSS file is loaded. Then, two JavaScript files are required. The first one is the locale script (in this case, we use English), and the second one is the `jqGrid` core library file `jquery.jqGrid.min.js`.

The next step is to modify the singer list view, `textitlist.jspx`, to use `jqGrid`. Here you can see the revised page:

```
<?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>
```

We declare a `<table>` tag with an ID of `list` for displaying the grid data. Under the table, a `<div>` section with an ID of pager is defined, which is the pagination part for `jqGrid`. Within the JavaScript, when the document is ready, we instruct `jqGrid` to decorate the table with an ID of `list` into a grid and provide detail configuration information. Some main highlights of the scripts are as follows:

- The `url` attribute specifies the link for sending `XMLHttpRequest`, which gets the data for the current page.
- The `datatype` attribute specifies the data format, in this case JSON. `jqGrid` also supports the XML format.
- The `mtype` attribute defines the HTTP method to use, which is GET.
- The `colNames` attribute defines the column header for the data to be displayed in the grid, while the `colModel` attribute defines the detail for each data column.
- The `jsonReader` attribute defines the JSON data format that the server will be returning.
- The `pager` attribute enables pagination support.
- The `onSelectRow` attribute defines the action to take when a row was selected. In this case, we will direct the user to the show singer view with the singer ID.

For a detailed description on the configuration and usage of jqGrid, please refer to the project's documentation site (`http://trirand.com/jqgridwiki/doku.php?id=wiki:jqgriddocs`).

Enabling Pagination on the Server Side

On the server side, there are several steps to take to implement pagination. First we will use the Spring Data Commons module's pagination support. To enable this, we need only to modify the `SingerRepository` interface to extend the `PagingAndSortingRepository<T,ID extends Serializable>` interface instead of the `CrudRepository<T,ID extends Serializable>` interface. You can see the revised interface here:

```
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>  {
}
```

The next step is to add a new method in the SingerService interface to support retrieving the data by page. The revised interface is depicted here:

```
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);
}
```

As shown previously, a new method called findAllByPage() is added, taking an instance of the Pageable interface as an argument. The following code snippet shows the implementation of the findAllByPage() method in the SingerServiceImpl class. The method returns an instance of the Page<T> interface (belonging to Spring Data Commons and in the package org.springframework.data.domain). As expected for this simple scenario, the service method just calls the repository method findAll(..), which is provided by the PagingAndSortingRepository<T,ID extends Serializable> interface.

```
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);
  }
}
```

The next step is the most complex, which is to implement the method in the SingerController class to take the Ajax request from jqGrid for page data. The following code snippet shows the implementation:

```
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);

 // Process order  by
 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);
 }

 // Constructs page  request for  current  page
 // Note: page  number  for  Spring Data   JPA starts with 0,
 // while jqGrid starts  with 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);

 // Construct the grid  data that  will  return as  JSON data
 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;
  }
  ...
}
```

The method handles the Ajax request, reads the parameters (page number, records per page, sort by, and sort order) from the request (the parameter names in the code sample are `jqGrid`'s defaults), constructs an instance of the `PageRequest` class that implements the `Pageable` interface, and then invokes the `SingerService.findAllByPage()` method to get the page data. Then, an instance of the `SingerGrid` class is constructed and returned to `jqGrid` in JSON format. The following code snippet shows the `SingerGrid` class:

```
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;
  }
```

```
  // other getters and  setters
  ...
}
```

Now we are ready to test the new singer list view. Make sure the project is rebuilt and deployed and then invoke the singer list view. You should now see an enhanced grid view of the singer listing.

You can play around with the grid, browse the pages, change the number of records per page, change the sort order by clicking the column headers, and so on. i18n is also supported, and you can try to see the grid with Chinese labels.

`jqGrid` also supports data filtering. For example, we can filter data by first names containing `John` or when the birth date is between date ranges.

Handling File Upload

The singer information has a field of BLOB type to store a photo, which can be uploaded from the client. This section shows how to implement file uploading in Spring MVC.

For a long time, the standard servlet specification didn't support file upload. As a result, Spring MVC worked with other libraries (the most common one being the Apache Commons FileUpload library, `http://commons.apache.org/fileupload`) to serve this purpose. Spring MVC has built-in support for Commons FileUpload. However, starting from Servlet 3.0, file upload has become a built-in feature of the web container. Tomcat 7 supports Servlet 3.0, and Spring has also supported Servlet 3.0 file upload since version 3.1.

In the following sections, we discuss how to implement the file upload functionality using Spring MVC and Servlet 3.0.

Configuring File Upload Support

In a Servlet 3.0–compatible web container with Spring MVC, configuring file upload support is a two-step process.

First, in the Java-based configuration class that defines everything needed to create the `DispatcherServlet` definition, we need to add a bean of type `StandardServletMultipartResolver`. This is a standard implementation of the `MultipartResolver` interface, based on the Servlet 3.0 `javax.servlet.http.Part` API. The following code snippet depicts the declaration of this bean that needs to be added to the `WebConfig` class:

```
package com.apress.prospring5.ch16.config;
...
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = {"com.apress.prospring5.ch16"})
public class WebConfig implements  WebMvcConfigurer {
...
 @Bean StandardServletMultipartResolver multipartResolver() {
  return new StandardServletMultipartResolver();
 }
...
}
```

The second step is to enable `MultiParsing` in Servlet 3.0 environments; this means that the `WebInitializer` implementation needs some changes. There is a method called `customizeRegistration(..)` that is defined in the `AbstractDispatcherServletInitializer` abstract class, which is the class extended by `AbstractAnnotationConfigDispatcherServletInitializer`. This method must be implemented to register an instance of `javax.servlet.MultipartConfigElement`. The enriched version of the `WebInitializer` class is shown here:

```
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);
 }
}
```

The first parameter of `MultipartConfigElement` is a temporary location where files should be stored. The second is the maximum file size allowed for upload, which is 5MB in this case. The third represents the size of the request, which is also 5MB here. The last one represents the threshold after which files will be written to disk.

Modifying Views for File Upload Support

We need to modify two views for file upload support. The first one is the edit view (`edit.jspx`) to support photo upload for a singer, and the second one is the show view (`show.jspx`) for displaying the photo.

The following JSPX snippet depicts the changes required in the `edit.jspx` view:

```
<?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>
```

In the previous snippet within the `<form:form>` tag, we need to enable the multipart file upload support by specifying the attribute `enctype`. Next, the file upload field is added to the form.

We also need to modify the show view to display the photo for a singer. The following JSPX snippet shows the changes required to the view `show.jspx`:

```
<?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>
```

In the previous view template, a new row is added to the table for displaying the photo by pointing to the URL for photo download.

Modifying Controllers for File Upload Support

The final step is to modify the controller. We need to make two changes. The first change is to the `create()` method to accept the upload file as a request parameter. The second change is to implement a new method for photo download based on the supplied singer ID. The following code snippet shows the revised `SingerController` class:

```
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());

// Process upload  file 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();
 }
 ...
}
```

First, in the `create()` method, a new request parameter of interface type `javax.servlet.http.Part` is added as an argument, which Spring MVC will provide based on the uploaded content in the request. Then the method will get the content saved into the `photo` property of the `Singer` object.

Next, a new method called `downloadPhoto()` is added to handle the file download. The method just retrieves the photo field from the `Singer` object and directly writes into the response stream, which corresponds to the `<img>` tag in the show view.

To test the file upload function, redeploy the application and add a new singer with a photo. Upon completion, you will be able to see the photo in the show view.

We also need to modify the edit function for changing the photo, but we will skip that here and leave it as an exercise for you.

Securing a Web Application with Spring Security

Suppose now we want to secure our singer application. Only those users who logged into the application with a valid user ID can add a new singer or update existing singers. Other users, known as *anonymous users*, can only view singer information.

Spring Security is the best choice for securing Spring-based applications. Although mostly used in the presentation layer, Spring Security can help secure all layers within the application, including the service layer. In the following sections, we demonstrate how to use Spring Security to secure the singer application.

Spring Security was introduced in Chapter 12, when you secured a REST service. For web applications, the possibilities are diverse, as there is also a tag library that can be used to secure only certain elements of a page. The Spring security tag library is part of the `spring-security-web` module.

Configuring Spring Security

In a previous section, we mentioned that for this chapter we are using a full annotations configuration, using Java-based configuration classes. Until Spring 3.*x*, enabling security in a Spring web application was done by configuring a filter in the web deployment descriptor (`web.xml`). This filter was named `springSecurityFilterChain`, and it was applied to any requests, except the one for static components. In Spring 4.0, a class has been introduced that can be extended to enable Spring Security: `AbstractSecurityWebApplicationInitializer`.

```
package com.apress.prospring5.ch16.init;

import org.springframework.security.web.context.
 AbstractSecurityWebApplicationInitializer;
public class SecurityWebApplicationInitializer
  extends AbstractSecurityWebApplicationInitializer {
}
```

By providing an empty class that extends `AbstractSecurityWebApplicationInitializer`, you are basically telling Spring that you want `DelegatingFilterProxy` enabled so that `springSecurityFilterChain` will be used before any other registered `javax.servlet.Filter`.

Besides this, the Spring Security context must be configured using a Java-based configuration class, obviously, and this class must be added to the root `WebApplicationContext` configuration. The `SecurityConfig` configuration class is depicted here:

```
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();
 }
}
```

The method `configure(HttpSecurity http)` defines the security configuration for HTTP requests. The `.antMatchers("/*").permitAll()` chained calls specify that all users are allowed to enter the application. We will see how we can protect the function by hiding the editing options in the view using Spring Security's tag library and controller method security. Then `.formLogin()` defines the support for form login. All calls after that until the `.and()` call are methods that configure the login form details. As we discussed in the layout, the login form will be displayed on the left. We provide a logout link as well through the `.logout()` call.

In Spring Security 4, the possibility of using CSFR tokens in Spring forms to prevent cross-site request forgery was introduced.[5] In this example, because we wanted to keep things simple, the usage of CSFR tokens was disabled by calling `.csrf().disable()`. By default a configuration without a CSFR element configuration is invalid, and any login request will direct you to a 403 error page stating the following:

```
Invalid CSRF Token 'null' was  found on  the request parameter
'_csrf' or  header 'X-CSRF-TOKEN'.
```

The method `configureGlobal(AuthenticationManagerBuilder auth)` defines the authentication mechanism. In the configuration, we hard-code a single user with the role USER assigned. In a production environment, the user should be authenticated against the database, LDAP, or an SSO mechanism.

Also, until Spring 3, the default login URL value is `/j_spring_security_check`, and the default names for the authentication keys are `j_username` and `j_password`. Starting with Spring 4, the default login URL value is `/login`, and the default names for the authentication keys are `username` and `password`.

In the previous configuration, the username, password, and login URL are explicitly set, but if in the view the default names are used, the following part of the configuration can be skipped.

```
.usernameParameter("username")
.passwordParameter("password")
.loginProcessingUrl("/login")|
```

Although the contents of the `WebInitializer` class were depicted earlier, the configuration will be covered here again to underline where the `SecurityConfig` class is used.

```
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<?>[]{
  WebConfig.class
   };
  }
  ...
}
```

[5]This is a type of attack that consists of hacking an existing session in order to execute unauthorized commands in a web application. You can read more about it at https://en.wikipedia.org/wiki/Cross-site_request_forgery.

Adding Login Functions to the Application

To add the login form to the application, two views must be changed. Here is the view header.jspx file to display the user information if the user is logged in:

```
<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>
```

First, the tag library with the prefix sec is added for the Spring Security tag library. Then, a <div> section with the <sec:authorize> tag is added to detect whether the user is logged in. If yes (that is, the isAuthenticated() expression returns true), the username and a logout link will be displayed.

The second view to be modified is the menu.jspx file, in which the login form is added; the New Singer option will display only if the user is logged in.

```
<?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>
```

The Add Singer menu item will render only when the user is logged in and has the role `USER` granted (as specified in the `<sec:authorized>` tag). Second, if the user is not logged in (the second `<sec:authorized>` tag, when the expression `isAnonymous()` returns true), then the login form will be displayed.

Redeploy the application, and it will display the login form, noting the new singer link is not shown.

Enter **user** in both the username and password fields and click the Login button. The user information will be displayed in the header area. The new singer link is also shown.

We also need to modify the show view (`show.jspx`) to show the edit singer link for only logged-in users, but we will skip that here and leave it as an exercise for you.

When the login information is incorrect, the URL to handle this will be at `/security/loginfail`. So, we need to implement a controller to handle this login fail scenario. The following code snippet shows the `SecurityController` class:

```
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;
  }
}
```

The controller class will handle all URLs with the prefix `security`, while the method `loginFail()` will handle the login fail scenario. In the method, we store the login fail message in the model and then redirect to the home page. Now redeploy the application and enter the wrong user information; the home page will be displayed again with the login fail message.

Using Annotations to Secure Controller Methods

Hiding the new singer link in the menu is not enough. For example, if you enter the `http://localhost:8080/users?form` URL in the browser directly, we can still see the add singer page, even though you are not logged in yet. The reason is that we haven't protected the application at the URL level. One method for protecting the page is to configure the Spring Security filter chain to intercept the URL for only authenticated users. However, doing this will block all other users from seeing the singer list view.

An alternative for solving the problem is to apply security at the controller method level, using Spring Security's annotation support. Method security is enabled by annotating the `SecurityConfig` class with `@EnableGlobalMethodSecurity(prePostEnabled = true)`, and the `prePostEnabled` attribute enables the support of pre- and post-annotations to be used on methods.

Now we can use the `@PreAuthorize` annotation for the controller method we want to protect. The following code snippet shows an example of protecting the `createForm()` method:

```
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";
  }
```

We use the `@PreAuthorize` annotation to secure the `createForm()` method, with an argument being the expression for security requirements.

Now you can try to directly enter the new singer URL in the browser, and if you are not logged in, Spring Security will redirect you to the login page, which is the singer list view as configured in the `SecurityConfig` class.

Creating Spring Web Applications with Spring Boot

Spring Boot was introduced really early in the book because it is a practical tool to create applications fast. In this section, we cover how to create a full-blown web application with security and web pages created using Thymeleaf. Thymeleaf is an XML/XHTML/HTML5 template engine that can work both in web and nonweb environments. It is really easy to integrate it with Spring. It is the best-suited template engine for Spring because its creator and project lead Daniel Fernandez started this project because he wanted to give Spring MVC the template engine it deserved.[6] Thymeleaf is a practical alternative to JSP or Tiles, and the SpringSource team fancies it quite a bit, so knowing how to configure it and use it could be useful in your future career.

The first Thymeleaf version was released in April 2011. At the time of writing, Thymeleaf 3.0.7 was recently published, including an update to the new integration module for Spring 5. There are quite a few extensions for Thymeleaf, written and maintained by the official Thymeleaf team (for example, Thymeleaf Spring Security Extension[7] and Thymeleaf Module for Java 8 Time API compatibility[8]).

Anyway, let's leave Thymeleaf for later and dive into the creation of a Spring Boot web application. To create a full-blown Spring web application, this means you need a DAO layer and a service layer in place. This means you need the specific Boot starter library for persistence and transactions. Here you can see a Gradle configuration snippet, depicting the libraries needed to create the application. Each of them will be detailed at the appropriate time.

```
//pro-spring-15/build.gradle
ext {
 //spring libs
 bootVersion = '2.0.0.M3'

 bootstrapVersion = '3.3.7-1'
 thymeSecurityVersion = '3.0.2.RELEASE'
 jQueryVersion = '3.2.1'
 ...
```

[6]You can find a full discussion on the official Thymeleaf forum at `http://forum.thymeleaf.org/why-Thymeleaf-td3412902.html`.
[7]The Thymeleaf Extras Spring Security library provides a dialect that allows you to integrate several authorization and authentication aspects of Spring Security (versions 3.*x* and 4.*x*) into Thymeleaf-based applications. See `https://github.com/thymeleaf/thymeleaf-extras-springsecurity`.
[8]This is a Thymeleaf Extras module, not part of the Thymeleaf core (and as such it follows its own versioning schema), but it is fully supported by the Thymeleaf team. See `https://github.com/thymeleaf/thymeleaf-extras-java8time`.

```
 spring = [
   ...
   springSecurityTest:
  "org.springframework.security:spring-security-test:$springSecurityVersion"

 ]

 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"
 ]
}

//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
}
```

Setting Up the DAO Layer

The Spring Boot JPA library `spring-boot-starter-data-jpa` contains autoconfigured beans that can be used to set up and generate an H2 database if the H2 library is in the classpath. All that is left for the developer to do is to develop an entity class and a repository. For this example, things will be kept really simple, so we'll use a simple version of the `Singer` class.

```
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;

 //setters and getters
 ...
}
```

Also, because of the simplicity, the simplest extension of a `CrudRepository` instance will be used.

```
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>  {

}
```

Setting Up the Service Layer

The service layer is also simple; it is made of the SingerServiceImpl class and of the DBInitializer class used to initialize the database and populate it with Singer records. The SingerServiceImpl class is depicted here. The initializer class was covered in the previous sections, so it won't be depicted again.

```
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;
 }

}
```

Setting Up the Web Layer

The web layer consists only of the simplest version of SingerController. The @RequestMapping annotations were replaced with equivalent annotations that no longer need to specify the HTTP method. The class is shown here:

```
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();
 }
}
```

The updateForm and createForm methods return the same view. The only difference is that the updateForm receives as an argument an ID of an existing Singer instance, which is used as a model object for the update view. The update view contains a button that, when clicked, calls the createSinger method. If the Singer instance had an ID, an update of the object is performed in the database; otherwise, a new Singer instance is created and saved to the database. Validation and proper handling of the validation failures are not covered in this example. This will be left as an exercise for you.

The tasks of discovery and creation of controller beans are supported by the Spring Boot web starter library spring-boot-starter-web.

Setting Up Spring Security

Spring Boot provides a Spring Security starter library called spring-boot-starter-security. If this library is in the classpath, Spring Boot automatically secures all HTTP endpoints with basic authentication. But the default security settings can be further customized. For this section, let's assume that only the front page (the home page) of the application is accesible using the root (/) context (http:\localhost:8080/). The customized configuration is depicted here:

```
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");
  }
}
```

Creating Thymeleaf Views

Before jumping into creating views using the Thymeleaf templating engine, let's cover the basic structure of a Spring Boot web application. A Spring Boot web application uses the `resources` directory as a web resources directory, so no `webapp` directory is needed. If the contents of the `resources` directory are organized following the default structure requirements of Spring Boot, a lot of configuration is not needed to be written because preconfigured beans are provided by the Spring Boot starter libraries. To use the Thymeleaf templating engine with the default configuration, the Spring Boot Thymeleaf starter library `spring-boot-starter-thymeleaf` must be in the classpath of the project.

Figure 16-5 shows the transitive dependencies of the `spring-boot-starter-thymeleaf` library, along with other starter libraries you might find interesting.

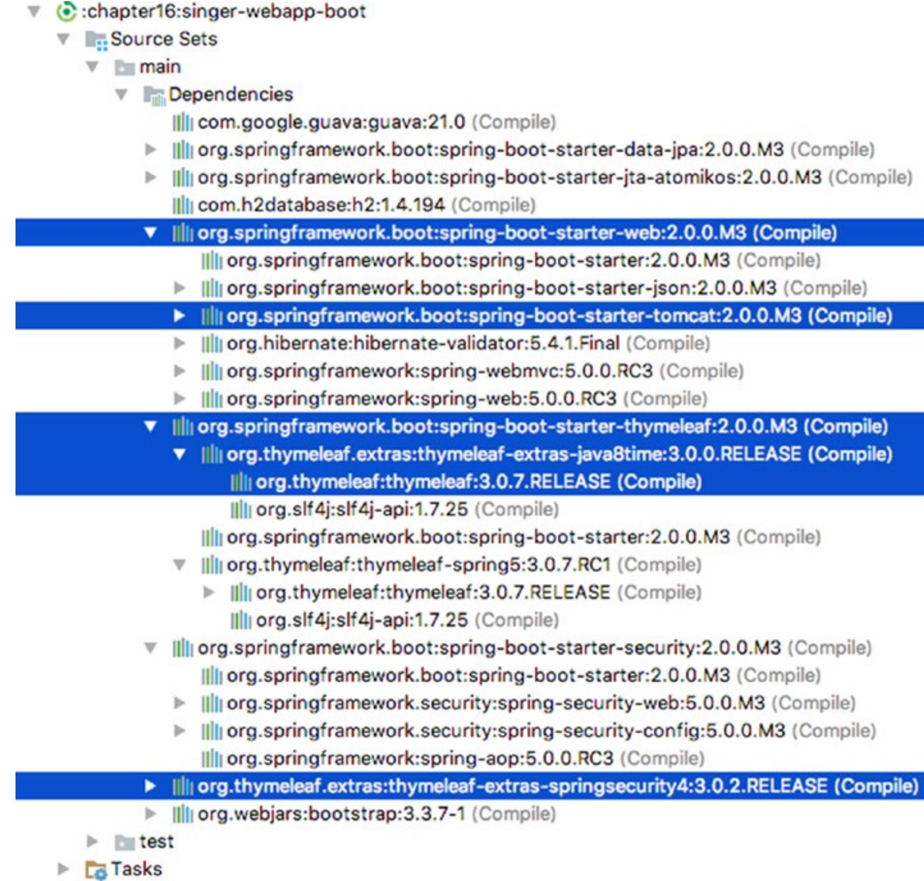

Figure 16-5. *Spring Boot starter libraries and starter dependencies*

As you can see, Spring Boot Thymeleaf starter version 2.0.0.M3 comes with Thymeleaf 3.0.7, which is the most recent released version at the time of writing.

Notice that `spring-boot-starter-web` has an embedded Tomcat server as a dependency that will be used to run the application.

Now that we have all the libraries in the classpath, let's analyze the structure of the `resources` directory, which is depicted in Figure 16-6. By default Spring Boot configures the Thymeleaf engine to read template files from the `templates` directory. If Spring MVC is being used alone, without Spring Boot, the Thymeleaf engine needs to be explicitly configured by defining three beans: a bean of type `SpringResourceTemplateResolver`, a bean of type `SpringTemplateEngine`, and a `ThymeleafViewResolver` bean.[9] Thus, with Spring Boot, all the developer has to do is start creating templates and drop them into the `/resources/templates` directory.

[9]The Thymeleaf official site provides a good tutorial on working with Thymeleaf and Spring. See `http://thymeleaf.org/doc/tutorials/3.0/thymeleafspring.html#spring-mvc-configuration`.

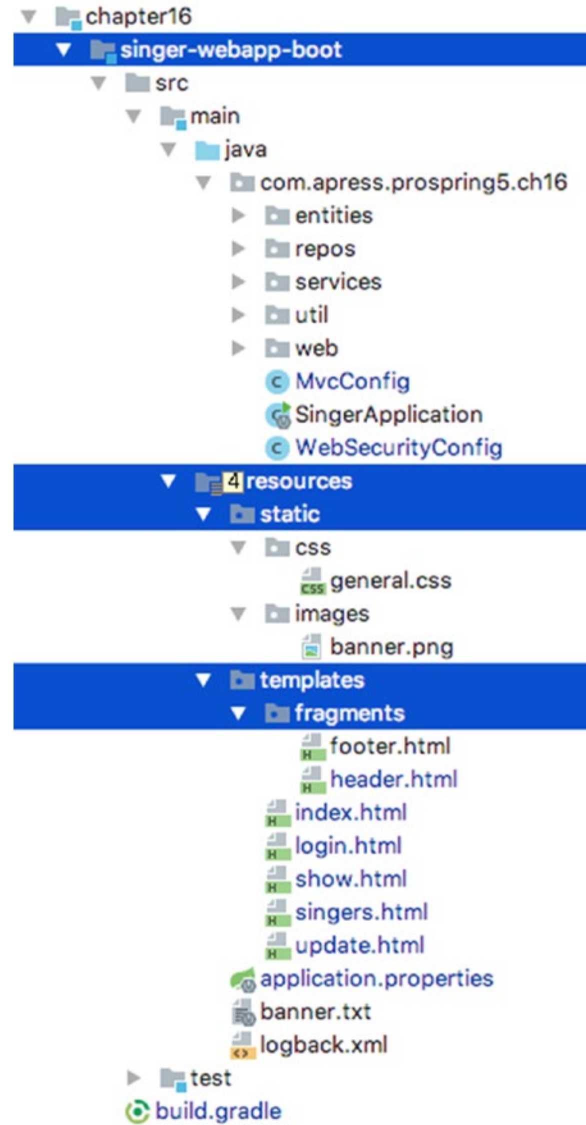

Figure 16-6. *Default structure of the resources directory for a Spring Boot web application using Thymeleaf*

Creating the Thymeleaf template is easy, and the syntax is simple HTML. Let's start with a simple example.

```
<!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>
```

The line `<html xmlns:th="http://www.thymeleaf.org">` is the Thymeleaf namespace declaration. This is important, and without it, the template will just be a simple static HTML file. As a template engine, Thymeleaf allows you to define and customize how the templates will be processed to a fine detail. Under the hood, the templates are handled by a processor (or several), which recognizes certain elements written in the Thymeleaf standard dialect. Most of the standard dialect consists of attribute processors; these allow a browser to correctly display templates, even without the files being processed, because the unknown attributes will just be ignored. Let's consider the following example of a form input text field:

```
<input type="text" name="singerName" value="John  Mayer" />
```

The previous declaration is a static HTML component, which the browser knows how to interpret. If you were to write the same element using Spring form elements, it would look like this:

```
<form:inputText name="singerName" value="${singer.name}" />
```

A browser would not be able to display the earlier declaration, so when writing templates using Apache Tiles, every time we want to view the form in a browser, we have to compile the project. Thymeleaf solves this problem by using normal HTML elements, which can be configured using Thymeleaf attributes. This is how the previous element looks written with Thymeleaf:

```
<input type="text" name="singerName" value="John Mayer"
   th:value="${singer.name}" />
```

With the previous declaration, the browser is able to display the element, and you can actually set a value for the element so we can see how components come together in the page. This value will be replaced by the result of the evaluation of `${singer.name}` during processing of the template.

As mentioned in the previous sections, when writing interfaces for a web application, we might need to isolate common parts and include them in other templates, to avoid code duplication. For example, the header, footer and menus are common parts of all the pages of an application. These common parts are represented with Thymeleaf by special templates called *fragments*. These partial templates should be defined by default under the `/templates/fragments` directory, but the name of the directory can be a different one if the situation requires it.[10] The syntax to define a fragment is depicted here, where a `footer.html` example is covered:

```
<!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>
```

[10] A web application can have multiple themes, and each of them can be represented by a set of fragments.

```
<body>
<div th:fragment="footer" th:align="center">
  <p>Copyright (c) 2017 by Iuliana Cosmina and Apress. All rights reserved.</p>
</div>
</body>
</html>
```

Notice the `th:fragment` attribute that declares the name of the fragment, which can be used later in other templates by inserting it or replacing certain elements. You can see the syntax to use the fragment here, where a simple `index.html` template is shown:

```
<!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>
```

There are three ways of using a fragment:

- `th:replace` replaces its host tag with the specified fragment.
- `th:insert` inserts the specified fragment as the body of its host tag.
- `th:include` inserts the contents of this fragment.[11]

If you want to know more about Thymeleaf templates, the best resources are available on the official site at http://thymeleaf.org/documentation.html.

The last directory that we need to cover here is the `static` directory. The `static` directory contains static resources for the application templates, such as CSS files and images. To include custom CSS classes in a Thymeleaf template, you can use the following syntax:

```
<!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>
```

[11]As of version 3.0, this is not recommended.

```
  ProSpring5 Singer Boot Application
   </title>
 </head>
 <body>
 ...
</body>
</html>
```

The href attribute contains the path relative to the template and is used when the template is opened in the browser so that the general.css styles are loaded. The th:href attribute is used when the template is processed, and {css/general.css} is resolved to the URL of the application.

Using Thymeleaf Extensions

In the application covered in this section, two Thymeleaf extensions are used. The thymeleaf-extras-springsecurity4 extension is needed because we are building a secured application using Thymeleaf templates. For example, when the user is not logged in, the Log In option should be shown, and when the user is logged in, the Log Out option should be shown.

This is a Thymeleaf Extras module, not part of the Thymeleaf core, but it's fully supported by the Thymeleaf team. Although the number 4 is in the name, all that is needed is the support for some special security utility objects. Thus, it is compatible with Spring Security 5.0.*x*, which is a transitive dependency of the Spring Boot starter security library. Let's take a look at a more extended header.html file that contains a navigation menu with different menu items for authorized and unauthorized users:

```
<!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>
```

This module provides a new dialect called `org.thymeleaf.extras.springsecurity4.dialect.SpringSecurityDialect` that includes the `#authorization` object (used in the previous example), which is an expression utility object with methods for checking the authorization based on expressions, URLs, and access control lists.[12] This dialect is configured out of the box by Spring Boot.

The second Thymeleaf extension used in this application is `thymeleaf-extras-java8time`. This dependency has to be explicitly added in the classpath of the project and provides support for the Java 8 Time API. This extension is also fully supported by the official Thymeleaf team. It adds a `#temporals` (and others like `#dates`) object to `ApplicationContext` as a utility object processor during expression evaluations. This means expressions in the Object-Graph Navigation Language (OGNL) and Spring Expression Language (SpringEL) can be evaluated. Let's consider the template needed to display a singer's details; it's called the `show.html` template.

```
<!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>
```

[12]As this extension is not in the scope of the book, if you are curious, you can visit the official site at https://github.com/thymeleaf/thymeleaf-extras-springsecurity for more information.

```
<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">
    <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>
```

In the previous example, singer.birthDate of type java.util.Date is formatted using the #dates utility object. In this case, no date formatter object is needed, and the accepted date pattern is hard-coded in the template.

This is not too practical, right?

Assume we have a DateFormatter class defined as shown in the following code snippet:

```
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);
    }
}
```

Let’s use a minimal web configuration class to configure this formatter in the application classpath.

```
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();
    }
}
```

Once the date formatter is registered in the application, it can be used in Thymeleaf templates via the double-bracket syntax. So this:

```
<p class="form-control-static"
        th:text="${#dates.format(singer.birthDate, 'dd-MMM-yyyy')}">
          Singer BirthDate
</p>
```

can be rewritten like this:

```
<p class="form-control-static" th:text="${{singer.birthDate}}">
      Singer BirthDate
</p>
```

So, the hard-coding of the date pattern was removed, and the date formatter class can be modified externally, without any change to the Thymeleaf template. That's more practical, right?

Using Webjars

In the most recent Thymeleaf template example, you might have noticed some strange links in the <head> element.

```
<!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>
```

Creating an HTML web page that looks amazing has become easier since Bootstrap[13] appeared. Bootstrap is the most popular HTML, CSS, and JS framework for developing responsive, mobile-first projects on the Web. To use Bootstrap in your templates, you just have to link to the Bootstrap CSS file in your template. For a long time, CSS styles and JavaScript code were part of a web application and were manually copied by developers into a special directory. But lately, the most commonly used frameworks for creating web pages, like jQuery and Bootstrap, can be used differently, by adding them as dependencies to your application, packaged as Java archives, called *webjars*.[14] Webjars are deployed on the Maven central repository and will be downloaded and added to your application automatically by your build tool (Maven, Gradle, etc.) once they've been declared as dependencies for your project.

In the example covered earlier, JQuery and Bootstrap webjars are declared using `th:href` attributes. Having them linked in the template file makes sure all Bootstrap classes are applied to the elements in the template (e.g., `class="container-fluid"`) and all jQuery functions are accessible (`onclick="$('#form').submit();`) when the template is processed and the application is deployed to a server.

After all this has been set up, if you run the `SingerApplication` class depicted next and you access `http://localhost:8080/`, you should see the `index.html` processed template.

```
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();
    }
}
```

Notice how no other annotation except `@SpringBootApplication` is needed. Entities, controllers, and repositories are automatically picked up, just by adding the JPA and web Spring Boot starter libraries in the classpath. Now if you access the `http://localhost:8080/` URL in a browser, you should see the home page of the application. If all went well and the Bootstrap webjar was processed correctly, the page should look like Figure 16-7.

[13]Here is the official site: `http://getbootstrap.com/`.
[14]See `https://www.webjars.org/`.

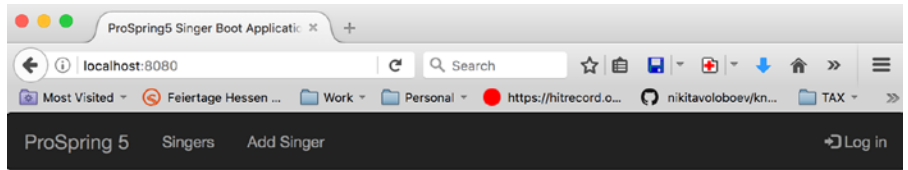

Figure 16-7. *The front page of the Spring Boot application built in this section*

Summary

In this chapter, we covered many topics related to web development using Spring MVC. First, we discussed the high-level concepts of the MVC pattern. Then we covered Spring MVC's architecture, including its `WebApplicationContext` hierarchy, request-handling life cycle, and configuration.

Next we learned how to develop a sample singer management application using Spring MVC, with JSPX as the view technology. During the course of developing the samples, we elaborated on different areas. Main topics included i18n, theming, and template support with Apache Tiles. Moreover, we learned how to use jQuery, jQuery UI, and other JavaScript libraries to enrich the interface. Samples included the date picker, rich-text editor, and data grid with pagination support. How to secure a web application with Spring Security was discussed too.

We also went through some functionality of Servlet 3.0–compatible web containers such as code-based configuration rather than using a `web.xml` file. We demonstrated how to handle file upload within a Servlet 3.0 environment.

And because Spring Boot is the prodigy feature of the Spring team, how to build a full-blown web application using it had to be covered. Of course, the most Spring-suited template engine, Thymeleaf had to be introduced as well.

In the next chapter, we cover more features that Spring brings in terms of web application development by introducing WebSocket.

CHAPTER 17

WebSocket

Traditionally, web applications have utilized the standard request/response HTTP functionality to provide communication between the client and server. As the Web has evolved, more interactive abilities have been required, some of which demand push/pull or real-time updates from the server. Over time, various methods have been implemented, such as continuous polling, long polling, and Comet. Each has its pros and cons, and the WebSocket protocol is an attempt to learn from those needs and deficiencies, creating a simpler and more robust way to build interactive applications. The HTML5 WebSocket specification defines an API that enables web pages to use the WebSocket protocol for two-way communication with a remote host.

This chapter covers a high-level overview of the WebSocket protocol and the main functionality provided by the Spring Framework. Specifically, this chapter covers the following topics:

- *Introduction to WebSocket*: We provide a general introduction of the WebSocket protocol. This section is not intended to serve as a detailed reference of the WebSocket protocol but rather as a high-level overview.[1]
- *Using WebSocket with Spring*: In this section, we dive into some of the details of using WebSocket with the Spring Framework; specifically, we cover using Spring's WebSocket API, utilizing SockJS as a fallback option for non-WebSocket-enabled browsers, and sending messages using Simple (or Streaming) Text-Oriented Message Protocol (STOMP) over SockJS/WebSocket.

Introducing WebSocket

WebSocket is a specification developed as part of the HTML5 initiative, allowing for a full-duplex single-socket connection in which messages can be sent between a client and a server. In the past, web applications requiring the functionality of real-time updates would poll a server-side component periodically to obtain this data, opening multiple connections or using long polling.

Using WebSocket for bidirectional communication avoids the need to perform HTTP polling for two-way communications between a client (for example, a web browser) and an HTTP server. The WebSocket protocol is meant to supersede all existing bidirectional communication methods utilizing HTTP as a transport. The single-socket model of WebSocket results in a simpler solution, avoiding the need for multiple connections for each client and less overhead—for example, not needing to send an HTTP header with each message.

[1]For details on the protocol, refer to RFC-6455 at `http://tools.ietf.org/html/rfc6455` or `https://www.websocket.org/`.

© Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho 2017

I. Cosmina et al., *Pro Spring 5*, https://doi.org/10.1007/978-1-4842-2808-1_17

WebSocket utilizes HTTP during its initial handshake, which in turn allows it to be used over standard HTTP (80) and HTTPS (443) ports. The WebSocket specification defines a `ws://` and a `wss://` scheme to indicate nonsecure and secure connections. The WebSocket protocol has two parts: a handshake between the client and server and then data transfer. A WebSocket connection is established by making an upgrade request from HTTP to the WebSocket protocol during the initial handshake between the client and the server, over the same underlying TCP/IP connection. During the data transfer portion of the communication, both the client and server can send messages to each other simultaneously, which as you can imagine opens the door to add more robust real-time communication functionality to your applications.

Using WebSocket with Spring

As of version 4.0, the Spring Framework supports WebSocket-style messaging as well as STOMP as an application-level subprotocol. Within the framework, you can find support for WebSocket in the `spring-websocket` module, which is compatible with JSR-356 (Java WebSocket).[2]

Application developers must also recognize that although WebSocket brings new and exciting opportunities, not all web browsers support the protocol. Given this, the application must continue to work for the user and utilize some sort of fallback technology to simulate the intended functionality as best as possible. To handle this case, the Spring Framework provides transparent fallback options via the SockJS protocol, which will we go into later in this chapter.

Unlike REST-based applications, where services are represented by different URLs, WebSocket uses a single URL to establish the initial handshake, and data flows over that same connection. This type of message-passing functionality is more along the lines of traditional messaging systems. As of Spring Framework 4, core message-based interfaces such as `Message` have been migrated from the Spring Integration project into a new module called `spring-messaging` to support WebSocket-style messaging applications.

When we refer to using STOMP as an application-level subprotocol, we are talking about the protocol that is transported via WebSocket. WebSocket itself is a low-level protocol that simply transforms bytes into messages. The application needs to understand what is being sent across the wire, which is where a subprotocol such as STOMP comes into play. During the initial handshake, the client and server can use the `Sec-WebSocket-Protocol` header to define what subprotocol to use. While the Spring Framework provides support for STOMP, WebSocket does not mandate anything specific.

Now that we have an understanding of what WebSocket is and the support Spring provides, where we might use this technology? Given the single-socket nature of WebSocket and its ability to provide a continuous bidirectional data flow, WebSocket lends itself well to applications that have a high frequency of message passing and require low- latency communications. Applications that may be good candidates for WebSocket could include gaming, real-time group collaboration tools, messaging systems, time-sensitive pricing information such as financial updates, and so on. When designing your application with the consideration of using WebSocket, you must take into account both the frequency of messages and latency requirements. This will help determine whether to use WebSocket or, for example, HTTP long polling.

Using the WebSocket API

As mentioned earlier in this chapter, WebSocket simply transforms bytes into messages and transports them between client and server. Those messages still need to be understood by the application itself, which is where subprotocols such as STOMP come into play. In the event you want to work with the

[2]See `www.oracle.com/technetwork/articles/java/jsr356-1937161.html`.

lower-level WebSocket API directly, the Spring Framework provides an API that you can interact with to do so. When working with Spring's WebSocket API, you would typically implement the `WebSocketHandler` interface or use convenience subclasses such as `BinaryWebSocketHandler` for handling binary messages, `SockJsWebSocketHandler` for SockJS messages, or `TextWebSocketHandler` for working with `String`-based messages. In this example, for simplicity we will use a `TextWebSocketHandler` to pass `String` messages via WebSocket. Let's start by taking a look at how we can receive and work with WebSocket messages at a low level, utilizing the Spring WebSocket API.

Each sample in this chapter can also be configured via Java configuration if that is your preference. In our opinion, the XML namespace represents the configuration aspects in a succinct fashion, and it will be used throughout the chapter. Please consult the reference manual for more information on Java configuration. First let's start by adding the required dependencies. The following Gradle configuration snippet lists these libraries:

```
//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"
   ]
}
...
//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
```

In the next configuration snippet, you can see the contents of the `WEB-INF/web.xml` file that we need to configure so we can use WebSocket with a standard Spring MVC dispatcher servlet:

```
<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>
```

We first create the servlet definition utilizing Spring's `DispatcherServlet`, providing it with a configuration file (`/WEB-INF/spring/root-context.xml`). We then provide servlet mapping, indicating all requests should go through `DispatcherServlet`.

Now let's move on and create the root context file, which contains the WebSocket configuration as shown here:

```
<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>
```

First, we configure a static resource called `index.html`. This file contains static HTML and JavaScript that is used to communicate with the back-end WebSocket service. Then, using the `websocket` namespace, we configure our handlers and corresponding bean to handle the request. We define a single handler mapping in this example, which receives requests at `/echoHandler` and uses the bean with the ID of `echoHandler` to receive a message and respond by echoing the provided message back to the client.

The previous configuration is probably vaguely familiar to you as we haven't used many XML configurations in this book. So, let's switch to Java configuration classes. Here is the configuration for 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");
        }
}
```

Let's continue with the class that replaces `web.xml` to configure `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[]{"/"};
    }
}
```

The `WebConfig` class contains the infrastructure of a Spring MVC application, and because we want to respect the principle of separation of concerns when using Java configuration, we need a different configuration class to support WebSocket communication. The class must implement the `WebSocketConfigurer` interface that defines callback methods to configure the WebSocket request handling.

```
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();
    }
}
```

The @EnableWebSocket annotation needs to be added to a @Configuration class to configure processing WebSocket requests.

Now we are ready to implement a subclass of TextWebSocketHandler (src/main/java/com/apress/prospring5/ch17/EchoHandler.java) to help us deal with String-based messages in a convenient way, as shown here:

```
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()));
    }
}
```

As you can see, this is a basic handler that takes the provided message and simply echoes it back to the client. The content of the received WebSocket message is contained in the getPayload() method.

That's pretty much all that is needed on the back end. Given that EchoHandler is a typical Spring bean, you can do anything you would in a normal Spring application, such as inject services, to carry out any functions this handler may need to do.

Now let's create a simple front-end client where we can interact with the back-end WebSocket service. The front end is a simple HTML page with a bit of JavaScript that uses the browser's API to make the WebSocket connection; it also contains some jQuery to handle button-click events and data display.

The front-end application will have the ability to connect, disconnect, send a message, and display status updates to the screen. The following code snippet shows the code for the front-end client page (`src/main/webapp/static/index.html`):

```
<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.");
                    }
                });

            $('#send')
                .click(function () {
                    if (typeof websocket != 'undefined') {
                        websocket.send($('#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>
```

The following code snippet provides a UI that allows us to call back into the WebSocket API and watch real-time results appear on the screen.

Build the project and deploy it into your web container. Then navigate to `http://localhost:8080/websocket-api/index.html` to bring up the UI. After clicking the Connect button, you will notice a `CONNECTED` message in the Status Output text area, and every three seconds a ping message will display in the Ping Output text area. Go ahead and type a message in the Message text box and then hit the Send button. This message will be sent to the back-end WebSocket service and displayed in the Message Output box. When you have finished sending messages, feel free to click the Disconnect button, and you will see a `DISCONNECTED` message in the Status Output text area. You will not be able to send any further messages or disconnect again until you reconnect to the WebSocket service. While this example utilizes the Spring abstraction on top of the low-level WebSocket API, you can clearly see the exciting possibilities this technology can bring to your applications. Now let's take a look at how to handle this functionality when the browser does not support WebSocket and a fallback option is required. You can test your browser for compatibility by using a site such as `http://websocket.org/echo.html`.

Using SockJS

Because all browsers may not support WebSocket and applications still need to function correctly for end users, the Spring Framework provides a fallback option utilizing SockJS. Using SockJS will provide WebSocket-like behavior as close as possible during runtime without the need for changes to application-side code. The SockJS protocol is used on the client side via JavaScript libraries. The Spring Framework's `spring-websocket` module contains the relevant SockJS server-side components. When using SockJS to provide a seamless fallback option, the client will first send a GET request to the server by using a path of `/info` to obtain transport information from the server. SockJS will first try to use WebSocket, then HTTP streaming, and finally HTTP long polling as a last resort. To learn more about SockJS and its various projects, see `https://github.com/sockjs`.

Enabling SockJS via the websocket namespace support is simple and requires only an additional directive inside the `<websocket:handlers>` block. Let's build a similar application as with the raw WebSocket API but using SockJS. The `src/main/webapp/WEB-INF/spring/root-context.xml` file will now look like this:

```
<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>
```

Notice that the `<websocket:sockjs>` tag has been added. At the most basic level, this is all that is needed to enable SockJS. We can reuse the `EchoHandler` class from the WebSocket API example, as we will be providing the same functionality.

This `<websocket:sockjs/>` namespace tag also provides other attributes to set configuration options such as handling session cookies (enabled by default), custom client library loading locations (at the time of this writing, the default is `https://d1fxtkz8shb9d2.cloudfront.net/sockjs-0.3.4.min.js`),

heartbeat configuration, message size limits, and so on. These options should be reviewed and configured appropriately, depending on your application needs and transport types. In the `web.xml` file not much needs to be added to reflect our SockJS servlet, as shown here:

```
<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>
```

As you probably suspect by now, the Java configuration follows here. There are two modifications to be made to support WebSocket communication using SockJS. First we need to support asynchronous messaging, which is enabled in the previous configuration using `<async-supported>true</async-supported>`. This is done by annotating a Java configuration class (a class already annotated with `@Configuration`) with another annotation: `EnableAsync`. If we look in the official Spring Javadoc, you find that this annotation enables Spring's asynchronous method execution capability, thus enabling annotation-driven async processing for an entire Spring application context.[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;
```

[3]You can find official site of the Spring Javadoc here: https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/scheduling/annotation/EnableAsync.html.

```
@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");
        }
}
```

The second change must be done in `WebSocketConfig` to enable SockJS support for our handler.

```
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();
        }
}
```

Next we will need to create an HTML page as we did in the WebSocket API sample, but this time utilizing SockJS to take care of the transport negotiation. The most notable differences are that we use the SockJS library rather than WebSocket directly and utilize a typical `http://` scheme rather than `ws://` to connect to the endpoint. The simple HTML client code is shown here:

```
<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>
```

With the new SockJS code implemented, build and deploy the project to the container and navigate to the UI located at `http://localhost:8080/sockjs/index.html`, which has all the same features and functionality of the WebSocket sample. To test the SockJS fallback functionality, try disabling WebSocket support in your browser. In Firefox, for example, navigate to the `about:config` page and then search for `network.websocket.enabled`. Toggle this setting to false, reload the sample UI, and reconnect. Utilizing a tool such as Live HTTP Headers will allow you to inspect the traffic going from browser to server for verification purposes. After verifying the behavior, toggle the Firefox setting `network.websocket.enabled` back to true, reload the page, and reconnect. Watching the traffic via Live HTTP Headers will now show you the WebSocket handshake. In the simple example, everything should work just as with the WebSocket API.

Sending Messages with STOMP

When working with WebSocket, typically a subprotocol such as STOMP will be used as a common format between the client and server so both ends know what to expect and react accordingly. STOMP is supported out of the box by the Spring Framework, and we will use this protocol in the sample.

STOMP, a simple, frame-based messaging protocol modeled on HTTP, can be used over any reliable bidirectional streaming network protocol such as WebSocket. STOMP has a standard protocol format; JavaScript client-side support exists for sending and receiving messages in a browser and optionally for plugging into traditional message brokers that support STOMP such as RabbitMQ and ActiveMQ. Out of the box, the Spring Framework supports a simple broker that handles subscription requests and message broadcasting to connected clients in memory. In this sample, we will utilize the simple broker and leave the full-featured broker setup as an exercise for you.[4]

For a full description of the STOMP protocol, see `http://stomp.github.io/stomp-specification-1.2.html`.

In the STOMP sample, we will create a simple stock-ticker application that displays a few predefined stock symbols, their current price, and the timestamp upon price change. New stock symbols and starting prices can also be added through the UI. Any connecting clients (that is, other browsers in tabs or totally new clients on other networks) will see the same data as they are subscribed to the message broadcasts. Every second, each stock price will be updated to a new random amount and the timestamp updated.

To ensure that your clients will be able to use the stock ticker application, even if their browser does not support WebSocket, we will utilize SockJS again to transparently handle any transport switching. Before diving into the code, it is worth noticing that STOMP messages support is provided by the `spring-messaging` library.

[4]See `http://docs.spring.io/spring/docs/current/spring-framework-reference/html/websocket.html#websocket-stomp-handle-broker-relay-configure` for more.

Now let's first create the Stock domain object, which holds information about the stock such as its code and price, as shown here:

```
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;
    }
    //setters and getters
    ...
}
```

Now we need to add an MVC controller to handle the incoming requests, as shown here:

```
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);
    }
}
```

The controller does a couple of things here. First, we add a few predefined stock symbols to the list and their starting prices for demonstration purposes. We then define a method `addStock`, which takes a `Stock` object, adds it to the list of stocks, and then broadcasts the stocks to all subscribers. When broadcasting the stocks, we iterate through all the stocks that have been added, updating the price for each, and then send them out to all subscribers of `/topic/price` by using the wired `SimpMessagingTemplate`. You also use a `TaskExecutor` instance to continuously broadcast the updated list of stock prices to all subscribed clients every second.

With the controller in place, let's now create the HTML UI for display to clients (`src/main/webapp/static/in-dex.html`), as shown in the following HTML snippet:

```
<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>
```

Similar to past examples, we have some HTML mixed in with JavaScript to update the display.[5] We utilize jQuery to update HTML data, SockJS to provide transport selection, and the STOMP JavaScript library `stomp.js` for communication with the server. Data sent via STOMP messages is encoded in JSON format, which we extract on events. Upon a STOMP connection, we subscribe to `/topic/price` to receive stock-price updates.

[5]The reason you combine HTML with JavaScript, although it is not respecting commonsense programming rules, is to keep the Spring MVC configuration as simple as possible.

Now let's configure the built-in STOMP broker in `root-context.xml` (`src/main/webapp/WEB-INF/spring/root-context.xml`).

```
<beans ...">

    <mvc:annotation-driven />

    <mvc:default-servlet-handler/>

    <mvc:view-controller  path= "/" view-name="/static/index.html" />

    <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>
```

For the most part, this configuration should look familiar. In this example, we configure `message-broker` by using the WebSocket namespace, define a STOMP endpoint, and enable SockJS. We also configure the prefix that subscribers will use to receive messages from. The configured `TaskExecutor` is used to provide stock quotes on the defined interval in the controller class. When using the namespace support, `SimpMessagingTemplate` is automatically created for us and available to inject into our beans.

Now all that's left to do is configure our `web.xml` file (`src/main/webapp/WEB-INF/web.xml`), as shown in the following configuration snippet:

```
<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>
```

The XML configuration has been covered, so let's switch to the nontraditional configuration using Java configuration classes. To enable Spring asynchronous calls and task execution, you must annotate `WebConfig` with `@EnableAsync`, and you must declare a bean of type `org.springframework.core.task.TaskExecutor`.

```
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();
        }
}
```

The more serious changes are needed in the class `WebSocketConfig`. This class must now implement `org.springframework.web.s` or extend the `AbstractWebSocketMessageBrokerConfigurer` abstract class, which will help us decide on what methods we actually want to implement. The class will now define methods for configuring message handling with simple messaging protocols like STOMP from WebSocket clients. Also, it has to be annotated with a different annotation called `@EnableWebSocketMessageBroker`, which will enable broker-backed messaging over WebSocket using a higher-level messaging subprotocol.

```
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");
        }
}
```

The methods that we are overriding in the previous configuration are the equivalents of the XML elements they are commented with. They are used to configure a STOMP endpoint and the message broker.

Summary

In this chapter, we covered the general concepts of WebSocket. We discussed the Spring Framework's support for the low-level WebSocket API and then moved on to using SockJS as a fallback option to select the appropriate transport, depending on the client browser. Finally, we introduced STOMP as a WebSocket subprotocol for passing messages between the client and server. For all examples, we presented the XML and Java configuration classes because the overall tendency in the business is to drop XML completely.

In the next chapter, we will discuss Spring subprojects that you can mix into your applications to provide even more robust functionality.

CHAPTER 18

Spring Projects: Batch, Integration, XD, and More

This chapter presents a high-level overview of a few projects that are part of the Spring portfolio, notably Spring Batch, Integration, XD, and a few notable features added in Spring Framework version 5. This chapter is not intended to cover each project in detail but to provide just enough information and a sample to get you started. The Spring portfolio contains many more projects than the ones discussed in this chapter, but we feel the ones presented here are widely used, and some are new and upcoming projects. You can view the full list of Spring projects at `http://spring.io/projects`. This chapter covers the following topics:

- *Spring Batch*: We cover the core concepts of the Spring batch-processing framework, including what it provides you as a developer, and we touch on the new JSR-352 support as of Spring Batch 3.0.
- *Spring Integration*: Integration patterns are used in many enterprise applications, and Spring Integration provides a robust framework for implementing these patterns. We build upon the batch example to show how Spring Integration can be used as part of a workflow to initiate a batch job.
- *Spring XD*: Spring XD ties together many of the existing Spring projects to provide a unified and extensible system for big data applications. Spring XD is a distributed system focusing on data ingestion, real-time analytics, batch processing, and data export. We show how to implement the applications from the Batch and Integration samples by utilizing Spring XD via simple DSL in the shell interface.
- *Notable features introduced in Spring 5*: There were a lot of discussions about Spring Framework 5 new features, but as the official release date was getting close, the list became fixed. Aside from under-the-hood features such as migrating the full code base to Java 8,[1] finally fixing the logging mess by integrating the Commons Logging bridge module (named `spring-jcl` instead of the standard Commons Logging[2]

[1]The intention was for Spring 5 to be fully compatible with Java 9, but as its release is about 18 months late already, the Spring team decided to stick with Java 8. However, Java 9 is planned to be released in September 2017, and release 5.*x* probably will be fully integrated with it.
[2]You might remember that the project setup was done in such a way to avoid Commons Logging because of its problematic runtime discovery algorithm; You can read about this on Spring's official reference page at `https://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#overview-avoiding-commons-logging`.

© Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho 2017

I. Cosmina et al., *Pro Spring 5*, https://doi.org/10.1007/978-1-4842-2808-1_18

and autodetecting Log4j 2.*x*, SLF4J, and JUL without any extra bridges), adding the *candidate component index* as an alternative to classpath scanning, and more,[3] there were a few other notable improvements. Three of them are covered in this chapter.

- *The Functional Web Framework*: `spring-webflux` is a complement to `spring-mvc` and was built on a reactive foundation. As the Reactive Streams API is an official part of Java 9, the Spring Framework 5 streaming support was built upon Project Reactor (`http://projectreactor.io/`), which implements the Reactive Streams API specification.
- *Full interoperability with Java 9*: The Spring Framework RC3 release became available in July 2017 and was advertised to be fully tested against the recent JDK 9 release candidate. Java 9 introduces quite a few interesting features, including the following: the Jigsaw project/Java modularity, a new HTTP client that supports the HTTP 2 protocol and WebSocket handshake, an improved process API, improved syntax for features such as `try-with-resources`, diamond operator and interface private methods, a publish-subscribe framework for reactive programming, and a set of new APIs. The official list of changes and features is available on the Oracle site,[4] but there are basically only two features that will be relevant for Spring: the modular functionality of JDK and the reactive framework.
- *Full support of JUnit 5 Jupiter*[5]: JUnit 5's Jupiter programming and extension models are fully supported in Spring Framework 5, including support for parallel test execution in the Spring TestContext Framework.

As each of these topics could have their own chapter or even books, covering all the details of each project and its various offerings would be impossible. We hope the introductions and basic samples will capture your interest to explore these topics further.

Spring Batch

Spring Batch, a framework for batch processing, is part of the Spring portfolio of projects. It's lightweight, flexible, and designed to provide developers with the ability to create robust batch applications with minimal effort. Spring Batch comes with a number of off-the-shelf components for a variety of technologies, and in most cases you may even be able to build your batch application by solely using the provided components.

Typical batch applications include daily invoice generation, payroll systems, and extract, transform, load (ETL) processes. While these are typical examples people may think of up front, Spring Batch can be used for any process that needs to run unattended, not just for these scenarios. As with all other Spring projects, Spring Batch builds upon the core Spring Framework, and you have full access to all its capabilities.

At a high level, a batch job contains one or more steps. Each step can provide the ability to either execute a single unit of work, which is represented by a tasklet implementation, or participate in what's called *chunk-oriented processing*. With chunk-oriented processing, a step utilizes an `ItemReader` to read some form of data, an optional `ItemProcessor` to do any transformations required on that data, and finally an `ItemWriter` to write the data out. A step also has various configuration attributes such as the ability to configure a chunk size (the amount of data to process per transaction), enable multithreaded execution,

[3]Check out the full list at `https://github.com/spring-projects/spring-framework/wiki/What's-New-in-the-Spring-Framework#whats-new-in-spring-framework-5x`.
[4]See `https://docs.oracle.com/javase/9/whatsnew/toc.htm`.
[5]You can find the official site here: `http://junit.org/junit5/docs/current/user-guide/`.

skip limits, and so on. Listeners can be used at the step level as well as the job level to receive notifications of various events that occur during the batch job life cycle, for example, before a step starts, when a step ends, during a chunk-oriented processing scenario, and so on.

While most jobs can run perfectly fine in a single-threaded, single-process manner, Spring Batch also provides options for scaling and parallel processing of jobs. Currently, Spring Batch provides the following scalability options out of the box:

- *Multithreaded steps*: This is the simplest way to make a step multithreaded. Simply add a `TaskExecutor` instance of your choice to the step configuration, and each chunk of items in a chunk-oriented processing setup will be processed in its own thread of execution.
- *Parallel steps*: Let's say, for example, you need to read in two large files with different data at the start of your job. At first you may create two steps, and one will execute after the other. If both of these data file loads do not depend on each other, why not process them both at the same time? For this case, Spring Batch allows you to define a split that contains flow elements and encapsulates these tasks to be executed in parallel.
- *Remote chunking:* This scalability option allows you to take a step and remotely distribute the work to a number of remote workers and communicate via some sort of durable middleware such as AMQP or JMS. Remote chunking is typically used when the reading of data is not the bottleneck in the process, yet the writing and optionally processing of chunk data is. Chunks of data are sent through the middleware for slave nodes to pick up and process, which then communicate back to the master about the status of their processing of that chunk.
- *Partitioning*: This scalability option is generally used when you want to process a range of data, utilizing threads for each range. A typical scenario is a database table filled with data that has a numerical identifier column. With partitioning, you can "partition" the data to be processed in separate threads with a certain number of records. Spring Batch provides the ability for you as the developer to hook into this partitioning scheme, as it's highly dependent on the use case at hand. Partitioning can be done in local threads or farmed out to remote workers (similar to the remote chunking option).

One basic yet common use case of batch processing is reading in a file of some sort, usually a flat file in a delimited format (for example, comma separated), which then needs to be loaded into a database, with each record optionally processed prior to writing to the database. Let's take a look at how we would implement this use case in Spring Batch. First we need to add the required dependencies, as shown in the following Gradle configuration:

```
//pro-spring-15/build.gradle
ext {
   //spring libs
   ...
   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",

        // needed  for  the Batch JSR-352 module
        hsqldb: "org.hsqldb:hsqldb:2.4.0"
   dbcp : "commons-dbcp:commons-dbcp:1.4",
   ]
}
...
//pro-spring-15/chapter18/build.gradle
dependencies {
      if (!project.name.contains("boot")) {
         compile(spring.jdbc) {
           // exclude these as batchCore will bring them
           // on as transitive dependencies
           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
      }
}
```

In the previous configuration, you can see the core dependencies we need to add to a Spring Batch project (not a Spring Boot project). That is why we have the `f (!project.name.contains("boot"))` condition; it prevents libraries with the version explicitly set to mingle with dependencies of Spring Boot projects in this chapter.

With the dependencies in place, let's dive into the code. First we create a domain object that represents a `Singer` based on the data in the file we will read, as shown here:

```
package com.apress.prospring5.ch18;

public class Singer {
    private String firstName;
    private String lastName;
    private String song;

    ... // setters & getters

    @Override
    public String toString()  {
        return "firstName: " + firstName + ", lastName: "
          +  lastName +  ", song: " +  song;
    }
}
```

Next, let's create an implementation of an ItemProcessor used to transform the first name, last name, and song of each singer represented by the Singer object to uppercase, as shown here:

```
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;
    }
}
```

Please note that ItemProcessors are not required in a chunk-oriented processing scenario; only ItemReader and ItemWriter are. We are using ItemProcessor here to serve as an example of how you can transform data prior to writing.

Next up, we will create a StepExecutionListener implementation that resides at the Step level and will tell you how many records were written after the step has completed, as shown in the following code snippet:

```
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;
    }
}
```

StepExecutionListener also allows us to modify the returned ExitStatus value if needed; otherwise, simply return null to keep it unchanged. At this point, we have the core components assembled, but before moving on to the configuration and invocation code, let's take a look at the data model and the data itself. The data model for this job is simple (src/main/resources/support/singer.sql) and is shown here with the src/main/resources/support/test-data.sql file that contains test data:

```
-- 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)
);

-- test-data.sql
John,Mayer,Helpless
Eric,Clapton,Change The World
John,Butler,Ocean
BB,King,Chains And Things
```

Now all we need is to create the Spring Batch configuration file, define the job, and set up the embedded database and related job components. As the XML configuration is cumbersome and was covered in the previous edition of this book, this chapter will focus only on the Java configuration classes. In the spirit of commonsense programming rules, you will decouple the batch configuration from the data source configuration by creating a separate configuration class for each. Here is the DataSourceConfig configuration class:

```
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;
      }
   }

}
```

As this configuration should be familiar by now, we will explain only the `schema-h2.sql` file. This file is part of the `spring-batch-core` library and contains DML statements needed to create the Spring Batch utility tables.

The `DataSourceConfig` class will be imported into the `BatchConfig` class, which is depicted here:

```
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;
    }
}
```

Although `BatchConfig` seems big, it is not nearly as big as the XML configuration would have been. Now it is time to explain each bean defined in it.

- The `@EnableBatchProcessing` annotation works in a similar way to all the `@Enable*` Spring annotations. This one provides a base configuration for building batch jobs. By annotating a configuration class with this annotation, the following things happen:
    - An instance of `org.springframework.batch.core.scope.StepScope` is created. Objects in this scope use the Spring container as an object factory, so there is only one instance of such a bean per executing step.
    - A set of specific batch infrastructure beans is made available for autowiring: `jobRepository` (of type `JobRepository`), `jobLauncher` (of type `jobLauncher`), `jobBuilders` (of type `JobBuilderFactory`, `stepBuilders` (of type `StepBuilderFactory`). This means they do not have to be explicitly declared (like in XML).
- The `job` bean is the batch job called `singerJob` that is created by calling `JobBuilderFactory.get(...)`.
- The `step1` bean is created by calling `StepBuilderFactory.get(...)` and is configured for chunk-oriented processing. The Spring container will inject the `ItemReader`, `ItemProcessor`, and `ItemWriter` instances found in the context automatically. The `StepExecutionStatsListener` bean must be set explicitly, though.
- The `database` bean is used in the declaration of `ItemWriter`, as this is the bean that will be used to write `Singer` instances to the embedded database.

Finally, we need a driver program to launch the job, as shown here:

```
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();
   }
}
```

This code should be familiar to you, as for the most part we are creating our context, obtaining a few beans, and calling methods on them. One thing you may notice is the `JobParameters` object. This object encapsulates parameters that are used to distinguish one instance of a job from another. Job identity is important in determining the last state of the job, if any, which also plays into other things such as the ability to restart a job. In this example, we simply use the current date as the `Job` parameter. `JobParameters` objects can be of many types, and these parameters can be accessed in the job as reference data. At this point, we are ready to test the new job. Compile the code and run the `SinxgerJobDemo` class. You will see some log statements display on the screen, and the ones of interest are as follows:

```
o.s.b.c.l.s.SimpleJobLauncher - Job: [SimpleJob: [name=singerJob]] launched with the
   following parameters: [{date=1501418591075}]
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.l.s.SimpleJobLauncher - Job: [SimpleJob: [name=singerJob]] completed with the
  following parameters: [{date=1501418591075}] and the following status: [COMPLETED]
```

That's all there is to it. You have now built a simple batch job that reads data from a CSV file, transforms the data via `ItemProcessor` to change the singer's first and last name and song to uppercase, and then writes the results to a database. You also used `StepListener` to output the number of items that were written in the step. For more information on Spring Batch, please see its project page at `http://projects.spring.io/spring-batch/`.

JSR-352

JSR-352 (Batch Applications for the Java Platform) was heavily influenced by Spring Batch. If you choose to utilize JSR-352 for your jobs, you will notice more and more similarities between the two and should feel comfortable if you are already a Spring Batch user. For the most part, Spring Batch and JSR-352 have similar constructs, and Spring Batch has fully supported this JSR as of Spring Batch 3.0. Like Spring Batch, JSR-352 jobs are configured via an XML schema in what is referred to as the Job Specification Language (JSL). Because JSR-352 defines a spec and an API, no off-the-shelf infrastructure components are provided as you may be used to when working with Spring Batch. If you strictly adhere to the JSR-352 API, this means implementing JSR-352 interfaces and writing all of the infrastructure components such as ItemReaders and ItemWriters on your own.

In this example, we will convert the previous batch example to utilize the JSR-352 JSL, but rather than rolling our own infrastructure components, we will utilize the same ItemReader, ItemProcessor, and ItemWriter as well as take advantage of Spring for dependency injection and so on. Implementing this job 100 percent to comply with the JSR-352 specification will be left as an exercise for you.

In this sample, as mentioned, we will reuse most of the code from the pure Spring Batch sample, with a few minor changes that we will explain here. If you haven't yet, this would be a good time to get the Spring Batch example working and then go forward with applying the changes in this section.

For this example, the H2 database will be replaced with HSQLDB and DBCP 2 with DBCP because newer versions are not supported. Also, JSR requires the singerJob.xml configuration file to be declared under src/main/resources/META-INF/batch-jobs/, and when starting a job, all that is needed is the file name without the .xml extension. So yeah, there's no way around it with JSR-352; XML must be used. The XML configuration file for this example is depicted here:

```
<?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>

    <!-- no transaction manager needed -->
    <bean id="stepExecutionStatsListener" ../>
    <bean id="itemReader" ../>
    <bean id="itemProcessor" ../>
    <bean id="itemWriter" ../>
</beans>
```

The beans that are not shown previously have the same definition when using Spring Batch. You can find the complete XML definition in the sources attached to the book. And now, because we must, here is how the previous job is configured with Spring Batch 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: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>
```

Thus, these two configurations look pretty much the same, except the job definition uses the JSR-352 JSL, and we are able to remove a few beans (`transactionManager`, `jobRepository`, and `jobLauncher`) as they are already provided for us in one way or another. You will also notice an additional schema definition using `jobXML_1.0.xsd`. Support for this schema is obtained via the JSR-352 API JAR and brought down automatically as a transitive dependency when using a build tool such as Gradle. If we need to obtain the dependency manually, see the project page listed at the end of this section. The second part that we need to modify is the `SingerJobDemo` class, as we are now using JSR-352-specific code to invoke the job, as shown here:

```
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);
    }
}
```

This is a bit different from other examples, where we work with ApplicationContext and beans directly. When creating a JSR-352 job, we use JsrJobOperator to start and control jobs. Rather than taking a JobParameters object to provide parameters to the job, a Properties object is used instead. The Properties object used is a standard java.util.Properties class, and job parameters should be created with both a String key and value. Another interesting change you may notice is the waitForJob() method. JSR-352 by default launches all jobs asynchronously. Thus, in the stand-alone program, as shown, we need to wait for the job to be in an acceptable state before the program terminates. If your code is running in a container such as an application server of some sort, this code may not be needed. Now let's compile and run the SingerJobDemo class, which will yield the following relevant log statements:

```
o.s.b.c.r.s.JobRepositoryFactoryBean - No database type set,
  using meta data indicating: HSQL
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
```

The log output looks pretty much the same, and you have now utilized JSR-352 to define and run this job along with using Spring's functionality for dependency injection and infrastructure components from Spring Batch rather than writing your own.

For more information on JSR-352, please see its project page at https://jcp.org/en/jsr/detail?id=352.

Spring Boot Batch

As expected, here comes Spring Boot, with a special starter library for Spring Batch to declutter the configuration even more. The swell part is that with the Spring Boot starter library for Spring Batch in the classpath, you won't need to bother too much with the dependencies. The bad part is that when it comes to configuration, well, the Spring Batch details cannot be reduced that much.

However, let's try to modify the batch example we have kept running so far and replace StepExecutionStatsListener with a JobExecutionStatsListener class; this class will extend the JobExecutionListenerSupport class, which is an empty abstract implementation of the JobExecutionListener interface that is used to provide callbacks at specific points in the life cycle of a job. In this case, the JobExecutionStatsListener class will query the database to check that our singer entries were indeed saved there.

The class JobExecutionStatsListener is depicted here:

```
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())
         );
      }
   }
}
```

As we can see, lambda expressions were heavily used just for fun in the previous example, but it is quite obvious what is going on in the `afterJob` callback method's body.

Also, let's see what you can do with the `BatchConfig` class.

```
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();
   }
```

```
    //adding lambda expressions
    @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() {
    ... //same as before
    }
}
```

So, aside from lambda expressions being heavily used in the declaration of the `itemReader` bean, the bean that changed the most is the `job` bean. The execution step is no longer autowired by the Spring container based on a qualifier, and the `flow` method is called to create a new job builder that will execute a step or sequence of steps.

That's pretty much it with Spring Boot; all that is left is to start the application by executing the typical `Application` class.

```
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();
   }
}
```

Eventually you can check the logs for the expected results printed by the `JobExecutionStatsListener` class, as shown here:

```
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]
```

Spring Integration

The Spring Integration project provides out-of-the box implementations of the well-known enterprise integration patterns (EIP). Spring Integration focuses on message-driven architectures. It provides a simple model for integration solutions, asynchronous abilities, and loosely coupled components, and it is designed for extension as well as testability.

At its core, a `Message` wrapper plays a central role in the framework. This generic wrapper around a Java object is combined with metadata used by the framework (more specifically the payload and headers) and is used to determine how to handle that object.

A `Message` channel is the pipe in a pipes-and-filters architecture in which producers send messages to this channel and consumers receive from it. `Message` endpoints, on the other hand, represent the filter of the pipes-and-filters architecture, and they connect application code to the messaging framework. Some `Message` endpoints that are provided out of the box by Spring Integration are `Transformers`, `Filters`, `Routers`, and `Splitters`, each providing their own roles and responsibilities.

Spring Integration also provides a plethora of integration endpoints (20+ at the time of this writing), which can be located in the documentation section "Endpoint Quick Reference Table" at `http://docs.spring.io/spring-integration/ reference/htmlsingle/#endpoint-summary`. These endpoints provide the ability to connect various resources such as AMQP, files, HTTP, JMX, Syslog, and Twitter. Going even beyond what is provided out of the box by Spring Integration, another project named Spring Integration Extensions is a community-based contribution model located at `https://github.com/spring-projects/spring-batch-extensions` that contains even more integration possibilities, including Amazon Web Services (AWS), Apache Kafka, Short Message Peer-to-Peer (SMPP), and Voldemort. Between the out-of-the-box and extension project components, Spring Integration provides a wealth of off-the-shelf components, which means the likelihood of having to write your own is greatly reduced.

In this example, we will build upon the previous batch examples, but this time we will introduce Spring Integration to show how we can use it to monitor a directory at a given interval. When a file arrives, we detect that file and kick off the batch job for processing.

In this sample, we are again going to build on the initial "pure" Spring Batch project that we started with in this chapter. Please be sure to take a look and have that running before proceeding, as we will be explaining only the new classes and configuration modifications here.

Obviously, some new dependencies must be added to the project.

```
//pro-spring-15/build.gradle
ext {
   //spring libs
   ...
   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",

          // needed for the Batch JSR-352 module
          hsqldb: "org.hsqldb:hsqldb:2.4.0"
   dbcp   : "commons-dbcp:commons-dbcp:1.4",
   ]
}
...

//pro-spring-15/chapter18/build.gradle
dependencies {
      if (!project.name.contains("boot")) {
         compile(spring.jdbc) {
            // exclude these as batchCore will bring them
            // on as transitive dependencies
            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
      }
   }
```

With the new dependencies in place, let's create a class that acts as a Spring Integration transformer. This Transformer instance will receive a Message instance from an inbound channel that represents a found file and launch the batch job with it, as depicted in the code snippet here:

```
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());
    }
}
```

Now, let's modify the BatchConfig class to support batch integration. What is really needed is to modify the itemReader bean, to be created every time a new file needs to be processed. This means that the path location must be injected into the bean, and it also means that the bean can no longer have the singleton scope.

```
package com.apress.prospring5.ch18.config;
...
@Configuration
@EnableBatchProcessing
@Import(DataSourceConfig.class)
@ComponentScan("com.apress.prospring5.ch18")
public class BatchConfig {
   ... // autowired  beans
```

```
   @Bean
   public Job singerJob() {
      return jobs.get("singerJob").start(step1()).build();
   }

   @Bean
   protected Step step1() {
      ...// no change
   }

   @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() {
      ...//no change
   }
}
```

The @StepScope is a convenient annotation that is equivalent to @Scope(value="step", proxyMode=TARGET_CLASS), thus making the itemReader bean have the scope proxy and naming this scope step so it is clear how this bean is being used.

Now that we have the batch configuration in place, let's add the integration typical configuration. We'll do this using an XML configuration file because at the time of writing, it really is more practical like this:

```
<?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>
```

The main additions to the configuration are the sections that are prefixed with the `int:` and `batch-int:` namespaces. First we create a few named channels to pass data through. We then configure an `inbound-channel-adapter` specifically for watching the specified directory on a given interval of one second. We then configure the `Transformer` bean, which receives files as standard `java.io.File` objects wrapped in a `Message`. Next we configure `job-launching-gateway`, which receives `job-launch` requests from `Transformer` to actually invoke the batch job. Last but not least, we create `logging-channel-adapter`, which will print out informational notices after the job completes. As you can see by following the configured `Channel` attributes, each component either consumes messages or produces messages, or both, via `Channel` instances. Now let's create a simple driver class that loads the configuration file. All this driver class does is load the application context with the configuration and remain running until you kill the process as it continuously polls the specified directory.

```
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();
   }
}
```

Now compile the code and run the FileWatcherDemo class. When the application starts, you may notice that some log messages are printed to the screen, but eventually nothing else happens. This is because the Spring Integration file adapter is waiting for files to be placed in the configured location on a polling interval, and nothing will happen until it detects a file in that location. In the src/main/resources/support/ directory, you will find four CSV files, named singer1.csv to singer4.csv. Each of them contains a single line with the first and last names of a singer and one of the singer's song names. Copy this files one by one into the tmp directory and watch the console to see what happens.

```
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}]
```

As you can see from the log statements, Spring Integration detected the file, created a `Message` from it, invoked the batch job that transformed the contents of the CSV file, and then wrote the contents to the in-memory database. While this is a simple example, it demonstrates how you can build complex and decoupled workflows between various types of applications by using Spring Integration.

For more information on Spring Integration, please see its project page at `http://projects.spring.io/spring-integration/`.

Spring XD

Spring XD is an extensible runtime service designed for distributed data ingestion, real-time analytics, batch processing, and data export. Spring XD builds upon many of the existing Spring portfolio projects, most notably the Spring Framework, Batch, and Integration. The goal of Spring XD is to provide a unified way to integrate many systems into a cohesive big data solution that helps solve the complexity of many common use cases.

Spring XD can run in a single stand-alone mode, typically for development and testing purposes, as well as in a fully distributed mode, providing the ability to have high-availability master nodes along with any number of worker nodes. Spring XD enables you to manage these services via a shell interface (utilizing the Spring shell) as well as through a graphical web interface. These interfaces allow you to define how to assemble various components to accomplish your data processing needs through either by using a DSL-type syntax via the shell application or by entering data into the web interface, which will build the definitions for you.

The Spring XD DSL is based around a few concepts, notably, streams, modules, sources, processors, sinks, and jobs. By combining these components via concise syntax, it is easy to create flows to connect various technologies to ingest data, process it, and eventually output the data to an external source, or even run a batch job for further processing. Let's take a quick look at these concepts:

- A stream defines how data will flow from a source to a sink and may pass through any number of processors. A DSL is used to define a stream; for example, a basic source-to-sink definition may look like `http | file`.
- A module encapsulates a reusable unit of work that streams are composed of. Modules are categorized by type based on their role. At the time of this writing, Spring XD contains modules of type source, processor, sink, and job.
- Sources in Spring XD either poll an external resource or are triggered by some sort of event. Sources provide output only to downstream components, and the first module in a stream must be a source.
- Processors are similar in nature to what we saw in Spring Batch. The role of a processor is to take an input, perform transformation or business logic on the provided object, and return an output.
- On the flip side of a source, a sink takes an input source and outputs that data to its destination resource. A sink is the final stop in the stream.
- Jobs are modules that define a Spring Batch job. These jobs are defined in the same way as we described at the start of this chapter and are deployed into Spring XD to provide batch-processing capabilities.
- Taps, as indicated by their name, listen to data flowing through the stream and also allow you to process that tapped data in a separate stream. The tap concept is similar to the WireTap enterprise integration pattern.

As you would expect, Spring XD provides a number of out-of-the-box sources, processors, sinks, jobs, and taps. As a developer, you are also not limited just to what's provided out of the box but are free to build your own modules and components as well. For more details on creating your own modules and components, see the reference manual on these customization points of interest:

- *Modules*: http://docs.spring.io/spring-xd/docs/current/reference/html/#_creating_a_module
- *Sources*: http://docs.spring.io/spring-xd/docs/current/reference/html/#creating-a-source-module
- *Processors*: http://docs.spring.io/spring-xd/docs/current/reference/html/#creating-a-processor-module
- *Sinks*: http://docs.spring.io/spring-xd/docs/current/reference/html/#creating-a-sink-module
- *Jobs*: http://docs.spring.io/spring-xd/docs/current/reference/html/#creating-a-job-module

In this example, we will show you how to use Spring XD's off-the-shelf components to replicate what we created with the Batch and Integration examples, all through a simple command-line configuration utilizing the XD shell and DSL.

Before you can begin, you must install Spring XD. Refer to the user manual section "Getting Started" at http://docs.spring.io/spring-xd/docs/current/reference/html/#getting-started, which provides details on the various ways of installing XD on your machine. The method of installation you choose is a matter of preference and does not impact the sample. Once you have XD installed, start the runtime in single-node mode as described in the documentation.

To replicate what we created in the Batch and Integration samples in XD, we need to do only a few basic tasks. First we need to create a CSV file to import, as shown here, and place that in `/tmp/singers.csv`:

```
John,Mayer,Helpless
Eric,Clapton,Change The World
John,Butler,Ocean
BB,King,Chains And Things
```

In your Spring XD shell console, type the following command:

```
job create singerjob --definition "filejdbc --resources=file:///tmp/singers.csv
     --names=firstname,lastname,song --tableName=singer
     --initializeDatabase=true" --deploy
```

Upon hitting Enter in the console, you should see a message stating, "Successfully created and deployed job 'singerjob.'" If not, inspect the console output in the terminal where you launched the single-node XD container for more details.

At this point, you have created a new job definition, but nothing has happened yet, as it has not been launched. In the shell, type the following command to launch the job:

```
job launch singerjob
```

The shell should now respond with a message indicating that a launch request for `singerjob` has been successfully submitted.

Breaking down the provided DSL, Spring XD knows that we wanted to create a batch job that reads from a file and outputs to a database via JDBC by the `job create` statement and the `filejdbc` source. It also automatically creates the table for us by using the `tableName` parameter, obtains the column names from the `names` parameter, and reads the data from the `resources` parameter in which we provided the file path to the CSV file.

If you want to inspect the imported data, use your favorite database tool to connect to the database you used during setup (either an embedded or real RDBMS) and select the records from the `Singer` table to verify. If you do not see data, check the log statements in the console that the single-node container is running in.

At this point, we have typed two commands into the shell but have not written any code or complex configuration. Yet we have imported the contents of the CSV file into the database with minimal effort. We accomplished this by utilizing Spring XD's prebuilt batch job, defining it with simple command-line DSL syntax, and then launched the job from the shell.

As you can see, Spring XD provides a lot of functionality out of the box that removes the need for the developer to create some of the more common use-case scenarios. Because we did some transformation to change the person's first and last names to uppercase in prior samples, we will leave that as an exercise for you to explore Spring XD further!

For more information on Spring XD, please see its project page at `http://projects.spring.io/spring-xd/`.

Spring Framework's Five Most Notable Features

At the time of writing, Spring Framework RC3 version was released. Spring 5.0 was a major revision of the core framework and came with a code base rewritten in Java 8, and it started being adapted to Java 9 with the RC3 release. There are a few major features announced for this version:

- It comes with the `spring-webflux` module, built on Reactor 3.1 with support for RxJava 1.3 and 2.1, which is also known as the Reactive Web Framework. This is a reactive complement to `spring-webmvc` that provides a web programming model designed for asynchronous APIs running on Tomcat, Jetty, or Undertow.
- Kotlin support is provided via a full null-safe API for bean registration and functional web endpoints.
- Integration with Java EE 8 APIs includes support for Servlet 4.0, Bean Validation 2.0, JPA 2.2, and the JSON Binding API (as an alternative to Jackson/Gson in Spring MVC).
- Full support for JUnit 5 Jupiter will allow developers to write tests and extensions in JUnit 5 and execute them in parallel with the Spring TestContext Framework.
- Java 9 interoperability was something desired by the Spring team for Spring Framework version 5. Because Oracle postponed the release of Java 9 for several months, the Spring Framework ended up being developed and released with Java 8 and using Project Reactor for the reactive programming support. But the promise to fully support Java 9 remained, and it started to be fulfilled with RC3.
- Many more features are available.[6]

The following sections will cover only three features from the list introduced previously.

[6]To keep track of the Spring project releases and contents, we recommend you follow the official Spring blog at `https://spring.io/blog`.

The Functional Web Framework

As stated earlier, the functional web framework (the `spring-webflux` module) is a reactive complement to `spring-webmvc` that provides a web programming model designed for asynchronous APIs. It was built according to reactive programming principles. *Reactive programming* can be explained in the simplest way as "programming with reactive streams." Streams are central to the reactive programming model, and they are used to support the asynchronous processing of anything. In a nutshell, reactive libraries offer the possibility for anything to be used as a stream (variables, user inputs, caches, data structures, etc.) and thus support stream-specific operations such as the following: filtering a stream to create another, merging streams, mapping values from one stream to another, and so on. The reactive part in reactive programming means that a stream will be an "observable" object that will be observed by a component and will react to it according to the objects emitted. A stream can emit three types of objects: values, errors, or "completed" signals.

To turn a normal application into a reactive application, the first logical step is to modify components to produce and handle data streams. There two types of data streams.

- `reactor.core.publisher.Flux`[7]: This is a stream of `[0..n]` elements. The easiest way to create a `Flux` is as follows:

  ```
  Flux simple = Flux.just("1", "2", "3");
  //or from an existing list: List<Singer> Flux<Singer>
  fromList = Flux.fromIterable(list);
  ```

- `reactor.core.publisher.Mono`: [8] This is a stream of `[0..1]` elements. The easiest way to create a `Mono` is as follows:

  ```
  Mono simple = Mono.just("1");
  //or from an existing object of type Singer
  Mono<Singer> fromObject = Mono.justOrEmpty(singer);
  ```

Both classes are implementations of `org.reactivestreams.Publisher<T>`, and you might wonder at this point if the `Mono` implementation is actually needed. The answer is yes, because of practical reasons: depending on the type of operation done with the values a stream is emitting, it is always useful to know the cardinality. Imagine a reactive repository class, for example: would it make sense for the `findOne` method to return a `Flux`?

This short introduction should be enough for you to grasp the basics of reactive programming and be able to understand the functionality that `spring-webflux` brings to the table.[9]

In this section, you will use the Spring Boot web application that was introduced in Chapter 16 that used Thymeleaf and transform it to use Spring WebFlux. The first step is to add the `spring-boot-starter-webflux` as a dependency and remove the unnecessary dependencies. In the following snippet, you can see the libraries needed, configured in the parent project, and used in the `webflux-boot` module:

```
//pro-spring-15/build.gradle
ext {
   //spring libs
   bootVersion = '2.0.0.M3'
   springDataVersion = '2.0.0.M3'
```

---

[7]You can find a detailed explanation at https://projectreactor.io/docs/core/release/reference/#flux.
[8]You can find a detailed explanation at https://projectreactor.io/docs/core/release/reference/#mono.
[9]If you need more resources about the reactive programming model and programming with reactive streams, you might want to try the reference documentation, code samples, and Javadoc of Project Reactor at https://projectreactor.io/docs or the really great introduction to reactive programming at https://gist.github.com/staltz/868e7e9bc2a7b8c1f754.

```
   junit5Version = '5.0.0-M4'
...
boot = [
...
         springBootPlugin:
            "org.springframework.boot:spring-boot-gradle-plugin:$bootVersion",
         starterWeb      :
            "org.springframework.boot:spring-boot-starter-web:$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"
   ]
}

//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
}
```

The `spring-boot-starter-webflux` module depends on a few reactive libraries that will be added to the application automatically. Figure 18-1 shows the dependencies of the application as depicted by the Gradle view in IntelliJ IDEA. Notice the `reactive-streams` library. This library contains the reactive streaming basic interfaces, the standard specification, and four interfaces: `Publisher`, `Subscriber`, `Subscription`, and `Processor`. You can read more about it at `www.reactive-streams.org`, as the streaming API is not the focus of this section. The streaming implementations that are used in the application in this section are provided by the `reactor-core` library, and you can read more about its contents at `https://projectreactor.io/`.

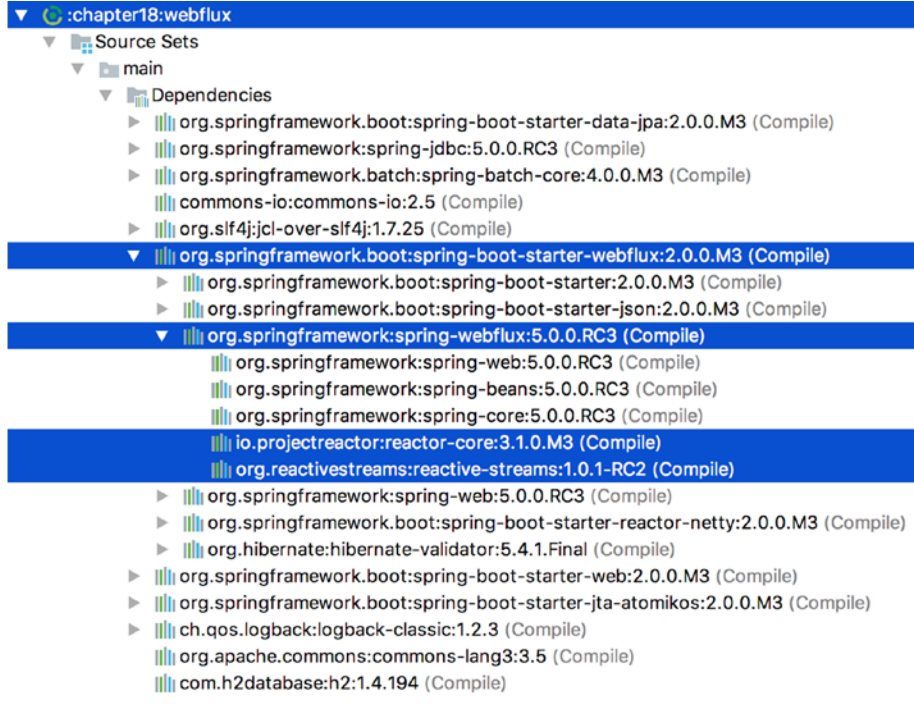

Figure 18-1. *Project dependencies depicted by the Gradle view in IntelliJ IDEA*

To keep things simple, the security layer will be removed, and we'll remove the interface as well. We'll use REST and use a set of test classes to test the application. The `Singer` class will be simplified by removing all validator annotations, as these are not the focus in this section.

```
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;
   }
   //setters and  getters
   ...
}
```

The repository implementation will be an empty extension of the `CrudRepository<Singer, Long>` interface, and it won't be depicted here again. The novelty here is that this repository will be used by a reactive repository implementation. Here you can see the contents of the `ReactiveSingerRepo` interface, which makes use of `Mono` and `Flux` to handle `Singer` objects:

```
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);
}
```

The implementation is shown here:

```
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()));
   }
}
```

Let's discuss each of the methods in this class separately.

- The `findById` method returns a simple stream, which will emit a `Singer` object if the call to `singerRepository.findById(id)` returns one; otherwise, it will emit an `onComplete` signal. This means that the returned object will be a `Mono<Singer>` object that will contain nothing.
- The `findAll` method returns a stream containing all the `Singer` objects returned by the `textitsingerRepository.findAll()` method.
- The `save` method receives a parameter of type `Mono<Singer>` instance, which contains one `Singer` instance. An instance of `java.util.function.Consumer<Singer>` is declared so that when the `Singer` object is emitted successfully, it will be persisted to the database by `singerRepository`. This method returns an empty `Mono` instance because there is nothing to return.

Now that we have the reactive repository, we need a reactive handler for it. Because the typical Spring @Controller would just consume the streams and return the contents in a view, we need to use something different. The class is called SingerHandler and is depicted here:

```
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));
   }
}
```

Requests are handled by a handler function, which takes as a parameter a ServerRequest and returns a Mono<ServerResponse>. These two interfaces are re-immutable and provide access to the underlying HTTP messages. Both are fully reactive; ServerRequest exposes the body as Flux or Mono, and Mono<ServerResponse> accepts any reactive stream as the body.

ServerRequest also provides access to other HTTP-related data such as the URI being handled, headers, and path variables. Access to the body is provided by the bodyToMono(...) method or equivalent bodyToFlux(...). ServerResponse provides access to the HTTP response. Being immutable, it has to be created using a builder class and the HTTP response status. The headers and body are set by calling various methods. All the examples in the previous code snippet create a response with a 200 (OK) status, JSON content-type, and a body.

But where does the name *functional web framework* comes from? Well, lambdas are the source of this. If you look at the previous code snippet, you will notice that list and create can be easily written as functions. The two lambda functions are shown here:

```
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)));
}
```

The HandlerFunction<ServerResponse> interface, which essentially is a Function<Request, Response<T>>, is side effect free because it returns the response directly, instead of taking it as a parameter. This makes this type of functions very practical because they can be easier to test, compose, and optimize.

Pretty neat, right? Just remember when it comes to lambdas, overusing them comes with the cost of readability and maintainability. Also, I know what you might be thinking right now: *OK, OK, reactive streams and lambdas are cool, but where are the mappings? How is the container supposed to know which function is mapped to an HTTP request?* Well, there are some new classes for that.

Here is the Spring application class that must be run to start the application and dissect it afterward:

```
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();
    }
}
```

The request-handling functions are exposed with the new functional web framework by using a `RouterFunction`, which is used to create an `HttpHandler` by the `RouterFunctions.toHttpHandler` utility method. This `HttpHandler` is used to create a `ServletRegistrationBean` that is a `ServletContextInitializer` used to register `Servlets` in a Servlet 3.0+ container.

A `RouterFunction` evaluates the request URI and checks whether there is a handler function that matches; otherwise, it returns an empty result. Its behavior resembles the `@RequestMapping` annotation, but the advantage of a `RouterFunction` is that expressing a route is not limited to what can be defined using annotation values and is not scattered inside a class. The code is in one place, and it can be overridden or replaced easily. The syntax for writing and composing `RouterFunctions` is really flexible; in the previous code sample, the most practical syntax was chosen.[10]

The easiest way to do an initial testing of the application is to run `SingerApplication` and access the URIs defined in `RouterFunction` in the browser. For example, if you access `http://localhost:8080/test`, you should see the "works!" text.

Still, we haven't provided an example of steams being used as streams, with each value being processed as it is being emitted. Let's create a REST controller, which returns directly a `Flux` object of singers but emits an object every two seconds.

```
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;
```

[10]If you want to take a deeper look into this, there is a blog entry on the official Spring Blog about it: https://spring.io/blog/2016/09/22/new-in-spring-5-functional-web-framework.

```
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
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();
   }
}
```

MediaType.TEXT_EVENT_STREAM_VALUE denotes a special type of plain-text response. This makes sure that the server will create a response that is served with text/event-stream Content-Type, which follows the server-sent events format. The response should contain a "data:" line followed by the message, in this case a text representation of a Singer object, followed by two "\n" characters to end the stream. This applies for only one message, meaning one object being emitted. In the case of multiple objects being emitted, multiple "data:" lines will be generated. Two or more consecutive lines beginning with "data:" will be treated as a single piece of data, meaning only one message event will be fired. Each line should end in a single "\n" (except for the last, which should end with two).

The oneByOne method contains an implementation that combines two streams. One is a Flux implementation returned by the reactive repository and contains singer instances. The other is a steam containing an interval of seconds, generated with the java.time.Duration class. Using the zip method that is used to combine two streams together, you wait for all the streams to emit one element and combine these

elements once into an output value (constructed by the provided combinator). The operator will continue doing so until any source completes. The combinator, in this case, is the `map(Tuple2::getT1)` statement, represented here as a method reference; it can also be written as `map(t -> t.getT1())`, which makes it easier to understand that basically one element in a stream is mapped to the other.

Let's test two new handler functions. There are two ways to do this, using the browser and using the `curl` command. In the browser, if you try to access `localhost:8080/`, all you will get is a blank page, and the browser will ask you if you want to save it `all` to a file. If you do, a pop-up will be displayed showing you the file download in progress. At some point, the download will finish, and the contents of the file will be displayed to you in an editor of your choosing. If you open the file, you will see its contents. If invisible characters are shown, you should be able to see the `"\n"` line terminators. Figure 18-2 shows the contents of the downloaded file. But still, aside from the slow download time, we do not clearly see that a singer is emitted every two seconds. This can be seen only by using the `curl` command, which is available on Unix-based systems. For Windows, you can try `Invoke-RestMethod` in PowerShell. If you call `curl` with the URIs mapped in the previous example, this is what you will see. The output for `curl http://localhost:8080/all` will take a while to execute, and a line will be printed every two seconds.

```
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
```

Figure 18-2. *Response with Content-Type text/event-stream*

```
$  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"}

$
```


For the `http://localhost:8080/one/1` URI, the data is automatically transformed into text, because that is the default accept header, as there is only one entry that is being returned. For `http://localhost:8080/all`, the request hangs, and each message is transformed into text as it arrives. Notice the different syntax; for the stream with multiple messages, the `"data:"` prefix is used for each element sent.

Another way to access the stream returned by an HTTP request for `http://localhost:8080/all` is to use a reactive web client. An interface for that is provided, of course. And here we explain how it is used. Because we need to share the context, we will declare the client in the same Spring Boot `Application` class, and if you look in the console, you will see that the client gets run immediately after starting up the application.

```
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);
      };
   }
}
```

The `WebClient` bean is a reactive, nonblocking alternative to `RestTemplate`, which will replace `AsyncRestTemplate` in future Spring releases. In Spring 5, it is already marked as deprecated.

There are a lot of new components in Spring WebFlux that can be used to write reactive web applications; if we covered them all here, this book would double in size. Just follow Spring's official blog at `https://spring.io/blog` to quickly be updated with all the new components introduced and how they are used. Now let's get into the alternative.

Java 9 Interoperabiliy

Because the Java 9 release date kept being postponed, Spring 5 had to be released based on Java 8, but the team continued working in parallel to adapt it to Java 9 by working with early builds of the JDK. At the time of this writing, most of the novelty components of Java 9 have been stabilized, and the planned release date of September 2017 seems to be realistic. Thus, interoperability with Java 9 is covered in this book based on an early access build. Let's dive right in.

JDK Modularity

JDK modularity is thought to be the biggest improvement in Java 9. Making the JDK modular has the benefit of scalability, as now Java can be deployed on smaller devices. The modularity functionality is known as Project Jigsaw, and the plan was for it to be a part of Java 8, but it was pushed to Java 9 because it wasn't considered stable enough. So, Java 9 introduces the module concept, which is a unit of software that is configured in a file named `module-info.java` that resides in the source code directory of a project and contains the following information:

- *A module name that should follow the package naming convention*: By convention, the module should have the same name as the root package.
- *A set of packages that are exported*: These are considered public APIs and are usable by other modules. If a class is in an exported package, it can be accessed and used outside the module.
- *A set of packages that are required*: These are modules that this module depends on. All public types in packages exported by these modules are accessible and can be used by dependent modules.

This basically means that accessibility is no longer represented by the four Java classifiers: `public`, `private`, `default`, and `protected`. Because the module configuration decides what another module can use from its dependent module, you now have the following:

- Public at module level, to everyone who reads this module (`exports`)
- Public at module level, to a selected list of modules (yes, filtered access, `export to`)
- Public inside the module and public to every other class inside the module
- Private inside the module, typical priv ate access
- `<default>` inside the module, typical default access
- Protected inside the module, typical protected access

This model was applied to the JDK, and apparently a lot of cyclical and unintuitive dependencies were removed. The operation also involved some cleanup because there were packages that were going beyond the scope of JSE.[11]

This topic is important for Spring interoperability because components of the JDK might not be accessible directly anymore. Until now in the book, we didn't really cover Java 9 because the project was built using Java 8, as Java 9 was not stable enough to use. But now the time has come. So, after changing the JDK, the Gradle installation must be upgraded to the latest 4.2 milestone release to support Java 9. After that, the first build of the project will fail with the following message:

```
$ 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
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.
1 error

FAILURE: Build failed with an exception.
```

What happened here is that the package `javax.annotation` is not one of those exported, and any of the annotations in it can no longer be used. The reason why this package and a lot of related ones are no longer accessible is that they contained enterprise components that were more suitable to be part of the JEE. Those packages are part of a module named `java.se.ee`, and according to everything mentioned before, we should just add the following module to our file and everything should work:

```
module collections.main {
   requires java.se.ee;
}
```

But this is not the case because if we open `java.se.ee module-info.java`, here is what we find:

```
@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;
}
```

[11]Read more about this at https://blog.codecentric.de/en/2015/11/first-steps-with-java9-jigsaw-part-1/.

As you can see, there is no line containing `exports java.xml.ws.annotation;`, which is the module that contains the `javax.annotation` package. So, what is the solution here? We add the JEE dependency that contains that package, of course. Here it is as defined in the Gradle configuration file:

```
misc = [
   ...
   jsr250        : "javax:javaee-endorsed-api:7.0"
   ...
]
```

Of course, `misc.jsr250` will have to be added everywhere annotations from that package are used. Let's try that again. Depending on when this book reaches you, you might also see something like this:

```
$ gradle clean build -x test
# all good until here
...
: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
```

What happened here is that `aspectj-aspects` could not be compiled because aspects are not recognized. This happens because the JAR needed to do so is not in the right place according to the Gradle `aspects` plug-in. The plug-in is clearly outdated and does not match the new JDK internal structure, but there is a small fix for this: just copy the `jrt-fs.jar` file from under the `$JAVA_HOME/libs` directory in the location where the plug-in looks for it.

When the build is run again, there will be a lot of deprecation warnings, but at least the build will be successful. Now we could start adding `module-info.java` files for all the modules if we want. Let's take the `chapter02/hello-world` project and define `module-info.java` because it is a small project and should be easy.

```
// 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;
}
```

That's about it. To keep its module names simple, the Spring team decided to break convention; otherwise, instead of `requires spring.context;`, we would have to add `requires org.springframework.context`.[12]

The modularity added in by Project Jigsaw does more than split up the JDK and restrict access to certain packages and components (reflection does not work on nonexported modules); it detects circular dependencies at compile time, improves readability, and detects duplicate module dependencies that differ only by version, eliminating conflicts or chaotic behavior of an application. And because of all these development benefits, this novelty feature of Java 9 deserves its own section in the book.

Reactive Programming with Java 9 and Spring WebFlux

A few sections ago, we introduced the reactive model and mentioned the standard API. Well, move away `reactive-streams.jar`, because there's a new boy in town! We introduce you to the `java.base` module, which exports the package `exports java.util.concurrent`, which among other things contains four functional interfaces with the same purpose all defined in the `java.util.concurrent.Flow` final class.

- `Flow.Processor`, which is the equivalent of `org.reactivestreams.Processor<T>`
- `Flow.Publisher`, which is the equivalent of `org.reactivestreams.Publisher<T>`
- `Flow.Subscriber`, which is the equivalent of `org.reactivestreams.Subscriber<T>`
- `Flow.Subscription`, which is the equivalent of `org.reactivestreams.Subscription<T>`

These interfaces have the same functions as the ones defined by the Reactive Streams project and are used to support the creation of publish-subscribe applications, meaning reactive applications. JDK 9 provides a simple implementation of `Publisher` that can be used for simple use cases and can also be extended depending on the requirements. `SubmissionPublisher<T>` is an implementation class of the `Flow.Publisher<>` interface that also implements `AutoCloseable` and can be used in a `try-with-resources` block.

RxJava[13] is one of the reactive implementation for the JVM designed to support sequences of data/events and adds operators that allow stream composition and filtering by hiding low-level concerns as thread synchronization and thread safety. At the time of writing, there are two versions on RxJava: RxJava 1.*x* and RxJava 2.*x*. RxJava 1.*x* is an implementation of ReactiveX (Reactive Extensions)[14] that is an API for asynchronous programming with observable streams. RxJava1 will be dropped, because it was completely rewritten on top of the Reactive Streams API introduced in the previous section. Spring 5 can work with both, but because RxJava 1.*x* will be dropped in 2018, let's focus on the future.

In the Spring reactive programming example with Project Reactor, we implemented a reactive repository and a reactive client. we'll do this here by using RxJava2. we'll define a reactive repository interface named `Rx2SingerRepo`.

---

[12]Check out the discussion that led to this decision: https://jira.spring.io/browse/SPR-14579.
[13]See https://github.com/ReactiveX/RxJava.
[14]See https://reactivex.io/.

```
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);
}
```

After looking attentively at the previous snippet, you probably noticed the equivalences: the `Flowable` class is the implementation of a stream of elements, and `Single` is the implementation for a stream of `[0..1]` elements. Creating them is just as easy, and you will notice a small API difference for the `Single` type regarding creating an empty stream.

```
Flowable simple = Flowable.just("1", "2", "3");
//or from an existing list: List<Singer>
Flowable<Singer> fromList = Flowable.fromIterable(list);

Single simple = Single.just("1");
//or from an existing object of type Singer
Single<Singer> fromObject = Single.just(null);
```

So, let’s see how the reactive repository implementation looks with RxJava2:

```
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);
   }
}
```

As we can see, the syntax is similar, but there are small differences in the API to take into account. If we were to rewrite the mapping methods as well, this is how they would look:

```
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);
   }
...
}
```

It seems a little more complicated to implement the zipping function with RxJava2, doesn't it?

The test the mappings, you can still use the Spring reactive `WebClient`, which is pretty cool. Or, you can take the new HTTP client from JDK 9 for a test run.

```
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());
```

Just make sure to add a dependency in the `jdk.incubator.httpclient` module in your `module-info.java` file.

This is all that can be said about Java 9 interoperability. The biggest changes that are relevant to the Spring Framework are the modularization and the new reactive API, RxJava2, that Spring already supports. All other Java 9 novelties are covered by another really interesting Apress book, called *Java 9 Revealed*.[15]

Spring Support for JUnit 5 Jupiter

JUnit 5 was mentioned shortly in Chapter 13, and now it is time to dig a little deeper. If you are wondering what all the fuss is about, you can find the simplest answer in the official documentation at `http://junit.org/junit5/docs/current/user-guide/#overview` (see Figure 18-3).

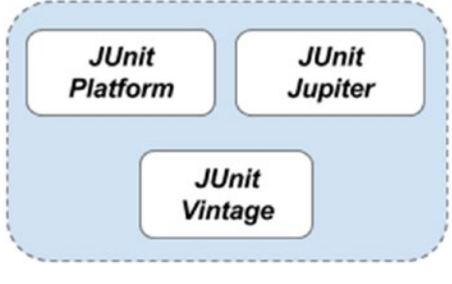

Figure 18-3. *JUnit 5*

The JUnit platform serves as a foundation for launching testing frameworks on the JVM. It comes with a Console Launcher used to launch the platform from the command line and build plug-ins for Gradle and Maven. This launcher can be used to discover, filter, and execute tests; thus, there's no need for Surefire or customizations for the Gradle test task. Also, third-party libraries such as Spock, Cucumber, and FitNesse can plug into the JUnit platform's launching infrastructure by providing a custom `TestEngine`.

JUnit Jupiter is the combination of the new programming model (based on a set of new JUnit5 annotations that reside in the `org.junit.jupiter.api` package) and extension model (the `Extension` API with its `@ExtendWith` annotation is designed to replace JUnit 4's `Runner @Rule` and `@ClassRule`) for writing tests and extensions in JUnit 5. The Jupiter subproject provides a `TestEngine` for running Jupiter-based tests on the platform.

JUnit Vintage as you probably guess provides a `TestEngine` for running JUnit 3 and JUnit 4–based tests on the platform.

[15]You can order it from `www.apress.com/la/book/9781484225912`.

But, let's see it in action, by testing the SingerHandler class. We'll start with a negative test method; specifically, let's try to obtain a singer who does not exist.

```
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(){
     //get singer
     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);
   }
...
}
```

ExchangeFunction is a functional interface that is used to exchange a ClientRequest for a delayed ClientResponse and can be used as an alternative to WebClient. Using such an implementation request can be sent to a running server, and responses can be analyzed. In the previous example, you probably noticed the following things:

- The @BeforeAll annotation is similar to @BeforeClass from JUnit classic; thus, methods annotated with it must be executed before any methods annotated with @Test (or derivate annotations like RepeatedTest) in the current class. What is special about it is that it can be set on nonstatic methods if the @TestInstance(Lifecycle.PER_CLASS) annotation is used at the class level.
- The @Test annotation from package org.junit.jupiter.api is the equivalent of the same annotation from JUnit classic; the only difference is that this annotation does not declare any attributes since test extensions in JUnit Jupiter operate based on their own dedicated annotations.
- The assertNull statement from class org.junit.jupiter.api.Assertions is also analogous to the same annotation in JUnit classic. There are analogous implementations and some extra ones for all the static functions in the Assertions class that lend themselves well to being used with Java 8 lambdas.

Let's see a more detailed example. This time, we are testing the editing of a singer instance. First we test that we definitely retrieved the correct singer, by checking firstName and lastName with grouped assertions.

```
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()  {
      //get  singer
      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());

      //edit singer
      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());
   }
}
```

We can do even more. In the previous example, we could condition the execution of the `assertEquals` assertions by the `assetNotNull` assertions above them. In the previous case, both `assertEquals` statements are executed and fail if the wrong singer is returned.[16] The code to declare the previous `assertEquals` execution dependent of the `assertNotNull` execution is shown here:

```
assertAll("singer", () ->   {
        assertNotNull(singer);
   assertAll("singer",
          () -> assertEquals("John", singer.getFirstName()),
          () -> assertEquals("Mayer", singer.getLastName()));
});
```

We'll stop here with JUnit-specific testing components and move on to more Spring testing novelties. Aside from the `WebClient` interface, for the reactive programming model, `spring-test` now includes `WebTestClient` for integrating testing support for Spring WebFlux. The new `WebTestClient`, similar to `MockMvc`, does not need a running server. That doesn't mean it cannot be used with an existing one. In the following sample, a `WebTestClient` is used with the application run from the `SingerApplication` class:

```
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
   public static void init() {
      client = WebTestClient
            .bindToServer()
            .baseUrl("http://localhost:8080")
            .build();
   }
```

[16]Just try to change the URI to `http://%s:%d/singers/2`.

```
    @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();
    }
}
```

WebTestClient is the main component for testing WebFlux server endpoints. It has a similar API to WebClient and delegates most of the work to an internal WebClient that is focused mainly on providing a test context. To run integration tests on an actual running server, the bindToServer() method must be called. The previous example is quite simple. WebTestClient is bound to an address where an actual application is running; thus, it does not need to define its own mappings or routing functions. But in certain situations when this is necessary, it can be done because there are many more configuration methods available. The test method in the following code snippet uses a custom RouterFunction by calling 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();
}
```

Another feature that is worth mentioning here is that with Spring JUnit 5, integration tests can be run in parallel. And now we have to leave the Singer application behind so that we can keep the code samples on point. To show you how tests can be run in parallel, a simple Spring Boot application will be used that declares only a bean of type FluxGenerator, which, as the name says, generates Flux instances. The code of the configuration/entry point to the application and the simple bean type FluxGenerator is depicted here:

```
//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);
   }
}

//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();
   }
}
```

Next, we will create two test classes, `IntegrationOneTest` and `IntegrationTwoTest`, that will have two test methods declared each; they will not test anything and will just use the `FluxGenerator` bean to get a `Flux` instance and print its contents. The two classes are depicted here, and as you can see, the sets of values are different so we can track the execution in the command line and make sure it is done in parallel.

```
//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) {
         }
      }
   }

//IntegrationTwoTest.java
package com.apress.prospring5.ch18.test;
... // same imports as  above
@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);
   }
}
```

The class to execute those tests is defined here. It contains two test methods—one that executes the test in parallel and one that executes them in a linear manner, one after the other.

```
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);
    }
}
```

JUnitCore is a facade for running tests. It supports JUnit 4, 3.8 tests, and mixtures. It receives as a parameter a Computer instance, or an extension of the ParallelComputer instance. The Computer class is used to execute tests normally, in a linear manner one after the other. The ParallelComputer instance allows running tests in parallel, and its constructor receives two Boolean arguments. The first one is for classes' execution in parallel; the second is for methods. In the previous example, we set them both to "true" to tell the JUnit runner that we want classes executed in parallel and methods in them as well. That is why IntegrationOneTest is slightly different than its counterpart IntegrationTwoTest class.

If you run the executeTwoInParallel methods, you should see in the console a log similar to the output depicted here:

```
...
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
...
```

The last subject we want to touch here is the JUnit 5 Extension API. At its core is the Extension interface, which is just a marker interface for components that can be registered explicitly with @ExtendWith or automatically via Java's ServiceLoader mechanism. In Spring 5, the SpringExtension has been added to the spring-test module, which implements a lot of Jupiter interface derivatives of @ExtendWith to integrate the Spring TestContext Framework into JUnit 5's Jupiter programming model. To use this extension, simply annotate a JUnit Jupiter-based test class with @ExtendWith(SpringExtension.class), @SpringJUnitConfig, or @SpringJUnitWebConfig.

Let's see a simple example. We will create a class namedTestConfig that will declare a bean of type FluxGenerator and create a test class to test this bean. The two classes are depicted here:

```
//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();
   }
}

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;

//JUnit5IntegrationTest.java
@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());
   }
}
```

There is still a lot of work being done on Spring JUnit 5 support. Thus, the source code introduced in this section might suffer some modifications in future releases of Spring. The area is so new it hasn't even been introduced in the Spring documentation yet,[17] which will probably be updated at some point in the future.

Summary

In this chapter, we provided high-level overviews of a few projects in the Spring portfolio. We took a look at Spring Batch, JSR-352, Integration, XD, WebFlux, and Spring's support of JUnit 5, each providing its own unique capabilities aimed at simplifying specific tasks at hand for you as the developer. Some of these projects are new, and some have been proven stable and solid, serving as ideal foundations for other frameworks. We encourage you to take a look at these projects in deeper detail, as we think they will greatly simplify your Java projects in general.

[17]The Testing chapter of the Spring reference does not even mention JUnit 5 yet; see https://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#testing.

APPENDIX A

Setting Up Your Development Environment

This appendix helps you set up the development environment that you will use to write, compile, and execute the Spring applications discussed in this book.

Introducing Project pro-spring-15

Project `pro-spring-15` is a three-level Gradle project. `pro-spring-15`[*] is the root project and is the parent of the projects named `chapter02` through `chapter18`. Each of these `chapter**` projects is also a parent project for the module projects nested under it. The project was designed in this way to help you get oriented with the code and easily match each chapter to the source code associated with it.

*At the time when this Appendix was being written, the project was still in raw form. It was later completed, renamed to pro-spring-5 and hosted on the Apress official repository located at: `https://github.com/Apress/pro-spring-5`. All written above is still valid, except for the project name.

© Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho 2017

I. Cosmina et al., *Pro Spring 5*, https://doi.org/10.1007/978-1-4842-2808-1

Figure A-1 shows the project structure.

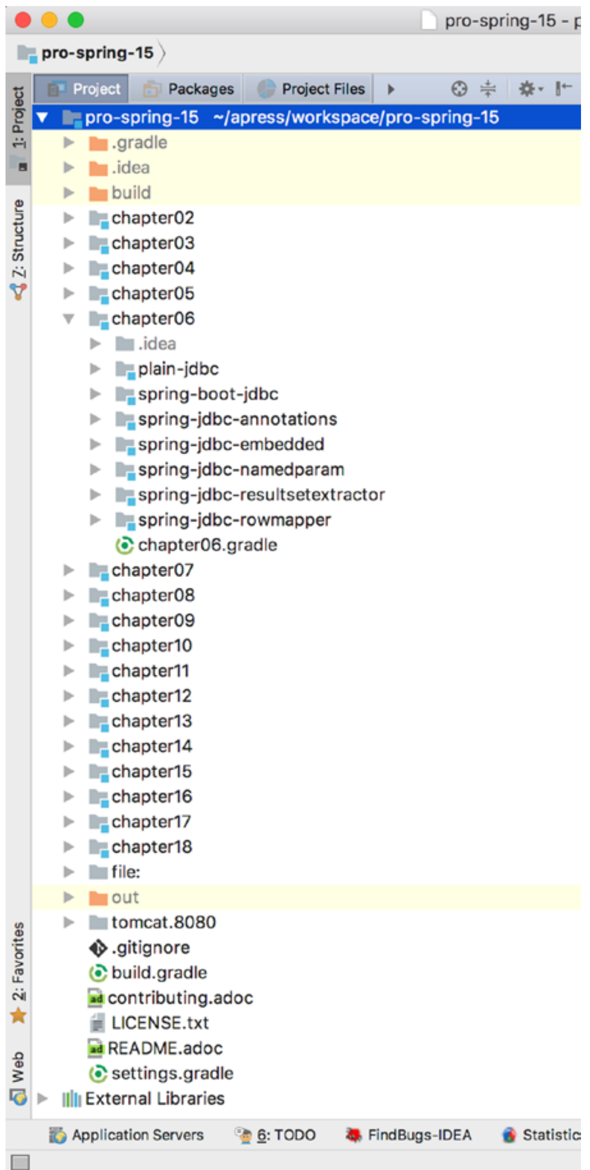

***Figure A-1.** pro-spring-15 project structure in Intellij IDEA*

Understanding the Gradle Configuration

The pro-spring-15 project defines a set of libraries available for the child modules to use, and it has the Gradle configuration in a file typically named build.gradle. All the modules on the second level (in other words, the chapter** projects) have a Gradle configuration file named [module_name].gradle (for example, chapter02.gradle) so you can quickly find the configuration file with the settings that are common to the projects for that chapter. Also, there's a closure element in pro-spring-15/settings.gradle that verifies at build time that all chapter modules have their configuration file present.

```
rootProject.children.each { project ->
    project.buildFileName = "${project.name}.gradle"
    assert project.projectDir.isDirectory()
    assert project.buildFile.exists()
    assert project.buildFile.isFile()
}
```

If a Gradle build file is not named the same as the module, then when executing any Gradle task, an error is thrown. The error is similar to the one depicted in the following code snippet, where the chapter02.gradle file was renamed to chapter02_2.gradle:

```
$ 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'.
> 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
```

This was a development choice; the configuration file of a chapter module is more visible in an editor this way. Plus, if you want to modify the configuration file for a chapter module, you can easily find the file in IntelliJ IDEA using a unique name.

In the pro-spring-15 project, each module on the third level (in other words, the children of chapter**) has a Gradle configuration file named build.gradle. This was also a development choice; in Gradle, having uniquely named Gradle configuration files is not possible at this level in this particular configuration.

Another approach for a multimodular project would have been to have in the main build.gradle file specific closures to customize the configuration for each module. But in the spirit of good development practices, we decided to keep the configurations for the modules as decoupled as possible and in the same location as the module contents.

The `pro-spring-15/build.gradle` configuration file contains a variable for each software version being used. These variables are used to customize the dependency declaration strings that are grouped in arrays named for a specific purpose or technology. This file also contains a list of public repositories from where the dependencies are downloaded.

In the following configuration, you can see the versions and the array for the persistency libraries used in the project:

```
ext {
   ...
   //persistency libraries
   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"
}
```

In the projects on the second level of the hierarchy (in other words, the `chapter**` projects), you'll find the `chapter**.gradle` configuration files. These dependencies are referred to by their array name and the name associated with then. These files also contain the project group name that is specific to that chapter, additional plug-ins, and extra Gradle tasks. Here you can see the `chapter08.gradle` file:

```
subprojects {
   group 'com.apress.prospring5.ch08'
   apply plugin: 'java'

   /*Task that copies all the dependencies under 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
   }
}
```

The if (!project.name.contains("boot")) is necessary here because under project chapter08 one or more Spring Boot projects are nested, and as Spring Boot comes with its own fixed set of dependencies and versions for them, you do not want the configuration defined in this file to be inherited, which can lead to conflicts or unpredictable behavior.

For projects on the third level, you can further customize the configurations inherited from the chapter** parent by adding their own dependencies and declaring their own manifest file specifications, their own plug-ins, and their own tasks. The following is a simple file called chapter12\spring-invoker\build.gradle (if you want to check out a heavily customized Gradle configuration, check out the chapter08\jpa-criteria\build.gradle file):

```
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(' '))
   }
}
```

Building and Troubleshooting

After you clone the source code (or download it), you need to import the project in the IntelliJ IDEA editor. To do this, follow these steps:

1. Select from the Intellij IDEA menu File ➤ New ➤ Project from Existing Sources (as shown in Figure A-2).

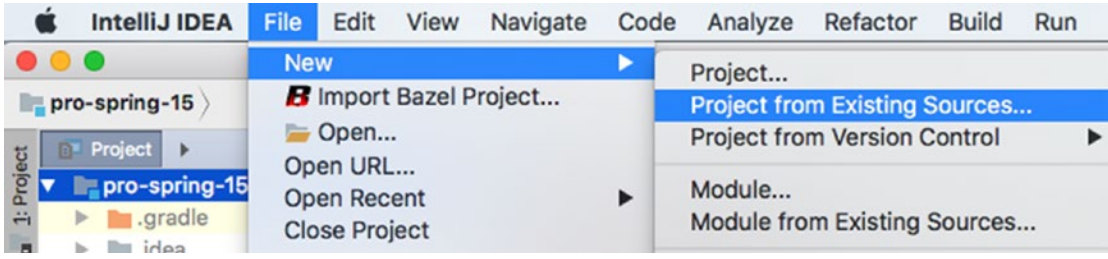

Figure A-2. Project import menu options in Intellij IDEA

After selecting the proper option, a pop-up window will appear requesting the location of the project (as shown in Figure A-3).

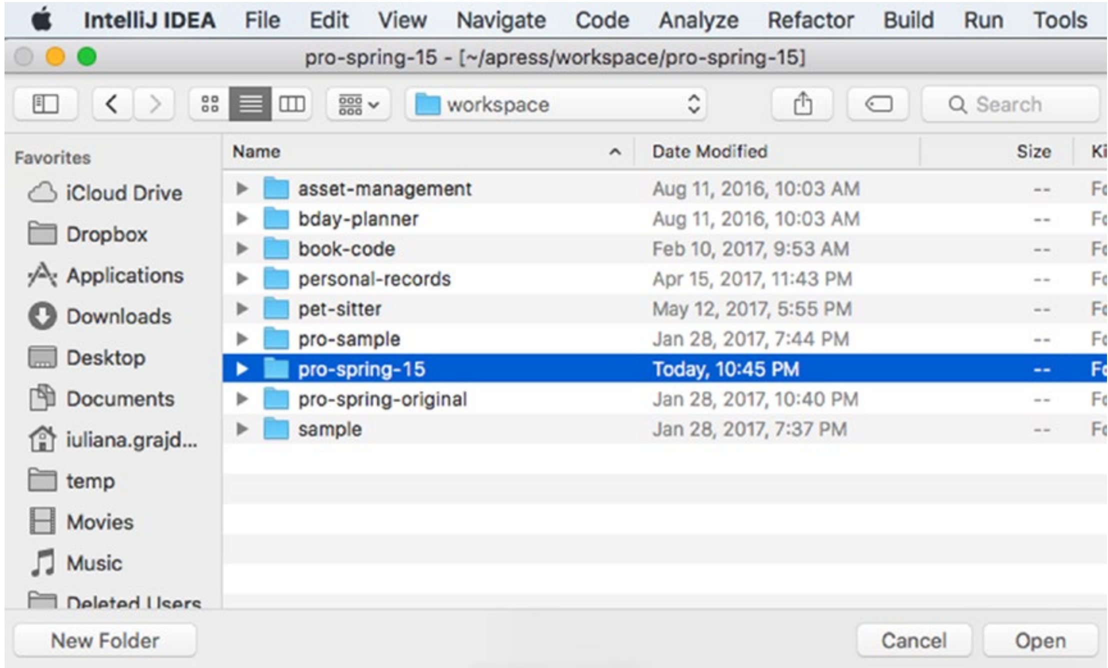

Figure A-3. Selecting the project root directory pop-up in Intellij IDEA

2. Select the `pro-spring-15` directory. A pop-up will ask for the project type. Intellij IDEA can create its own type of project from the selected sources and build it with its internal Java builder, but this option is not useful here because `pro-spring-15` is a Gradle project.

3. Check the "Import project from external model" radio button and select Gradle from the menu, as depicted in Figure A-4.

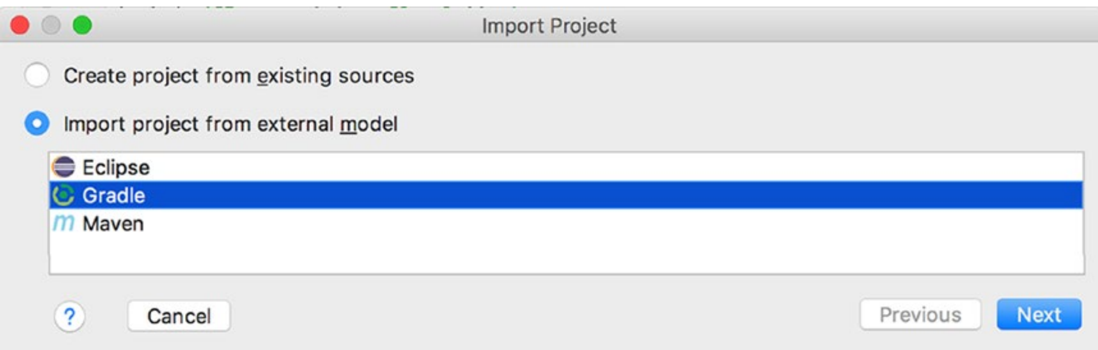

***Figure A-4.** Selecting the project type Intellij IDEA*

4. The last pop-up window will appear and ask for the location of the `build.gradle` file and the Gradle executable. The options will be already populated for you (as shown in Figure A-5). If you have Gradle installed on the system, you might want to use it.

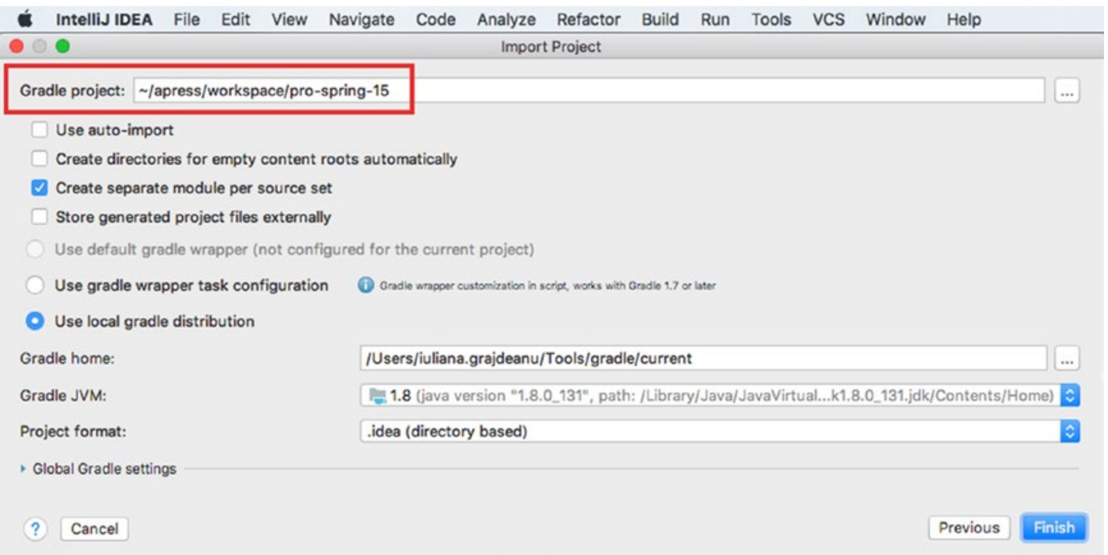

***Figure A-5.** Last pop-up for project import in Intellij IDEA*

Before getting to work, you should build the project. This can be done from Intellij IDEA by clicking the Refresh button, as shown by the (1) in Figure A-6. Clicking this button will cause IntelliJ IDEA to scan the configuration of the project and resolve dependencies. This includes downloading missing libraries and doing an internal light build of the project (just enough to remove compile-time errors caused by missing dependencies).

The Gradle `compileJava` task, marked with (3) in Figure A-6, executes a full build of the project. You can also do the same this from the command line, by executing the Gradle build command, as depicted below:

```
.../workspace/pro-spring-15 $ gradle build
```

Or you can do this from Intellij IDEA, by double clicking the build task, as depicted by the (2) in Figure A-6.

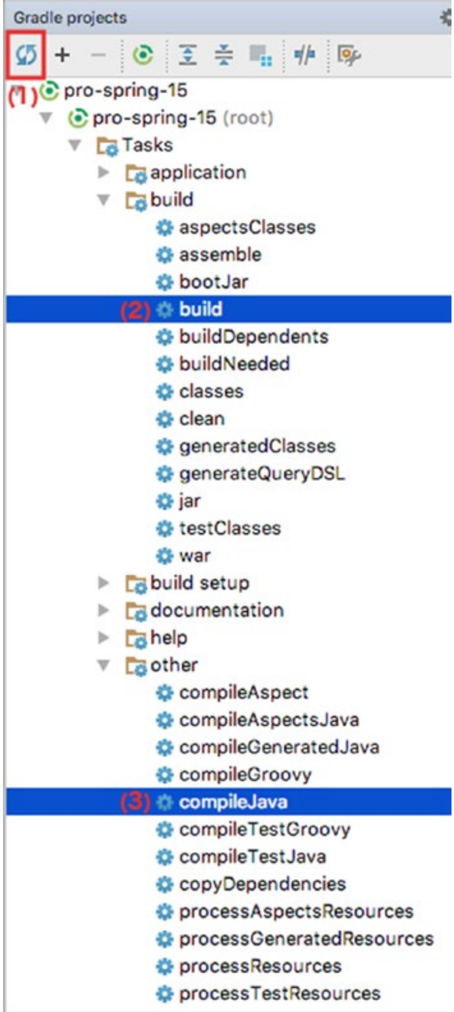

Figure A-6. *Last pop-up for project import in Intellij IDEA*

This will execute the following set of tasks on every module:

```
: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
...
```

In the previous example, the tasks were depicted only for module `chapter02`. The Gradle build task will execute all the tasks it depends on. As you can see, it does not run the `clean` task, so you need to make sure to run this task manually multiple times when building a project to make sure the most recent versions of the classes are used.

Because this project contains tests that were designed to fail, executing this task will fail. You can instead just execute tasks **`clean`** and **`compileJava`**. Another option is to execute the Gradle `build` task, but to skip the tests using `"-x"` argument:

```
.../workspace/pro-spring-15 $ gradle build -x test
```

Deploy on Apache Tomcat

There are a few web applications under the `pro-spring-15` project. Most of them are Spring Boot applications that run on an embedded server. But there are certain advantages in using an external container like the Apache Tomcat server. Starting the server in debug mode and using breakpoints to debug an application is much easier to do, but that's only one advantage. An external container can run multiple applications at a time without the need to stop the server. Embedded servers are useful for testing, fast development, and educational purposes, like Spring Boot is, but in production, application servers are preferred.

Here is what you have to do if you are interested in using an external server. First download the latest version of Apache Tomcat from the official site.[1] You can get version 8 or 9; they will both work with the sources of this book. Unpack the archive somewhere on your system. Then configure an IntelliJ IDEA launcher to start the server and deploy the chosen application. This is quite easy to do; follow these steps:

1. From the runnable configuration menu, choose Edit Configurations, as shown by (1) in Figure A-7. A pop-up window will appear and list a set of launchers. Click the plus (+) sign and select the Tomcat Server option. The menu will expand; select Local, as shown by (2) in Figure A-7, because you are using a server installed on your computer.

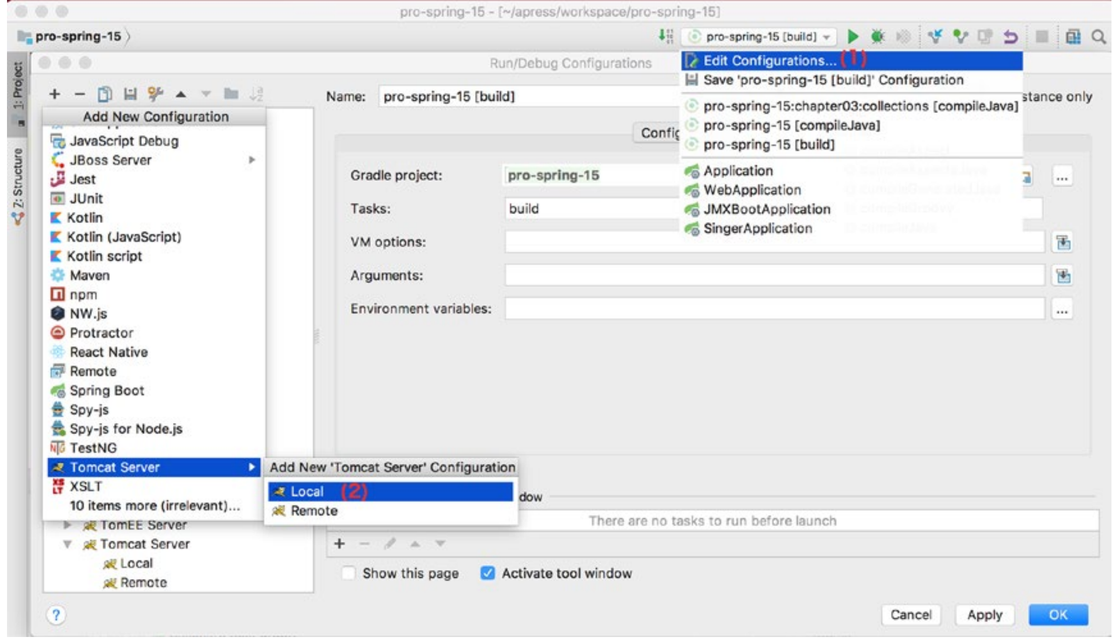

Figure A-7. *Menu options to create a Tomcat launcher in Intellij IDEA*

2. A pop-up window like the one in Figure A-8 will appear and will request some information.

[1]The Apache Tomcat official site is at `http://tomcat.apache.org/`.

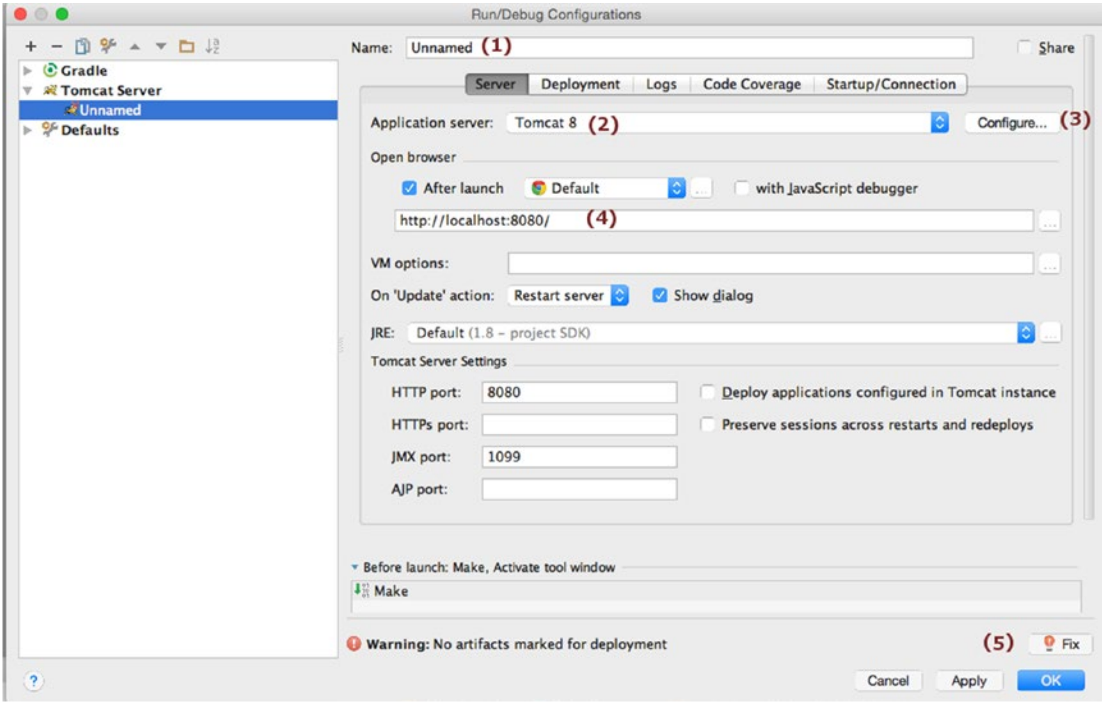

Figure A-8. *Pop-up to create a Tomcat launcher in Intellij IDEA*

In Figure A-8, you'll see that some items are numbered. Here is what the numbers mean:

- 1 is the launcher name. You can put anything here, preferably the name of the application you are deploying.
- 2 is the Tomcat instance name.
- 3 is the button that will open the pop-up window to insert the Tomcat instance location, as depicted in Figure A-9.

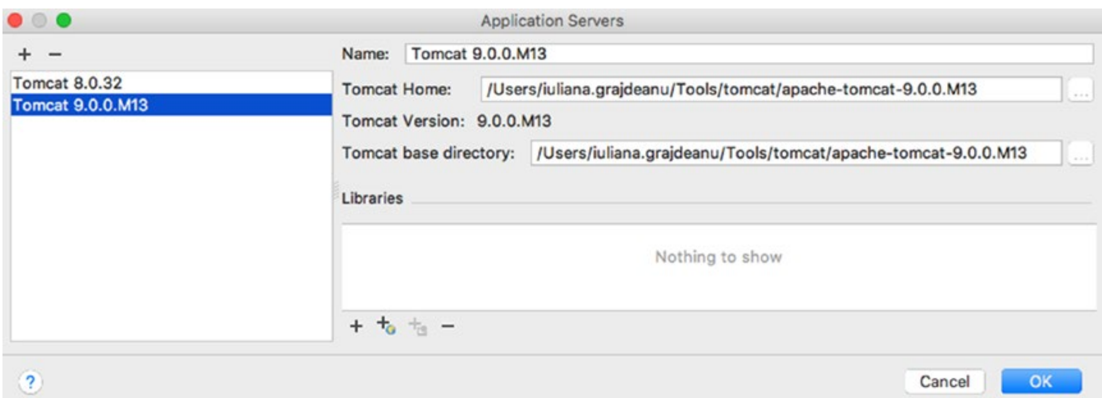

Figure A-9. *Configuring the Tomcat instance in Intellij IDEA*

- 4 is the URL where the Tomcat server can be accessed.
- 5 is the Fix button, which is used to choose an artifact to deploy on the Tomcat instance. If there is no web archive set to be deployed to Tomcat, this button will be displayed with a red lightbulb icon on it.

3. Click the Fix button and select an artifact. IntelliJ IDEA will detect all artifacts available (Figure A-10) and present them to you in a list. If you intend to open the server in debug mode and use breakpoints in the code, select an artifact with the name postfixed with `(exploded)`; this way, Intellij IDEA manages the contents of the exploded WAR and can link the actions in the browser with the breakpoints in the code.

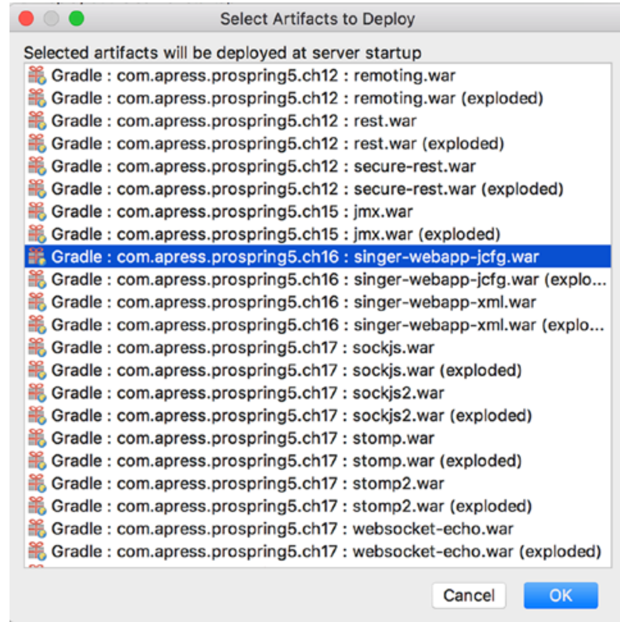

Figure A-10. *Deployable artifact list in Intellij IDEA*

4. Complete the configuration by clicking the OK button. You can specify a different application context by inserting a new value in the Application Context field. Choosing a different application context will tell Tomcat to deploy the application under the given name. The application will be accessible via this URL:

   `http://localhost:8080/[app_context_name]/.`

Other application servers can be used in a similar way as long as IntelliJ IDEA provides plug-ins for them. Launcher configurations can be duplicated, and multiple Tomcat instances can be started at the same time as long as they function on different ports, which can make for faster testing and comparisons between implementations. IntelliJ IDEA is really flexible and practical, and that's why we recommend it for practicing the exercises in this book. The Gradle projects can also be imported in Eclipse and other Java editors that support Gradle.

Index

A

Advanced Message Queuing Protocol (AMQP)
- JMS, 606
- rabbitListenerContainerFactory, 609–611
- RabbitMQ, 608, 611
- RPC calls, 608
- Spring Boot, 612–613
- WeatherService, 607–608
- XML configuration, 609

After-returning advice
- KeyGenerator, 224–227
- SimpleAfterReturningAdvice, 224–225

ApplicationContext interface
- ApplicationEvent, 171–173
- event usage, 173–174
- features, 167
- getMessage(), 170
- MessageSource, 168, 170

ApplicationContextAware interface, 148–150

Application programming interfaces (APIs), 10

Around advice
- MethodInterceptor, 227
- ProfilingDemo, 229–230
- ProfilingInterceptor, 228–229

AspectJ
- AnnotationAdvice, 285–286
- aop namespace, 286
- AOP configuration, 291
- configuration, 287–288
- Java configuration, 284–285
- pointcut expression, 246
- singleton aspects, 292–294
- Spring Boot, 289–290
- test methods, 288–289

Aspect-oriented programming (AOP), 2, 10
- advice types, 218
  - after-returning, 224–227
  - around, 227–230
  - before, 219–224
  - throws, 230–232
- AgentDecorator, 214–215
- alliance, 214
- aop namespace
  - advice class, 278
  - configuration, 281
  - MethodBeforeAdvice, 278
  - MethodInterceptor, 282
  - pointcut, 283
  - simpleBeforeAdvice, 279
  - sing(), 280
- declarative configuration, 271
- definition, 212–213
- dynamic, 213
- introductions (*see* IntroductionInterceptor)
- joinpoints, 216
- OOP, 211
- ProxyFactory class, 217
- static, 213

AssertTrue, 534

Auditing strategies, 451–452

B

BatchSqlUpdate class
- insertWithAlbum(), 343–346
- test method, 346–347

Bean life-cycle management
- ApplicationContext, 130
- default-init-method, 131
- destruction, 139
- initialization callback, 128
- InitializingBean, 132–134
- init-method, 129–130
- interface mechanism, 127
- JSR-250, 134–136
- post-initialization and pre-destruction, 127

Bean-managed transactions (BMTs), 467

Before advice
- SecureBean, 221, 223–224
- SecurityAdvice, 222–224

© Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho 2017
I. Cosmina et al., *Pro Spring 5*, https://doi.org/10.1007/978-1-4842-2808-1

Before advice (*cont.*)
  SecurityManager, 221–222
  SimpleBeforeAdvice, 219
  sing(), 220
BookwormOracle class, 43

C

Castor XML
  @Controller, 586
  field handler, 583–584
  mapping file, 582–583
  properties, 584
  @RequestMapping, 586
  @ResponseBody, 586
  SingerController, 584–585
  Spring web application, 587–590
CGLIB, 249–251
ClassPathXmlApplicationContext, 34
Closure, 639, 643–644
Container-managed transactions (CMTs), 467
Contextualized dependency lookup (CDL), 38–40
curl command, 590–592

D

Data Access Object (DAO), 2
  components, 317
  JdbcSingerDao, 317–318
  service layer, 735
  Spring Boot, 733–734
  Spring Security, 737–738
  web layer, 735–737
Database connections. *See* DataSource
Database operation. *See* Query data
DataSource
  configuration, 311–312
  DAO (*see* Data Access Object (DAO))
  DriverManagerDataSource, 310
  embedded database, 315–316
  JNDI data source, 314
  test class, 312–313
DbUnit, 634–637
DefaultPointcutAdvisor, 236
Dependency injection (DI)
  annotations, bean naming, 100–105
  AOP, 10
  ApplicationContext, 48, 49
  autowiring, 112–122
  BeanFactory Implementations, 46–48
  bean inheritance, 122–123
  bean instantiation mode, 105–108
  bean-naming structure, 95–99
  bean scope, 108
  collection injection, 77–84
  components, 50–52, 54
  configuration options, 49
  constructor, 40
  constructor injection, 58–63
  EL, 11
  field injection, 64–65
  injecting beans, 73–74
  injecting simple values, 66–68
  injection and ApplicationContext nesting, 75–77
  injection parameters, 66
  interface-based design, 8
  Java 9, 10
  JavaBeans (POJOs), 8
  Java configuration, 54–56
  Lookup Method Injection, 85–91
  Method Injection, 84
  Method Replacement, 92–95
  resolve dependencies, 109–111
  setter injection, 41, 56–58
  SpEL, 69–72
  XML configuration, 49
Destroy-method
  destructiveBean, 139–140
  DisposableBean, 141–142
  GenericXmlApplicationContext, 139–140
  @PostConstruct, 145
  shutdown hook, 146
Development environment
  gradle configuration file, 831–833
  Intellij IDEA
    application, 838
    deployable artifact, 840
    pop-up window, 834, 838–839
    project import, 834–836
    project type, 835
    Tomcat launcher, 838
  project pro-spring-15, 829–830
displayInfo() method, 78, 88
doSomething() method, 86
DyanmicMethodMatcherPointcut, 239–242
Dynamic language code, 652, 654

E

Enterprise integration patterns (EIP), 17
Enterprise JavaBeans (EJB), 2
EntityManagerFactory, 479
Entity Versioning. *See* Hibernate Envers
Environment abstraction. *See* PropertySource abstraction
Expression Language (EL), 11

F

FactoryBeans
accessing, 156
factory-method attributes, 157–158
MessageDigester
configuration, 153–154
implementation, 152
isSingleton() property, 153
Java configuration, 155
factory-method attributes, 157–158
Field formatting
ConversionServiceFactoryBean, 523–524
custom formatter, 521–523
Field injection, 41, 64–65
formatMessage() method, 92
Front-end unit test, 637

G

GenericXmlApplicationContext, 54, 88
getFriendlyName() method, 81
getMySinger() method, 86
Google Web Toolkit (GWT), 13
Groovy, 201–204
closure, 643–644
dynamic typing, 642
Spring
age category rule, 650–652
DSLs, 644–645
dynamic language code, 652, 654
rule engine, 646–648
rule factory, 648–650
syntax, 643

H

Hibernate Envers
adding tables, 452–453
auditing strategies, 452
EntityManagerFactory, 453–455
history retrieval, 456–457
properties, 455–456
testing, 458–459
Hibernate module
AbstractEntity, 390–391
annotating entity and fields, 390
configuration
DatabasePopulator, 388
data source, 386–387
DbIntializer, 389
entities, 386, 389
in-memory database, 388
data model, 356–357
deleting data, 384, 386
inserting data, 378–382
internal mechanism, 391
JCP, 355
mapping (*see* Object-relational mapping (ORM))
properties, 361–362
SessionFactory, 358–361
session interface (*see* Session interface)
updating data, 382–384
Hibernate Query Language (HQL), 372

I

InitializingBean
afterPropertiesSet(), 138
init-method, 137–138
@PostConstruct, 138
Integration test
configuration class, 629–630
dependencies, 623
infrastructure classes, 627
Java configuration, 625–626
service-layer testing, 623–624
TestExecutionListener, 627–629
IntelliJ IDEA, 831
Interface-based design evolution, 8–9
IntroductionInterceptor
Advisor, 268–269
Contact, 264
interface structure, 264
IsModified, 266
mixin, 266–267
object modification detection, 265–266
per-instance, 264–265
PointcutAdvisor, 264
Inversion of control (IoC), 1
capabilities
bean life cycle, 125
configuration enhancements, 126
FactoryBeans, 125
Groovy, 126
JavaBeans PropertyEditors, 125
Java classes, 126
portability, 126–127
Spring ApplicationContext, 126
Spring aware, 125
Spring Boot, 126
CDL, 39–40
dependency injection, 45
dependency pull, 38
injection *vs.* lookup, 41–42
setter injection *vs.* constructor injection, 42–44
IsModified interface, 266

J

Java 9, 10
  interoperability, 774, 811
  reactive programming with, 814–817
  support for, 10
Java classes
  ApplicationContext
    AppConfigOne, 181
    @ComponentScan, 182–184
    MessageRenderer, 176–177
    @PropertySource, 180–181
    @Service, 182
    StandardOutMessageRenderer, 176–177
    static internal class, 179–180
    XML configuration, 177–179
  Spring mixed configuration, 185–186
Java Community Process (JCP), 8
Java Database Connectivity (JDBC), 2
  extensions, 350
  generated key, 340–342
  implementation, 297
  lambda expression, 329
  MySQL, 305
  ORM, 350
  PlainSingerDao, 307
  relational database, 297
  simple data model
    database, 300
    DEBUG level, 303, 304
    entities, 300–303
    entity-relationship, 298
    MySQL, 299
    resources, 299
  SingerDao, 305–307
  Spring Boot, 351–353
  used packages, 309–310
Java Data Objects (JDO), 2, 355
Java Development Kit (JDK), 21
  proxies, 250
Java Enterprise Edition (JEE), 1
Java Management Extensions (JMX) monitoring
  hibernate statistics, 659–661
  JEE applications, 655
  managed beans, 655
  Spring bean, 656–657
  Spring Boot, 661–662, 664
  VisualVM, 657–659
Java Message Service (JMS)
  configuration classes, 572–573
  Gradle configurations, 570
  HornetQ, 571
  @JmsListener, 573–574
  queue, 570
  sending messages, 574–576
  topic, 570
Java Persistence API (JPA), 355
  criteria API
    CriteriaBuilder, 426
    CriteriaQuery.from(), 426
    CriteriaQuery.select(), 426
    execution, 428
    findByCriteriaQuery(), 424–425
    meta-model class, 423–424
    predicate, 426
    Root.fetch(), 426
    testing, 427
  Spring Boot
    adding dependencies, 460–461
    configuration, 460–461
    DBInitializer, 462–463
    InstrumentRepository, 463–464
    run(), 465
    SingerRepository, 464–465
Java Runtime Environment (JRE), 21
Java Standard Tag Library (JSTL), 13
Java Transaction API (JTA) transactions
  configuration, 491, 493–494
  EntityManagerFactory, 494
  findAll(), 498
  infrastructure, 491
  principle, 507
  rollback, 500–501
  save(), 496–497
  ServicesConfig, 494, 496
  Spring Boot, 501, 503–506
  testing, 499
  UserTransactionService, 496
Java Transaction Service (JTS), 469
Java virtual machine (JVM), 21
JdbcTemplate class
  DAO class, 321–323
  DAO implementation, 331
  dataSource, 332
  JSR-250 annotation, 331
  named parameters, 324
  RowMapper<T>, 325–327
JEE 7 container[7], 18
JPA 2.1
  component scan, 398
  dataSource bean, 398
  EntityManagerFactory, 394–398
  native queries, 421–422
  ORM mapping, 398, 400
  query data
    createQuery(), 414–415
    deleting data, 419–420
    findAll(), 404
    findAllWithAlbum(), 406, 408

findById(), 409
getSingleResult(), 409
inserting data, 415–417
@NamedQuery, 404
SingerService, 400–404
SingerSummary, 412–413
testFindAll(), 406
testFindAllWithAlbum(), 408
TypedQuery.getResultList(), 405
untyped results, 410–412
updating data, 417–419
transactionManager bean, 398
JpaRepository, 436
jQuery and jQuery UI
CKEditor, 714–715
jqGrid, 715–717
pagination
SingerController, 719–720
SingerGrid, 720–721
SingerService, 718
view, 712–713
JSR-352, 783–785
JSR-250 annotations, 143–144
JSR-330 annotations, 201
MessageProvider and
ConfigurableMessageProvider, 198
MessageRenderer and
StandardOutMessageRenderer, 199
JSR-349 bean validation, 527

K

KeyHelper instance, 85

L

Logic unit tests
create(), 621–623
dependencies, 619
listData(), 620–621
MVC controllers, 620

M

Many-to-many mappings, 369–371
MappingSqlQuery<T>
SelectAllSingers, 333–334
setDataSource(), 334–335
SqlUpdate, 338–340
test method, 336–337
Message-oriented middleware
(MOM), 606
MessageSourceResolvable interface, 171
Mixin, 266–267
Model-View-Controller (MVC), 2, 674–675

N

Native queries
create and execute, 421
SQL ResultSet mapping
findAllByNativeQuery(), 422
@SqlResultSetMapping, 421
NewsletterSender interface, 44

O

Object modification detection, 265–266
Object-relational mapping (ORM)
@Column, 364
data model, 363
@Entity, 364
JPA annotations, 398, 400
libraries, 355
many-to-many mappings, 369–371
one-to-many mappings, 367–369
simple mapping, 364
Album class, 365–367
Singer class, 363–365
@Table, 364
Object/XML Mapping (OXM), 12, 23
One-to-many mappings, 367–369

P

performLookup(), 40
PicoContainer, 17
PlatformTransactionManager, 469–470
Pointcuts
Advisors and, 233–249
AnnotationMatching
Pointcut, 247–249
AspectJ's pointcut expression, 246
ClassFilter, 234
composable, 259–262
control flow, 256–258
DefaultPointcutAdvisor, 236
DyanmicMethodMatcher
Pointcut, 239–241
implementations, 235–236
name matching, 242–244
ProxyFactory, 233
regular expressions, 244–245
StaticMethodMatcherPointcut, 236–239
Project object model (POM), 20, 362
PropertyEditor
configuration, 162–163
custom, 164, 166–167
implementations, 159–162
spring-beans, 158–159
String, 163–164

PropertySource abstraction
ApplicationContext, 193
AppProperty, 196
MutablePropertySources, 195
ProxyFactoryBean class
ApplicationContext, 274
configuration, 273–274
implementation, 272–273
introductions, 275–277
ProxyFactory class
CGLIB, 249–251
JDK, 250
performance, 251–254
testing, 255

■ Q

Query data
createQuery(), 414–415
deleting data, 419–420
findAll(), 404
findAllWithAlbum(), 406, 408
inserting data, 415–417
@NamedQuery, 404
SingerService, 400–404
SingerSummary, 412–413
testFindAll(), 406
testFindAllWithAlbum(), 408–412
TypedQuery.getResultList(), 405, 406
updating data, 417–419

■ R

RabbitMQ connection, 608
Refreshable beans, 648
Remoting support
AMQP (*see* Advanced Message Queuing Protocol (AMQP))
ContactService
configuration, 564–566
HttpInvokerConfig, 567
SingerService, 562–563
Spring HTTP invoker, 568–569
data model, 558–560
dependencies, 560–561
RESTful-WS
curl, 590–592
dependencies, 580
manipulating resources, 580
RestTemplate, 592–597
Spring Boot, 602–606
Spring MVC, 581–582
Spring security, 597–602
XMLHttpRequest and properties, 581
RestTemplate class, 592–597
ResultSetExtractor interface
findAllWithAlbums(), 327–328
lambda expressions, 329
test class, 329–330
Rich Internet applications (RIAs), 13

■ S

Scripting support
Groovy (*see* Groovy)
in Java, 640–641
Selenium, 638
Service layer
DAO, 670
data model, 666–669
SingerService, 670–671
Session interface
querying
associations fetching, 375–378
HQL, 372
lazy fetching, 372–375
@Resource, 371
SingerDaoImpl class, 371
setDependency() method, 41
setOracle() method, 74
Simple (or Streaming) Text-Oriented Message Protocol (STOMP)
MVC controller, 766–767, 769
stock-ticker application, 765–766
WebConfig, 771–772
Spring
community, 15
Context, 2
data access, 12
description, 1
dynamic scripting support, 14
GitHub, 20
Google Guice, 17
guides, 19
inversion of control, 1
JAR files, 19
Java developers, 1
JBoss Seam Framework, 17
JDK, 21
JEE, 13
JEE 7, 18
job scheduling support, 14
mail support, 14
MVC, web tier, 13
object/XML mapping, 12
origins of spring, 15
packaging options, 19
PicoContainer, 17
remoting support, 14

simplified exception handling, 15
Spring 0.9, 2
Spring Batch and integration, 17
Spring Boot, 16
Spring Core, 2
Spring GitHub repository, 20
Spring Security project, 16
Spring 1.x, 2
Spring 2.x, 2
Spring 2.5.x, 3–4
Spring 3.0.x, 4
Spring 3.1.x, 5
Spring 3.2.x, 5
Spring 4.0.x, 6
Spring 4.2.x, 6–7
Spring 4.3.x, 7
Spring 5.0.x, 7
STS, 16
test suite and documentation, 19
transaction management, 13
validation, 11
WebSocket support, 14
Spring aware
ApplicationContext, 147
BeanNameAware, 147–148
Spring Batch
chunk-oriented processing, 774
configuration, 781–782
DataSourceConfig, 778–781
Gradle configuration, 775–776
ItemProcessor, 777–778
JSR-352 (*see* JSR-352)
scaling and parallel processing, 775
Spring Boot, 16, 786–790
ApplicationContext, 206
Artemis, 576–579
configurations, 205
Gradle Projects, 205, 208–209
“opinionated” approach, 204
RESTful-WS, 602–606
Spring Data JPA
custom queries
AuditorAwareBean, 449
@Column, 444
@EntityListeners, 444, 448
@MappedSuperclass, 444
@Param, 438
@Query, 437
SINGER_AUDIT, 441–444
SingerAuditRepository, 447
SingerAuditService, 445–446
testing, 439, 449–451
library dependencies, 429
repository abstraction, 430–436
Spring Expression Language (SpEL), 4, 69–72
Spring Framework
features, 798
functional web framework
content-Type text/event-stream, 809
handler function, 804–807
repository implementation, 802–803
REST controller, 807–808
RouterFunctions, 807
types of data streams, 799
WebFlux, 799–801
Java 9, 811
WebFlux, 814–816
Spring Integration
configuration, 792–795
JUnit 5
FluxGenerator, 823–827
TestEngine, 817–820
WebTestClient, 821–823
Message, 790–792
Spring modules
application, 24
ApplicationContext configuration, 32
configuration classes, 34–35
getBean() method, 33
Gradle, 26
Hello World application, 27–31
JAR files, 21–23
ListableBeanFactory, 31
Maven repository, 24–25
MessageProvider interface, 32
MessageRenderer interface, 28, 31
spring-context.jar, 33
spring documentation, 26
StandardOutMessageRenderer, 28–30
Spring MVC
configuration, 678–680
DispatcherServlet, 683–684
file upload support
configuration, 721–723
modifying controllers, 724–726
modifying views, 723–724
folder structure, 686
life cycle, 676–678
list view, 685–686
SingerController, 684–685
Tiles, 698–699
view, 681–682
WebApplicationContext, 675–676
Spring profiles
FoodProviderService, 187–190
Java configuration, 190–193
Spring Security, 597–602

configuration, 726–728
login functions, 729–731
project, 16
secure controller methods, 731
Spring testing
annotations, 618
categories, 616–617
DbUnit, 634–637
front-end unit test, 637
integration test
configuration class, 629–630
dependencies, 623
infrastructure classes, 627
Java configuration, 625–626
service-layer testing, 623–624
TestExecutionListener, 627–629
logic unit tests
create(), 621–623
dependencies, 619
listData(), 620–621
MVC Controllers, 620
unit test, service layer, 630–633
Spring Tool Suite (STS), 16
Spring transaction abstraction layer, 468–469
Spring type conversion
arbitrary types, 517–520
ConversionService, configure, 515–517
custom converter, 514–515
PropertyEditors, 511–514
Spring XD, 796–797
SQLException class, 319–321
SqlFunction class, 347–349
StandardLookupDemoBean class, 89
StaticMethodMatcherPointcut class, 236–239

T

Task Scheduling
annotation, 548–550
AppConfig, 546
asynchronous task
execution, 551–554
CarService, 544
CrusRepository, 543
DBInitializer, 542–543
dependencies, 537–538
execution, 546–547
JPA entity, 540–542
TaskExecutor, 554–556
task-namespace, 545
TaskScheduler abstraction, 539
trigger, task, and scheduler, 539
Theming and templating, 691–693
Throws advice, 230–232
Thymeleaf
extensions, 743–747
fragments, 741–743
Spring Boot, 738–740
Webjars, 747–749
Transaction management
AOP configuration, 486–487
data model
classes, 475–478, 480
countAll(), 485
findAll(), 482
findById(), 483
@Query, 484
save(), 483
Spring JPA project, 473–475
@Transactional annotation, 480–481
declarative approach, 490
global transactions
JTA transactions (*see* Java Transaction API (JTA) transactions)
PlatformTransactionManager, 469–470
programmatic transactions, 488–489
Spring transaction abstraction layer, 467–469
TransactionDefinition
interface, 471–472
TransactionStatus interface, 472–473

U

Unit test, service layer, 630–633
UnsatisfiedDependencyException, 116

V

Validation, Spring
AssertTrue, 534
bean validation, 528–529
custom validator, 531–534
dependencies, 510
JSR-349, 527, 535
object properties, 527–528
Spring validator interface, 525–527
testing, 530–531

W, X, Y, Z

Web applications
DAO (*see* Data Access Object (DAO))
i18n
configuration, 688–689
List View, 690–691
JSR-349, 709–711
requirements, 665
service layer, 735
SingerService, 672–673
singer view

adding, 708
edit, 703–707
show(), 700–703
Spring Boot, 732–733
Thymeleaf views (*see* Thymeleaf)
URLs views, 699
view templating, 693–698
WebSocket
API
front-end application, 758–759
Java configuration, 753
SockJS, 760–764
Spring MVC, 754–756
TextWebSocketHandler, 757
WebSocketConfigurer, 756–757
STOMP, 752